A scheduler for SCADA-based multi-source fusion systems

Rafael Corchuelo *, Miguel Toro

University of Seville, ETSI Informática, Avda. Reina Mercedes s/n, Sevilla E-41012, Spain

A B S T R A C T

Keywords:
Multi-source fusion
SCADA systems
Schedulers

In this article, we report on our experience regarding devising, implementing, and deploying a scheduler for multi- source fusion in the context of SCADA systems (Supervisory Control and Data Acquisition). They are challenging because they commonly rely on low-end boards with very limited computing, memory, and storage capabilities, but have to run hundreds if not thousands of agents that co-ordinate by means of complex multi-way rendez- vouses. Our work was carried out in the context of a solar plant in which we could easily confirm that not scheduling the rendez-vouses fairly may easily drive the system into as many as 3 779.10 critical-failure states per hour, whereas a straightforward solution to the problem can reduce the figure to 1 094.76 critical-failure states per hour. Unfortunately, that is far from zero, which is the ideal number. In the literature, there are several proposals to deal with this problem, but most of them could not be adapted to our context, namely: some of them can deal with two-way rendez-vouses only, whereas ours involve an average of 12.89 agents; others require to instrument the agents, but many of them are hardware devices that cannot be modified; a few others cannot work with rendez-vouses that can get intermittently enabled and disabled along an execution, which makes them of little interest in our context; and some require to use shared memory, which is an advanced hardware feature that is not supported by our low-end computing boards. The two proposals that we managed to adapt were not efficient enough in our context since they led to an average of 1 102.77 and 1 458.65 critical-failure states per hour, respectively. That motivated us to work on a new proposal that does not have any of the previous problems. It relies on a incremental approach that was implemented very efficiently using bounded counters and queues. Furthermore, the experimental results and the corresponding statistical analysis confirm that it works very well in practice.

1. Introduction

We work in the context of multi-source fusion systems that support industrial production plants. Such systems are commonly supported by Supervisory Control and Data Acquisition (SCADA) systems [8] that run a collection of agents that work co-ordinately so that the plants work safely and productively. The agents can be sensors, which are devices that sense the physical world and produce data, processors, which apply rules to the data in order to make decisions regarding which actions must be performed, and actuators, which perform the selected actions in the physical world.

Such fusion systems have some common features, namely: a) Typically, several sensors, processors, and actuators must co-ordinate in order to carry out a single piece of work. This implicitly requires implementing multi-way co-ordination protocols that are far more involved than in the case of two agents. b) Furthermore, the sensors and the actuators are hardware devices that communicate by means of standardised protocols, which means that the chances that they can be instrumented are very scarce. Instrumentation refers to changing the code of an agent so that it can be integrated into a new framework. c) In addition, the same agent may require co-ordination with several independent groups of agents, which implies that they must make decisions that do not neglect any of the groups too often because that might easily result in critical-failure states. d) Another issue is that the computing boards on which they run are very simple; in particular, it is not generally expected that they can provide shared memory. e) Last, but not least, their computing, memory, and storage capabilities are very scarce, which is not commonly due to budget constraints, but technical problems and verification requirements [39].

Remote-method invocation is the de-facto standard used in the context of SCADA systems. It is very adequate in cases in which the interactions amongst the agents can be easily implemented by means of message-passing protocols. Unfortunately, the complexity of the protocols increases as the number of agents involved in a piece of co-ordinated work increases. For instance, think of a system in which there are several actuators that must open or close some pumps depending on the

* Corresponding author.
E-mail addresses: corchu@us.es (R. Corchuelo), migueltoro@us.es (M. Toro).

decisions that they gather from several processors that need data from several sensors. Implementing a protocol that co-ordinates this piece of work is far from trivial and the result is not commonly simple to verify and test. Using a standard rendez-vous library helps abstract away from the details involved in the co-ordination and focus on the work to be done. Open MPI [33,40] is a well-known library in this context, but there are others that are specifically tailored for web services [15,27], reconfigurable systems [52], integration technologies [18–20], or Java systems [32].

Rendez-vouses help raise the level of abstraction, which is very appealing. Unfortunately, our experience proves that the scheduler must be carefully designed so that every rendez-vous that is enabled sufficiently often is executed sufficiently often since, otherwise, the system may easily enter critical-failure states. In our experimental study, we confirmed that a scheduler that does not take the problem into account may lead to as many as 3 779.10 critical-failure states per hour; we also confirmed that a straightforward solution can reduce the figure to 1 094.76 critical-state failures per hour, which is still very far from zero. A critical-failure state is a situation in which a part of the system is not working well. If such a state lasts for too long, then the system might malfunction, which may result in serious component damage, production interruptions, environmental pollution, or human injuries, just to mention a few common problems. Such states are typically due to a component that is malfunctioning (e.g., faulty sensors or actuators, or processors that work wrongly due to faulty boards, network failures, or wrong rules), but also due to actuators that do not perform their actions timely due to scheduling problems, which is our focus in this article. When the system enters such a state, the engineers get an alarm and execute an automated recovery procedure first; if it does not work, then the engineers override the system and try to keep it under control. If no solution is found, then the system is stopped and reset, which results in production interruptions and business losses.

The previous problem is related to a liveness property called fairness [16,25,34,53]. In the literature, there are many proposals to enforce fairness, but most of them cannot be adapted to our context: some proposals were designed to deal with two-way rendez-vouses only [3,5,35,49,56,57,60], but typical rendez-vouses in our context coordinate many more agents; others require to instrument the agents to build a scheduler into them [1,4,7,10,26], which can only be simulated by means of wrappers that are not generally possible to implement in low-end computing boards; some of them [4,7] cannot deal with rendez-vouses that get intermittently enabled or disabled along an execution, which hinders applying them to our context; and some of them require to use shared memory [42], which is an advanced hardware feature that is not available in low-end computing boards. There are two proposals that could be adapted [30,45], but our experimental analysis confirms that they are not efficient enough in our context because they drove our experimental system into as many as 1 102.77 and 1 458.65 critical-failure states per hour, respectively.

In this article, we report on the design of a scheduler for multi-source fusion systems that are supported by SCADA systems. We introduce it at a high level of abstraction that helped us prove that it is correct and complete; we also delve into how to implement it efficiently using bounded counters and queues, which is accompanied by a detailed study on the impact on correctness, completeness, time complexity, and space complexity; we deployed our proposal to a solar plant and performed a series of experiments that confirm that it is able to run the system without entering any critical-failure states; the intuitive conclusions were confirmed to be statistically sound using a well-established statistical method.

The rest of the article is organised as follows: Section 2 presents the related proposals and compares them to ours, Section 3 presents the scheduler, Section 4 delves into its implementation, Section 5 reports on the experimental analysis, and, finally, Section 6 summarises our conclusions.

2. Related work

In this section, we present the related work. First, we summarise the literature and then discuss on the problems that motivate our work.

2.1. Summary of the literature

We have found several architectures that were specifically designed to support distributed multi-source fusion systems, namely: Wang et al. [54] presented an architecture that arranges agents into four layers: a sensor driver layer, a logical sensor layer, a fusion unit layer, and a task unit layer; Lyu et al. [38], presented a three-layer architecture in which sensors are deployed to the bottom layer and report to a gateway layer that keeps the processors in the application layer away from their peculiarities; de Alba et al. [13] presented an architecture in which processors are implemented in a distributed blackboard that helps them work co-ordinately; and Rodríguez et al. [44] devised a similar architecture that was implemented using an agent-oriented platform.

The previous articles implicitly assume that there is a scheduler that is provided by the underlying implementation platform. Our focus is on the design of such a scheduler in a context in which computing power, memory, and storage are scarce and fairness must be enforced to prevent the system from entering critical-failure states. Fairness is a liveness property that basically requires the scheduler to make decisions that allow both the system and its individual components to progress as quickly as possible [16,25,34,53].

Olderog and Apt [42] set the foundations to enforce fairness by means of a scheduler for non-deterministic multi-choice commands. Their idea was to augment each thread with a collection of local counters that track how many times each alternative in a multi-choice command may be neglected in spite of being enabled. This simple idea set the foundation for many succeeding proposals, including ours.

The idea to enforce fairness using a scheduler was soon adopted in the context of distributed systems [48], where fairness is used to ensure that no resource remains idle forever or that no agent can starve forever. In the context of information fusion, there are some proposals that focus on resources that can be accessed by means of specific-purpose IoT networks [38,47] or general-purpose communication networks [11,23,36,46,50]; there are also some proposals that deal with arbitrary resources [3,5,35,49,56,57,60]. Regarding agents, there are several proposals that focus on allocating them to the data centres [12], the machines [22,31,41,55,58,59], or the cores [14,24,43] that have more computing power available, since this helps prevent starvation.

The previous proposals were devised in the context of systems whose agents interact by means of remote-method invocations. The problem is far more involved when the agents interact by means of rendez-vouses, chiefly if they co-ordinate more than two agents, aka. multi-way rendez-vouses. The subject of fairness in the previous proposals are the resources or the agents; in our context, the subject is a set of rendez-vouses: guaranteeing that they are executed fairly is the cornerstone to guaranteeing that the agents can progress. Francez and Forman [17] adapted Olderog and Apt's [42] proposal to this context. Some authors soon started to work on solutions that do not require any shared memory. Unfortunately, this research path was not straightforward because Tsay and Bagrodia [51] proved that no scheduler can guarantee fairness unless the agents ready only one rendez-vous stochastically, arbitrary delays can be introduced in the presence of a rendez-vous that is known to be enabled, or every computational step in every agent can be controlled by the scheduler. (Joung [29] characterised the exact subset of topologies in which these impossibility results hold.) The first research path was explored by Joung [30], the second one was explored by Ruiz et al. [45], but, to the best of our knowledge, nobody has explored the third one since it does not seem realistic. Recently, André et al. [1], Bensalem et al. [4], Bonakdarpour et al. [7], Brook et al. [10], and Jongmans and Arbab [26] have focused on enforcing fairness by instrumenting the agents so that a distributed scheduler is built into them.

2.2. Discussion

A good solution to implement a scheduler for multi-source fusion systems that are supported by SCADA systems must fulfil the following requirements:

– It must support multi-way rendez-vouses. We need a solution that supports both two-way and multi-way rendez-vouses since the latter are very common in our context. For instance, in our experimental analysis we dealt with a system in which the rendez-vouses co-ordinate an average of 12.89 agents.
– It must not require to instrument the agents. We need a solution that does not rely on instrumentation since the sensors and the actuators are black boxes that are provided by third-party vendors. The reader might argue that they might be wrapped using proxies that simulate the instrumentation; although that solution makes sense in a general context, it is not appropriate in our context because the proxies must be deployed to additional computing boards or share the existing computing boards with the processors.
– It must implement a strong fairness notion. Some authors have worked on a weaker notion of fairness that is not as stringent as ours. Instead of requiring a rendez-vous to be enabled from time to time, they require the rendez-vouses to be permanently enabled from a specific point in the execution onwards, which is far too restrictive in our context.
– It must not require any shared memory. We need a solution that does not require any shared memory, since the usual hardware that is available in our context does not support this feature.
– It must be efficient in terms of both time and space. We need a solution with a low time and space complexity due to the small computing, memory, and storage capabilities of the hardware used in our context.

The proposal by Olderog and Apt [42] does not deal with multi-way rendez-vouses, but multi-choice commands; this means that it provides a foundation, but adapting it, if possible, is not straightforward. Unfortunately, it requires to instrument the source code of the agents to which it is applied, which is not possible in a context in which sensors and agents are black boxes provided by third parties; it also requires shared memory, which is not available in our context; furthermore, the proposal requires to know the status of each alternative before making a decision on which must be executed, which is not efficient enough.

The proposals that focus on resource that are connected through a communications network [11,23,36,38,46,47,50] do not seem easy to adapt because they depend on a variety of parameters that are highly dependent on the network and difficult to extrapolate to other contexts, e.g., signal strength or waveform. The proposals that focus on general resources [3,5,35,49,56,57,60] might be applicable by mapping each rendez-vous onto a shared resource for which the agents have to compete; the problem is that they were not designed to deal with resources that must be shared by two or more agents. The proposals that focus on agents [12,14,22,24,31,41,43,55,58,59] are not applicable because their goal is to guarantee that agents can progress by allocating them to the computing devices that have more computing power available, independently from whether they co-ordinate or not with others; in these proposals, an agent that is readying a rendez-vous is considered to be waiting for input/output, which excludes it from the collection of agents whose progress must be guaranteed.

The adaptation of Olderog and Apt's [42] proposal by Francez and Forman [17] can deal with multi-way rendez-vouses, but it requires to instrument the agents, which is not possible in our context in the case of sensors and actuators, and it requires to know the status of every rendez-vous before making a decision on which must be executed, which is inefficient. The stochastic approach by Joung [30] might be used, but our experimental results prove that it is far too inefficient in our context; furthermore, it cannot deal with systems in which a single agent can stop executing naturally (i.e., it was programmed to work for a finite period of time only) or systems in which an agent increases its execution time monotonically (e.g., an agent that processes a log in a non-incremental manner). The approach by Ruiz et al. [45] might also be used in our context, but our experimental results also prove that it is not efficient enough because it requires to perform some global computations that are costly and it periodically requires to freeze the whole system to know the status of every rendez-vous. The recent proposals by André et al. [1], Bensalem et al. [4], Bonakdarpour et al. [7], Brook et al. [10], and Jongmans and Arbab [26] also require to instrument the agents to build a distributed scheduler into them, which is not generally possible in our context due to the limitations of the computing boards; furthermore, Bensalem et al.'s [4] and Bonakdarpour et al.'s [7] proposals deal with a weaker level of fairness that cannot deal with rendez-vouses that get enabled or disabled depending on the logic of the system.

Summing up: none of the proposals in the literature seems to be appropriate in our context, which motivated us to work on a new one.

3. The scheduler

In this section, we describe the scheduler. First, we present some preliminaries, then describe the proposal, and finally prove that it is correct and complete.

3.1. Preliminaries

Our preliminaries focus on the mathematical foundations and some fundamental concepts regarding systems, executions, and fairness.

Definition 1 (Mathematical foundations)**.** A transition is a tuple of the form $C_1 \xrightarrow{e} C_2$, where C_1 denotes a source configuration, C_2 denotes a target configuration, $\longrightarrow$ denotes a transition relation on configurations, and e denotes the event that fires the transition from the source to the target configuration. A configuration is a tuple that models the data and the execution state of a system. A transition chain is a sequence of transitions of the form $\langle C_i \xrightarrow{e_{i+1}} C_{i+1} \rangle_{i \geq 0}$. We use notation $\exists^\infty x$: $P(x)$ to mean that there are infinitely many values of x that satisfy predicate P. A map is a correspondence amongst the elements of two sets; we represent them using notation $\{k_i \mapsto v_i\}_{i=1}^{n}$, where $\{k_1, k_2, \ldots, k_n\}$ is the domain of the map and $\{v_1, v_2, \ldots, v_n\}$ is its range ($n \geq 0$). If $\leq$ is an order on the range of map μ, then $\hat{\mu}$ denotes its flattening, which is defined as a re-ordering of the domain such that $\mu(\hat{\mu}(i)) \leq \mu(\hat{\mu}(i+1))$, for every $1 \leq i < n$, $n \geq 0$); simply put: $\hat{\mu}$ sorts the domain of map μ according to its range. We use *RND* to denote a random variable that is uniformly distributed in the set of natural numbers.

Definition 2 (Systems)**.** A system is a tuple (R, A), where R denotes a set of rendez-vouses and A denotes a set of agents. A rendez-vous is a component that allows several agents to exchange data co-ordinately; they have a state that is composed of variables that the agents that co-ordinate on them can read and/or write. Rendez-vouses that co-ordinate only two agents are referred to as two-way rendez-vouses, whereas rendez-vouses that co-ordinate more than two agents are referred to as multi-way rendez-vouses. An agent is a component that consumes and/or produces the data that are exchanged through the rendez-vouses; the agents can be further classified as sensors, which sense the physical world and write data to the rendez-vouses, processors, which read data from the rendez-vouses, process them, and write the results to the rendez-vouses, and actuators, which read data from the rendez-vouses and perform actions in the physical world. Agents execute scripts that can either perform some local computation, ready some rendez-vouses, or execute a rendez-vous (which allows it to perform co-ordinated computations). There are a variety of technologies available to write the scripts [15,18–20,27,32,40,52]. An agent may ready several rendez-vouses at the same time, but only one of them may be selected for execution because agents are single-threaded in our context.

Remark 1. Multi-source information fusion is a process that combines data from many sources to produce an output that is somewhat useful. The previous idea fits our model perfectly: data are gathered from the physical world using sensors, they are processed using processors that produce an output, which is fed into some actuators that perform actions. Our model assumes that the agents co-ordinate by means of rendez-vouses, which can co-ordinate two or more agents. We are interested in multi-source fusion systems that are supported by SCADA systems [8]. (SCADA stands for Supervisory Control and Data Acquisition.) Such systems are commonly used to monitor and control the individual components of industrial plants in fields as diverse as telecommunications, water and waste control, oil and gas refining, transportation, or energy, just to mention a few examples. Typical SCADA systems are composed of sensors and actuators that are provided by different vendors and processors that are implemented in-house. The sensors and the actuators are very commonly implemented as black-box devices that communicate by means of standardised protocols; simply put: the chances that they can be instrumented are scarce (by instrumentation we mean changing their source code so as to adapt them to specific purposes). The processors are commonly deployed to low-end computing boards that provide very little computing power (in the range of a few MIPS), memory (in the range of a few KiB), or storage (in the range of a few hundreds of KiB).

Definition 3 (Executions)**.** We assume that the operational semantics of the technology used to implement a system was defined by means of a transition relation $\longrightarrow$ on configurations of the form $(a_1, a_2, \ldots, a_n, r_1, r_2, \ldots, r_m)$, where each a_i denotes the data and the execution state of an agent and each r_j denotes the data and execution state of a rendez-vous ($n \geq 0$, $m \geq 0$, $1 \leq i \leq n$, $1 \leq j \leq m$). An execution of a system is a transition chain of the form $\langle C_i \xrightarrow{r_{i+1}} C_{i+1} \rangle_{i \geq 0}$, which indicates that rendez-vous r_{i+1} is executed to transition from configuration C_i to configuration C_{i+1} ($i \geq 0$). A rendez-vous is said to be enabled if every agent that can ready it is readying it; we use notation *enabled*(*r, C*) to denote that rendez-vous r is enabled in configuration C.

Definition 4 (Fairness)**.** Fairness is a liveness property that requires that no rendez-vous that is enabled from time to time in an execution is executed finitely many times (possibly never). Formally speaking, execution $\langle C_i \xrightarrow{r_{i+1}} C_{i+1} \rangle_{i \geq 0}$ is fair as long as $\forall r : (\exists^\infty i : enabled(r, C_i)) \Rightarrow (\exists^\infty i : r = r_i)$. In the literature, there are some authors who consider a weaker notion that requires that no rendez-vous that is enabled permanently from a configuration onwards is executed finitely many times. Formally speaking, the previous execution is weakly fair if $\forall r : (\exists k : \forall i \geq k : enabled(r, C_i)) \Rightarrow (\exists^\infty i : r = r_i)$. We focus on the initial formulation since it is more general and stringent [16].

Remark 2. The notion of fairness is intimately related to the idea of infiniteness. The systems in which we are interested are designed to work twenty-four hours a day, seven days a week. Theoretically, their executions are infinite; for practical purposes that means that their lengths are not bounded by any pre-defined constant. Implementing fairness is challenging insofar the scheduler must guarantee that no rendez-vous that is enabled from time to time is executed finitely many times (possibly never). Note that one cannot check the previous property in finite time just taking a look at a partial execution. One the contrary, one must prove that the scheduler cannot produce any unfair executions by means of a theoretical analysis that commonly relies on a reductio ad absurdum argument. That is: one must assume that the scheduler may produce an unfair execution and then find a sequence of entailments that result in contradictions. In practice, enforcing fairness is appealing because it helps give every rendez-vous that is enabled from time to time a fair chance to be executed. In our context, this is particularly important in order prevent the system from entering critical-failure states that might result in disaster.

Example 1. Fig. 1 provides an excerpt from the definition of the system that we used in our experimental analysis. The actual system has 10 158 sensors of 321 different types, 124 processors of 97 different types, 2 034 actuators of 112 different types, and 215 rendez-vouses that co-ordinate 12.89 agents in average. Thus, the example focuses on a tiny part that exhibits a common co-ordination pattern that allows us to illustrate many of the core concepts.

Fig. 1.a sketches the scripts that are executed by a specific type of sensor, a specific type of processor, and a specific type of actuator when they co-ordinate on a collection of rendez-vouses of a specific type. Fig. 1.b sketches an instantiation of this system with three sensors, three processors, three actuators, and three rendez-vouses. We implicitly assume that there is a loader that instantiates the agents and the rendez-vouses and distributes them according to a predefined topology; we also assume that the loader provides an integer index to each agent or rendez-vous, which is passed as parameter i to the corresponding scripts.

The sensor script consists in a loop that senses data from the physical world and immediately readies rendez-vouses $RV(i)$ and $RV(i+1)$ in order to send the data to agents that can also ready these rendez-vouses (we implicitly assume that $i+1$ is computed modulo the number of rendez-vouses in the system). The processor script consists in a loop that readies rendez-vous $RV(i)$; when it is executed, the script reads the data sent by the corresponding sensor, applies a rule to compute an action, and writes the action to the rendez-vous. The actuator script also consists in a loop that first readies rendez-vous $RV(i)$ and then performs the action that it reads from that interaction. Recall that the script of the sensors and the actuators is built in the corresponding hardware devices; one can only configure the rendez-vouses to which they broadcast their data or the rendez-vouses from which the actuators read them.

Note how simple it is to specify such a complex system using rendez-vouses since each of them encapsulates the protocol required to synchronise multiple entities so that they can exchange data co-ordinately. This sample system sketches a pattern that is very common in practice: three different agents need to co-ordinate (a sensor, a processor, and an actuator) and there are some agents that may ready several rendez-vouses in order to provide their data to different agents (the sensors). Realise that this is an instance of the well-known Dining Philosophers problem; the reader can consult any textbook on distributed algorithms to realise how difficult it is to implement a solution to this problem using a protocol that relies on remote-method invocation [37].

Note, too, that a solution in which the sensors or the processors buffer data is not possible because of the tiny amount of memory or storage that is available in our context. In such a context, only a piece of data can be stored and it must be distributed to the agents that are interested in it at a specific point in time. Missing some data is not important as far as the system executes fairly. Unfortunately, our experiments prove that fairness does not happen naturally and must then be enforced by the scheduler, cf. Section 5. For instance, an execution of the previous system in which rendez-vouses $RV(1)$ and $RV(2)$ are executed intermittently and rendez-vous $RV(3)$ is never executed happens too often in practice, which may easily drive the system into critical-failure states.

3.2. *Description*

Our proposal works as follows: the scheduler associates a priority with every rendez-vous; the priority is a counter that indicates the number of times that a rendez-vous may be rejected for execution despite it is enabled; the scheduler arranges the rendez-vouses in a sequence in decreasing priority order (the smaller a counter, the higher the priority); it then searches the sequence for the first rendez-vous that is enabled; if it does not exist, then the system terminates; if it exists, then it is executed and its priority is reset to a random natural value, whereas the

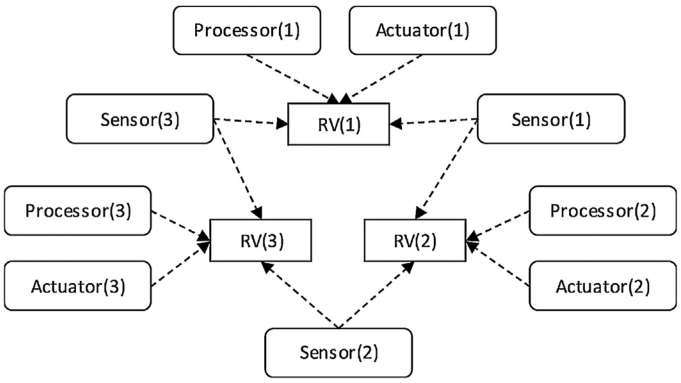
Fig. 1. Sketch of a sample system.

```
agent $Sensor(i)$
  loop
    sense $data$
    ready $RV(i)$, $RV(i+1)$ do
      write $data$
    end
  end
end
```

```
agent $Processor(i)$
  loop
    ready $RV(i)$ do
      read $data$
      compute $action$
      write $action$
    end
  end
end
```

```
agent $Actuator(i)$
  loop
    ready $RV(i)$ do
      read $action$
    end
    perform $action$
  end
end
```

a) Scripts of the sensors, processors, and actuators.



b) Instantiation of the system.

priorities of the rendez-vouses before it are kept and the others are increased by one. The key is that the scheduler needs to find only the first enabled rendez-vous in decreasing order of priority; the other rendez-vouses may be enabled or disabled, but the scheduler guarantees that they will be executed if they are enabled from time to time.

To formalise this idea, we need to generalise the transition relation $\longrightarrow$ that defines the operational semantics of our systems so that it works on extended configurations of the form (C, ρ), where C denotes a configuration of the system and ρ is a map that associates a priority with each rendez-vous. In the initial configuration, we assume that the state of every agent and rendez-vous is initialised to default values. We also assume that the initial priority map is initialised so that every rendez-vous gets a priority using random variable *RND*. After that, the system executes according to the following inference rule:

$$\dfrac{\begin{array}{l} C \xrightarrow{\widehat{\rho}(k)} C' \\ \forall 1 \le i < k : \neg enabled(\widehat{\rho}(i), C) \\ enabled(\widehat{\rho}(k), C) \end{array}}{(C, \rho) \xrightarrow{\widehat{\rho}(k)} (C', update(\rho, k))}$$

The rule has three antecedents and a consequent. The first antecedent requires that the system can transition from configuration C to configuration C' by executing rendez-vous $\widehat{\rho}(k)$, for some k; the second antecedent constraints the value of k so that no rendez-vous in $\widehat{\rho}$ in a position before k is enabled; the third antecedent requires the rendez-vous at position k in $\widehat{\rho}$ to be enabled. Summing up, the antecedents of the inference rule constraint k so that it denotes the position in $\widehat{\rho}$ of the first enabled rendez-vous, if any. The consequent states that if such a k exists, then the system can transition from configuration C to configuration C', but the priorities must be updated as follows:

$$\begin{aligned} update(\rho, k) \quad = \quad & \{r \mapsto \rho(r) \mid \exists 1 \le j \le k-1 : r = \widehat{\rho}(j)\} \cup \\ & \{r \mapsto RND \mid r = \widehat{\rho}(k)\} \cup \\ & \{r \mapsto \rho(r) - 1 \mid \exists k+1 \le j \le |\rho| : r = \widehat{\rho}(j)\} \end{aligned}$$

Simply put: the update keeps the priority of the rendez-vouses before position k, resets the priority of the selected rendez-vous, and increases the priority of the other rendez-vouses.

3.3. Correctness

Proving that the scheduler is correct amounts to proving that it cannot generate any unfair executions.

Theorem 1 (Correctness). *The scheduler does not generate any unfair executions.*

Proof. We proceed by reductio ad absurdum. Assume that the scheduler generates the following execution:

$$\lambda = \langle\, (C_i, \rho_i) \xrightarrow{r_{i+1}} (C_{i+1}, \rho_{i+1})\, \rangle_{i \ge 0}$$

If λ is unfair, that means that there must exists a rendez-vous r that is enabled from time to time in λ, but it is executed finitely many times only (possibly never). Let R denote the set of rendez-vouses. It can be partitioned into R_1, which has the rendez-vouses that are executed finitely many times in λ, and R_2, which has the rendez-vouses that are executed infinitely many times. There must exist a configuration (C_z, ρ_z) ($z \ge 0$) such that it is the last configuration in λ that is reached by executing a rendez-vous in R_1. Consider now a succeeding transition of the following form:

$$(C_i, \rho_i) \xrightarrow{r_{i+1}} (C_{i+1}, \rho_{i+1}) \ (i \ge z)$$

Assume that $\widehat{\rho_i}$ is of the form $\langle\, r_1, \dots, r_{q-1}, r_q, r_{q+1} \dots, r_n \,\rangle$. Assume, too, that the neglected rendez-vous is r_q. If neither $r_1, r_2, \dots, r_{q-1}$ are enabled, this is contradictory because the scheduler would have checked them and would have selected $r = r_q$ for execution since it is the first enabled rendez-vous in $\widehat{\rho_i}$. Consequently, there must exists an index p ($1 \le p < q$) such that r_p is the first enabled rendez-vous in $\widehat{\rho_i}$, i.e, $r_p = r_{i+1}$.

This means that when r_p is executed, ρ_i is updated before reaching configuration (C_{i+1}, ρ_{i+1}) so that the counter of the preceding rendez-vouses remain unchanged whereas the counter associated with r_p is assigned a random natural value (possibly zero), and the counters associated with the rest, including r_q, are decreased by one. Realise that the counters of the rendez-vouses before r_p are reset to random natural values or remain constant every time that they are checked, whereas the counter of rendez-vous r_q is decreased by one every time that it is not checked. Then, there must exist a configuration (C_s, ρ_s) $(s \geq z)$ such that r_q is enabled and $\rho_s(r_q)$ must be necessarily the minimum counter in that configuration. Consequently, r_q is the first rendez-vous in $\hat{\rho}_s$ and must have been selected for execution in that configuration, which concludes the proof. □

3.4. Completeness

Proving that the scheduler is complete amounts to proving that it can generate the whole set of fair executions, which we do in two steps: first, we introduce a lemma that proves that every execution of a system can be characterised with a series of priority maps in which the rendez-vous that is executed in each transition has priority one; next, we prove that the scheduler can generate executions with such priority maps, which renders it complete.

Lemma 1 (A characterisation of priority maps). *Let λ be a fair execution of the following form:*

$$\lambda = \langle\, (C_i,\ \rho_i) \xrightarrow{r_{i+1}} (C_{i+1}, \rho_{i+1})\,\rangle_{i \geq 0}$$

The following is a characterisation of the priority maps in which the rendez-vous that is executed in each transition has priority one:

$$\rho_i(r) = \begin{cases} \min\{j \geq i \mid r = r_{j+1}\} - i + 1 & \text{if } \exists s \geq i : r = r_{s+1} \\ \max\{j \geq i \mid enabled(r, C_j)\} - i + 1 & \text{if } (\nexists s \geq i : r = r_{s+1}) \wedge \\ & \quad (\exists s \geq i : enabled(r, C_s)) \\ 0 & \text{otherwise} \end{cases}$$

Proof. The intuition behind the definition is as follows:

1. Case 1: rendez-vous *r* is eventually executed from configuration (C_i, ρ_i) onwards. In this case, the value assigned to $\rho_i(r)$ is the number of configurations between (C_i, ρ_i) and the first succeeding configuration, including (C_i, ρ_i), in which *r* is executed.
2. Case 2: rendez-vous *r* is eventually enabled but never executed. In this case, the value assigned to $\rho_i(r)$ is the number of configurations between (C_i, ρ_i) and the configuration in which *r* is enabled for the last time, which might also be (C_i, ρ_i).
3. Case 3: rendez-vous *r* is never enabled from configuration C_i onwards. In this case, we define $\rho_i(r) = 0$.

The proof is straightforward: the rendez-vous that is executed in each transition falls within Case 1, so $\rho_i(r_{i+1}) = \min\{j \geq i \mid r = r_{j+1}\} - i + 1 = i - i + 1 = 1$; note, too, that the only smaller counter is zero, but according to the definition, this value is assigned to rendez-vouses that are never again enabled. Thus, the rendez-vous executed in each transition has priority one. □

Theorem 2 (Completeness). *The scheduler can generate the whole set of fair executions of a system.*

Proof. Our goal is to prove that the scheduler can generate executions in which the priority map in each configuration is characterised by the previous lemma. Consider any transition of the following form:

$$(C_i, \rho_i) \xrightarrow{r_{i+1}} (C_{i+1}, \rho_{i+1}) \ (i \geq 0)$$

Configuration (C_0, ρ_0) is special since it must be constructed by means of a procedure that initialises the data and execution state of every agent and rendez-vous and also initialises the priority map to random natural numbers. That is, nothing prevents the scheduler from generating a priority map as characterised by the previous lemma.

Thus, we have to focus on the succeeding transitions. In a configuration (C_i, ρ_i) $(i \geq 1)$, $\hat{\rho}_i$ has the following structure:

$$\hat{\rho}_i = \langle\, \overbrace{r_1, \ldots, r_{z-1}}^{\rho_i(r)=0}, \overbrace{r_z, \ldots, r_{z+p}}^{\rho_i(r)=1}, \overbrace{r_{z+p+1}, \ldots, r_n}^{\rho_i(r)>1} \,\rangle \ (z \geq 1, p \geq 0)$$

That is, there can be $z-1$ initial rendez-vouses whose priority equals zero, next $p+1$ rendez-vouses whose priority equals one, and then $n - z - p$ rendez-vouses whose priority is greater than one ($z - 1 \geq 0, p + 1 \geq 1, n - z + p \geq 0$). Therefore, there are three cases to analyse:

1. Case $r \in \{r_1, r_2, \ldots, r_{z-1}\}$: the rendez-vouses whose priority is zero are not enabled in (C_i, ρ_i) or any succeeding configuration. Otherwise, we can apply the first or the second case of the definition of ρ_i and they both lead to counters greater than or equal to one. Therefore, if these rendez-vouses are permanently disabled from configuration (C_i, ρ_i) onwards, then $\rho_{i+1}(r) = 0$, which is consistent with the update that the scheduler performs because it does not modify the counters that are associated with the disabled rendez-vouses at the beginning of $\hat{\rho}_i$.
2. Case $r \in \{r_z, r_{z+1}, \ldots, r_{z+p}\}$: the executed rendez-vous belongs to the set of rendez-vouses whose priority equals one, so it is one of the rendez-vouses in this case. We can assume that r_z is selected for execution because any arrangement of the rendez-vouses with the same priority is valid according to the definition of the scheduler. Thus, there are two sub-cases:
   (a) Case $r = r_z$: in this case, $\rho_{i+1}(r) \geq 0$, which is consistent with the update performed by the scheduler since it resets the counter of rendez-vous *r* to a random natural value.
   (b) Case $r \in \{r_{z+1}, \ldots, r_{z+p}\}$: in this case, the scheduler decreases the counter associated with *r* by one, so we need to prove that $\rho_{i+1}(r) = 0$, which indicates it will be permanently disabled from configuration (C_i, ρ_i) onwards. Indeed, because if $\rho_{i+1}(r)$ was greater than zero, this would imply that *r* might be enabled at least one more time from configuration (C_{i+1}, ρ_{i+1}) onwards, that is $\rho_i(r)$ should be greater or equal than two. Consequently, if this value is greater than zero in configuration (C_{i+1}, ρ_{i+1}), given that rendez-vous *r* has not been selected for execution in configuration (C_i, ρ_i), then $\rho_i(r) = 1 + \rho_{i+1}(r) \geq 2$ should hold, which contradicts the idea that $\rho_{i+1}(r)$ might be greater than zero.
3. Case $r \in \{r_{z+p+1}, r_{z+p+2}, \ldots, r_n\}$: the scheduler decreases the counter associated with *r* by one, i.e., we need to prove that $\rho_{i+1}(r) = \rho_i(r) - 1$. Indeed, because if $\rho_i(r)$ was greater than one, this would mean that there exists $\rho_i(r)$ configurations before *r* is selected or enabled for the last time, including (C_i, ρ_i). Therefore, this happens one less time in configuration (C_{i+1}, ρ_{i+1}), i.e., $\rho_{i+1}(r) = \rho_i(r) - 1$.

We have now explored every possible case and we have proved that the scheduler can update the priority map in each transition according to the characterisation of the previous lemma, which concludes the proof. □

4. The implementation

In this section, we describe the implementation. First, we present some preliminaries on how to implement maps, then describe the implementation of the scheduler, next analyse its time and space complexity, and, finally, analyse the impact of implementing it using bounded counters.

4.1. Preliminaries

The key to an efficient implementation is to select the right data structures. Our implementation relies heavily on maps, which we use to store a rendez-vous specification map in which each rendez-vous is mapped onto the agents that can eventually ready it, a readiness map in which they are mapped onto the agents that are readying them in a particular configuration, and a priority map in which they are mapped onto their corresponding priority counters.

```
method $scheduler(R,\ T)$
    $\rho := makeQueue()$
    $C := \emptyset$
    for $r \in R$ do
        $insert(\rho,\ r,\ RND)$
        $C(r) := \emptyset$
    end
    loop
        receive $Ready(S)$ from agent $a$
        for $s \in S$ do
            $C(s) := C(s) \cup \{a\}$
        end
        $(\rho,\ r) := transition(\rho,\ C,\ T)$
        if $\neg\ isEmpty(\rho)\ \wedge\ r \neq null$ then
            send $Execute(r)$ to agents in $C(r)$
            for $s \in R$ do
                $C(s) := C(s) \setminus T(r)$
            end
        end
    until $isEmpty(\rho)$
end
```

Fig. 2. Main method of the scheduler.

The specification and the readiness map can be implemented using a hash-table approach since the key to performance is to be able to fetch the sets of agents that may ready or are readying each rendez-vous as efficiently as possible. No other operation is performed on them.

The priority map requires a careful selection of the implementation because we need to flatten it many times. Unfortunately, a hash-table approach does not allow to perform this operation efficiently. Fortunately, Brodal [9] devised a priority map implementation that performs the following operations in $O(1)$ worst-case time: *makeQueue*(), which returns a new empty queue, *isEmpty*(ρ), which returns whether queue ρ is empty or not, *select*(ρ), which returns the element with the maximum priority in queue ρ, *insert*(ρ, e, p), which inserts element e with priority p into queue ρ; additionally, it implements the following operation in $O(\log n)$ worst-case time: *delete*(ρ), which deletes the element with the minimum priority from queue ρ and returns it. Furthermore, it does not require more than $O(n)$ space to store a queue with n elements. This makes Brodal's [9] approach a very good choice in our context.

4.2. Scheduler implementation

Fig. 2 shows the main method of the scheduler. It works on a set of rendez-vouses R and a specification map T that indicates which processes must ready each rendez-vous so that it becomes enabled. First, it initialises queue ρ, which implements the priority map, and readiness map C, which keeps track of which agents have readied which rendez-vouses. The former is initialised so that each rendez-vous is assigned a random natural value and the latter is initialised so that each rendez-vous is assigned an empty set of agents. The method then starts its main loop, which consists of the following steps: a) it first waits for a notification of the form *Ready*(S) from any agent a, which indicates that it is readying the rendez-vouses in set S; b) it then loops over S and updates map C to record that agent a is readying that subset of rendez-vouses; c) it then invokes method *transition*, which implements the inference rule that specifies the scheduler and returns the updated priority map and the rendez-vous selected for execution, if any; c) if the *transition* method returns a non-empty map and a non-null rendez-vous, then that rendez-vous may be executed immediately, which requires to send an execute notification to the agents that had readied it and to update the corresponding readiness sets. The main method terminates when method *transition* returns an empty priority map, which means that no rendez-vous is enabled and the system has stopped.

Fig. 3.a shows the ancillary *transition* method. It works on a priority map ρ, a readiness map C, and a specification map T. It firsts iterates over the priority map to find the first rendez-vous that is enabled. The iteration is implemented in a loop in which the rendez-vouses in queue ρ are iteratively transferred to the ancillary queue ρ'. In each iteration, the highest priority rendez-vous is first selected from queue ρ; if the set of processes that have readied that rendez-vous equals the set of processes indicated by the specification map T, then that rendez-vous is enabled. In this case, we transfer all of the elements that remain in ρ to queue ρ' by means of the ancillary *transferAll* method; otherwise, we move the disabled rendez-vous from queue ρ to queue ρ' using the *transferOne* method. If one rendez-vous is found to be enabled, then *transition* returns queue ρ', which is an updated version of ρ, and the rendez-vous selected for execution; otherwise, it returns an empty queue and a null rendez-vous.

The ancillary *transferOne* and *transferAll* methods are shown in Fig. 3.b and c. Both methods require their parameters to be in/out because Brodal's [9] proposal assumes that queues are handled by reference. Method *transferOne* works on two queues ρ and ρ'; it transfers the rendez-vous with the maximum priority in queue ρ to queue ρ'; the priority is not changed at all since the highest-priority rendez-vous is assumed to be disabled. Method *transferAll* works on two queues ρ and ρ', too; it transfers all of the rendez-vouses in ρ to ρ' so that the highest-priority one (which is assumed to be enabled) gets its priority reset to a random natural number and the remaining ones get their priorities increased.

4.3. Complexity analysis

In this section, we prove that the scheduler does not require more than $O(m\log m)$ time to process a ready notification from an agent and it does not consume more than $O(n\,m)$ space, where m denotes the number of rendez-vouses and n denotes the number of agents. These upper bounds are sensible to implement the scheduler in a SCADA context in which computing power, memory, and storage space are very limited.

The sketch of the proof is as follows: first, Lemma 2 proves that method *transferOne* does not require more than $O(\log m)$ time and $O(1)$ space; next, Lemma 3 proves that method *transferAll* does not require more than $O(m\log m)$ time and $O(1)$ space; then, Lemma 4 proves that method *transition* does not require more than $O(m\log m)$ time and $O(m)$ space; finally, Theorem 3 proves the time and space upper bounds regarding the scheduler.

Lemma 2 Method. *transferOne. Let m denote the number of rendez-vouses in a system. A call to transferOne*(ρ, ρ') *does not require more than* $O(\log m)$ *time and it does not consume more than* $O(1)$ *space.*

Proof. A call to *transferOne*(ρ, ρ') wraps a call to *delete*, which can be completed in $O(\log m)$ worst-case time, followed by a call to *insert*, which can be executed in $O(1)$ worst-case time. The call requires to temporarily store the element retrieved from ρ before inserting it in ρ', which does not exceed $O(1)$ space. As a conclusion, a call to *transferOne*(ρ, ρ') does not require more than $O(\log m)$ time and it does not consume more than $O(1)$ space. □

Lemma 3 Method. *transferAll. Let m denote the number of rendez-vouses in a system. A call to transferAll*(ρ, ρ') *does not require more than* $O(m\log m)$ *time and it does not consume more than* $O(1)$ *space.*

Proof. The first two statements do not require more than $O(\log m)$ time since they call the *delete* and the *insert* methods on a queue that has a maximum of m elements. The main loop calls again the *delete* and the *insert* methods a maximum of $m - 1$ times; in each iteration, the size of queue ρ decreases by one. Thus, the following is an upper bound to the time required to execute the main loop:

$$\log(m-1) + \log(m-2) + \log(m-3) + \cdots + \log 2 + \log 1 =$$

```
method $transition(\rho,\ C,\ T)$
    $enabled := false$
    $\rho' := makeQueue()$
    while $\neg\ enabled \wedge \neg\ isEmpty(\rho)$ do
        $(r,\ c) := select(\rho)$
        $enabled := (C(r)\ =\ T(r))$
        if $enabled$ then
            $transferAll(\rho,\ \rho')$
        else
            $transferOne(\rho,\ \rho')$
        end
    end
    if $enabled$ then
        return $(\rho',\ r)$
    else
        return $(makeQueue(),\ null)$
    end
end
```

a) The *transition* method.

```
method $transferOne(\rho\ [in\ out],\ \rho'\ [in\ out])$
    $(r,\ c) := delete(\rho)$
    $insert(\rho',\ r,\ c)$
end
```

b) The *transferOne* method.

```
method $transferAll(\rho\ [in\ out],\ \rho'\ [in\ out])$
    $(r,\ c) := delete(\rho)$
    $insert(\rho',\ r,\ RND)$
    while $\neg\ isEmpty(\rho)$ do
        $(r,\ c) := delete(\rho)$
        $insert(\rho',\ r,\ c\ -\ 1)$
    end
    $\rho := \rho'$
end
```

c) The *transferAll* method.

Fig. 3. Ancillary methods.

$$= \sum_{i=1}^{m-1} \log(m-i) < \sum_{i=1}^{m} \log m = \log \prod_{i=1}^{m} m = \log m^m = m \log m$$

Therefore, a call to *transfer*(ρ, ρ') does not require more than $O(m \log m)$ time. Furthermore, the call requires only a temporary variable to store the elements that are transferred from ρ to ρ', which means that it does not require more than $O(1)$ space. □

Lemma 4 Method. *transition. Let m denote the number of rendez-vouses in a system. A call to transition(ρ, C, T) does not require more than $O(m \log m)$ time and it does not consume more than $O(m)$ space.*

Proof. The first two statements run in $O(1)$ worst-case time. The main loop is executed a maximum of m times; in each iteration, the call to method *select* does not require more than $O(1)$ time, the check for enablement can be implemented in $O(1)$ worst-case time using hashed sets (a trivial implementation would not consume more than $O(n)$ time, where n denotes the number of agents in the system), and the call to the *transferOne* method was proved not to consume more than $O(\log m)$ time. So the main loop does not require more than $O(m \log m)$ time to execute; in the final iteration it might call method *transferAll*, which does not require more than $O(m \log m)$ time. The final check to return the value of the method does not require more than $O(1)$ time. The call requires to store a Boolean variable and a new queue ρ', only. As a conclusion, a call to *transition*(ρ, C, T) does not require more than $O(m \log m)$ time and it does not consume more than $O(m)$ space. □

Theorem 3 (The scheduler). *Let m and n denote the number of rendez-vouses and agents in a system, respectively. The scheduler does not require more than $O(m)$ time to initialise, it does not require more than $O(m \log m)$ time to process a ready notification from an agent, and it does not require more than $O(n\,m)$ space.*

Proof. The initialisation sentences do not require more than $O(m)$ time to execute because creating queue ρ and readiness map C can be accomplished in $O(1)$ worst-case time; the initialisation loop iterates over the set of rendez-vouses and initialises their priorities and the sets of agents that are readying them in $O(1)$ worst-case time. Thus, the conclusion is that the scheduler does not require more than $O(m)$ time and more than $O(m)$ space to initialise.

The main loop blocks on the receive statement until an agent sends a ready notification. It then updates the readiness map, which can be easily implemented in $O(m)$ worst-case time, calls the *transition* method, which does not require more than $O(m \log m)$ time, and, if possible, sends an execute notification to a subset of agents, which does not require more than $O(m)$ time, and updates the readiness map, which does not require more than $O(m)$ time. As a conclusion, the scheduler does not require more than $O(3\,m + m \log m) \subseteq O(m \log m)$ time to process a ready notification from an agent.

The scheduler must store the set of rendez-vouses, which requires $O(m)$ worst-case space, and the specification map, which does not require more than $O(n\,m)$ space. Furthermore, it must store queue ρ, which requires $O(m)$ worst-case space, readiness map C, which requires $O(n\,m)$ worst-case space, and every call to method *transition* requires $O(m)$ worst-case space. As a conclusion, the scheduler does not require more than $O(m + n\,m) \subseteq O(n\,m)$ space. □

4.4. Impact of using bounded counters

The counters on which our proposal relies must be implemented using data registers that provide a limited range of integer or natural numbers. In the following theorems, we analyse the impact of using bounded counters on correctness and completeness.

Theorem 4 (Correctness preservation). *An implementation in which counters range in interval $[-|R|+1, 0]$, where R denotes the set of rendez-vouses in a system, preserves correctness.*

Proof. The proof follows from reductio ad absurdum. Assume that an implementation of the scheduler produces an unfair execution of the following form:

$$\lambda = \langle\, (C_i, \rho_i) \xrightarrow{r_{i+1}} (C_{i+1}, \rho_{i+1})\, \rangle_{i>0}$$

Obviously, ρ_0 must set every counter to zero since this is the only random natural value that can be generated. If λ is unfair, it implies that there exists an element $r \in R$ that is enabled from time to time, but it is executed finitely many times only (possibly never). The set of rendez-vouses R can be partitioned into sets R_1 and R_2 so that R_1 is the subset of rendez-vouses that are executed finitely many times in λ and $R_2 = R \setminus R_1$. There is also a $z \geq 0$ such that the rendez-vouses executed from configuration $(C_z,\ \rho_z)$ onwards belong to set R_2.

If rendez-vous r is enabled from time to time in λ, then there are infinitely many configurations $(C_s,\ \rho_s)$ $(s \geq z)$ in which both r_{s+1} and r are enabled. Without any loss of generality, we can assume that $\widehat{\rho_s}$ is of the form $\widehat{\rho_s} = \langle\, r_1, r_2, \ldots, r_{p-1}, r_p, r_{p+1}, \ldots r_n\, \rangle$ $(p > 0)$ and that r_p is the rendez-vous selected for execution in configuration $(C_s,\ \rho_s)$. Obviously, the counter associated with the neglected rendez-vous must have a value

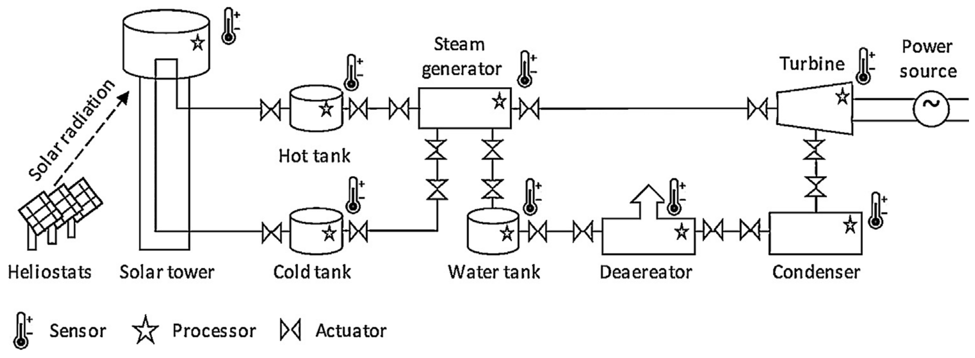


Fig. 4. Schema of our solar power plant.

greater than or equal to $\rho_s(r_p)$; otherwise, it would be before r_p in $\widehat{\rho_s}$ and, given that it is enabled, it would have been selected, which contradicts our hypothesis. Therefore, the neglected rendez-vous must be after r_p in $\widehat{\rho_s}$. In this situation, the counter associated with r_p is reset to zero, that is, it must be at the rear of $\widehat{\rho_{s+1}}$ in the next configuration. Without any loss of generality, we can assume that it is the last rendez-vous in $\widehat{\rho_{s+1}}$ because if there are other rendez-vouses with the same priority any ordering is valid according to the definition of the scheduler. Consequently, the priorities associated with $r_{p+1}, r_{p+2}, \ldots, r_n$ in the next configuration are decreased by one; that is, there should exist a configuration (C_q, ρ_q) $(q \geq s)$ such that the priority associated with the neglected rendez-vous must be necessarily higher than the priority associated with any other rendez-vous and it should be at the beginning of $\widehat{\rho_q}$. Since its counter is not altered, it should be selected when a configuration in which it is enabled is reached, which contradicts our hypothesis. Furthermore, a priority may be increased $|R| - 1$ times in the worst case because the corresponding rendez-vous can remain unchecked $|R| - 1$ times at most. As a conclusion, an implementation that provides counters in range $[-|R| + 1, 0]$ preserves correctness. □

Theorem 5 (Completeness preservation)**.** *An implementation that implements priorities with bounded counters does not preserve completeness.*

Proof. Assume that we have bounded counters that range in the integer interval $[-|R| + 1, m]$, for some $m \geq 0$, i.e., $|R| + m$ different integer values are available. Assume that there exists a fair execution in which rendez-vous r is selected only once at configuration $(C_{|R|+m}, \rho_{|R|+m})$. Realise that $\rho_0(r)$ should be at least $|R| + m + 1$, which is out of range. As a conclusion, implementing the scheduler with bounded counters does not preserve completeness. □

Remark 3. Common computing boards provide data registers that allow to operate with integer counters in range $[-2^{b-1}, 2^{b-1} - 1]$ or natural counters in range $[0, 2^b - 1]$, where b denotes the number of bits of the data registers. The largest priority that a rendez-vous may have is $-|R| + 1$. Thus, implementing the priorities using integer counters amounts to wasting the values in range $[-2^{b-1}, -|R|]$, which are quintessential to generate as many fair executions as possible. Thus, our proposal is to implement the priorities using natural counters, which range in interval $[0, 2^b - 1]$, and then use a simple linear translation in which $-|R| + 1$ corresponds to 0 and $2^b - |R|$ corresponds to $2^b - 1$. This way, the priorities range in an interval in which no value is wasted.

5. Experimental analysis

In this section, we describe our experimental analysis. First, we describe the solar plant in which we performed our experiments, then report on the experimental environment, the baselines and the competitors, and finally present our experimental results and analyse them statistically. The experimentation was performed in the context of an industrial research project with a company that wished to explore new models that minimise the gap between the specification produced by their research team and the implementation carried out by their engineering team [6]. Due to non-disclosure clauses, we cannot report on every detail from an engineering point of view, but we can provide an overall picture that is enough to assess our proposal from a scientific point of view.

5.1. *Description of the system*

Solar plants have proven to be very efficient and eco-friendly because they use molten salt and water that can be recycled many times and do not significatively pollute the environment. They have recently become economically feasible due to the positive balance between production costs and sales benefits [28].

Fig. 4 sketches the solar plant in which we conducted our experiments. The flow to generate electricity starts at the cold tank, which stores molten salt at roughly 290° C. At such temperatures, the salt is a fluid that can be easily pumped into the solar tower, where it is heated up to 565° C thanks to the solar radiation that is concentrated there by means of an array of heliostats. The salt is then pumped into the hot tank, where it can be stored at about 390° C. The salt is then pumped from the hot tank into the steam generator, where heat is transferred to water, which boils and becomes steam that is pumped into a standard turbine that generates electricity. The salt is returned to the cold tank, where it can start a new cycle and the steam is condensed, deareated (a process that removes dissolved oxygen), and returned to the water tank. The flows of salt, steam, and water need to be carefully controlled so that the plant works optimally depending on the weather conditions and the amount of electricity that must be produced.

In the actual system, there are 10 158 sensors, 124 processors, and 2 034 actuators. There are 321 types of sensors, which produce data regarding temperature, pressure, voltage, intensity, conductibility, vibrations, acoustics, force, or strain, just to mention a few examples. There are 97 types of processors, which execute rules that were learnt using a variety of machine-learning procedures, try to recover the system automatically when it enters a critical-failure state, or send data and statistics to an external machine-learning system that improves the rules incrementally. There are 112 types of actuators, most of which are pumps that help keep the flows of molten salt, water, and steam under control. There are also 215 rendez-vouses that co-ordinate 12.89 agents in average; the smallest rendez-vouses are two-way and the largest ones are 34-way.

5.2. *The experimentation environment*

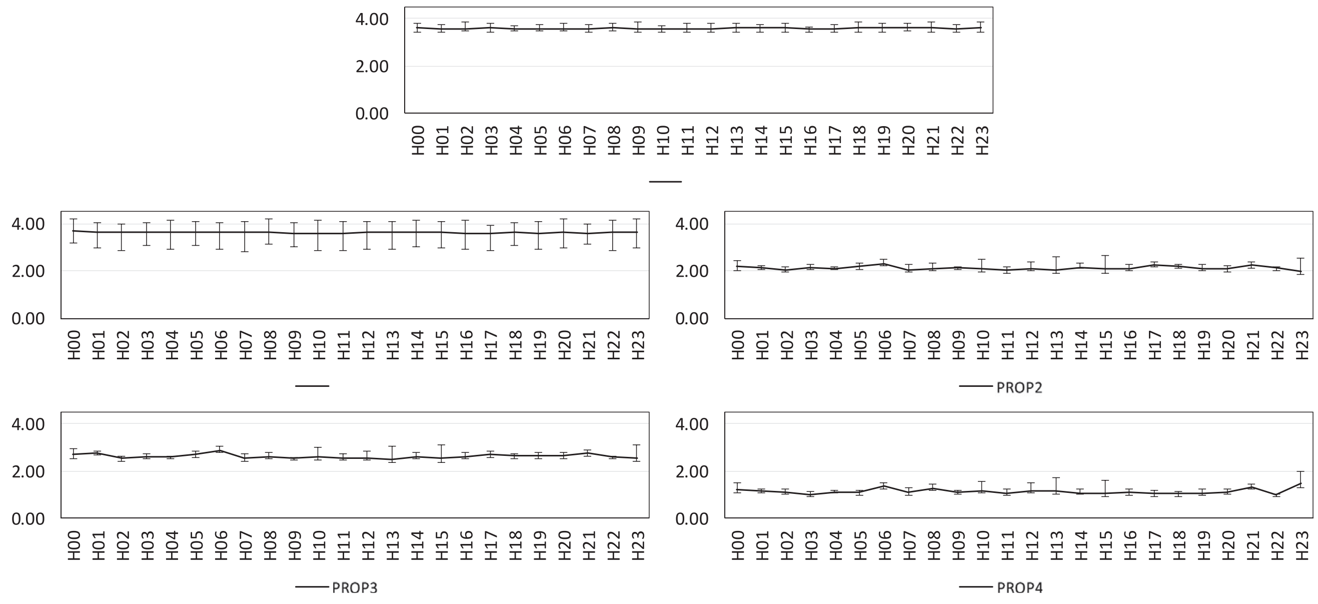
The sensors and the actuators are provided by many different vendors, but they all communicate by means of industrial standards that help integrate them with a variety of technologies, including ours.

The processors were developed in-house and they were deployed to an array of 124 Arduino Mega 2560 REV3 boards, each of which is equipped with one ATmega2560 CPU that delivers up to 16 MIPS when

Table 1
Average performance (millions of rendez-vouses per hour).

	SCH				PROP1				PROP2				PROP3				PROP4			
Hour	Min	Max	Avg	Dev	Min	Max	Avg	Dev	Min	Max	Avg	Dev	Min	Max	Avg	Dev	Min	Max	Avg	Dev
H00	3.46	3.82	3.61	0.09	3.16	4.18	3.68	0.30	2.00	2.43	2.18	0.11	2.14	3.14	2.73	0.30	0.33	2.12	1.24	0.49
H01	3.45	3.76	3.57	0.09	2.97	4.04	3.63	0.28	2.08	2.25	2.14	0.04	2.14	3.17	2.74	0.32	0.03	1.97	1.15	0.58
H02	3.47	3.85	3.59	0.10	2.85	3.97	3.63	0.27	1.94	2.15	2.06	0.05	2.03	3.10	2.56	0.32	0.04	1.82	1.14	0.50
H03	3.45	3.83	3.63	0.12	3.10	4.02	3.65	0.23	2.06	2.29	2.16	0.06	2.10	3.15	2.63	0.27	0.16	2.04	1.02	0.59
H04	3.49	3.69	3.58	0.05	2.91	4.15	3.64	0.31	2.05	2.16	2.10	0.03	2.16	3.03	2.59	0.29	0.23	1.96	1.13	0.53
H05	3.49	3.76	3.60	0.09	3.08	4.08	3.63	0.27	2.08	2.32	2.21	0.06	2.21	3.33	2.73	0.30	0.02	1.85	1.09	0.48
H06	3.48	3.79	3.59	0.10	2.90	4.05	3.62	0.29	2.20	2.47	2.32	0.06	2.44	3.34	2.90	0.28	0.10	2.30	1.37	0.53
H07	3.44	3.76	3.59	0.08	2.83	4.08	3.63	0.31	1.95	2.26	2.06	0.07	2.01	3.08	2.54	0.27	0.10	1.77	1.10	0.47
H08	3.50	3.82	3.62	0.10	3.12	4.17	3.64	0.27	2.03	2.31	2.11	0.06	2.13	3.06	2.61	0.26	0.13	1.93	1.28	0.45
H09	3.43	3.87	3.60	0.11	3.03	4.03	3.61	0.27	2.05	2.19	2.13	0.04	2.15	3.09	2.54	0.30	0.37	1.80	1.13	0.40
H10	3.45	3.69	3.58	0.07	2.88	4.13	3.61	0.29	1.97	2.48	2.09	0.10	2.14	3.02	2.61	0.31	0.03	2.32	1.19	0.59
H11	3.44	3.79	3.57	0.09	2.85	4.10	3.61	0.28	1.90	2.18	2.01	0.06	2.03	3.01	2.56	0.33	0.27	1.90	1.09	0.54
H12	3.44	3.81	3.60	0.09	2.94	4.08	3.66	0.29	1.99	2.40	2.08	0.08	2.17	2.97	2.54	0.26	0.09	2.00	1.20	0.59
H13	3.44	3.81	3.61	0.09	2.90	4.09	3.63	0.28	1.88	2.58	2.02	0.13	2.03	3.00	2.53	0.29	0.10	1.95	1.16	0.46
H14	3.45	3.75	3.60	0.07	3.02	4.15	3.62	0.29	2.09	2.32	2.14	0.05	2.18	3.13	2.61	0.28	0.10	1.88	1.08	0.56
H15	3.44	3.82	3.60	0.09	2.95	4.12	3.66	0.30	1.90	2.64	2.07	0.14	2.06	3.28	2.54	0.35	0.04	1.90	1.07	0.50
H16	3.43	3.66	3.55	0.06	2.94	4.16	3.59	0.25	2.02	2.29	2.11	0.06	2.11	3.12	2.61	0.30	0.27	1.87	1.09	0.48
H17	3.45	3.76	3.59	0.09	2.85	3.94	3.59	0.29	2.15	2.41	2.27	0.06	2.20	3.23	2.70	0.28	0.02	1.72	1.05	0.53
H18	3.45	3.84	3.60	0.11	3.10	4.05	3.62	0.28	2.09	2.29	2.20	0.05	2.28	3.10	2.66	0.25	0.10	1.98	1.06	0.59
H19	3.45	3.82	3.63	0.09	2.89	4.07	3.61	0.29	2.00	2.26	2.10	0.06	2.24	3.10	2.65	0.29	0.10	2.01	1.08	0.56
H20	3.48	3.79	3.63	0.10	2.96	4.17	3.62	0.30	1.97	2.21	2.07	0.07	2.06	3.16	2.65	0.33	0.22	1.92	1.12	0.59
H21	3.43	3.85	3.63	0.12	3.12	3.96	3.59	0.25	2.13	2.39	2.24	0.06	2.30	3.24	2.78	0.27	0.10	2.16	1.34	0.53
H22	3.43	3.78	3.59	0.09	2.89	4.17	3.64	0.32	2.02	2.17	2.12	0.03	2.15	3.15	2.61	0.28	0.05	1.69	1.01	0.49
H23	3.43	3.84	3.62	0.10	2.94	4.17	3.65	0.28	1.86	2.55	2.01	0.13	1.92	3.03	2.56	0.29	0.56	1.97	1.47	0.42

a) Raw experimental results.



b) Average/deviation chart.

running at 16 MHz, 256 KiB of flash memory (of which 8 KiB are dedicated to the standard boot loader), 8 KiB of SRAM memory, and 4 KiB of EEPROM memory. Note that ATmega2560 CPUs have 8-bit data registers, which means that doing computations with 16-bit counters reduces their computing power to 4–8 MIPS [2].

Each board was extended with an Arduino Ethernet REV3 shield that helps the processors communicate through a standard IEEE 802.3u network. This shield was selected because it has two additional advantages: first, it can power the boards from the Ethernet connector, which reduces the number of power cords and power sockets required; second, it has a Micro SD card socket that helped us store the statistics without consuming any storage in the main board.

Each board was triplicated to implement the main system, a fail-over system, and an additional test system in which engineers can experiment with new proposals without interfering with the other systems.

5.3. The baselines and the competitors

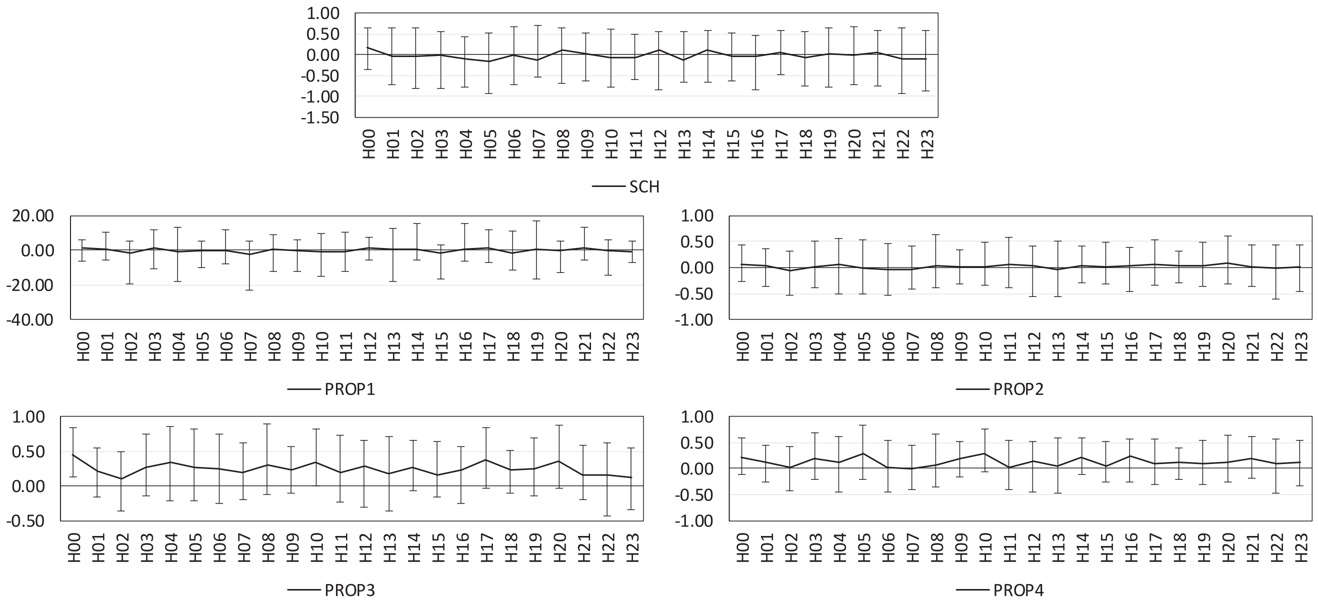
We implemented two straightforward proposals that serve as baselines for experimentation purposes, namely:

– PROP1: this baseline executes the rendez-vouses as soon as they are detected to be enabled. Obviously, PROP1 does not guarantee that the executions are fair; it is intended to prove that fairness must be enforced since, otherwise, the system may easily run into

Table 2
Deviation in performance (thousands of rendez-vouses per hour).

	SCH				PROP1				PROP2				PROP3				PROP4			
Hour	Min	Max	Avg	Dev	Min	Max	Avg	Dev	Min	Max	Avg	Dev	Min	Max	Avg	Dev	Min	Max	Avg	Dev
H00	-0.36	0.63	0.18	0.33	-6.57	5.80	1.03	2.76	-0.26	0.44	0.06	0.19	-0.28	1.21	0.45	0.44	-0.29	0.77	0.21	0.30
H01	-0.73	0.65	-0.03	0.38	-5.41	10.08	0.37	3.97	-0.35	0.35	0.03	0.22	-0.43	1.07	0.22	0.44	-0.38	0.69	0.12	0.29
H02	-0.82	0.65	-0.05	0.41	-19.30	5.13	-2.05	5.40	-0.53	0.32	-0.07	0.24	-0.48	1.28	0.10	0.51	-0.46	0.62	0.02	0.34
H03	-0.81	0.56	0.00	0.39	-10.54	11.84	1.09	4.83	-0.39	0.50	0.01	0.26	-0.27	1.67	0.26	0.43	-0.50	0.74	0.19	0.30
H04	-0.80	0.44	-0.10	0.35	-18.08	13.22	-0.69	5.60	-0.52	0.55	0.05	0.23	-0.40	1.25	0.35	0.50	-0.50	0.76	0.12	0.33
H05	-0.94	0.52	-0.15	0.40	-10.32	4.95	-0.42	3.30	-0.52	0.52	-0.03	0.27	-0.53	1.06	0.27	0.45	-0.47	0.96	0.28	0.41
H06	-0.73	0.68	0.00	0.34	-7.94	11.81	-0.22	3.74	-0.54	0.47	-0.05	0.27	-0.30	0.93	0.24	0.38	-0.55	0.86	0.03	0.38
H07	-0.54	0.70	-0.14	0.38	-23.23	5.51	-2.45	5.52	-0.42	0.41	-0.03	0.24	-0.32	0.98	0.19	0.38	-0.33	0.53	0.00	0.27
H08	-0.71	0.63	0.10	0.36	-12.20	8.95	0.80	3.76	-0.40	0.62	0.03	0.24	-0.38	1.37	0.30	0.43	-0.36	0.71	0.07	0.34
H09	-0.63	0.53	0.02	0.34	-12.08	5.80	-0.37	3.54	-0.32	0.35	0.01	0.18	-0.43	1.14	0.22	0.36	-0.31	0.76	0.18	0.34
H10	-0.78	0.60	-0.06	0.31	-15.32	9.27	-0.93	5.49	-0.33	0.48	0.01	0.22	-0.26	1.15	0.34	0.41	-0.34	0.83	0.28	0.33
H11	-0.61	0.49	-0.06	0.27	-11.89	10.40	-0.61	4.66	-0.38	0.58	0.06	0.27	-0.43	1.20	0.20	0.44	-0.50	0.77	0.03	0.35
H12	-0.83	0.55	0.10	0.35	-5.64	7.66	1.02	3.11	-0.55	0.41	0.04	0.26	-0.39	0.92	0.28	0.32	-0.22	0.53	0.13	0.20
H13	-0.65	0.54	-0.12	0.36	-18.03	12.28	0.15	6.26	-0.55	0.51	-0.03	0.23	-0.54	1.10	0.17	0.40	-0.59	0.62	0.04	0.30
H14	-0.65	0.59	0.12	0.35	-5.40	15.27	0.68	4.24	-0.30	0.41	0.03	0.20	-0.30	1.12	0.27	0.38	-0.27	0.84	0.22	0.35
H15	-0.62	0.52	-0.03	0.33	-16.92	3.13	-1.48	5.08	-0.31	0.48	0.00	0.22	-0.46	0.90	0.16	0.45	-0.48	0.77	0.05	0.37
H16	-0.83	0.46	-0.02	0.33	-6.21	15.34	0.85	4.94	-0.45	0.37	0.03	0.23	-0.57	1.18	0.23	0.44	-0.30	0.78	0.23	0.35
H17	-0.48	0.58	0.05	0.30	-7.44	11.58	1.00	4.17	-0.35	0.52	0.06	0.24	-0.25	1.28	0.38	0.43	-0.44	0.74	0.10	0.33
H18	-0.76	0.55	-0.06	0.39	-11.77	11.14	-1.37	4.77	-0.29	0.32	0.04	0.14	-0.61	1.23	0.22	0.42	-0.43	0.77	0.12	0.31
H19	-0.78	0.63	0.01	0.40	-16.23	16.89	0.13	5.94	-0.36	0.47	0.03	0.20	-0.53	1.12	0.25	0.41	-0.38	0.71	0.09	0.27
H20	-0.71	0.66	0.00	0.38	-13.08	4.88	-0.59	4.14	-0.31	0.60	0.08	0.20	-0.57	1.20	0.35	0.46	-0.63	0.77	0.12	0.35
H21	-0.76	0.58	0.04	0.36	-5.61	13.29	1.44	3.78	-0.36	0.44	0.01	0.25	-0.35	0.96	0.16	0.38	-0.20	0.63	0.18	0.25
H22	-0.94	0.64	-0.11	0.50	-14.20	6.05	-0.10	5.22	-0.61	0.43	-0.02	0.25	-0.50	0.94	0.16	0.39	-0.56	0.84	0.10	0.39
H23	-0.86	0.57	-0.09	0.41	-7.33	4.88	-0.94	4.01	-0.45	0.43	0.01	0.22	-0.66	0.79	0.12	0.38	-0.31	0.63	0.13	0.28

a) Raw experimental results.



b) Average/deviation chart.

critical-failure states. It was implemented as follows: on reception of a ready notification, the scheduler checks if a rendez-vous is enabled and executes it immediately; otherwise, it updates the readiness map and waits for another notification.

– PROP2: this baseline requires the status of every rendez-vous to be known before making a decision on which one must be executed. Obviously, this guarantees fairness at the cost of introducing arbitrary delays; it is intended to prove that fairness must be enforced efficiently since, otherwise, the systems may also run into critical-failure states. It was implemented as follows: on reception of a ready notification, the scheduler firsts checks the status of every rendez-vous; if they all are known to be enabled or disabled, then the most priority one is executed; otherwise, it waits for another notification.

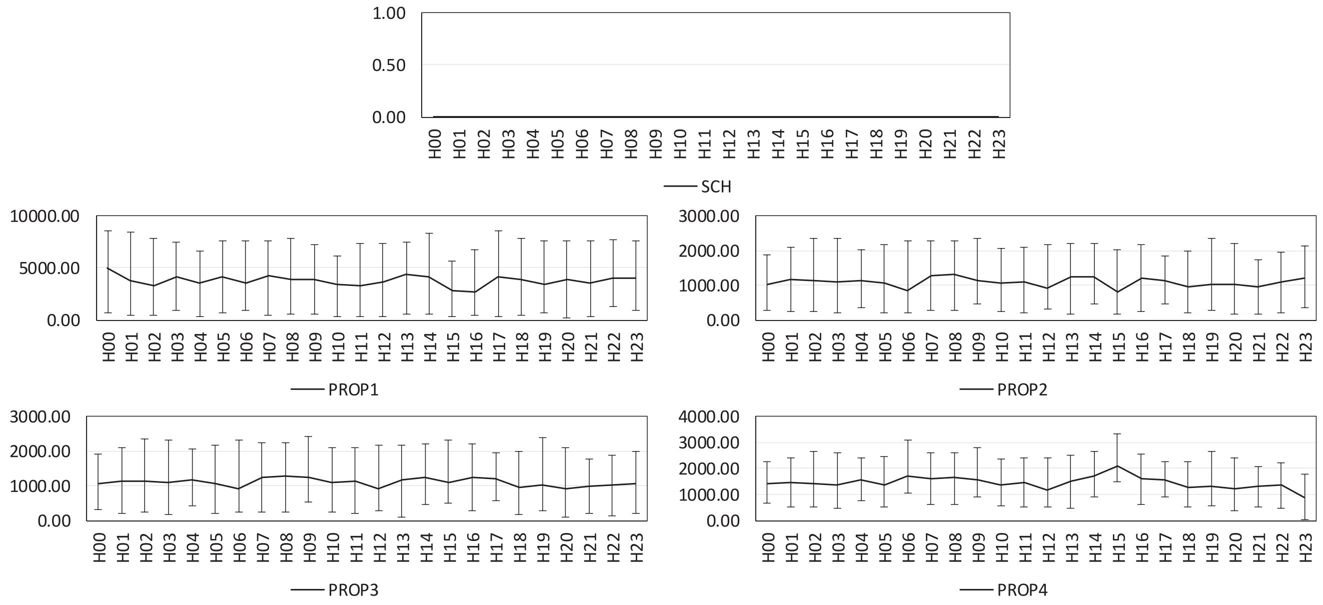
We also managed to adapt two other proposals in the literature that serve as competitors for experimentation purposes, namely:

– PROP3: this proposal uses the approach by Ruiz et al. [45], which guarantees fairness by introducing regular delays that freeze the system from time to time. This approach requires to perform a static analysis of the system that helps understand which rendez-vouses are linked to each other because there are agents that can ready them all; it also requires to analyse the status of the rendez-

Table 3
Critical-failure state alarms.

	SCH				PROP1				PROP2				PROP3				PROP4			
Hour	Min	Max	Avg	Dev	Min	Max	Avg	Dev	Min	Max	Avg	Dev	Min	Max	Avg	Dev	Min	Max	Avg	Dev
H00	0.00	0.00	0.00	0.00	688.00	8526.00	4984.83	2099.21	299.00	1869.00	1043.83	509.84	288.00	2216.00	1075.92	550.08	400.00	2774.00	1419.92	718.42
H01	0.00	0.00	0.00	0.00	402.00	8422.00	3724.92	2284.87	231.00	2104.00	1158.00	568.97	214.00	2070.00	1148.17	598.43	330.00	2712.00	1465.58	738.93
H02	0.00	0.00	0.00	0.00	476.00	7872.00	3322.88	2038.10	235.00	2337.00	1136.21	525.35	298.00	2196.00	1137.17	514.25	322.00	2674.00	1430.25	613.41
H03	0.00	0.00	0.00	0.00	936.00	7475.00	4141.00	1990.79	210.00	2335.00	1107.38	645.07	244.00	2102.00	1082.67	570.32	320.00	2790.00	1362.17	742.53
H04	0.00	0.00	0.00	0.00	321.00	6608.00	3507.17	2052.18	372.00	2038.00	1150.50	390.74	398.00	1936.00	1188.00	402.06	480.00	2526.00	1539.50	554.10
H05	0.00	0.00	0.00	0.00	660.00	7587.00	4115.08	2098.63	205.00	2161.00	1060.46	575.70	230.00	2060.00	1058.33	554.31	258.00	2538.00	1370.00	704.80
H06	0.00	0.00	0.00	0.00	927.00	7593.00	3490.67	1949.82	203.00	2273.00	862.04	507.12	264.00	2000.00	902.42	534.78	142.00	2965.00	1692.63	788.03
H07	0.00	0.00	0.00	0.00	488.00	7618.00	4214.50	2159.55	287.00	2278.00	1292.08	560.33	284.00	2054.00	1257.25	536.07	374.00	2554.00	1630.67	657.45
H08	0.00	0.00	0.00	0.00	519.00	7894.00	3941.50	2034.89	286.00	2272.00	1320.83	597.24	250.00	2176.00	1286.08	563.48	354.00	2656.00	1672.33	687.54
H09	0.00	0.00	0.00	0.00	558.00	7207.00	3958.79	1879.53	449.00	2342.00	1140.58	507.79	380.00	2170.00	1227.17	553.45	522.00	2702.00	1582.42	703.86
H10	0.00	0.00	0.00	0.00	285.00	6193.00	3367.92	2159.11	231.00	2058.00	1063.58	513.91	256.00	2036.00	1093.08	522.65	382.00	2602.00	1389.92	649.52
H11	0.00	0.00	0.00	0.00	296.00	7379.00	3294.83	2298.38	209.00	2082.00	1112.83	587.29	244.00	2160.00	1120.75	591.15	344.00	2638.00	1442.50	741.32
H12	0.00	0.00	0.00	0.00	342.00	7315.00	3666.04	1853.86	305.00	2177.00	932.54	488.25	354.00	1826.00	907.08	461.75	474.00	2324.00	1171.58	559.20
H13	0.00	0.00	0.00	0.00	634.00	7515.00	4376.58	2344.43	160.00	2218.00	1227.46	576.62	184.00	2100.00	1185.50	564.34	286.00	2614.00	1518.58	698.73
H14	0.00	0.00	0.00	0.00	614.00	8261.00	4154.08	2108.03	452.00	2197.00	1238.50	481.45	468.00	2094.00	1255.42	461.87	518.00	2618.00	1712.75	605.12
H15	0.00	0.00	0.00	0.00	321.00	5656.00	2780.88	1674.45	184.00	2019.00	800.42	532.37	706.00	1608.00	1100.67	284.15	1074.00	2924.00	2096.92	439.22
H16	0.00	0.00	0.00	0.00	489.00	6696.00	2708.50	1954.46	260.00	2171.00	1220.58	478.81	304.00	2150.00	1240.42	505.77	362.00	2626.00	1600.08	668.86
H17	0.00	0.00	0.00	0.00	394.00	8519.00	4178.54	2287.64	454.00	1837.00	1117.29	398.38	482.00	2008.00	1221.33	447.41	624.00	2446.00	1553.25	556.85
H18	0.00	0.00	0.00	0.00	398.00	7799.00	3854.71	2127.61	198.00	1981.00	951.42	492.24	188.00	1918.00	943.92	439.42	280.00	2304.00	1251.00	598.16
H19	0.00	0.00	0.00	0.00	677.00	7579.00	3399.13	2123.85	272.00	2366.00	1023.13	588.50	234.00	2104.00	1027.33	565.35	318.00	2568.00	1315.33	733.89
H20	0.00	0.00	0.00	0.00	272.00	7607.00	3933.75	2144.45	185.00	2206.00	1012.88	587.01	216.00	1994.00	921.08	499.48	318.00	2656.00	1225.50	669.50
H21	0.00	0.00	0.00	0.00	281.00	7652.00	3498.75	1793.76	186.00	1741.00	969.75	507.40	226.00	2076.00	993.08	533.69	244.00	2498.00	1318.58	710.98
H22	0.00	0.00	0.00	0.00	1272.00	7694.00	4059.88	1901.11	203.00	1941.00	1114.04	488.60	180.00	1724.00	1034.42	446.29	266.00	2466.00	1373.33	564.46
H23	0.00	0.00	0.00	0.00	998.00	7622.00	4023.58	2069.67	365.00	2137.00	1217.96	507.87	452.00	1682.00	1059.33	339.45	136.00	1712.00	872.75	490.71

a) Raw experimental results.



b) Average/deviation chart.

vouses at run-time to determine if they are stable or not, which has to do with whether they can become enabled in future or not.

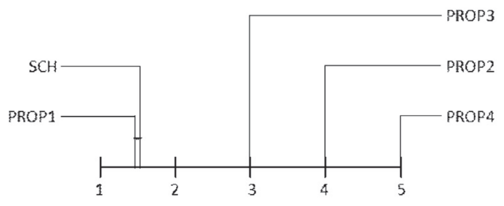
– PROP4: this proposal uses the approach by Joung [30], which consists of a distributed non-deterministic scheduler. Basically, each agent stochastically selects one of the rendez-vouses that it is readying and sends messages to the other agents that can ready it; the agents then wait for a pre-defined period of time called Δ; later, they check if the other agents have also selected the same rendez-vous, in which case, it is executed; otherwise, the procedure is re-initiated. We set Δ to the maximum time that the agents spent doing local computations during the experiments that we carried out with the other proposals. To implement this proposal, we had to wrap each sensor and actuator with a proxy that implements the previous protocol. This, obviously, introduced some additional workload to the computing boards.

5.4. *Experimental results*

Hereinafter, we refer to our proposal as SCH. The experiments consisted of running it, the baselines (PROP1 and PROP2), and the competitors (PROP3 and PROP4) on the test system for twenty-four days. We computed several statistics regarding average performance, deviation of performance, and number of critical-failure state alarms raised on an hourly basis.

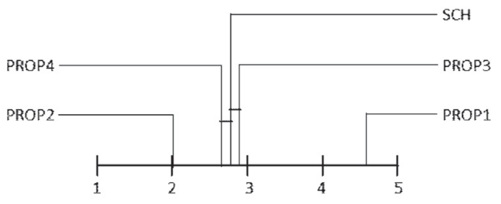
Table 4
Statistical analysis.

Ranking		Iman-Davenport		Bergmann-Hommel		
Proposal	Rank	Statistic (F)	P-Value	Comparison	Statistic (z)	Adjusted p-value
SCH	1.58	10037.25	0.0E+00	PROP1-SCH	0.16E+01	0.11E+00
PROP1	1.43			PROP1-PROP3	0.17E+02	0.12E-61
PROP2	4.00			PROP1-PROP2	0.28E+02	0.57E-165
PROP3	3.00			PROP1-PROP4	0.38E+02	0.49E-299
PROP4	5.00			SCH-PROP3	0.15E+02	0.75E-51
				SCH-PROP2	0.26E+02	0.94E-147
				SCH-PROP4	0.37E+02	0.89E-293
				PROP3-PROP2	0.11E+02	0.14E-25
				PROP3-PROP4	0.21E+02	0.86E-101
				PROP2-PROP4	0.11E+02	0.12E-25



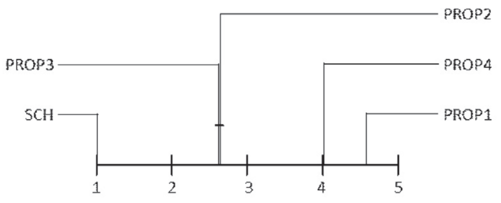
a) Average performance.

Ranking		Iman-Davenport		Bergmann-Hommel		
Proposal	Rank	Statistic (F)	P-Value	Comparison	Statistic (z)	Adjusted p-value
SCH	2.74	373.76	0.35E-247	PROP2-PROP4	0.59E+01	0.83E-08
PROP1	4.69			PROP2-SCH	0.71E+01	0.29E-11
PROP2	2.08			PROP2-PROP3	0.84E+01	0.19E-15
PROP3	2.86			PROP2-PROP1	0.28E+02	0.28E-171
PROP4	2.62			PROP4-SCH	0.13E+01	0.21E+00
				PROP4-PROP3	0.26E+01	0.31E-01
				PROP4-PROP1	0.22E+02	0.30E-107
				SCH-PROP3	0.13E+01	0.19E+00
				SCH-PROP1	0.21E+02	0.12E-95
				PROP3-PROP1	0.20E+02	0.41E-84



b) Performance deviation.

Ranking		Iman-Davenport		Bergmann-Hommel		
Proposal	Rank	Statistic (F)	P-Value	Comparison	Statistic (z)	Adjusted p-value
SCH	1.00	2075.89	0.0E+00	SCH-PROP2	0.18E+02	0.19E-72
PROP1	4.56			SCH-PROP3	0.18E+02	0.21E-69
PROP2	2.70			SCH-PROP4	0.33E+02	0.22E-238
PROP3	2.66			SCH-PROP1	0.38E+02	0.49E-299
PROP4	4.08			PROP2-PROP3	0.43E+00	0.67E+00
				PROP2-PROP4	0.15E+02	0.39E-49
				PROP2-PROP1	0.20E+02	0.81E-88
				PROP3-PROP4	0.15E+02	0.18E-51
				PROP3-PROP1	0.20E+02	0.20E-91
				PROP4-PROP1	0.52E+01	0.49E-06



b) Critical-failure state alarms.

Table 1 shows our experimental results regarding average performance. The average performances achieved by SCH, PROP1, PROP2, PROP3, and PROP4 are 3.60 ± 0.09, 3.63 ± 0.28, 2.13 ± 0.07, 2.63 ± 0.29, and 1.15 ± 0.52 million rendez-vouses per hour, respectively. According to these results, PROP1 seems to be the proposal that achieves the highest performance, followed by SCH (which is 0.82% less efficient), PROP3 (which is 27.55% less efficient), PROP2 (which is 41.32% less efficient), and PROP4 (which is 68.32% less efficient). The intuitive conclusion is that the scheduler does not seem to introduce a significant penalty regarding efficiency, since the difference with regard to PROP1 is only 0.82%; the penalty introduced by the other proposals is huge because PROP2 requires to freeze the system on every decision, PROP3 requires to freeze it from time to time, and PROP4 uses a stochastic approach that is very inefficient.

Table 2 presents our experimental results regarding the deviation of performance. It was straightforward to compute an estimate for the number of times that each rendez-vous should have been executed in an execution in which each rendez-vous executes as many times as it can and draws are broken in a balanced manner; this facilitated computing an estimation of the deviation of the actual number of executions with regard to that estimate. The deviations in performance introduced by SCH, PROP1, PROP2, PROP3, and PROP4 are -0.02±0.36, -0.15±4.51, -0.15 ± 0.02 ± 0.23, 0.25 ± 0.42, and 0.13 ± 0.32 thousands of rendez-vouses per hour, respectively. To make the difference more evident, we need to analyse the length of the intervals in which the deviations range, which are $|[-0.72, 0.58]| = 1.30$, $|[-11.70, 9.38]| = 21.08$, $|[-0.41, 0.46]| = 0.87$, $|[-0.43, 1.13]| = 1.55$, and $|[-0.41, 0.73]| = 1.14$ thousands of rendez-vouses per hour, respectively. Recall that PROP2 freezes the system before deciding on which rendez-vous must be executed next, which clearly explains why it results in the smallest range of deviation regarding the ideal execution; SCH introduces 166.67% deviation with respect to PROP2, but the deviations introduced by PROP1, PROP3, and PROP4 are 2 702.56%, 198.72%, and 146.15%, respectively. That is, the proposals that enforce fairness seem to introduce similar deviations to performance, which are clearly smaller than the deviation introduced when fairness is not enforced.

Table 3 shows our experimental results regarding critical-failure states. Running our experiments in a test system allowed us to count the number of times that the system would have entered critical-failure states without any real impact on the solar plant. Realise that SCH did not drive the system into any critical-failure states, whereas the other proposals resulted in an average of 3 779.10 ± 2 059.52, 1 094.76 ± 525.70, 1 102.77 ± 501.67, and 1 458.65 ± 649.82 alarms per hour, respectively. The results regarding PROP1 are not surprising at all since we already know that it does not enforce fairness, which means that it can easily lead to executions in which the number of times that each rendez-vous executes deviates significantly from the number of times that it might have executed. The results regarding PROP2, PROP3, and PROP4 also make sense: realise that they implement fairness at the cost of introducing arbitrary delays in the execution, which results in actuators that do not work as responsively as they should.

Summing up: our proposal does not seem to introduce a significant penalty on performance when compared to a scheduler that does not take fairness into account; it does not seem to introduce a large deviation to the performance when compared to a scheduler that knows the state of every rendez-vous before scheduling them; and it does not seem to drive the system into any critical-failure states, which is very important in our context.

5.5. *Statistical analysis*

The conclusions from our experimental analysis are clear. What remains to be done is to check that the differences in rank found in our experiments are statistically significant. We used the following statistical analysis [21] at the standard significance level $\alpha = 0.05$: a) first, we computed the experimental rank of each proposal regarding average performance, deviation in performance, and number of critical-failure states; b) then, we performed Iman-Davenport's test to check if the differences in rank are globally significant or not; c) if they were, then we performed Bergman-Hommel's test to compare the proposals to each other. Both tests return (adjusted) p-values that must be compared to the standard significance level; if the p-value is smaller, then the conclusion is that the experimental results support the hypothesis that the differences in rank are statistically significant; otherwise, they do not. Table 4 summarises the results; we have greyed the (adjusted) p-values that are smaller than the significance level and we have drawn the difference diagrams to facilitate the interpretation.

Regarding the average performance, the experimental ranking is PROP1 > SCH > PROP3 > PROP2 > PROP4. Iman-Davenport's test returns p-value 0.00, which clearly indicates that the experimental results support the hypothesis that there are global significant differences in the ranking. It then proceeds to compare the proposals to each other using Bergman-Hommel's test. The conclusion is that the experimental results do not support the hypothesis that there are significant differences between PROP1 and SCH, but it supports the hypothesis that the differences are significant regarding the other comparisons.

Regarding the deviation in performance, the experimental ranking is PROP2 > PROP4 > SCH > PROP3 > PROP1. Iman-Davenport's test returns a p-value that is very close to zero, which clearly indicates that the experimental results support the hypothesis that there are global significant differences in the ranking. Thus, it makes sense to compare the proposals using Bergman-Hommel's test. In this case, the test returns that the experimental results do not support the hypothesis that the differences between PROP4 and SCH or SCH and PROP3 are statistically significant, whereas it supports the hypothesis that they are statistically significant regarding the other comparisons.

Regarding the number of critical-failure state alarms, the ranking is SCH > PROP3 > PROP2 > PROP4 > PROP1. Since the p-value returned by Iman-Davenport's test is 0.00, the hypothesis that there are global differences in rank is clearly supported by the experimental results. Bergman-Hommel's test returns a p-value smaller than the significance level only for the comparison between PROP2 and PROP3, which means that the experimental results support the hypothesis that the differences in rank between these proposals are not statistically significant, but they are regarding the other comparisons.

Summing up: regarding average performance, the difference in ranking between SCH and PROP1 is not statistically significant, which confirms that the penalty introduced by our proposal is not actually significant; regarding deviation in performance, the difference in ranking between PROP2 and SCH is statistically significant; fortunately, the size effect is not important in this case, because the analysis confirms that the differences in rank regarding the number of critical-failure states entered is very significant. The conclusion is that our proposal is not detrimental at all to performance and can schedule the rendez-vouses in a system as complex as ours very well.

6. Conclusions

In this article, we have delved into devising, implementing, and deploying a scheduler for multi-source fusion systems in the context of SCADA systems. It addresses the problems that we have found in other proposals, namely: it can deal with multi-way rendez-vouses, it does not require any instrumentation, it implements a strong level of fairness, it does not require any shared memory, and it is efficient in terms of both time and space. Our analysis of the related work reveals that ours is the only proposal in the literature that was designed to address the problems in the context of SCADA systems, most of which originate from the fact that the computing boards that support them have very limited computing, memory, and storage capabilities. Our experimental analysis makes it clear that our proposal is good enough for practical purposes since it was the only that could run the experimental system without entering any critical-failure states. The conclusions were checked to be statistically sound using a well-established statistical analysis procedure.

Declaration of Competing Interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

CRediT authorship contribution statement

Rafael Corchuelo: Conceptualization, Methodology, Software, Validation, Formal analysis, Investigation, Resources, Data curation, Writing - original draft, Writing - review & editing, Visualization, Supervision, Funding acquisition. **Miguel Toro:** Conceptualization, Methodology, Formal analysis, Supervision, Funding acquisition.

Acknowledgements

The authors would like to thank Dr. David Ruiz for collaborating on the initial ideas regarding this article, Dr. Jesús, S. Aguilar for discussing on the evaluation of the proposal, Dr. Francisco Herrera, for sharing his statistic evaluation software with us, Dr. Gerth S. Brodal for sharing his knowledge on implementing queues efficiently with us, Dr. Rafael Z. Frantz for working on the Guaraná technology that lies at the heart of our implementation, and i2Factory, S.L. for providing the perfect project to put our ideas into practice, the experimentation environment, and some machinery. The work was supported by the Spanish and the Andalusian R&D programmes trough FEDER grants TIC2003-02737-C02-01, TIN2013-40848-R, TIN2016-75394-R, and P18-RT-1060.

References

[1] P. André, G. Ardourel, C. Attiogbé, Composing components with shared services in the Kmelia model, in: Intl. Conf. on Software Composition, 2008, pp. 125–140, doi:10.1007/978-3-540-78789-1_9.
[2] ATMEL, AVR202: 16-bit arithmetics, Technical report, 2002. http://ww1.microchip.com/downloads/en/AppNotes/doc0937.pdf

[3] S. Banerjee, P. Mukherjee, S. Kanrar, N. Chaki, A novel symmetric algorithm for process synchronization in distributed systems, Alg. Appl. (2018) 51–66, doi:10.1007/978-981-10-8102-6_4.

[4] S. Bensalem, M. Bozga, J. Quilbeuf, J. Sifakis, Optimized distributed implementation of multiparty interactions with Restriction, Sci. Comput. Program. 98 (2015) 293–316, doi:10.1016/j.scico.2014.02.013.

[5] E. Bini, Adaptive fair scheduler: fairness in presence of disturbances, in: Intl. Conf. on Real-Time Networks and Systems, 2016, pp. 129–138, doi:10.1145/2997465.2997468.

[6] B. Bonakdarpour, M. Bozga, J. Quilbeuf, Model-based implementation of distributed systems with priorities, Design Autom. for Emb. Sys. 17 (2) (2013) 251–276, doi:10.1007/s10617-012-9091-0.

[7] B. Bonakdarpour, S. Devismes, F. Petit, Snap-stabilizing committee coordination, J. Parallel Distrib. Comput. 87 (2016) 26–42, doi:10.1016/j.jpdc.2015.09.004.

[8] S.A. Boyer, SCADA: supervisory control and data acquisition, International Society of Automation, 4th edition, 2016.

[9] G.S. Brodal, Worst-case efficient priority queues1996, Annual ACM-SIAM Sympo- sium on Discrete Algorithms52–58, 10.1145/313852.313883.

[10] A. Brook, D.A. Peled, S. Schewe, Local and global fairness in concurrent systems, in: ACM/IEEE Intl. Conf. on Formal Methods and Models for Co-Design, 2015, pp. 2–9, doi:10.1109/MEMCOD.2015.7340461.

[11] C. Chao, H. Fu, Supporting fast and fair rendez-vous for cognitive radio networks and computer applications, J. Netw. Comput. Appl. (2018) 98–108, doi:10.1016/j.jnca.2018.03.032.

[12] L. Chen, S. Liu, B. Li, B. Li, Scheduling jobs across geo-distributed datacenters with max-min fairness, in: 2017 IEEE Conf. on Computer Communications, 2017, pp. 1–9, doi:10.1109/INFOCOM.2017.8056949.

[13] J.M.F. de Alba, R. Fuentes-Fernández, J. Pavón, Architecture for manage- ment and fusion of context information, Inf. Fusion 21 (2015) 100–113, doi: 10.1016/j.inffus.2013.10.007.

[14] J. Feliu, J. Sahuquillo, S. Petit, J. Duato, Perf&Fair: a progress-aware scheduler to enhance performance and fairness in SMT multicores, IEEE Trans. Comput. 66 (5) (2017) 905–911, doi:10.1109/TC.2016.2620977.

[15] A. Ferrara, Web services: a process algebra approach, in: ICSOC, 2004, pp. 242–251, doi:10.1145/1035167.1035202.

[16] N. Francez, Fairness, Springer, 1986.

[17] N. Francez, I.R. Forman, Interacting Processes: A Multiparty Approach to Co-Ordi- nated Distributed Processing, Addison-Wesley, 1996.

[18] R.Z. Frantz, A.M.R. Quintero, R. Corchuelo, A domain-specific language to design enterprise application integration solutions, Int. J. Cooper. Inf. Syst. 20 (2) (2011) 143–176, doi: 10.1142/S0218843011002225.

[19] R.Z. Frantz, R. Corchuelo, C. Molina-Jiménez, A proposal to detect errors in enterprise application integration solutions, J. Syst. Softw. 85 (3) (2012) 480–497, doi:10.1016/j.jss.2011.10.048.

[20] R.Z. Frantz, R. Corchuelo, F. Roos-Frantz, On the design of a maintainable software development kit to implement integration solutions, J. Syst. Softw. 111 (2016) 89–104, doi:10.1016/j.jss.2015.08.044.

[21] S. Garcia, F. Herrera, An extension on “statistical comparisons of classifiers over multiple data sets ” for all pairwise comparisons, J. Mach. Learn. Res. 9 (10) (2008) 2677–2694.

[22] K. Gardner, M. Harchol-Balter, E. Hyytiä, R. Righter, Scheduling for efficiency and fairness in systems with redundancy, Perform. Eval. 116 (2017) 1–25, doi: 10.1016/j.peva.2017.07.001.

[23] M. Hojeij, C.A. Nour, J. Farah, C. Douillard, Waterfilling-based proportional fairness scheduler for downlink non-orthogonal multiple access, IEEE Wirel. Commun. Lett. 6 (2) (2017) 230–233, doi: 10.1109/LWC.2017.2665470.

[24] S. Huh, S. Hong, Providing fair-share scheduling on multicore computing systems via progress balancing, J. Syst. Softw. 125 (2017) 183–196, doi:10.1016/j.jss.2016.11.053.

[25] Y. Jiang, A survey of task allocation and load balancing in distributed systems, IEEE Trans. Parallel Distrib. Syst. 27 (2) (2016) 585–599, doi:10.1109/TPDS.2015.2407900.

[26] S.T.Q. Jongmans, F. Arbab, Global consensus through local synchronization: a formal basis for partially-distributed coordination, Sci. Comput. Program. 115-116 (2016) 199–224, doi:10.1016/j.scico.2015.09.001.

[27] S.T.Q. Jongmans, F. Santini, M. Sargolzaei, F. Arbab, H. Afsarmanesh, Orchestrating web services using Reo: from circuits and behaviors to automatically generated code, Serv. Orient. Comput. Appl. 8 (4) (2014) 277–297, doi:10.1007/s11761-013-0147-1.

[28] P. Jordan, Solar Energy Markets: An Analysis of the Global Solar Industry, Elsevier, 2013.

[29] Y.J. Joung, Characterizing fairness implementability for multiparty interaction, Colloq. Autom. Lang. Program. 1099 (1996) 110–121, doi:10.1007/3-540-61440-0_121.

[30] Y.J. Joung, Two decentralized algorithms for strong interaction fairness for systems with unbounded speed variability, Theor. Comput. Sci. 243 (1–2) (2000) 307–338, doi:10.1016/S0304-3975(98)00224-2.

[31] E. Kalyvianaki, M. Fiscato, T. Salonidis, P.R. Pietzuch, THEMIS: fairness in federated stream processing under overload, in: Intl. Conf. on Management of Data, 2016, pp. 541–553, doi:10.1145/2882903.2882943.

[32] J.M. Kerridge, Experiments in multicore and distributed parallel processing using JCSP, Comm. Process Arch. Conf. (2011) 131–142, doi:10.3233/978-1-60750-774-1-131.

[33] S. Kurgalin, S. Borzunov, A Practical Approach to High-Performance Computing, Springer, 2019.

[34] T. Lan, D.T.H. Kao, M. Chiang, A. Sabharwal, An axiomatic theory of fairness in network resource allocation, in: IEEE Intl. Conf. on Computer Communications, 2010, pp. 1343–1351, doi:10.1109/INFCOM.2010.5461911.

[35] X. Liu, X. Zhang, W. Li, X. Zhang, Swarm optimization algorithms applied to multi-resource fair allocation in heterogeneous cloud computing systems, Computing 99 (12) (2017) 1231–1255, doi:10.1007/s00607-017-0561-x.

[36] Y. Liu, M. Elkashlan, Z. Ding, G.K. Karagiannidis, Fairness of user clustering in MIMO non-orthogonal multiple access systems, IEEE Commun. Lett. 20 (7) (2016) 1465–1468, doi:10.1109/LCOMM.2016.2559459.

[37] N.A. Lynch, Distributed Algorithms, Elsevier, 2009.

[38] Y. Lyu, F. Yan, Y. Chen, D. Wang, Y. Shi, N. Agoulmine, High-performance scheduling model for multisensor gateway of cloud sensor system-based smart-living, Inf. Fusion 21 (2015) 42–56, doi:10.1016/j.inffus.2013.04.004.

[39] Y. Mao, Y. Zhang, Q. Hua, H. Dai, X. Wang, A non-intrusive solution to guarantee runtime behavior of open SCADA systems, ICWS (2015) 739–742, doi:10.1109/ICWS.2015.105.

[40] MPI Forum, A Message-Passing Interface Standard, 2019. URL https://www.mpi-forum.org/

[41] Z. Niu, S. Tang, B. He, Gemini: an adaptive performance-fairness scheduler for data-intensive cluster computing, in: IEEE Intl. Conf. on Cloud Computing Technology and Science, 2015, pp. 66–73, doi:10.1109/CloudCom.2015.52.

[42] E.-R. Olderog, K.R. Apt, Fairness in parallel programs: the transformational approach, ACM Trans. Program. Lang. Syst. 10 (3) (1988) 420–455, doi:10.1145/44501.44504.

[43] A. Pathania, V. Venkataramani, M. Shafique, T. Mitra, J. Henkel, Distributed fair scheduling for many-cores, Design, Autom. Test (2016) 379–384, doi:10.3850/9783981537079.

[44] S. Rodríguez, J.F. de Paz, G. Villarrubia, C. Zato, J. Bajo, J.M. Corchado, Multi-agent information fusion system to manage data from a WSN in a residential home, Inf. Fusion 23 (2015) 43–57, doi:10.1016/j.inffus.2014.03.003.

[45] D. Ruiz, R. Corchuelo, M. Toro, Fairness in systems based on multiparty interactions, Concurr. Comput. 15 (11–12) (2003) 1093–1116, doi:10.1002/cpe.782.

[46] A.M. Saleh, N. Parveen, M.F. Uddin, Optimal scheduling of coordinated multi-point transmissions in cellular networks, Int. J. Commun. Syst. 31 (1) (2018), doi:10.1002/dac.3431.

[47] B. Shen, S. Rho, X. Zhou, R. Wang, A delay-aware schedule method for distributed information fusion with elastic and inelastic traffic, Inf. Fusion 36 (2017) 68–79, doi:10.1016/j.inffus.2016.11.008.

[48] A.S. Tanenbaum, M. van Steen, Distributed Systems: Principles and Paradigms, Createspace, 2016.

[49] G. Taubenfeld, Fair synchronization, J. Parallel Distrib. Comput. 97 (2016) 1–10, doi:10.1016/j.jpdc.2016.06.007.

[50] S. Timotheou, I. Krikidis, Fairness for non-orthogonal multiple access in 5G systems, IEEE Signal Process. Lett. 22 (10) (2015) 1647–1651, doi:10.1109/LSP.2015.2417119.

[51] Y.-K. Tsay, R.L. Bagrodia, Some impossibility results in interprocess synchronization, Distrib. Comput. 6 (4) (1993) 221–231, doi:10.1007/BF02242710.

[52] Z. Ul-Abdin, B. Svensson, Occam-π for programming of massively parallel reconfigurable architectures, Int. J. Reconfig. Comp. 2012 (2012) 1–17, doi:10.1155/2012/504815.

[53] H. Völzer, D. Varacca, Defining fairness in reactive and concurrent systems, J. ACM 59 (3) (2012) 13:1–13:37, doi:10.1145/2220357.2220360.

[54] J. Wang, J. Su, Y. Xi, COM-based software architecture for multisensor fusion system, Inf. Fusion 2 (4) (2001) 261–270, doi:10.1016/S1566-2535(01)00042-2.

[55] Q. Wang, X. Huang, A performance-fairness scheduler on Hadoop YARN, IEEE ICSESS (2016) 541–553, doi:10.1109/ICSESS.2016.7883019.

[56] W. Wang, B. Liang, B. Li, Multi-resource fair allocation in heterogeneous cloud computing systems, IEEE Trans. Parallel Distrib. Syst. 26 (10) (2015) 2822–2835, doi:10.1109/TPDS.2014.2362139.

[57] Z. Xiao, Z. Tong, K. Li, K. Li, Learning non-cooperative game for load balancing under self-interested distributed environment, Appl. Soft Comput. 52 (2017) 376–386, doi:10.1016/j.asoc.2016.10.028.

[58] Q. Xie, M. Pundir, Y. Lu, C.L. Abad, R.H. Campbell, Pandas: robust locality-aware scheduling with stochastic delay optimality, IEEE/ACM Trans. Netw. 25 (2) (2017) 662–675, doi:10.1109/TNET.2016.2606900.

[59] M. Zaharia, D. Borthakur, J.S. Sarma, K. Elmeleegy, S. Shenker, I. Stoica, Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling, in: EuroSys 2010, 2010, pp. 265–278, doi:10.1145/1755913.1755940.

[60] Q. Zhu, J.C. Oh, An approach to dominant resource fairness in distributed environment, in: Intl. Conf. on Industrial, Engineering and Other Applications of Applied Intelligent Systems, 2015, pp. 141–150, doi:10.1007/978-3-319-19066-2_14.