BEDWE: A Decentralized Workflow Engine for Best-Effort Infrastructures

Palakiyem Wallah[1], Cédric Tedeschi[2](✉), and Jean-Louis Pazat[2]

[1] Université de Kara, Kara, Togo
[2] Univ Rennes, Inria, CNRS, IRISA, Rennes, France
cedric.tedeschi@inria.fr

Abstract. We consider the problem of executing composite computing applications called *workflows* on top of unreliable computing infrastructures. Having in mind the situation of the electric delivery in the sub-saharan area, we propose BEDWE, a decentralized workflow engine able to dynamically assign portions of the workflow to currently live compute nodes. More precisely, in a point-to-point manner, each node can receive a part of the workflow and delegate a subpart of it to another node. This mechanism can be repeated recursively until the whole workflow is executed. BEDWE includes a mechanism to support nodes leaving the network due to power outage. We present a software prototype of BEDWE and its experimentation over the French nation-wide Grid'5000 platform.

Keywords: Workflows · Decentralized orchestration
Fault-tolerance · Best-effort infrastructures

1 Introduction

Computing reliably is a major challenge in countries struggling to deliver a constant electric power delivery. For instance, African countries from the sub-Saharan region are used to face power outages, due to an insufficient level of electric injection to satisfy the needs of people, administration and companies. This problem can be solved by cutting electricity in some area/while another one is supplied. These cuts are planned according to a predefined schedule that are publicly announced so people can organise themselves. Being able to ensure the completion of computations running on computers located in such an environment calls for fault-tolerance mechanisms to be injected in the system supporting these applications. This paper explores a specific problem in this area, where we consider a fully-decentralized computing platform composed of compute nodes running in an electric environment subject to power cuts. More specifically, we consider applications which are compositions of building block services, as followed by *service-oriented computing.*

Service-oriented computing has become one of the dominant paradigms to develop applications in both scientific and industrial contexts. This model advocates the composition of services as a programming model to build complex applications out of existing building blocks, or simpler services [1]. A sibling concept is the *workflow*. A workflow is a *temporal* composition of services completing a specific task. More precisely, each service in a workflow corresponds to a particular step in the workflow and may have precedence constraint dependencies with other services. These dependencies between services can be represented as a directed acyclic graph where nodes are the services and links are the dependencies/precedence constraints between services.

A workflow needs to be enacted. For this, it traditionally relies on a *workflow engine i.e.,*a program that takes a workflow's specification and deploys it on compute nodes, starting each service once its dependencies have been satisfied *i.e.,* when the services to get run before completed. This engine is traditionally a centralized, reliable component. If it fails, the whole workflow coordination is undermined and the workflow may not complete. Having this centralized engine is no longer possible when the workflow is supposed to run over unreliable platforms, where having one reliable node on which to run the engine is no longer possible. The problem becomes even more complicated when considering these emerging computing platforms described above. Recently, decentralizing workflow execution has been the focus of some researches aiming at providing solutions where centralized enactment systems cannot be used any more.

The objectives of our work is to propose: (1) a truly decentralized workflow execution whose management is shared by each participant node and (2) a mechanism to support crashes and delays for common workflow patterns. Our solution, called BEDWE (*Best Effort Decentralized Workflow Execution*) runs on every node taking part in the workflow engine. It uses a specific protocol allowing to send, receive and process portions of the workflow.

The remainder of this paper is organized as follows. In Sect. 2, we define our model and describes, by syntactic means, the type of workflows supported. Section 3 describes BEDWE execution model and illustrates its execution and deployment over the nodes for different workflow patterns. In Sect. 4, we discuss the problems brought about by potential crashes (for instance due to power cuts) and devise a simple solution, based on a heartbeat protocol to deal with failures in BEDWE. Experimental results using the Montage workflow over the Grid'5000 platform are presented in Sect. 5. Finally, Sect. 7 presents some applications perspective and concludes the paper.

2 Workflow Patterns and Grammar

Workflow computing is a major paradigm in both business process management and many scientific fields. We distinguish between (1) a workflow specification module that allows to describe services and the ordering amongst them, and (2) a workflow enactment system that coordinates the execution of the services in the correct order. Our contribution is focused on the workflow enactment system level which should handle most workflow models through an appropriate language.

2.1 Workflow Patterns

We here describe the type of workflows that we want to support in BEDWE. A workflow is a finite set of services that are composed in some specific logical order to accomplish a specific process/application. Most commonly, patterns found in workflows are the three following models, as illustrated in Fig. 1: (a) the sequential pattern, (b) the parallel pattern, and (c) conditional pattern.

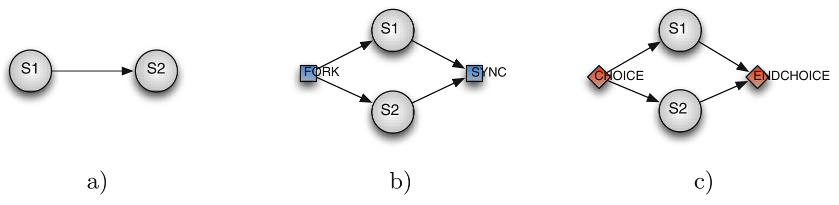


Fig. 1. The three basic workflow patterns.

In order to express more complex patterns, these basic patterns can be combined. Therefore, the loop structure will not be considered in this dissertation. As an example, Fig. 2 shows a concrete workflow which integrates 10 services. The first two, S1 and S2 services are sequentially processed. After service S2, the execution diverges into two parallel branches. The execution of branches will be merged before the execution of service S9. Also, after the execution of service S3, the two outgoing branches are associated with the choice condition and either S3 S4 S6 or S3 S5 S6 path will be selected. Consequently, after S3, a sequential order process is obtained with S6.

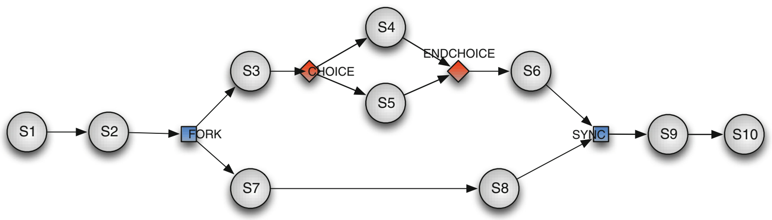


Fig. 2. Example of workflow.

2.2 Workflow Description Language

Defining a workflow consists in describing the patterns that compose it and the relationships among them so, any programmable analyzer can interpret it. As mentioned in Sect. 2.1, a workflow pattern is defined by control-flow dependencies between services. In order to build our workflow grammar, we summarize the possible workflow definition as follows: (1) a workflow is composed of patterns; (2) a pattern is either a parallel, a conditional or a sequential model, or a composition of two or three different models; (3) a parallel workflow pattern starts with the FORK keyword which splits the workflow, followed by a list of branches and ends with the SYNC keyword which merges incoming result of branches; (4) a conditional workflow pattern starts with the CHOICE keyword which splits the workflow, followed by a list of switch branches and ends with the ENDCHOICE keyword indicating a simple merge step. (5) a sequential workflow pattern is one or a set of services with a basic dependency (a simple transition). Using some compiling languages principles and rules it is easy to convert above summary in a grammar annotation similar to the BNF notation that is depicted in Table 1.

Table 1. Workflow grammar.

WF $\rightarrow$ t {WF$_s$ | WF$_p$ | WF$_c$ }*	with:
WF$_s$ $\rightarrow$ t$^+$	t: service identifier, BRC: branch
WF$_p$ $\rightarrow$ FORK(BRC {,BRC}$^+$)SYNC	WF$_s$: sequential pattern
WF$_c$ $\rightarrow$ CHOICE(BRC {,BRC}$^+$)ENDCHOICE	WF$_p$: parallel pattern
BRC $\rightarrow$ WF$_s$ | WF$_p$ | WF$_c$	WF$_c$: conditional pattern

According to this grammar, the workflow in Fig. 2 is easily described as follow, where `Si` is a service identifier:

```
S1 S2 FORK(S3 CHOICE(S4, S5)ENDCHOICE S6, S7 S8)SYNC S9 S10
```

We developed a parser program based on Flex and Bison to validate this grammar [2].

3 BEDWE Workflow Processing

3.1 Pattern Extraction Model

BEDWE decentralized procedure relies on workflow extraction: any workflow w in our grammar can be written as $w = y.z$, y being the first atomic service to be executed on the local node, and z the rest of the workflow definition to be sent in a *message pack* to another node able to execute it. More precisely:

- if y is a sequential pattern, then the first service of the sequence is extracted;
- if y is a parallel pattern, then a list of its branches is extracted;
- if y is a conditional pattern, then from the extracted list of branches, one is chosen according to the choice condition.

3.2 Workflow Execution in BEDWE

In the following, we describe how BEDWE operates, by illustrating its behaviour on a particular workflow composed of the three basic patterns. We assume a set of agents (processes) running on a set of possibly distributed compute nodes, and that are able to communicate through message passing. All these agents run the same software stack; they can invoke services, check the well-formedness of the workflow (or part of a workflow) received and communicate via message-passing. These agents are thus interchangeable and any one of them can be selected to manage the workflow or part of it, as we will now describe. Let us use the example of Fig. 2 that contains the three basic patterns to illustrate different main phases of the process illustrated in Figs. 3, 4, 5, 6 and 7.

Initialization. A first contact node (N1 in Fig. 3), gets a complete description of a workflow from the client, it extracts the first pattern, here S1, it assembles with the rest of definition into a message pack which will be sent to a node N2 that can invoke S1.

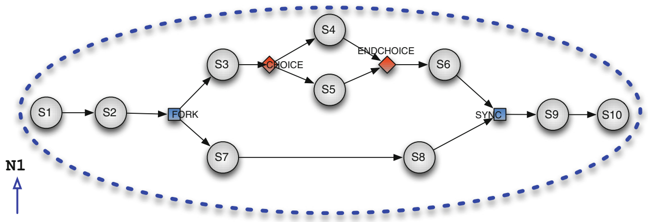


Fig. 3. BEDWE workflow execution (initialisation).

Processing of a Sequential Pattern. As illustrated in Fig. 4, some node N2 gets a message pack from N1, it invokes service S1 and, in the same way extracts from the content of definition the first pattern (here it is S2), it assembles with the rest of definition into a message pack which will be sent to a node N3, that can invoke S2.

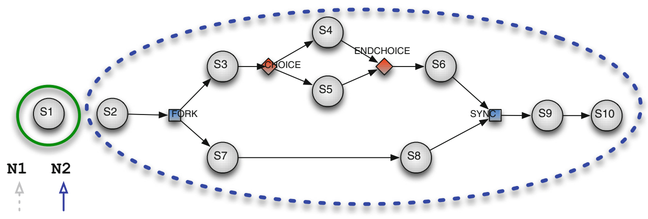
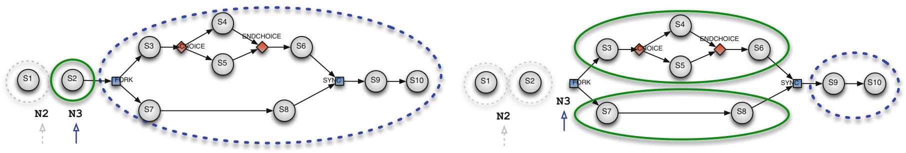


Fig. 4. BEDWE workflow execution (sequential pattern).

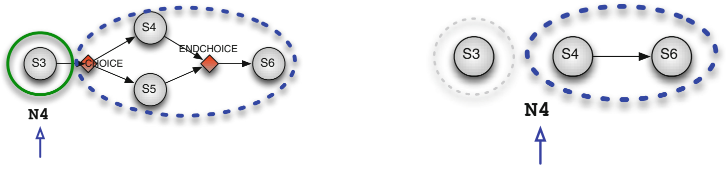
Processing of a Parallel Pattern. As illustrated in Fig. 5, some node N3, receives a message pack from N2, it invokes service S2 and try to extract from the content of definition the first pattern, here it is a parallel model. It extracts all branches belonging to that pattern, here there are "S3 CHOICE(S4, S5)ENDCHOICE S6" and "S7 S8". It then assembles message pack from each branch and sends them to different nodes (here are N4 and N5) which are respectively able to invoke service S3 and service S7. It keeps the following patterns of the parallel model (here: S9 S10) that will be processed after synchronization. Nodes N4 and N5 receive their message packs from the node N3 and they concurrently process to it. Node N5 will process as sequential pattern while node N4 will operate on a conditional pattern after service S3.



(a) before branching. (b) after branching.

Fig. 5. BEDWE workflow execution (parallel pattern).

Processing of a Conditional Pattern. A conditional pattern execution is illustrated in Fig. 6. Node N4, after invocation of service S3 and tries to extract from the content of definition the first pattern, here it is a conditional model. It extracts the two branches belonging to that pattern, here it is S4 and S5. Then, it evaluates the condition of selection, we suppose S4 is selected. The selected conditional branch forms with rest of the branch a sequential model (S4 S6) which will be processed as sequential phase pattern.



(a) before choice. (b) after choice.

Fig. 6. BEDWE workflow execution (conditional pattern).

Processing of a Synchronization. As illustrated in Fig. 7, the nodes which has invoked services S6 and S8 send their results to the node N3 who had split branches for parallel processing. At the end of synchronization, node N3 reads the following of achieved parallel pattern and processes it as any pattern. For our example case, we have a pattern S9 S10 which is a sequential model.

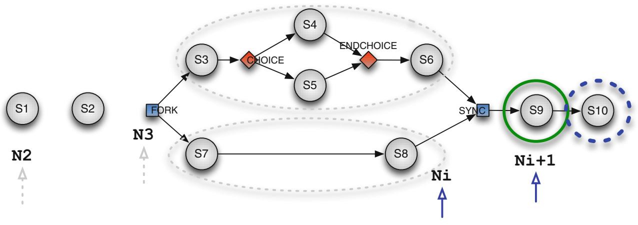


Fig. 7. BEDWE workflow execution (synchronisation pattern).

Ending Output. Still as shown in Fig. 7, after the invocation of service S10, the workflow has completed its execution. The output result of S10 is sent to the client as such a good process of the workflow.

4 Fault Tolerance in BEDWE

A service's execution time can significantly vary between invocations, even for different invocations within the same workflow. This can be due to network's and CPU's load fluctuation. Incidentally, processes may crash. This raises the common question of detecting crashes of computing nodes hosting services: how much time should we wait for an answer before considering a service invocation as *failed*. In BEDWE, we rely on the classical *heartbeat* mechanism and fix a particular duration above which no heartbeat received from a process makes it considered as *failed*.

4.1 BEDWE's Resilience Principle

The implementation of the heartbeat protocol in BEDWE is illustrated in Fig. 8. Assume Node 1 sends a message to Node 2 to request it to process the next workflow pattern. We can consider that Node 1 becomes the client for Node 2 which becomes the server for this particular part of the workflow. During the service's execution on Node 2, Node 2 sends a particular heartbeat message to Node 1 periodically, informing it that it is still running the task. Upon receipt, Node 1 updates its the liveness status of Node 2 as *alive*. Also, periodically, Node 1 checks this liveness status and resets it to *crash*. If, at the next checking time, the status is not back to *alive*, it means Node 2 did not send the heartbeat and thus will be considered as *crashed* by Node 1. When, the task completion's notification is received, Node 1 stops its periodic checking of Node 2's liveness. Note that a node can be both a server and a client for different tasks.

4.2 Fault Tolerance for Each Pattern

Let us now review more precisely how the resilience is done for each possible pattern. Note that the resilience mechanism is recursively done in recursive patterns

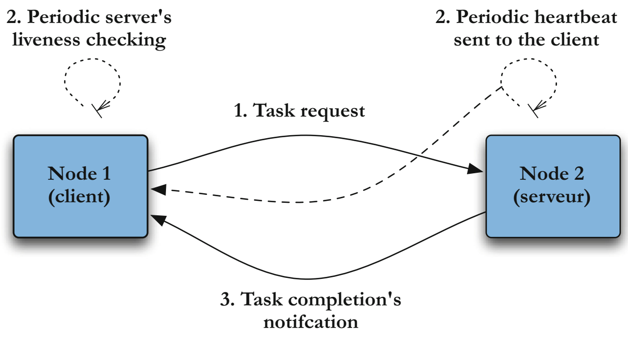

2. Periodic server's liveness checking
2. Periodic heartbeat sent to the client
1. Task request
Node 1 (client)
Node 2 (serveur)
3. Task completion's notifcation

Fig. 8. BEDWE's resilience principle.

such as nested parallel patterns. The key assumptions in the following are that: (1)A node cannot be responsible for the execution of two consecutive services in a sequence. (2) The two nodes responsible for two consecutive services cannot fail at the same time.

Sequential Pattern. In a sequence of services, the heartbeat protocol presented previously is repeated between each pair of nodes managing consecutive services. Given a sequence of three services whose execution is hosted by N1, N2 and N3 respectively, the process is first started when N2 starts executing, N2 being the server and N1 the client. Once N2 completes, it sends the subsequent part of the workflow to N3. At this point, the heartbeat mechanism is also triggered between N2 which is now the client and N3 which is the server. Once the mechanism is started between N2 and N3, N2 sends the notification of completion and the heartbeat protocol between N1 and N2 is stopped.

Parallel Pattern. Assume a node N1 starting a parallel pattern whose first nodes are respectively N2 and N3. In this case, N1 sends one distinct workflow branch to N2 and N3. Two concurrent instances of the heartbeat protocol are started between N1 and N2 on one hand and N1 and N3 on the other hand. When N2 (respectively N3) completes its service and if there are other services in this branch, then N2 (resp. N3) forwards the residual branch to another node say N4, (resp. N5). When N4 and N5 receive their respective residual branch, N2 and N3 stops sending hearbeats to N1 but two new heartbeats protocols are started between N2 and N4 on one hand, and N3 and N5 on the other hand. The process of shifting the heartbeat along the branch is done in parallel inside the two branches until reaching their ends.

Conditional Validation. A conditional pattern is converted to sequential one after the selection test. So, their validation is similar to the sequential case.

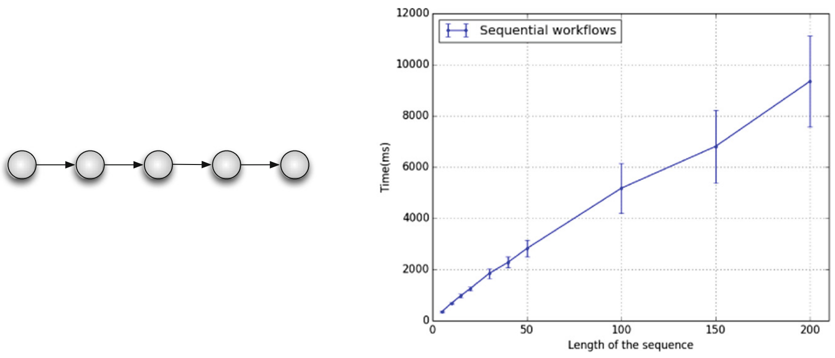


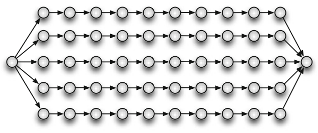
Fig. 9. Experiments with sequential workflows.

5 Experimental Validation

To validate BEDWE, we developed a Java-based software prototype implementing the algorithms described above. It represents more than 1700 lines of code. The prototype has been enhanced with the resilience mechanism described in Sect. 4. Deploying BEDWE means deploying BEDWE agents (as described in Sect. 3) implementing the algorithms described above, and using sockets to communicate portions of workflows, acks and heartbeats. The prototype was deployed over the Grid'5000 platform which gathers more than 8000 compute cores distributed over 8 geographically distributed sites [3]. In the following experiments, a set of BEDWE agents were deployed over computing cores of Grid'5000, and different workflows were submitted to these agents. All the experiments were conducted on the *parapide* cluster located in Rennes, composed of Intel Xeon X5570 CPUs with 8 cores and 24 GBs of memory each. For whole set of experiments, 40 BEDWE agents were deployed over 40 different cores. Agents are initially *idle*: they are listening for an incoming workflow (or sub-workflow) to be submitted.

The first experiments were done on sequential workflows, *workflows* composed of a single sequential pattern whose length varied between 5 and 200, as illustrated in the right part of Fig. 9). The results are given in the right part of Fig. 9. We observe an execution time growing linearly with the number of services. This is to be expected in the sense that such a workflow does not require to have many agents to work at the same time. At most two agents are required to be active at the same time due to the resilience mechanisms: an agent that completed a service waits for the completion of the next service in the workflow, running in another agent, before becoming idle again.

The second set of experiments conducted was performed using parallel workflows. These workflows were composed of one fork giving birth to a number of branches varying between 2 and 20. The number of sequential services within a branch was kept constant, at 10. Such a workflow is illustrated for a number of branches of 5 in the left part of Fig. 10. The objective of this experiment was to



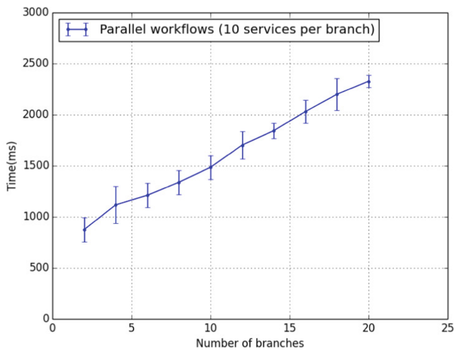


Fig. 10. Experiments with parallel workflows.

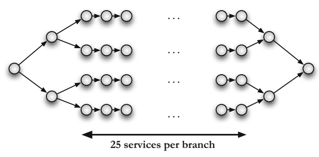
test the ability of BEDWE to tackle high parallelism. The result is shown in the right part of Fig. 10. It suggests that the BEDWE prototype is able to leverage the parallelism: recall that in the previous, sequential experiment, 200 sequential services were completed in more than 9 s. Now, when we have 20 branches, we have also 200 services, but the length of the branches are only 10, and the completion time is between 2 and 2.5 s, which is far closer to the case of a sequential case of 10 services (even if still significantly longer).

The last experiment was performed on workflows with nested forks. The workflows used in this part are all composed of 100 services, but the number of nested forks varies from 1 to 5. This supposes to adapt the number of services per branch for each case. The particular case illustrated in the left part of Fig. 11 is for the case of 2 nested forks, resulting in 25 services in each branches. For cases where the total number of services cannot be divided by the number of branches, few isolated services were added between forks. In our experiments, as plotted in the right part of Fig. 11, we observed a slightly increasing completion time, which can be explained, in spite of the higher degree of parallelism, by the increased complexity of forks management. In particular, the more nested forks, the more nodes waiting for the completion of the fork they are responsible for, making less nodes available to run tasks and an increase in sequentiality.

6 Related Work

Workflow execution is a topic which is not settled yet, as highlighted by the recent articles covering it in literature [4,5]. Traditionally, workflow execution is centralized, would it be in an industrial [1] or more academic/scientific context [6, 7]. While very mature and efficient, these tools cannot be used in a context where the continuous presence of an orchestrating tool is mandatory.

The idea of describing direct interactions between services in the execution of a composition was proposed in the concept of *choreography of web services* [8]. Choreography allows to describe at once all the interactions that will take place at execution time. It allows to be more precise than orchestration in which the



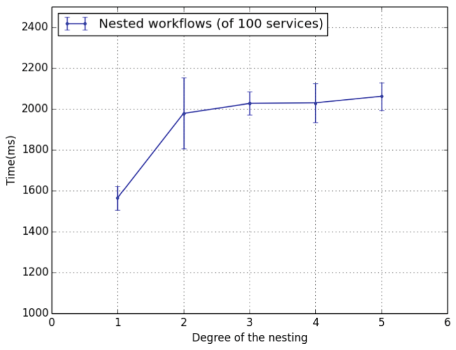


Fig. 11. Experiments with nested parallel workflows.

execution is seen from a single point of view, and to describe interactions between different organizations each providing part of the services to be combined [9].

Decentralising the workflow execution by relying on direct interactions (based on messaging) between services has been proposed in [10–12]. In particular, A continuation-passing style, where information on the remainder of the execution is carried in messages, has been proposed in [11]. Nodes interpret such messages and thus conduct the execution of services without consulting a centralised engine. However, nodes need to know explicitly which nodes to interact with and when, in a synchronous manner. A similar idea, based on the dynamic partitioning of the workflow as its execution moves forward has also been studied in [13]. Bedwe follows a similar principle but focus on parallel splits and choices, while extending such mechanisms with a particular protocol for fault-tolerance.

7 Conclusion

This paper has proposed BEDWE to decentralize workflow execution over unreliable platforms. The platform envisioned in this work is a set of compute nodes in regions subject to power cuts according to a predefined schedule, due to an insufficient level of electric injection to satisfy all the needs. Ensuring the completion of workflow execution on computers located in such an environment calls for decentralization and fault-tolerance. The BEDWE engine is supposed to run on every node taking part in the workflow engine. Engines use a specific protocol allowing to send, receive and process portions of the workflow, to split the workflow into several parallel executions, and synchronize their outcomes. Nodes taking care of contiguous portions of the workflow are watching each others. A BEDWE prototype was implemented in Java and validated over the Grid'5000 platform. Our future work will include devising a solution based on BEDWE for executing workflows in an African context, specifically for the city of Lome, Togo, which is subject to planned power cuts. With such a solution, a workflow execution will dynamically move from power outages area to some neighborhood supplied with power. While the targeted platform is very different from the platform used for the experiment, the present article was about validating the approach.

References

1. Weerawarana, S., Curbera, F., Leymann, F., Storey, T., Ferguson, D.F.: Web Services Platform Architecture: SOAP, WSDL, WS-Policy, WS-Addressing, WS-BPEL, WS-Reliable Messaging and More. Prentice Hall PTR, Upper Saddle River (2005)
2. Levine, J., John, L.: Flex & Bison, 1st edn. O'Reilly Media Inc., Sebastopol (2009)
3. Bolze, R., et al.: Grid'5000: a large scale and highly reconfigurable experimental grid testbed. Int. J. High Perform. Comput. Appl. **20**(4), 481–494 (2006)
4. Hi-WAY: Execution of Scientific Workflows on Hadoop YARN, 21–24 March 2017
5. Marozzo, F., Duro, F.R., Blas, F.J.G., Carretero, J., Talia, D., Trunfio, P.: A data-aware scheduling strategy for workflow execution in clouds. Concurr. Comput. Pract. Exp. **29**(24) (2017)
6. Wolstencroft, K., et al.: The Taverna workflow suite: designing and executing workflows of web services on the desktop, web or in the cloud. Nucl. Acids Res. **41**(Webserver–Issue), 557–561 (2013)
7. Ludäscher, B., et al.: Scientific workflow management and the Kepler system. Concurr. Comput. Pract. Exp. **18**(10), 1039–1065 (2006)
8. Barros, A., Dumas, M., Oaks, P.: A critical overview of the web services choreography description language. BPTrends (2005)
9. Qiao, X., Wei, J.: A decentralized services choreography approach for business collaboration. In: International Conference on Services Computing (SCC 2006), Chicago, USA, pp. 190–197 (2006)
10. Micillo, R.A., Venticinque, S., Mazzocca, N., Aversa, R.: An agent-based approach for distributed execution of composite web services. In: Proceedings of the 17th IEEE International Workshops on Enabling Technologies, Rome, Italy, June 2008
11. Yu, W.: Consistent and decentralized orchestration of BPEL processes. In: Proceedings of the 24th ACM Symposium on Applied Computing (SAC), Honolulu, March 2009
12. Downes, P., Curran, O., Cunniffe, J., Shearer, A.: Distributed radiotherapy simulation with the webcom workflow system. Int. J. High Perform. Comput. Appl. **24**, 213–227 (2010)
13. Atluri, V., Chun, S.A., Mukkamala, R., Mazzoleni, P.: A decentralized execution model for inter-organizational workflows. Distrib. Parallel Databases **22**(1), 55–83 (2007)