A Dynamic Rescheduling Algorithm for Resource Management in Large Scale Dependable Distributed Systems

Alexandra Olteanu[a], Florin Pop[1a], Ciprian Dobre[a], Valentin Cristea[a]

[a]*Faculty of Automatic Control and Computers, University POLITEHNICA of Bucharest, Splaiul Independentei 313, 060042 Bucharest, Romania*

Abstract

Scheduling is the key to distributed applications performance in large scale heterogeneous environments. For such systems resilience in case of faults can be approached at the level of rescheduling mechanisms. The performance of rescheduling is very important in the context of large scale distributed systems and dynamic behavior. The paper proposes a generic rescheduling algorithm, which can be used with a wide variety of scheduling heuristics which can be selected by users in advance, depending on system's structure. The dynamic behavior of the rescheduling component is issued by a monitor, which periodically provides information about the systems' states. The rescheduling component is designed as a middleware service that aims to increase the dependability of large scale distributed systems. The monitor component, which triggers the rescheduler when an error occurs, is also a middleware component. The rescheduling service was implemented for a real Grid environment. The proposed approach supports fault tolerance and offers an improved mechanism for resource management. The evaluation of the proposed rescheduling algorithm was performed using modeling and simulation. We present results confirming its performance and capabilities.

Keywords: Monitoring, Performance evaluation, Rescheduling, Resource Management, Dependable Distributed Systems
2010 MSC: 68M20, 68M14, 68U20

---

1. Introduction

Ensuring dependability in Large Scale Distributed Systems (LSDS), is a difficult issue, due to the geographical distribution of resources and users, the volatility of resources, the QoS requirements of the applications, and constraints imposed by the resource owners. It represents a hot research subject and, despite the valuable results obtained in many projects developed so far [1], no solution was found that responds to different important requirements of dependable systems and at the same time, offers high performance by exploiting the capabilities of modern systems.

One of the important services that are responsible for the performance of Dependable LSDS (D-LSDS) is the resource allocation management. This is very complex firstly

[1]Corresponding author

because of the large distribution of resources and users, which claim frequently for remote operations and data transfers. They decrease the systems' safety and reliability and make them more vulnerable to security threats [2, 3]. Systems must ensure the correct and complete execution of applications even when resources join and leave them dynamically, or when resources become unavailable due to faults. The management of distributed systems is also complicated by the high requirements of applications and constraints imposed by resources' owners. In many cases, these requirements and constraints are conflicting between each other. For example, an application needs a high execution time and performs database operations, while the resources' owner makes them available in a restricted time interval and does not allow database operations. Another important aspect that also complicates the problem is represented by tasks dependencies [4], which add constraints on the order of their execution.

Since the scheduling problem is NP-complete in its general form, a large number of heuristics based methods have been proposed in the literature [4, 5]. We take a similar approach to build efficient plans for better use of available resources. The contribution is to add a monitoring service for detecting execution context changes (including faults and changes in resource status), and a rescheduling service for supporting fault tolerance and resilience. In addition, the proposed rescheduling algorithm can use scheduling heuristics that are different from those previously employed.

The paper is organized as follows. In Section 2 the related work regarding the techniques for re-scheduling in LSDS is presented. Section 3 describes the proposed algorithm for re-scheduling. The simulation environment is described in Section 4. Section 5 presents the test scenarios and experimental results. Our conclusions are highlighted in Section 6.

2. Related work

Rescheduling is associated in the literature with two goals: optimizing the scheduling performance, to obtain better execution times, and providing fault tolerance. The policy proposed in [6], considers rescheduling at some carefully selected points along execution. After the initial schedule is obtained, it selects a set of unfinished tasks for rescheduling, if the run time performance variation exceeds a predefined threshold. In GrADS Project [7], once launched, execution is tracked by the contract monitor, which detects anomalies and invokes, when necessary, the rescheduler to take corrective action by using one of the two provided mechanisms: simple stop/migrate/restart approach to rescheduling Grid applications and a process-swapping approach to rescheduling.

Another approach considers the allocation of jobs to resources using batch mode methods. These methods are able to provide fast planning by exploring characteristics of distributed and highly heterogeneous systems. In evaluating these methods, four parameters of the system are measured: makespan, flowtime, resource usage and matching proximity [8]. The scheduling problem can be considered in immediate mode, in which jobs are allocated as soon as they arrive in the system. This type of scheduling arises in many grid-based applications, especially, in real-time applications [9].

Most evaluations and analysis studies of various heuristics, surprisingly, showed that similar values are obtained for results quality, identifying the same strengths and weaknesses, the differences being given only by few percents [6]. By contrast, to overcome the issues risen up in a dynamic distributed environment context, this work presents a dynamic, generic and adaptive rescheduling algorithm which uses various fault tolerant mechanisms and can be associated with almost any scheduling heuristic [10].

The research in D-LSDS led to the description of fault tolerance mechanisms for preserving application execution despite the presence of a computing node fail. These mechanisms are classified into two major categories according to the level at which errors are treated:

- Mechanisms in the first category consider task level failures, for which information about the task is sufficient to redefine the status of a failed task. According to [11], from task level mechanisms we can mention: retry, alternate resource, checkpoint and task duplication. After detecting the failure, the retry approach simply considers a number of attempts to execute a failed task on the same resource. The checkpoint saves the computation state periodically, such that it migrates the saved work of failed tasks to other processors. The alternate resource mechanisms chooses other resource for executing failed tasks. The task duplication mechanism selects tasks for replication, hoping that at least one of the replicated tasks will finish successfully.

- Mechanisms in the second category treats application level fails, where much more information is necessary to redefine the entire state of an application. The category of application level mechanisms [11] contains: rescue file, redundancy, user-defined exception handling and rewinding mechanisms. The rescue file mechanism consists in the resubmission of uncompleted portions of a DAG (Directed Acyclic Graph) when one or more tasks resulted in failure. The user-defined exception handling allows users to give a special treatment to a specific type of failures. The rewinding mechanism seeks to preserve the execution of the application.

An important issue is related to schedule categories [4][12][13]: scheduling independent tasks aims to increase the total system performance, and scheduling tasks with dependencies aims to reduce the execution time without violating the tasks precedence constraints. Scheduling algorithms use two different insertion policies: 1) the insertion based policy looks for the insertion of a task in the earliest idle time slot, between two scheduled tasks on a processor, and 2) the non-insertion based policy considers the possible insertion of a task only after the last scheduled task on a processor. Well-known scheduling algorithms, that are cited and used as reference for new algorithms in a large number of papers, are described below.

MCP (Modified Critical Path) algorithm is based on lists with two phases: the priority and selection of resources [14]. *CCF* (Cluster ready Children First) is a dynamic scheduling algorithm based on lists. In this algorithm the graph is visited in topological order, and tasks are submitted as soon as scheduling decisions are taken. *ETF* (Earliest Time First) algorithm is based on keeping the processors as busy as possible. It computes, at each step, the earliest start times of all ready nodes and selects the one having the smallest start time [4]. *HLFET* (Highest Level First with Estimated Times) uses a hybrid of the list-based and level-based strategy. The algorithm schedules a task to a processor that allows the earliest start time [15]. *Hybrid Remapper PS* (Hybrid Remapper Minimum Partial Completion Time Static Priority) is a dynamic list scheduling algorithm specifically designed for heterogeneous environments [4] [16].

Zomaya et al. investigates in [17] the effectiveness of rescheduling using cloud resources to increase the reliability of job completion. Specifically, schedules are initially generated using grid resources, and cloud resources (relatively costlier) are used only for rescheduling to cope with a delay in job completion. Therasa et al. address the fault tolerance in terms of resource failure using periodic checkpointing, which periodically saves the jobs state. An inappropriate checkpointing interval leads to delay in the job execution, and reduces the throughput, and they proposed a strategy to achieve fault tolerance by dynamically adapting the checkpoints based on current status and history of failure information of the resource, which is maintained in the information server [18].

According to all presented scheduling methods the resource management service is responsible to ensure the tasks execution, the resource availability and allocation offering a fault tolerant environment in LSDDS. We present in the next section a method for rescheduling and recovering from errors using the presented algorithms. The dynamic aspect of this method is sustained by using the real time monitoring information and by dynamic change of fault tolerant mechanisms in a error case.

3. Rescheduling for LSDDS

3.1. Preliminaries

In this paper we use DAG (Directed Acyclic Graph) for modeling the workflow, where nodes represent computation and edges represent communication, data flow, between nodes. A task graph is represented as $G(V, E, c, \tau)$ where:

- V is a set of nodes (tasks). We will refers to the nodes using $n_1, n_2, \ldots$ notation;
- E is a set of directed edge (dependencies), noted as $e(n_i, n_j)$;
- $w : V \to R_+$ is a function that associates a weight $w(n_i)$ to each node $n_i \in V$; $w(n_i)$ represent the execution time of the task T_i, which is represented by the node n_i in V;
- ew_n is a function $ew_n : E \to R_+$ that associates a weight to a directed edge; if n_i and n_j are two nodes in V then $ew_n(n_i, n_j)$ denotes the inter-tasks communication time between T_i and T_j (the time needed for data transmission betwwen processors that execute tasks T_i and T_j). When two nodes are scheduled on the same processing element P, the cost of the connecting edge becomes zero.

The graph also has two virtual nodes with cost zero: the start node is the starting point of the program and the end node represents the end of the program. The graph nodes must be assigned to the available resources, and communication that is represented by graph edges must appear in the network between resources to which the nodes are assigned. Under this assumption, the scheduler must determine both what resources are assigned to each graph node and which is the execution order if more nodes are assigned to the same resources.

In addition, we defined an **inner node** as a DAG node for the execution of which depends other nodes and an **leaf node** as a node for the execution of which no other node depends on. A sub graph is a part of application DAG formed by the current node and all the others nodes that depends on its' execution.

A task graph example is shown in Figure 1. Two items are associated with each node: the task id T_i is represented in the upper half of the node n_i (the circle), while the execution time $w(n_i)$ is represented in the lower half. Each edge is labeled with the inter-tasks communication time, $ew_n(n_i, n_j)$.

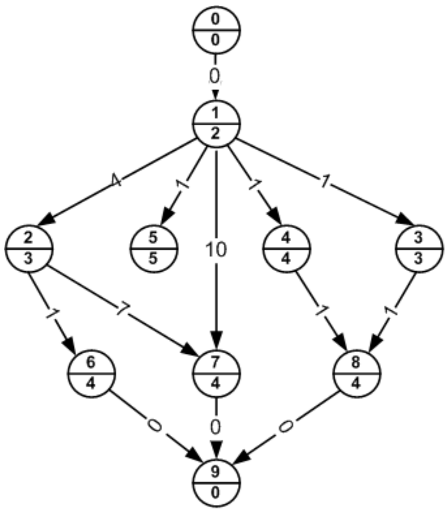

Figure 1: Task DAG Example

If we denote $st(n_i)$ the start time and $ft(n_i)$ the finish time for task n_i, and define $makespan = max_{n_i \in V} \{ft(n_i)\}$ we can formulate the goal of optimizing DAG scheduling as follows: minimize the *makespan*. The main problem raised by the workflow consists of submitting the scheduled tasks to Grid resources without violating the structure of the original workflow [19].

A scheduler is considered efficient if the *makespan* is short and respects resource constrains, such as a limited number of processors, memory capacity, available disk space, etc. Many types of scheduling algorithms for DAG are based on the list scheduling technique. Each task has an assigned priority, and scheduling is done according to a list priority policy: select the node with the highest priority and assign it to a suitable machine. According to this policy, two attributes are used for assigning priorities:

- *tlevel* (top-level) for a node n_i is the weight of the longest path from the source node to n_i.
- *blevel* (bottom-level) for a node n_i is the weight of the longest path from n_i to exit node.

The time-complexity for computing *tlevel* and *blevel* is $O(|V| + |E|)$, so there are no penalties for scheduling algorithms.

We can define the *ALAP* (*As Late As Possible*) attribute for a node n_i to measure how far the node's start-time, $st(n_i)$ can be delayed without increasing the *makespan*. This attribute will have an important role for load balancing constrains because it show if we can delay the execution start of a task T_i. The critical path (CP) is the weight of the longest path in the DAG and offers an upper limit for the scheduling cost. Algorithms based on CP heuristics produce the best results on average. They take into consideration the critical path of the scheduled nodes at each step. However, these heuristics can result in a local optimum, failing to reach the optimal global solution [20].

The DAG scheduling problem is an NP-complete problem [21]. A solution for this problem consists of a series of heuristics [22], where tasks are assigned priorities and placed in a list ordered by priority. The method through which the tasks are selected to be planned at each step takes into consideration this criterion, thus the task with higher priority receives access to resources before those with a lower priority. The heuristics used vary according to task requirements, structure and complexity of the DAG.

Most scheduling algorithms are based on list scheduling technique. The basic idea of list scheduling is to make a scheduling list (a sequence of nodes for scheduling) by assigning them some priorities, and then repeatedly execute the following two steps until all the nodes in the graph are scheduled: 1. *Remove the first node from the scheduling list*; 2. *Allocate the node to a processor which allows the earliest start-time.*

If the communication between tasks is considered, there are three models of communication delay:

- *intra-task-communication*: communication delays are hidden in the topology of the multiprocessor tasks,
- *inter-task-communication*: communication delays occur if dependend uni-proccesor tasks are not processed by the same processor of machine,
- *combination of both*: in the case of divisible task scheduling a multiprocessor task can be partitioned into smaller tasks, one partial task is processed by the current processor and the other parts are distributed among the grid processors, phases of inter-task-communication and computation (with intra-task-communication) alternate with each other.

3.2. The Re-Scheduling Algorithm

To easily include a rescheduling service in a system, we have designed a rescheduling algorithm that can be used in combination with a wide variety of scheduling algorithms which can be selected in advance depending on the system structure. The scheduling algorithm could be preconfigured by the system administrator considering factors like the number and characteristics of available resources, and the structure and attributes of the graph task that we want to schedule, to achieve optimal results.

We consider the following Grid scheduling problem: a DAG of task is submitted to a cluster in Grid and the cluster manager is responsible for task execution. The schedule decision is taken by a specific component implemented on the top of cluster manager. The optimization criteria are based on makespan and it is oriented to the applications. The error indication mechanism is based on periodic interrogation of task status in the cluster.

Our rescheduling algorithm contains a series of characteristics so it can take the most appropriate mapping decisions, to what resources we send the rescheduled tasks, when rescheduling is needed for ensuring dependability. First of all it may use different scheduling algorithms for better use of available resources. Other feature is represented by the different treatment provided by the algorithm for each DAG node type. Furthermore, for each of these types a series of fault tolerance mechanisms for preserving application execution despite the presence of a processor fail can be defined and used.

Node type	**Rescheduling strategies**
1. Leaf node	a) task is rescheduled on the same processor / resource b) task is rescheduled on another processor / resource than the one where the error occurred
2. Inner node	a) task is rescheduled on the same processor / resource without the re-scheduling of all nodes to which exist a way from the node where the error occured. b) task is rescheduled on another processor / resource than the one on which the error occurred without the re-scheduling of all nodes to which exist a way from the node where the error occured. c) it is rescheduled the whole sub graph, containing the nodes to which there is a way from the node where the error occured, with the same algorithm used for scheduling. d) it is rescheduled the whole sub graph, containing the nodes to which there is a way from the node where the error occured, with different algorithm than the one used for scheduling.

Table 1: Nodes types and re-scheduling strategies

When a task fails, all the other tasks that depend on its' correct execution must be rescheduled, too. We must analyze if the task where the error occurred can be rescheduled alone or with all the others tasks that depend on its' execution. If the current task is an leaf node it is rescheduled alone. If the task is an inner node it can be rescheduled alone or a dependency sub-graph having as root the current task is rescheduled. Based on this assumptions there are two types of input data in terms of the tasks set: one task and a tasks' sub-graph. This two situations are treated differently like we also underlined before.

Considering the rescheduling need for a set of tasks, the classification of nodes are described in Table 1. The classification is made by nodes position within a DAG and the rescheduling strategies that can be used for each case and considered by the developed algorithm and our implementation.

In determining the rescheduling strategies, we have considered some of the fault tolerance mechanisms previously presented. For what case they were used is presented in Table 2.

Re-scheduling strategies (Tab. 1)	**Fault tolerance mechanisms**
1.a), 2.a)	Retry
1.b), 2.b)	Alternate resource
2.c), 2.d)	Rescue file
2.c), 2.d)	User-defined Exception Handling

Table 2: Fault tolerance mechanisms used by the described re-scheduling strategies

The algorithm also needs to have as input data the scheduling algorithm that should be used for rescheduling and the system set of the available resources. After rescheduling, the tasks' execution is reordered and new associations (task, CPU) are built and submitted for execution.

According with presented issues for fault tolerance we proposed a generic, adaptive and dynamic rescheduling algorithm that aims to offer adaptability to dynamics changes during the application execution giving the possibility of changing the scheduling algorithm used for rescheduling and also permitting a pre-configuration of the rescheduling strategy. Moreover we recommended the selection of different strategies, that employ fault tolerant mechanisms as presented in tables 1 and 2, by considering the type of node where the error occurred. The algorithm description is presented below:

```
H0 = scheduling heuristic,
HR = heuristic used for rescheduling,
H_current = used heuristic,
S1 = rescheduling strategy for inner nodes,
S2 = rescheduling strategy for edge nodes,
P_current = current scheduling for unfinished nodes,
P = previous scheduling,
A = application DAG,
A_current = unfinished DAG tasks,
ERRN = the node where the error had occurred,
AUX1 = is true if just the current node is rescheduled,
AUX2 = is true if we reschedule the current node on another resource

1. H_current = H0;
2. A_current = A;
3. P = null;
4. R = get_available_resources();
5. P_current = schedule(H_current, A_current, R);
6. if(P.task_asoc_to_res <> P_current.task_asoc_to_res)
7.   P = P_current;
8.   submit(A_current, P_current);
9. if(DAG unfinished)
10.   if(error detected)
11.     if (ERRN = inner node)
11.        HR = select_rescheduling_heuristic(S1, AUX1);
12.        if(AUX1 = true)
13.          goto 19;
14.       else
15.          H_current = HR;
16.          A_current = subgraph_cons(A, ERRN);
17.          goto 4;
18.     if (ERRN = edge node)
19.        HR = select_rescheduling_heuristic(S2, AUX2);
20.        A_current = ERRN;
21.        if(AUX2 = true)
```

```
22.           H_current = HR;
23.           R = get_available_resources()\P_current.get_res_for(ERRN);
22.           goto 5;
23.         else
24.           goto 8;
25.   goto 9;
```

The selection of rescheduling heuristic procedure calls a scheduling algorithm (like HLFET, CCF, MCP, ETF, HybridRemapper), pointed by the used heuristic, to schedule the entire graph or just a section of it. The proper scheduling heuristic is used after we analyze what rescheduling strategies are configured for the current node type. In this approach, when the tasks are created, a monitoring service is switched on to monitor their progress. Monitoring a LSDS environment is special because tasks completation cannot be guaranteed, due to unforeseen factors such resource failure or interruptions by higher priority tasks. Consequently, the tasks progress must be monitored so that the scheduler can act dynamically to mask the occurrence of unforeseen events that can result in faults. The actions that can be taken are: rescheduling, migration, forced tasks completation and resource renegotiation.

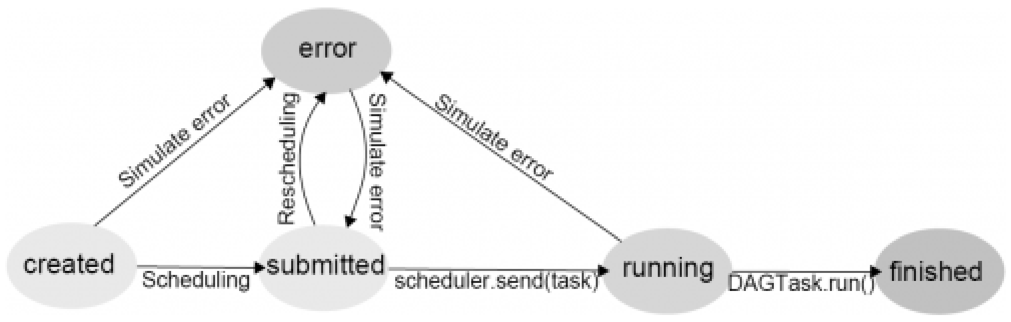

Figure 2: States of tasks and the actions of passing from one state to another

Figure 2 presents how a task goes from one state to another and the actions required to change state. When tasks are for the first time introduced in the system it is called that they are in the *created* state. Next, the new created tasks are sent for scheduling. After all tasks have been mapped to a processor they pass into the *submitted* state. From their creation to their completion the tasks may reach the *error* state. For our tests this happen because of the error simulation component which is called whenever a task is executed and considers randomly which task to be sent in the *error* state. Once a task is switched to the *error* state it can get out of this state only by using the rescheduling component which send the task to the *submitted* state. Once the tasks entered the submission state they are sent to run (*running* state). Then, after the task execution is complete, the task switch to *finished* state. This is the end of task's execution.

3.3. Analyzed Scheduling Algorithms for Re-Scheduling Model

HLFET (Highest Level First with Estimated Times) - uses a hybrid of the list-based and level-based strategy. The algorithm schedules a task to a processor that allows the earliest start time [15] (see 1).

Algorithm 1 Highest Level First with Estimated Times (HLFET) algorithm

```
 1: for each node do
 2:     Calculate the static blevel
 3: end for
 4: Make a ready list in a descending order of static blevel. Initially, the ready list
    contains only the entry nodes. Ties are broken randomly.
 5: repeat
 6:     Schedule the first node in the ready list to a processor that allows the earliest
        execution, using the noninsertion approach.
 7:     Update the ready list by inserting the nodes that are now ready.
 8: until all nodes are scheduled
```

CCF (Cluster ready Children First) dynamic scheduling algorithm based on lists. The graph is visited in topological order, and tasks are submitted as soon as scheduling decisions are taken. The algorithm assumes that when a task is submitted for execution it is inserted into the `RUNNING-QUEUE`. If a task is extracted from the `RUNNING-QUEUE`, all its successors are inserted into the `CHILDREN-QUEUE`. The running ends when the two queues are empty.

MCP (Modified Critical Path) - algorithm based on lists with two phases: the prioritization and selection of resources. Parameter used to prioritize nodes is ALAP (As Late As Possible) [14] (see 2).

Algorithm 2 Modified Critical Path (MCP) algorithm

```
 1: for each node do
 2:     Compute the $ALAP$ time
 3: end for
 4: for each node do
 5:     Create a list which consists of the ALAP times of the node itself and all its children
        in a descending order
 6: end for
 7: Sort these lists in an ascending lexicographical order.
 8: Create a node list according to this order.
 9: repeat
10:     Schedule the first node in the node list to a processor that allows the earliest
        execution, using the insertion approach.
11:     Remove the node from the node list.
12: until the node list is empty
```

ETF (Earliest Time First) - algorithm based on keeping the processors as busy as possible. It computes, at each step, the earliest start times of all ready nodes and selects the one with the shortest start time [4] (see 3).

Hybrid Remapper PS (Hybrid Remapper Minimum Partial Completion Time Static Priority) is a dynamic list scheduling algorithm specifically designed for heterogeneous environments. The set of tasks is partitioned into blocks so that tasks in a block do not have any data dependencies among them. Subsequently the blocks are

Algorithm 3 Earliest Time First (ETF) algorithm

```
1: for each node do
2:     Calculate the static blevel
3: end for
4: Initially, the pool of ready nodes includes only the entry nodes.
5: repeat
6:     for each node in the ready pool do
7:         Calculate the earliest start-time on each processor.
8:     end for
9:     Pick the node-processor pair that gives the earliest time using the non-insertion
       approach.
10:    Ties are broken by selecting the node with a higher static blevel.
11:    Schedule the node to the corresponding processor.
12:    Add the newly ready nodes to the ready node pool.
13: until all nodes are scheduled
```

executed one by one [4] [16].

4. Simulation Solution

Because of the complexity of the LSDS, involving many resources and many jobs being concurrently executed in heterogeneous environments, there are not many simulation tools to address the general problem of distributed computing. The simulation instruments tend to narrow the range of simulation scenarios to specific subjects, such as scheduling or data replication. The simulation model provided by MONARC is more generic that others, as demonstrated in [23]. It is able to describe various actual distributed system technologies, and provides the mechanisms to describe concurrent network traffic, to evaluate different strategies in data replication, and to analyze task scheduling procedures.

MONARC is built based on a process oriented approach for discrete event simulation, which is well suited to describe concurrent running programs, network traffic as well as all the stochastic arrival patterns, specific for such type of simulations [24]. Threaded objects or *Active Objects* (having an execution thread, program counter, stack) allow a natural way to map the specific behavior of distributed data processing into the simulation program [25]. Furthermore, of the strengths of MONARC is that it can be easily extended, even by users, and this is made possible by its layered structure. The first two layers contain the core of the simulator and models for the basic components of a distributed system, and they are fixed parts on top of which some particular components can be built.

In order to test a new algorithm the user should present it to the job scheduler. MONARC provide a feature that allow an user to only supply a corresponding DAG scheduling algorithm to be used by specifying the name of a class implementing a particular interface. In addition, for this case a DAGJob is used to carry enough information about the corresponding input DAG topology as required by a DAG scheduling algorithm. This type of job executes three actions: it first waits for data from all inner tasks, it then processes data according to the specified weights in the DAG, and it finally sends data to all corresponding outer tasks.

The means to achieve resiliance are fault prevention, fault tolerance, fault removal and fault forecasting [26]. Among these fault tolerance is the most difficult to achieve because of the complexity of the distributed systems. We consider that the simulation model of MONARC is adequate to be augmented as presented in this paper for testing various methods, techniques and technologies related to the different phases of fault tolerance achievement in distributed systems: error and fault detection, recovery and fault masking (redundancy).

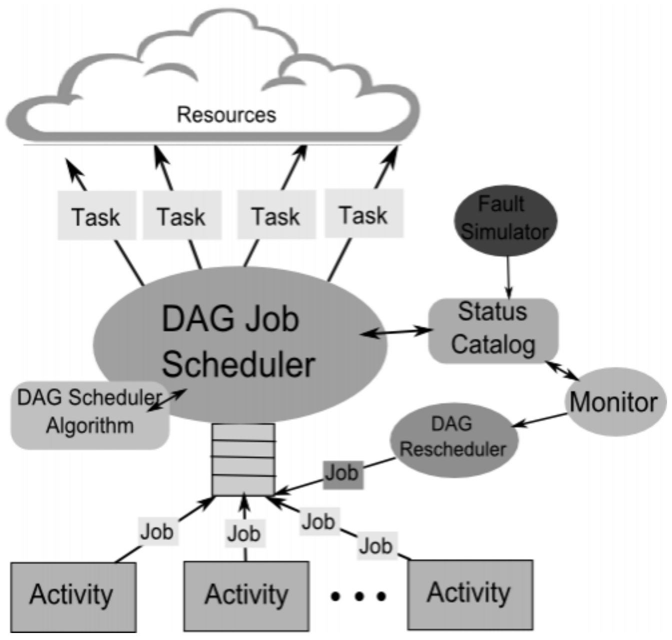

Figure 3: MONARC Simulator architecture and its extensions

For modeling the rescheduling service for a DAG in case of errors occurrence, the MONARC simulator has been extended with components for error generation, catalog of states, monitoring and rescheduling, which are presented in Figure 3. Due to the fact that the simulator's default behavior does not take into account the situations when one or more tasks fail it did not provide an error simulation mechanism. Therefore, we extended the simulator's default behavior to simulate the errors' appearance by implementing a catalog that contains the tasks' states and a method for error generation. The error generation is accomplished by setting a task state to error state. This is done considering the maximum percentage or number of errors and delay between the errors' occurrence. Furthermore, if the simulator is configured to generate errors, a monitoring component, that periodically analyzes the catalog of states, calls the rescheduling component when it finds a task that is in error state. The rescheduling component is described by the previous presented algorithm. It decides to act based on the information provided by the task where the error occurred by analyzing and deciding what rescheduling strategy to use for each case. After having determined what rescheduling strategy should be used two situations can be distinguished: reschedule a single node or build a sub-graph which is sent to the scheduler as a job so that all component tasks will be rescheduled. Rescheduling and monitoring are triggered when the job is submitted for the first time in the system. In this way, the system provides for every created task a checkpoint immediately after its creation.

Rescheduling component implemented for the MONARC simulator is called by the monitor in case of error detection. It receives the node where the error occurred and, after deciding if the node is an inner node or an leaf node, calls the rescheduling strategies which are predefined for each case. Choosing an appropriate strategy is very important because it can influence the rescheduling performance. Based on the classification described in Table 1 the rescheduling algorithm, previously presented, was implemented for the MONARC simulator:

- the monitor implements the iterations, if needed, and check if nodes are in the error state
- the scheduling procedure chooses the rescheduling strategy, considering the node type, which can be inner or leaf node, and sends for rescheduling the node or the corresponding sub-graph.

Due to simulator's openness and modularity we were able to develop simulation scenarios using user provided DAG scheduling algorithms to assigns tasks for execution on existing simulated resources and to test the rescheduling algorithm with them. The tests scenarios and experimental results are presented in the next section.

5. Tests Scenarios and Experimental Results

5.1. System setup

The basic requirements for rescheduling algorithm simulating using MONARC simulator are: a configuration file which specifies the components of the distributed system we want to model, a set of tasks that will be executed in the virtual distributed system, a scheduling algorithm for tasks with dependencies which assigns to each task a processing unit (CPU), one or more activities set by user to send to the distributed system tasks for scheduling and execution, a scheduling algorithm for tasks with dependencies which would be used for rescheduling, which generally it is the same with the one stated previously, and the rescheduling strategies, defined in the previous section, for both inner and leaf nodes.

The purpose of the simulation is to check the quality and the reliability of the rescheduling algorithm. The rescheduling algorithm evaluation was made using two test scenarios. The characteristics of the system configuration and of the tasks sets used for these test scenarios are:

- Set 1: synthetic test case used for analyzing how the scheduling algorithms succeed to provide a good load balancing and for classifying these algorithms in the case of rescheduling. Were used 3 processors with similar characteristics, aspect that don't present interest in this analysis, and set of 14 tasks for scheduling and executing;

- Set 2: realistic test case that aims to provide a complex evaluation of the rescheduling algorithm. For this case were used 50 processors with similar characteristics and sets of 100/200/300/400/500 tasks which were generated with Stencil parallel algorithm pattern and different CCR (communication to computation ratio) values, but with random costs for communication and computation.

The CCR value describe the importance of communication in a task graph, which strongly determines the scheduling behavior. Based on CCR we classify task graphs in:

- $CCR < 1$ - coarse grained graph
- $CCR = 1$ - mixed
- $CCR > 1$ - fine grained graph

For costs analysis CCR parameter can ve defined as:

$$CCR = \frac{\sum\limits_{n_i, n_j} ew_n(n_i, n_j) * |V|}{\sum\limits_{n_i} w(n_i) * |E|}$$

More details about DAGs' generation, error simulation and system characteristics will be given next along with results presentation and interpretation.

5.2. Evaluation results

In [4] and [25] we analyzed the performance of Grid DAG scheduling algorithms. We consider that there are a number of factors that determine which algorithm is more suitable for a particular application. For example the schedule length when the application needs to execute as fast as possible, and the load balancing schedule with idle times reduced as much as possible. Taking too much time for the scheduling process is not always recommended especially in time critical applications. However, if there is a chance to improve the resulted schedule then there should be a compromise in order to run a more complex and time consuming algorithm to obtain better results. In prvious experiments, CCF algorithm offers the best load balancing and the minimum total time. ETF has the same total schedule time as HLFET for all considered tests.

For the proposed re-scheduling algorithm the first configurations set are used for evaluation of the rescheduling algorithm performances with different scheduling algorithms considering the average time of the CPU usage metric for the scheduled tasks. To make a pertinent comparison of how each algorithm behaves in case of rescheduling, for these tests we have forced crashes on the same graph nodes. This represent more a synthetic study case that is used for partial determining which scheduling algorithm, from our set of implemented algorithms, seems to give the best results in order to be used for larger tests.

The first set of tests are introducing the $Tasks * Time$ parameter (the area under the graphics). The $Tasks * Time$ represents an average time of the CPU usage for the scheduled tasks metric. When errors don't occur in the system we can observe that the CCF has the smallest value for this parameter (see Figure 4), followed by MCP (see

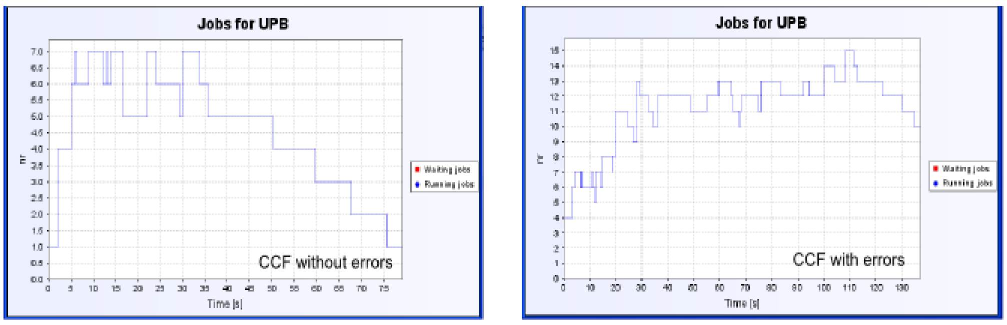

Figure 4: The obtained results in case of the number of concurrent tasks in the system for the scheduling CCF algorithm

Figure 6). This means that CCF, followed by MCP, has the best resource utilization score. For a better classification this parameter is analyzed along with how the resources load-balancing is made. We also observed that the CCF algorithm offers the best resources load-balancing and this mean that it provides a better workload distribution along resources.

When errors occur is observed that CCF still has the smallest value for the parameter previous described. Note also the fact that for CCF the graphic is balanced, there are no sudden variations, indicating a good resources utilization and load-balancing.

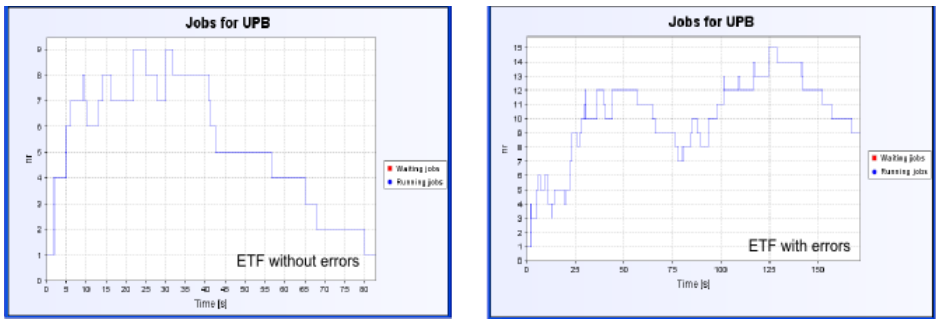

Figure 5: The obtained results in case of the number of concurrent tasks in the system for the scheduling ETF algorithm

Another important observation that is confirmed by the figures 4 to 8 is that insertion approach is better than the non-insertion approach. For example we can analyze the MCP scheduling algorithm in comparison with ETF algorithm. Because MCP is trying to fill the available gaps by scheduling the task in these gaps, provides a better load-balancing and a better execution time.

The lowest CPU utilization was obtained for the CCF scheduling algorithm, for both cases with and without errors occurrence, which is explained by the higher number of transfers between different processors involved in the simulation experiment. Furthermore CCF offers a better load balancing. An appropriate execution time for scheduled tasks was obtained for MCP scheduling algorithm. Another interesting observation is

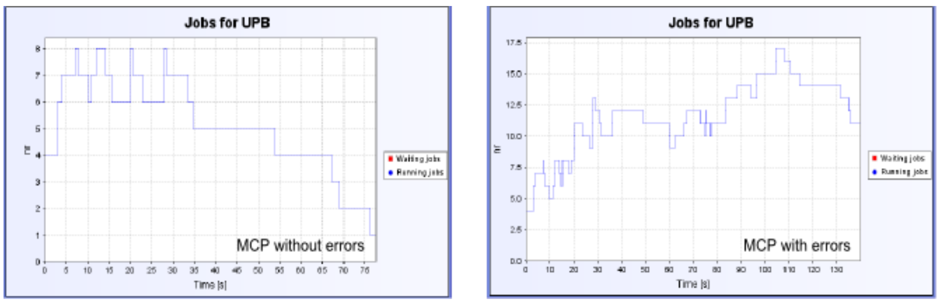

Figure 6: The obtained results in case of the number of concurrent tasks in the system for the scheduling MCP algorithm

about ETF algorithm which, in spite of its strategy to keep the CPU as occupied as possible, the results puts it on the last place.

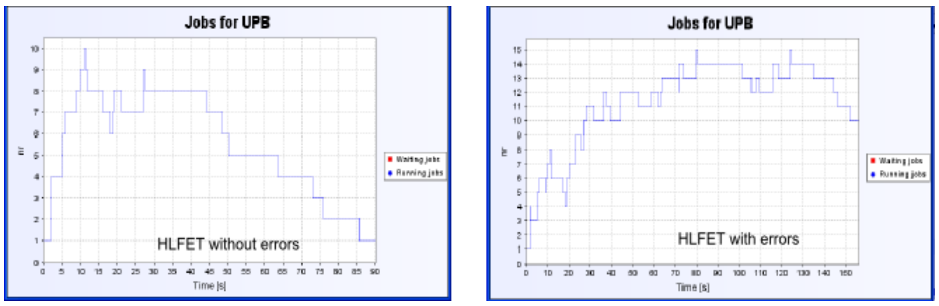

Figure 7: The obtained results in case of the number of concurrent tasks in the system for the scheduling HLFET algorithm

For the second set of tests we were more interested in evaluating the finalization times and the total number of rescheduled nodes, considering that when an error occur at an inner node an entire sub-graph is rescheduled. The errors number that can appear at some time moment is limited at 1% of the total nodes number. The probability distribution of the error states is a normal one.

The second set of tests is used to test the performance of rescheduling algorithm in case of larger graphs. For this, error simulation was made limiting the number of errors at some time to 1% of the number of nodes. This percent was choosed after analyzing the logs offered by MONALISA [27]. We observed in our experiments that the percentage of error has not exceeded this threshold, and due to this we consider it to be the upper limit.

Lets consider t_{rs} *total time for error occurance* (with rescheduling) and t_s *total time without errors*. We consider different sets of tasks with 10/100/200/300/400 tasks per set. The CCR parameter for this sets is considered between 0.49 (high computational tasks with dependencies) and 3.48 (data intensive tasks with dependencies).

We consider the *HLFET* (Highest Level First with Estimated Times) algorithm because it uses a hybrid of the list-based and level-based strategy, according with different types of DAGs. The error occurrence is simulated at the same nodes for each one of the

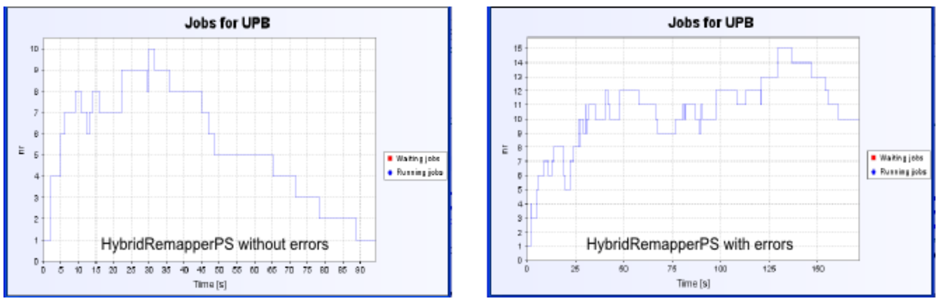

Figure 8: The obtained results in case of the number of concurrent tasks in the system for the scheduling HybridRemapperPS algorithm.

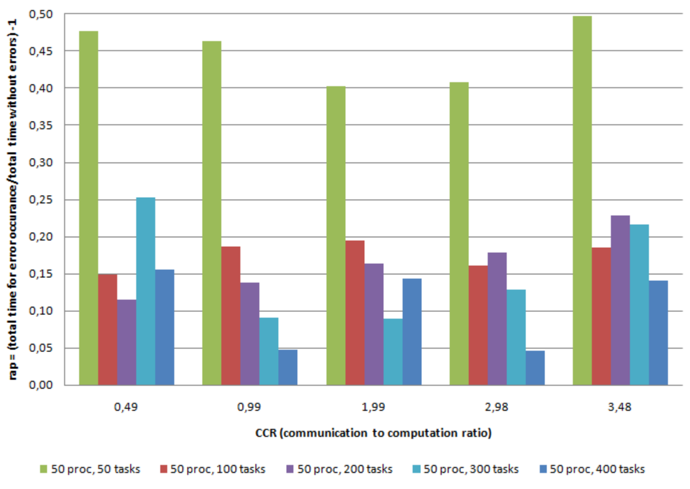

Figure 9: Error occurance at the same task (Random costs for communication and computation, Scheduling Algorithm: HLFET).

tests set. We define:

$$rap = \frac{t_{rs}}{t_s} - 1$$

as percentage of increasing time considering errors. Figure 9 shows the results for the considered experimental sets on 50 processors.

The Figure 9 highlights the performance of our proposed solution for rescheduling. The *rap* parameter decreases when the number of tasks increase. This is a normal behavior because the number of errors is lower in comparison with number of tasks and the rescheduling phase was designed for performance improvement. Another important result shows that the variation or *rap* parameter is lower for the same number of tasks with different structure (different values for CCR). This explain that our rescheduling method performs for all types of dependencies.

6. Conclusions

We proposed in the paper a new rescheduling algorithm for LSDS. The algorithm has an important feature: is generic, because it can be used with a large variety of rescheduling heuristics. It may also use more re-scheduling strategies whose classification was elaborated according to the node position in graphic and depending on the fault tolerance mechanisms that can be used. Choosing a good scheduling approach is also an important issue for the performance of an application launched onto a distributed systems environment.

The proposed algorithm for rescheduling is a solution to improve the resource management for dependable distributed systems. We used a simulation environment, based on MONARC, to evaluate the performance of the algorithm with different scenarios. MONARC is able to model failures in distributed systems according with realistic experience. We analyzed the existing scheduling algorithms, studied and compared, but there are few studies comparing the performance of scheduling algorithms. These studies considering at the same time the distributed system structure on which we want to schedule the tasks, the type of directed acyclic graph (DAG), in which graph nodes represent tasks and graph edges represent data transfers, and the type of tasks, for example CPU-bound vs. I/O bound.

Choosing the best known scheduling algorithm can improve performance of an application if all the aspects previous enumerated are considered. Our future work is aimed to propose a method for choosing, in a dynamic manner, the most appropriate scheduling algorithm for a particular distributed system, which is analyzed.

Acknowledgments

The research presented in this paper is supported by romanian projects: "*DEPSYS - Models and Techniques for ensuring reliability, safety, availability and security of Large Scale Distributes Systems*" (Contract No. 618/15.09.2009, Project CNCSIS-PN-IDEI ID: 1710) and "*SORMSYS - Resource Management Optimization in Self-Organizing Large Scale Distributes Systems*" (Contract No. 5/28.07.2010, Project CNCSIS-PN-II-RU-PD ID: 201). The work has been co-funded by the Sectorial Operational Program Human Resources Development 2007-2013 of the Romanian Ministry of Labor, Family and Social Protection through the Financial Agreement "*POSDRU/89/1.5/S/62557*".

References

[1] A. Avizienis, J.-C. Laprie, B. Randell, C. Landwehr, Basic Concepts and Taxonomy of Dependable and Secure Computing, IEEE Trans. Dependable Secur. Comput. 1 (1) (2004) 11–33, ISSN 1545-5971, doi:http://dx.doi.org/10.1109/TDSC.2004.2.

[2] K. Christodoulopoulos, V. Sourlas, I. Mpakolas, E. Varvarigos, A comparison of centralized and distributed meta-scheduling architectures for computation and communication tasks in Grid networks, Comput. Commun. 32 (7-10) (2009) 1172–1184, ISSN 0140-3664, doi: http://dx.doi.org/10.1016/j.comcom.2009.03.004.

[3] F. Xhafa, A. Abraham, Computational models and heuristic methods for Grid scheduling problems, Future Gener. Comput. Syst. 26 (4) (2010) 608–621, ISSN 0167-739X.

[4] F. Pop, C. Dobre, V. Cristea, Performance Analysis of Grid DAG Scheduling Algorithms using MONARC Simulation Tool, in: ISPDC'08: Proceedings of the 2008 International Symposium on Parallel and Distributed Computing, IEEE Computer Society, Washington, DC, USA, ISBN 978-0-7695-3472-5, 131–138, 2008.

[5] F. Xhafa, A. Ajith, A compendium of heuristic methods for scheduling in computational grids, in: IDEAL'09: Proceedings of the 10th international conference on Intelligent data engineering and automated learning, Springer-Verlag, Berlin, Heidelberg, ISBN 3-642-04393-3, 978-3-642-04393-2, 751–758, 2009.

[6] R. Sakellariou, H. Zhao, A low-cost rescheduling policy for efficient mapping of workflows on grid systems, Sci. Program. 12 (4) (2004) 253–262, ISSN 1058-9244.

[7] F. Berman, H. Casanova, A. Chien, K. Cooper, H. Dail, A. Dasgupta, W. Deng, J. Dongarra, L. Johnsson, K. Kennedy, C. Koelbel, B. Liu, X. Liu, A. Mandal, G. Marin, M. Mazina, J. Mellor-Crummey, C. Mendes, A. Olugbile, J. M. Patel, D. Reed, Z. Shi, O. Sievert, H. Xia, A. YarKhan, New grid scheduling and rescheduling methods in the GrADS project, Int. J. Parallel Program. 33 (2) (2005) 209–229, ISSN 0885-7458, doi:http://dx.doi.org/10.1007/s10766-005-3584-4.

[8] F. Xhafa, L. Barolli, A. Durresi, Batch mode scheduling in grid systems, Int. J. Web Grid Serv. 3 (1) (2007) 19–37, ISSN 1741-1106.

[9] F. Xhafa, J. Carretero, L. Barolli, A. Durresi, Immediate mode scheduling in grid systems, Int. J. Web Grid Serv. 3 (2) (2007) 219–236, ISSN 1741-1106.

[10] A. Benoit, M. Hakem, Y. Robert, Contention awareness and fault-tolerant scheduling for precedence constrained tasks in heterogeneous systems, Parallel Comput. 35 (2) (2009) 83–108, ISSN 0167-8191, doi:http://dx.doi.org/10.1016/j.parco.2008.11.001.

[11] I. Hernandez, M. Cole, Reactive grid scheduling of DAG applications, in: PDCN'07: Proceedings of the 25th conference on Proceedings of the 25th IASTED International Multi-Conference, ACTA Press, Anaheim, CA, USA, 92–97, 2007.

[12] I. Hernandez, M. Cole, Reliable DAG scheduling on grids with rewinding and migration, in: GridNets '07: Proceedings of the first international conference on Networks for grid applications, ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), ICST, Brussels, Belgium, Belgium, ISBN 978-963-9799-02-8, 1–8, 2007.

[13] M. A. Bender, C. A. Phillips, Scheduling DAGs on asynchronous processors, in: SPAA '07: Proceedings of the nineteenth annual ACM symposium on Parallel algorithms and architectures, ACM, New York, NY, USA, ISBN 978-1-59593-667-7, 35–45, 2007.

[14] S. Bansal, P. Kumar, K. Singh, An improved two-step algorithm for task and data parallel scheduling in distributed memory machines, Parallel Comput. 32 (10) (2006) 759–774, ISSN 0167-8191.

[15] S. Jin, G. Schiavone, D. Turgut, A performance study of multiprocessor task scheduling algorithms, J. Supercomput. 43 (1) (2008) 77–97, ISSN 0920-8542, doi:http://dx.doi.org/10.1007/s11227-007-0139-z.

[16] A. Gandhi, H. Akkary, R. Rajwar, S. T. Srinivasan, K. Lai, Scalable Load and Store Processing in Latency Tolerant Processors, in: ISCA '05: Proceedings of the 32nd annual international symposium on Computer Architecture, IEEE Computer Society, Washington, DC, USA, ISBN 0-7695-2270-X, 446–457, 2005.

[17] Y. C. Lee, A. Y. Zomaya, Rescheduling for reliable job completion with the support of clouds, Future Gener. Comput. Syst. 26 (2010) 1192–1199, ISSN 0167-739X.

[18] A. L. Therasa.S, Sumathi.G, A. Dalya.S, Article: Dynamic Adaptation of Checkpoints and Rescheduling in Grid Computing, International Journal of Computer Applications 2 (3) (2010) 95–99, published By Foundation of Computer Science.

[19] M. Maheswaran, S. Ali, H. J. Siegel, D. Hensgen, R. F. Freund, Dynamic mapping of a class of

independent tasks onto heterogeneous computing systems, J. Parallel Distrib. Comput. 59 (1999) 107–131, ISSN 0743-7315.

[20] Y.-K. Kwok, I. Ahmad, Static scheduling algorithms for allocating directed task graphs to multi-processors, ACM Comput. Surv. 31 (1999) 406–471, ISSN 0360-0300.

[21] J. D. Ullman, NP-complete scheduling problems, J. Comput. Syst. Sci. 10 (1975) 384–393, ISSN 0022-0000.

[22] O. Sinnen, L. Sousa, On Task Scheduling Accuracy: Evaluation Methodology and Results, J. Supercomput. 27 (2004) 177–194, ISSN 0920-8542.

[23] I. C. Legrand, H. B. Newman, The MONARC toolset for simulating large network-distributed processing systems, in: WSC '00: Proceedings of the 32nd conference on Winter simulation, Society for Computer Simulation International, San Diego, CA, USA, ISBN 0-7803-6582-8, 1794–1801, 2000.

[24] C. Dobre, C. Stratan, V. Cristea, Realistic Simulation of Large Scale Distributed Systems using Monitoring, in: ISPDC '08: Proceedings of the 2008 International Symposium on Parallel and Distributed Computing, IEEE Computer Society, Washington, DC, USA, ISBN 978-0-7695-3472-5, 434–438, 2008.

[25] C. Dobre, F. Pop, V. Cristea, Towards Scalable Simulation of Large Scale Distributed Systems, in: NBIS '09: Proceedings of the 2009 International Conference on Network-Based Information Systems, IEEE Computer Society, Washington, DC, USA, ISBN 978-0-7695-3767-2, 103–108, 2009.

[26] B. Cho, H. Youn, E. Lee, Performability analysis method from reliability and availability, in: ICHIT '09: Proceedings of the 2009 International Conference on Hybrid Information Technology, ACM, New York, NY, USA, ISBN 978-1-60558-662-5, 401–407, 2009.

[27] A. Costan, C. Dobre, V. Cristea, R. Voicu, A Monitoring Architecture for High-Speed Networks in Large Scale Distributed Collaborations, in: ISPDC '08: Proceedings of the 2008 International Symposium on Parallel and Distributed Computing, IEEE Computer Society, Washington, DC, USA, ISBN 978-0-7695-3472-5, 409–416, 2008.