

Data Management for the RedisDG Scientific Workflow Engine

Leila Abidi, Souha Bejaoui, Christophe Cérin, Jonathan Lejeune, Yanik Ngoko, Walid Saad

▶ **To cite this version:**

Leila Abidi, Souha Bejaoui, Christophe Cérin, Jonathan Lejeune, Yanik Ngoko, et al.. Data Management for the RedisDG Scientific Workflow Engine. IEEE International Conference on Computer and Information Technology, Dec 2016, Nadi, Fiji. pp.599 - 606, 10.1109/CIT.2016.55 . hal-01517163

HAL Id: hal-01517163
https://hal.science/hal-01517163

Submitted on 2 May 2017

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Data Management for the RedisDG Scientific Workflow Engine

Leila Abidi*, Souha Bejaoui*, Christophe Cérin*, Jonathan Lejeune*, Yanik Ngoko* and Walid Saad*†

Université de Paris 13, LIPN* and University of Tunis, LATICE†
* 99, avenue Jean-Baptiste Clément, 93430 Villetaneuse, France
† ENSIT, 5 av. Taha Hussein, B.P. 56, Bab Mnara, Tunis, Tunisia
Email: {leila.abidi,souha.bejaoui,christophe.cerin,jonathan.lejeune,yanik.ngoko,walid.saad}@lipn.univ-paris13.fr

***Abstract*—In this paper we investigate the general problem of controlling a scientific workflow service in terms of data management. We focus on the data management problem for the RedisDG scientific workflow engine. RedisDG is based on the Publish/Subscribe paradigm for the interaction between the different components of the system, hence new issues appear for scheduling. Indeed, the Publish/Subscribe paradigm utilization introduces different challenging problems, among them the design of effective solutions for managing data, on the fly, when tasks are published. Our contributions are twofold. First we add new functionalities to the RedisDG workflow engine with scheduling decisions related to the allocation of data intensive jobs to compute units and according to an efficient management of data and second we introduce a large set of experiments to validate our approaches. We analyze our results and we also sketch perspectives and insights. Experiments are conducted on the Grid'5000 testbed and the paper is a step forward to implement a 'Workflow engine as a Service' (WaaS).**

***Index Terms*—Large scale cloud applications including Internet/Web computing, Volunteer computing, Data management, Scientific workflow, Scheduling, Experimental evaluation on large scale systems.**

1. Introduction

In this paper, the problem of controlling the propagation of data is directly linked to the problem of finding a 'good' allocation of tasks of the workflow and also to resource utilization. We consider the problem as a balance between multiple objectives. To exemplify our work we can imagine a user, connected to a cloud. Then he pays for N computing units according to his budget, downloads his workflow description and executable codes, then he launches his application. The cloud system deploys the infrastructure, activates the N computing units, executes the workflow. Basically, the cloud user needs to be sure that two objectives are fulfilled: (1) all the reserved computing units are used and (2) the execution time of the workflow is as low as possible. For this purpose, we propose in particular to optimize the data placement during the execution of the workflow.

The architectural context of the RedisDG system that we are designing is very important to understand because it underlines the specific difficulties of the allocation problem and strategies we are designing. RedisDG is based on the Publish/Subscribe paradigm which is an asynchronous mode for communicating between entities [1], [4]. This communication mode is multipoint, anonymous and implicit which increases the scalability by eliminating many sorts of explicit dependencies between participating entities. Eliminating dependencies reduces the coordination needs and consequently the synchronizations between entities. One challenge of this system is to 'see' the scheduling/allocation mechanisms as interactions between software components and not as a separate component with no relationship with its environment. Nodes may join or leave the workflow system at any time to mimic a dynamic system or a volunteer based system. The Publish/Subscribe model helps in realizing this vision. We mention this property for the sake of completeness of the potential of our RedisDG system [2], [3]. However we do not explore this possibility in this paper and we stay in a confined environment such as with a data center or a grid in a conventional sense.

Google Cloud Pub/Sub[1] for instance offers asynchronous messaging that allows for secure and highly available communication between independently written applications. Google Cloud Pub/Sub helps developers to quickly integrate systems hosted on the Google Cloud Platform and externally. For example, a large queue of tasks can be efficiently distributed among multiple workers, such as Google Compute Engine instances or an order processing application can place an order on a topic, from which it can be processed by one or more workers. Another example corresponds to a residential sensor streaming data (through a channel) to back-end servers hosted in the cloud.

The readers should be aware that the design of scheduling decisions according to the Publish/Subscribe paradigm is innovative and non conventional. In comparison to classical workflow scheduling approaches, we deal with online scheduling with unavailability constraints. However, we have a special unavailability characterized by the fact that after a publish message, the more we wait, the more we

1. https://cloud.google.com/pubsub/overview

have candidate workers for the execution of our tasks. However, the more we wait, the more we delay the execution of the workflow. In addition, the readers also need to assume that, in using the Publish/Subscribe paradigm:

- we have specific problems, for instance those related to the management of Publish/Subscribe events that lead to fairness problems (the nodes with the lowest latency have more chances to be selected). We do not show any experimental proof of this phenomenon because of space limit which is part of another paper. Here we control the number of jobs done by any worker. This is one of the performance metric. Another one is the total execution time as explained later in the main tabular for experimental results.
- we can investigate solutions to the allocation problem, based on the Publish/Subscribe, which are unusual as explained before. The solutions are general in the sense that we are controlling the problems inside the RedisDG system making possible the utilization of any Publish/Subscribe layer and not only Redis[2]. But, it is important to notice that we focus on optimized data placement. We also do not formalize the problem (with complexity results etc) but we do prefer to give practical insights to solve it effectively.

The contributions, as explained in the paper, are twofold: a) solving the allocation problem under the Publish/Subscribe paradigm throughout effective data-aware heuristics b) do large set of experiments (and hidden implementation stuff) to demonstrate the potential of our approaches. In conclusion, we investigate the general problem of allocating resources but in the context of a different paradigm than the ones we traditionally find in the literature. This opens new directions that are motivated in the paper.

The organization of the paper is as follows. In section 2 we introduce our context and we recall some notions about the RedisDG system that we turn into a workflow engine. Section 3 is related to our data-aware approaches and we introduce our heuristics. Section 4 deals with the experiments with the RedisDG system and we analyze the results. Section 5 presents the related works. Section 6 concludes the paper and draw perspectives.

2. Context of the work

In this section, we recall the coordination algorithm of RedisDG system which is the core of our paper. The goal and the difficulties are to define a middleware able to support workflows that is light enough to be integrated easily into a cloud and using current Web technologies (in our case the Publish/Subscribe paradigm).

In Figure 1, we depict the steps of an application execution. In RedisDG, a task may have five states: *WaitingTasks, TasksToDo, TasksInProgress, TasksToCheck* and *FinishedTasks*. These states are managed by five actors: a broker, a coordinator, a worker, a monitor and a checker. Taken separately, the behavior of each component in the system may appear simple, but we are rather interested in the coordination of these components, which makes the problem more difficult to solve. The key idea is to allow the connection of dedicated components (coordinator, checker...) in a general coordination mechanism in order to avoid building a monolithic system. The behavior of our system as shown in Figure 1 is as follows:

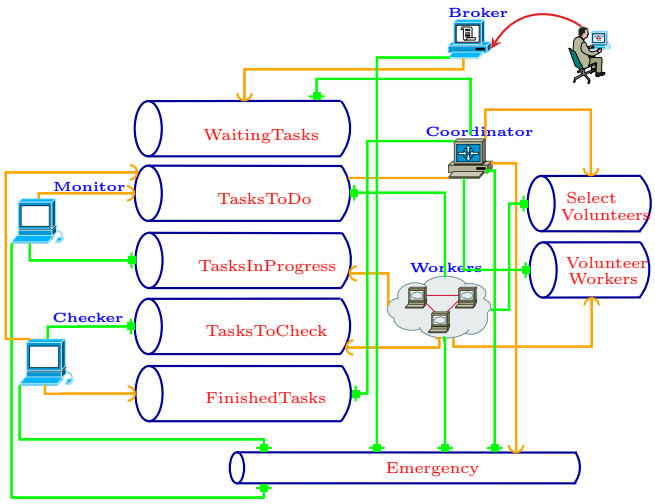


Figure 1. Interactions between components of the RedisDG system

1) Tasks batches submission. Each batch is a series-parallel graph of tasks to execute.
2) The Broker retrieves tasks and publishes them on the channel called *WaitingTasks*.
3) The Coordinator is listening on the channel *WaitingTasks*.
4) The Coordinator begins publishing independent tasks on the channel *TasksToDo*.
5) Workers announce their volunteering on the channel *VolunteerWorkers*.
6) The coordinator is aware of worker volunteering by listening the *VolunteerWorkers* channel.
7) The coordinator selects Workers according to several criteria (e.g. SLA).
8) The Workers, listening beforehand on the channel *TasksToDo* start executing the published tasks. The event 'execution in progress' is published on the channel *TasksInProgress*.
9) During the execution, each task is under the supervision of the Monitor whose role is to ensure the correct execution by checking if the node is alive. Otherwise the Monitor publishes again, tasks that do not arrive at the end of their execution.
10) Once the execution is completed, the Worker publishes the task on channel *TasksToCheck* by indicating information about task execution (e.g. time execution, CPU consumption, etc.).
11) The Checker verifies the result returned and publishes the corresponding task on the channel *FinishedTasks*.

2. Technically speaking we use http://redis.io Pub/Sub capabilities

12) The Coordinator checks dependencies between completed tasks and those waiting, and restarts the process in step (4).
13) Once the application is completed (no more tasks), the Coordinator publishes a message on the channel *Emergency* to notify all the components by the end of the process.

Summarizing, it is important to understand all the interactions in this protocol because we will explain later on some pitfalls leading to observational behaviors on real infrastructures. These unexpected behaviors, not visible in the modeling steps, are related to the scheduling policies that we introduce now.

3. Our Data-aware approach

In this paper we consider List Scheduling Algorithms. The method consists in building the list of tasks to be executed in considering the precedence between tasks but also a scoring to decide on the allocation. The principle is as follows. We may consider groups of machines, denoted by G_j, for modeling the case where the transfer between machines in that group has no cost. Task T_i will be allocated in the group G_j for which the intersect between the files resident on machines of G_j and files required by T_i is maximal.

Note that in the Publish/Subscribe context, this principle does not always mean that we need to wait for all workers. Assume for example that at the beginning, all the workers are equivalent. Then we can select one of them. Similarly, if we have a task that has no dependencies, all workers are equivalent. Finally, if we maintain internally a state table of files that workers have, we do not need to wait for a response of everyone to allocate tasks.

We also consider a centralized version for the data management and a decentralized one. The first one consists in using a central server where workers put and get their data. This model was the initial model for RedisDG and it now serves as a reference model for the performance. The later one tries to transfer data from machines to machines without intermediary.

3.1. Motivating example

Let us consider the direct acyclic graph of Figure 2, where tasks have the same execution time for sake of simplicity. Assume that the overall objective is the minimization of the data transfer costs. In this example, we weaken the problem by considering the criterion of the number of files to transfer. Table 1 lists the size of required files for the execution. When we start the execution, the broker publishes tasks from 1 to 9 on the WaitingTasks channel and, according to dependencies, they will be published on the TasksToDo channel by the coordinator. We now explain the different cases, mainly on physical machines while introducing our scheduling approaches. The case for containers is left as an exercise because of the space limit.

Execution on physical machines and minimization of the number of files to transfer: this method consists in examining, each time we have a ready task i.e. published on the TasksToDo channel, its inputs and to compare them with the files that exist on workers that have published their volunteering on the channel Volunteer-Workers. Then, we need to select the worker that hosts the greatest number of files for the considering task. Consider now the execution of our workflow on two workers $W1$ and $W2$.

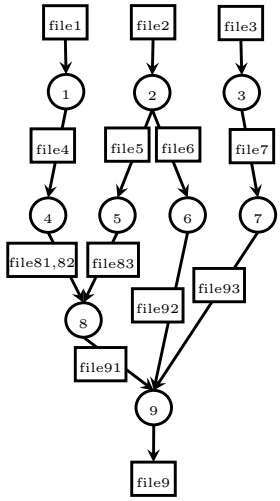


Figure 2. The direct acyclic graph of our illustrative example

TABLE 1. Sizes of files in our illustrative example

File	file81	file82	file83	file91	file92	file93
Size (MB)	2	3	8	6	3	15

The first task to be published by the coordinator is $T1$, and this task can be executed either on $W1$ or $W2$ since the inputs are already located on these machines. Assume and according to a FIFO strategy that $W1$ is answering first, then $W1$ executes $T1$. Symmetrically, assume that $W2$ is publishing first its volunteering for $T2$ and $W1$ for $T3$. Then we will get $W2 \longleftarrow T2$ and $W1 \longleftarrow T3$.

Once $T1, T2$ and $T3$ executed, $T4, T5$ and $T6$ become available and independent. They will be published on the TasksToDo channel in the previous order, since they have been published in the same order by the broker on the WaitingTasks channel.

For the allocation of task $T4$, it is obvious that $W1$ is the best choice since the single file required by $T4$ exists in $W1$. On the other hand, if $W2$ is chosen for the execution of $T4$, it needs to download first $file4$ from $W1$ before the execution. It is the same for tasks $T5$ and $T6$ that will be executed on $W2$ and $T7$ on $W1$, hence the allocations $W1 \longleftarrow T4, W2 \longleftarrow T5, W2 \longleftarrow T6$ and $W1 \longleftarrow T7$.

Once the execution of $T4$ and $T5$ completed, $T8$ is published on the TasksToDo channel. Both $W1$ and $W2$ contain inputs of $T8$. Since in our case, minimizing the number of files to exchange is equivalent to the selection of the worker with the maximal required files, we proceed as follows. $W1$ contains 2 files that come from the execution of $T4$ and $W2$ contains only one file that come from the execution of $T5$. Then, we chose $W1$ for the execution of $T8$ and we download $file83$ from $W2$, hence, according to

our notations $W1 \longleftarrow T8$.

The $T9$ task has 3 predecessors named $T6, T7$ and $T8$, that are respectively executed on $W2, W1$ and $W1$. As for $T8$, we select $W1$ for the execution of $T9$ in order to minimize the number of file to transfer, hence $W1 \longleftarrow T9$. Finally we obtain the allocation given on Table 2.

TABLE 2. Allocation for our example with 2 workers

Time/Event	W1	W2
t_0	T1, T3	T2
Publication of T4, T5, T6 and T7 on TasksToDo		
t_1	T4, T7	T5, T6
Publication of T8 on TasksToDo		
t_2	T8	
Publication of T9 on TasksToDo		
t_3	T9	

3.2. Presentation of our heuristics

The overall objective is the minimization of the data transfer costs to decide the tasks placement. Two directions are under concern with our heuristics that we manage with the Publication-Subscription schema making the originality of the implementation. The first one seeks for minimizing the total number of files we have to exchange. The second one seeks for minimizing the total size of files we have to exchange.

Initially, the RedisDG system was based on a FIFO (First In, First Out) heuristics: the worker who answered the first one was selected (modulo the compliance with certain capacity constraints such as the CPU type, the RAM or disk availability etc). Now, we can wait more or less over time to keep the first $k > 1$ workers. For instance and depending on the heuristics, the coordinator maintains a table of the past allocations according to a criterion based on a metric related to data. The coordinator pre-computes a set of 'best candidates' for the next tasks to execute. It publishes the task and waits for a reply of one/two/... 'best candidates'. Then it decides on the allocation. This protocol exemplifies the ways to cope with the Publish/Subscribe in designing data-aware approaches for the RedisDG system.

In our current work, we do not consider the latency and the bandwidth between any two pairs of machines but we have rather observed the following situation with a great impact on performance. Since the objective is to reuse the machines with the data in place (which are those having executed the root tasks), we tend to evict some machines and to choose almost the 'same machines'. Following this idea, there is no transfer if we use only one and the same machine. To counter-balance this phenomenon we decided to implement a 'round robin' algorithm (for the root tasks) in order to maximize the parallelism between tasks, to give the same chance to machines to fairly contribute, and to increase the performance.

3.2.1. Heuristics: one worker per node.

A. FIFO Heuristic (FIFO): RedisDG framework is able to run according to this method: the coordinator selects the first replying worker, then it changes its state from SUBMITTED to GIVEN and publishes to all the workers (to guarantee some liveness property) the information about the task's id and the selected worker.

B. Input Number Heuristic (IN): the allocation that follows this approach distinguishes between 2 types of tasks.

- Root tasks: they do not require any transfer since the inputs are already in place in the distribution that we deploy. In this case we keep the FIFO heuristic for allocating tasks to workers.
- Non root tasks: we compute a score for each worker. This score corresponds to the number of inputs required by the task. The value is computed as follows:

$$score(W_j, T_i) = card(I_{T_i} \cap DW_j)$$

This equation explains that the scoring associated to worker W_j for executing task T_i is the cardinality of the set of input files issued from T_i (denoted I_{T_i}) that is intersected with the set of files located on the worker W_j (denoted by DW_j). Thus:

$$score(W_j, T_i) = card(I_{T_i} \cap (\bigcup_{p \in Pred(i,j)} O_p))$$

where $Pred(i, j)$ be the set of predecessor tasks executed by the worker W_j, and O_p be the list of the outputs of tasks p. Since $Pred(i, j)$ is finite we have that:

$$\begin{aligned} score(W_j, T_i) &= card(\bigcup_{p \in Pred(i,j)} (I_{T_i} \cap O_p)) \\ &= \sum_{p \in Pred(i,j)} card(I_{T_i} \cap O_p) \end{aligned}$$

C. Input Size heuristic (IS): This heuristic uses the same reasoning as the previous one but in this case the score is computed as the sum of the required files sizes. Formally, the score is

$$score(W_j, T_i) = \sum_{p \in Pred(i,j)} \sum_{f \in (I_{T_i} \cap O_p)} Size(f)$$

The technical difficulty is about the computation of this value in the context of the Publication-Subscription paradigm. The tricky part was on reusing the FinishedTasks channel (see Figure 1) with a special message related to the file size, on modifying the format for messages and tasks. Thus the process is as follows. When a worker has completed a task, it publishes the sizes of its output files in the FinishedTasks channel. The coordinator receives this message as well, and it uses this information for the allocation of non-root tasks.

D. Fair Root heuristics (FRIN, FRIS): To reduce the effect of monopolizing the tasks by fewer workers, accentuated by both previous heuristics, we have implemented two other heuristics: Fair Root Input Number and and Fair root Input Size. Indeed, since these last tasks generate inputs for tasks in the next level, their distribution is

conditioning the allocation of the forthcoming tasks. Then, fair distribution of root tasks guarantee better management of resources and subsequently better execution time due to the effect of parallelization.

E. Fair Distribution heuristic (FD): This heuristic generalizes the previous one to all the levels of the tasks graph so that it is no more reserved for the root tasks only and gives us an other point of comparison to justify the use of the Data-Aware-Scheduling approach.

3.2.2. Heuristics: many workers per machine based on equivalence class.

We apply all the previous heuristics for the case of Docker containers. The different scores are computed by tacking account of the localization of each container. We compute a single score per machine. In this case, we consider that all the containers belong to an equivalence class. Consequently, we can apply the Fair Distribution case to balance the work among all the containers.

4. Experiments

4.1. Montage Workflow

The MONTAGE[3] project has been created by NASA/IPAC *Infrared Science Archive* as an open-source toolbox to generate personalized mosaic of the sky from images in the *Flexible Image Transport System* (FITS) format. The MONTAGE application has been represented as a workflow that can be executed on the Teragrid [4] infrastructure for instance. On Figure 3, we show a sample of a MONTAGE workflow (only 20 nodes) that has been generated from the workflow generator [5]. What is important to notice is the shape of the workflow and the corresponding independent part that could be executed in parallel. In Table 3, we show an instance of MONTAGE application, that we have executed with RedisDG. This example will be use in an intensive way in the paper. It exhibits a workflow with 1446 tasks and 3722 dependency links between tasks. The execution of this quite large instance requires 9423 input files (including the intermediary files) and it generates 2889 files (including the intermediary files).

TABLE 3. An instance of MONTAGE workflow with 1446 nodes

Level	Task	Number of tasks
1	mProject	301
2	mDiffFit	838
3	mConcatFit	1
4	mBgModel	1
5	mBackground	301
6	mImgtbl	1
7	mAdd	1
8	mShrink	1
9	mJPEG	1

3. http://montage.ipac.caltech.edu/index.html
4. http://www.teragrid.org/

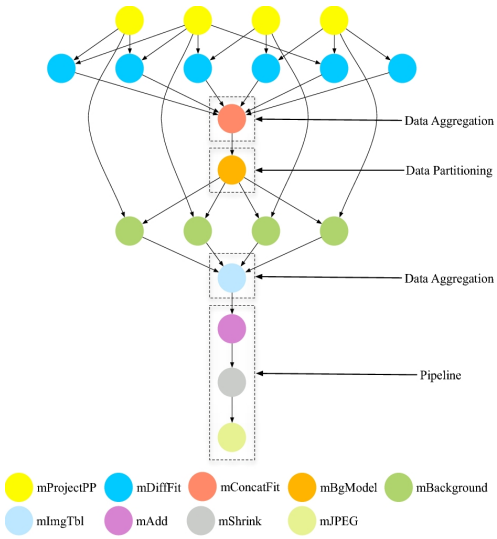


Figure 3. An instance (20 nodes) of MONTAGE workflow

4.2. Experimental plan and performance metrics

We use the Grid5000 testbed as the experimental platform and across different clusters inside Grid5000. We deployed a Grid5000 image containing the RedisDG workflow engine and all the compiled versions of the MONTAGE workflow. The input files of the MONTAGE workflow are also included inside this image. We use a moderate number of machines, leading to an execution time of one instance of the MONTAGE workflow below 10 minutes. In the reminder of the paper we give only representative examples of our experiments. The execution times are given as a mean value plus the standard deviation. We are experimenting either with bare machines or with Docker containers. In this last case and technically speaking we have reserved a Docker volume where any containers on the same machine can store and upload their input and output files. Initially, this volumes contain the files of the root tasks which is part of the Linux image that we deploy on each machine. The other files are generated on the fly by the application. The different performance metrics that we consider are as follows. The *Input transfer time* (Input) is the total time for transferring the inputs for all the tasks. The *Processing time* (Processing) is the total computing time for all the tasks. The *Output transfer time* (Output) is the total time for transferring the output files for all the tasks. The *Total execution time* (Total) is the sum of the 3 previous times. The *Execution time of the workflow* (Execution) is the difference in time between the beginning of the execution (date of the first publication by the broker) and the end of the workflow (at the moment of the publication of a message on the emergency channel). The *Distribution of tasks* (Distribution) is the number of tasks executed by each worker (physical machines or Docker containers).

4.3. Results and analysis

4.3.1. Homogeneous physical machines. In this scenario we use the Griffon cluster in Nancy: 1 machine for the master (coordinator and broker daemons) and 4 machines as workers. We reported in Table 4 the results for each heuristics and corresponding to mean values. We also reported the standard deviation. Since we have experimented in an homogeneous context the standard deviation for the Processing time is reported as zero. The result for a centralized data management and a FIFO scheduling is presented in line #01. Then line #02 is for a decentralized data management and a FIFO scheduling. At least we introduce the results for the Input Number Strategy (IN), for the Input Size strategy (IS), for the Fair Root Input Number strategy (FRIN), for the Fair Root Input Size strategy (FRIS) and for the Fair Distribution strategy (FD). Since the management for input and output files and the processing time do not greatly vary from one run to another, the determining performance factor is the distribution of tasks within workers. Indeed, we noticed a case where a worker has executed alone 36.4% of the tasks, that is equivalent to 30% of the total execution time. However, when tasks are distributed in a more 'fair' manner the execution time is better. Consequently we obtained a better resource utilization. From a data management point of view, reducing the data transfer can improve the total execution time of the workflow, especially for the MONTAGE application. As shown on Table 4, the processing time counts for 50% of the total execution time, while the other 50% corresponds to data transfers. On average, the decentralized data management has reduced by 16% the duration of execution compared with the centralized case, by 21.4% the total time of execution of tasks, by increasing the input transfer time of 10% (overhead for the search of predecessors and workers) and by reducing by 82% the output transfer time, which justifies our scheduling heuristics according to our new data management strategies. We note that the results given by the first family of heuristics (IN and IS on lines #03 and #04) are very close: this is explained by the homogeneity of the sizes of the files involved in the MONTAGE workflow, which are about 5MB. The ideal situation is probably in combining the two dimensions when scheduling other workflows. Regarding the experimental results on Table 4, the Input Number heuristic and the Input Size heuristic have reduced, on average and compared with the decentralized data management with FIFO scheduling (see line #02), by a factor of 10.7% (respectively 9.6%) the execution time and by a factor of 11.4% (respectively 13%) the total execution time. If we compare the new results with those of the original version of RedisDG (see line #01), we note, for the Input Number heuristics (Input Size heuristic respectively), a decrease in execution time by 25% (respectively by 24%) and a decrease in the total duration of task execution by 30% (respectively by 30%). These experimental results justify our data-aware approaches. Although the experimental results of the second family of heuristics (FRIN and FRIS on lines #05 and #06) coupling fairness and data transfer optimization, are better than those of the first series in terms of execution times (385s against 408s and 394s against 413s, on average). Indeed this particular case of homogeneous machines does not demonstrate the best value of our approaches. In fact, in the case of heterogeneous machines (not seen in this paper), the most efficient machines tend to monopolize the computation while the remaining workers have almost nothing to do. The better management of resources has led to lower execution time even in the case of homogeneous machines. From the execution time point of view, we went from 385s for the Fair Input Number heuristic to 454s for the Fair Distribution heuristic, with a loss of 21% of the execution time. Furthermore, by extending the fair scheduling to all tasks (line #07), we lost 31% of the transfer time of inputs, always comparing to the same heuristic. Moreover, the results of the Fair Distribution heuristic are very close to those of FIFO.

4.3.2. Docker. We present in this subsection and on Table 4 the results of the execution of the MONTAGE application on 5 machines in the Griffon cluster of the Nancy site: 1 master machine and 4 machines each containing 3 workers (Docker containers). On line #08 we noticed a decrease of 13% of the execution time comparing to the case of using physical machines but an increase of 25% for the total execution time. This increase is mainly due to the emphasis of the effect of data management centralization: 12 workers send and receive data instead of 4 in the case of the physical machines. The results on line #09 have been improved over the centralized data management case on line #08, but also compared to the performance with physical machines in the same context of decentralized data management (lines #02-#07): the execution time decreased by 17% and the total execution time of the workflow decreased by 10.4%. The executions with Docker as shown on lines #10, #11, #12, #13 and #14 not only confirmed, once again, our previous findings. They also underline the importance of an efficient resource management, both by distributing roots tasks equally on machines and in maximizing the effect of the parallelization through the use of several containers per node.

4.3.3. Synthesis. All the experiments confirmed that the decentralized approaches are better than the centralized approaches and this result was expected because we bypass an intermediary server. We have also demonstrated that the graph structure, in particular the number of root tasks (referenced as level 1 in tabular 3) coupled with a fair distribution of tasks, impacts the performance metrics. For instance we introduced the Fair Distribution heuristic to justify the involvement of the data transfer factor during the scheduling phase. Indeed, the best results we have obtained so far are for Fair Root heuristics. We also noticed an improvement between the FIFO scheduling case and the fair scheduling case for root tasks. We also confirmed, empirically, that the pair (Fairness, Data) is the best strategy (see the column Distribution) among those that we have, in the case of homogeneous physical machines.

TABLE 4. Experimental results. The variance for the Processing time is zero because we use an homogeneous system

				Input (s)	Processing (s)	Output (s)	Total (s)	Execution (s)	Distribution
Physical Machines	#01	Centralized		374 ± 13%	513 ±0%	386 ±4%	1273 ±5%	544±4%	361.5 ± 20.3%
	#02	Decentralized	FIFO	416 ±2%	515 ±0%	70 ±15%	1001 ±2%	457 ±4%	361.5 ± 19.4%
	#03		IN	272 ±3%	514 ±0%	101 ±8%	887 ±1%	408 ±1%	361.5 ±19.6%
	#04		IS	258 ±2%	514 ±0%	98 ±0%	870 ±1%	413 ±1%	361.5 ±27.8%
	#05		FRIN	302 ±1%	516 ±0%	104 ±18%	922 ±2%	385 ±5%	361.5 ± 1.7%
	#06		FRIS	297 ±1%	515 ±0%	113 ±2%	925 ±1%	394 ±2%	361.5 ± 4.5%
	#07		FD	437 ±1%	516±0%	115 ±4%	1068 ±0%	454 ±1%	361.5 ± 0.1%
Docker Containers	#08	Centralized		691 ±1%	528 ±0%	484 ±3%	1703 ±1%	441 ±5%	361.5 ± 41.7%
	#09	Decentralized	FIFO	483 ±2%	534 ±0%	100 ±9%	1117 ±1%	378 ±3%	361.5 ± 30.3%
	#10		IN	249 ±6%	533 ±0%	121 ±10%	903 ±2%	304 ±5%	361.5 ± 71.2%
	#11		IS	236 ±1%	530 ±0%	113 ±1%	879 ±0%	317±2%	361.5 ± 75.8%
	#12		FRIN	351 ±2%	532 ±0%	117 ±4%	1000 ±1%	293 ±4%	361.5 ± 1.5%
	#13		FRIS	375 ±5%	533 ±0%	132 ±12%	1040 ±3%	294 ±5%	361.5 ± 1.8%
	#14		FD	484 ±1%	533 ±0%	116 ±4%	1133 ±1%	455 ±2%	361.5 ± 0.1%

5. Related works

Applications in e-Science are becoming increasingly large-scale and complex. These applications are often in the form of workflows [5] such as MONTAGE, Blast [6], CyberShake [7] with a large number of software components and modules. Workflow and scheduling policies have been studied for one decade or two. In the reminder of this section we consider two categories. First the works that seek to optimize the execution time and/or QoS constraints of the workflows running in grid environments, and second works anchored in the Map-Reduce framework.

5.1. Conventional workflow scheduling policies

In [8] authors reviewed the solutions for allocating suitable resources to workflow tasks so that the execution can be completed to satisfy objective functions specified by users. The context is Grid computing and the authors first introduced workflow management systems that define, manage and execute workflows on computing resources. Our context is more related to volunteer computing and we want to design, as a whole, the interactions of components, especially the interactions between the workflow scheduling and data movement components. Moreover, there are two types of abstract workflow model, deterministic and non-deterministic. In a deterministic model, the dependencies of tasks and I/O data are known in advance, whereas in a non-deterministic model, they are only known at run time. In our case the dependencies are known in advance but nodes publish their volunteering. We may consider them as active and not passive, making a strong distinction with other works. The heuristics recalled in [8] are based on the performance estimation for task execution and I/O data transmission. In our work, we do not make any assumption about performance estimation, apriori known. Dependency mode scheduling algorithms intends to provide a strategy to order and map workflow tasks on heterogeneous resources based on analyzing the dependencies of the entire task graph, in order to complete these interdependent tasks at earliest time. The strategy ranks the priorities of all tasks in a workflow application at one time. On issue with this strategy is to set weights on tasks. One idea is to set the weight of each task and edge to be equal to its estimation execution time and communication time. In our case we assume that the execution context is fluctuating (nodes may enter/leave the system at any time) making the estimates of weights a challenging problem. Another strategy is duplication based scheduling that uses the idling time of a resource to duplicate some parent tasks, which are also being scheduled on other resources. In our case we can duplicate tasks but for the purpose of result certification since our context is the volunteering computing. All of our previous experimental results are accomplished with no duplication. Meta-heuristics are yet another approach for solving the workflow scheduling problems. In general, there are two phases with meta-heuristics approaches and for each iteration of the process: construction phase and local search phase. The construction phase generates a feasible solution. A feasible solution for the workflow scheduling problem is required to meet the following conditions: a task must be started after all its predecessors have been completed; every task appears once and only once in the schedule. A local search is then applied into the solution to improve it. In our case we try also to optimize the execution time but as a consequence of a fair strategy to use all the resources or candidates. We also do not consider to be compliant with any deadline which is yet another objective that leads to a specific family of heuristics. At last, many studies as those in [8], [9], [10], [11] that may serve as complimentary readings, validate the strategies through simulations. In our case our option is to run real world applications (MONTAGE) which requires a special effort for designing an experimental plan, to check the reproducibility of the experiments.

5.2. Map-Reduce oriented works

In [12] authors focused on the MapReduce framework for processing massive data over computing clouds. The major factor affecting the performances of map-reduce jobs is locality constraints for reducing data transfer cost in using poor network bandwidth. They proposed a scheduling approach that provides 1) a data pre-placement strategy for improving locality and concurrency and 2) a scheduling algorithm considering locality and concurrency. They

pointed a need for scheduling workflow services composed of multiple MapReduce tasks with precedence dependency in shared cluster environments. In this paper they also introduced the data cohesion score that we reuse in our paper.

Authors in [13] noticed a conflict between fairness in scheduling and data locality (placing tasks on nodes that contain their input data). They investigated this problem in designing a fair scheduler for a 600-node Hadoop cluster at Facebook. To address the conflict between locality and fairness, they proposed a simple algorithm called delay scheduling: when the job that should be scheduled next according to fairness cannot launch a local task, it waits for a small amount of time, letting other jobs launch tasks instead. The difference with our work is as follows. First authors consider multiple jobs and they try to optimize at this granularity level. We consider only the optimization of one job. Second they give a priority to local tasks, otherwise, they skip to another job. If a job has been skipped long enough, they start allowing it to launch non-local tasks to avoid starvation.

6. Conclusion

In this paper we investigated the problems of data management and allocation under the Publish/Subscribe paradigm in order to implement robust and fast data transfers for the RedisDG workflow engine. We investigated a large set of strategies starting from the key ideas of minimizing either the number of files to transfer or the total size of files to transfer. Thanks to the DAG abstract format we inserted some intelligence, for instance for controlling the execution of the root tasks, then the tasks at any level in the DAG. We showed experimentally, on the Grid5000 testbed that coupling efficient data management strategies and efficient allocation strategies provide the best compromise for performance. The context of our experiments includes the use of a single (homogeneous) cluster, Docker containers or bare machines. In the future we plan to augment the criteria for the allocation of tasks in tacking into account parameters related to the network bandwidth but also in coupling decisions. For instance, we could imagine to decide the allocation of tasks on a data-aware criteria plus an energy-aware criteria and/or a fairness-aware criteria and/or a load-aware criteria. This paper is a step into this direction but at the moment, in this paper, we consider only data-aware criteria. The general objective remains to offer a Workflow engine as a Service for a cloud provider.

Acknowledgments

We would like to thank the Pegasus Team and John Good from the MONTAGE project. Experiments presented were carried out using the Grid'5000 testbed, supported by a scientific interest group hosted by Inria and including CNRS, RENATER and several Universities as well as other organizations. This work has received funding from the Wendelin project, selected from "Informatique en nuage, cloud computing et big-data" PIA call.

References

[1] P. T. Eugster, P. Felber, R. Guerraoui, and A.-M. Kermarrec, "The many faces of publish/subscribe," *ACM Comput. Surv.*, vol. 35, no. 2, pp. 114–131, 2003.

[2] L. Abidi, J. Dubacq, C. Cérin, and M. Jemni, "A publication-subscription interaction schema for desktop grid computing," in *Proceedings of the 28th Annual ACM Symposium on Applied Computing, SAC '13, Coimbra, Portugal, March 18-22, 2013*, S. Y. Shin and J. C. Maldonado, Eds. ACM, 2013, pp. 771–778. [Online]. Available: http://doi.acm.org/10.1145/2480362.2480510

[3] W. Saad, L. Abidi, H. Abbes, C. Cérin, and M. Jemni, "Wide area bonjourgrid as a data desktop grid: Modeling and implementation on top of redis," in *26th IEEE International Symposium on Computer Architecture and High Performance Computing, SBAC-PAD 2014, Paris, France, October 22-24, 2014*. IEEE Computer Society, 2014, pp. 286–293. [Online]. Available: http://dx.doi.org/10.1109/SBAC-PAD.2014.50

[4] H. Abbes and J.-C. Dubacq, "Analysis of Peer-to-Peer Protocols Performance for Establishing a Decentralized Desktop Grid Middleware," in *Euro-Par Workshops*, ser. Lecture Notes in Computer Science, E. César, M. Alexander, A. Streit, J. L. Träff, C. Cérin, A. Knüpfer, D. Kranzlmüller, and S. Jha, Eds., vol. 5415. Springer, 2008, p. 235âĂŞ246.

[5] S. Bharathi, A. Chervenak, E. Deelman, G. Mehta, M.-H. Su, and K. Vahi, "Characterization of scientific workflows," in *Workflows in Support of Large-Scale Science, 2008. WORKS 2008. Third Workshop on*, Nov 2008, pp. 1–10.

[6] S. F. Altschul, T. L. Madden, A. A. SchÃd'ffer, J. Zhang, Z. Zhang, W. Miller, and D. J. Lipman, "Gapped blast and psiblast: a new generation of protein database search programs," *NUCLEIC ACIDS RESEARCH*, vol. 25, no. 17, pp. 3389–3402, 1997.

[7] R. Graves, T. H. Jordan, S. Callaghan, E. Deelman, E. Field, G. Juve, C. Kesselman, P. Maechling, G. Mehta, K. Milner, D. Okaya, P. Small, and K. Vahi, "CyberShake: A Physics-Based Seismic Hazard Model for Southern California," *Pure and Applied Geophysics*, vol. 168, pp. 367–381, 2011.

[8] J. Yu, R. Buyya, and K. Ramamohanarao, *Workflow Scheduling Algorithms for Grid Computing*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 173–214. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-69277-5_7

[9] H. Topcuoglu, S. Hariri, and M.-Y. Wu, "Performance-effective and low-complexity task scheduling for heterogeneous computing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 13, no. 3, pp. 260–274, Mar 2002.

[10] M. Rahman, S. Venugopal, and R. Buyya, "A dynamic critical path algorithm for scheduling scientific workflow applications on global grids," in *e-Science and Grid Computing, IEEE International Conference on*, Dec 2007, pp. 35–42.

[11] R. Sakellariou and H. Zhao, "A hybrid heuristic for dag scheduling on heterogeneous systems," in *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International*, April 2004, pp. 111–.

[12] D. Y. K. M. Sim, "A locality enhanced scheduling method for multiple mapreduce jobs in a workflow application."

[13] M. Zaharia, D. Borthakur, J. S. Sarma, K. Elmeleegy, S. Shenker, and I. Stoica, "Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling," in *European Conference on Computer Systems, Proceedings of the 5th European conference on Computer systems, EuroSys 2010, Paris, France, April 13-16, 2010*, C. Morin and G. Muller, Eds. ACM, 2010, pp. 265–278. [Online]. Available: http://doi.acm.org/10.1145/1755913.1755940