Self-Healing Distributed Scheduling Platform

Marc Frincu, Norha Villegas, Dana Petcu, Hausi Muller, Romain Rouvoy

Self-Healing Distributed Scheduling Platform

Marc E. Frincu*, Norha M. Villegas†, Dana Petcu*, Hausi A. Müller† and Romain Rouvoy‡
*Research Institute e-Austria, Timisoara, Romania
Email: {mfrincu,petcu}@info.uvt.ro
†Dept. of Computer Science, University of Victoria, Victoria, Canada
Email: {nvillega,hausi}@cs.uvic.ca
‡INRIA Lille - Nord Europe, ADAM Project-team, Villeneuve d'Ascq, France
Email: romain.rouvoy@lifl.fr

Abstract

Distributed systems require effective mechanisms to manage the reliable provisioning of computational resources from different and distributed providers. Moreover, the dynamic environment that affects the behaviour of such systems and the complexity of these dynamics demand autonomous capabilities to ensure the behaviour of distributed scheduling platforms and to achieve business and user objectives. In this paper we propose a self-adaptive distributed scheduling platform composed of multiple agents implemented as intelligent feedback control loops to support policy-based scheduling and expose self-healing capabilities. Our platform leverages distributed scheduling processes by (i) allowing each provider to maintain its own internal scheduling process, and (ii) implementing self-healing capabilities based on agent module recovery. Simulated tests are performed to determine the optimal number of agents to be used in the negotiation phase without affecting the scheduling cost function. Test results on a real-life platform are presented to evaluate recovery times and optimize platform parameters.

1. Introduction

With the continuous evolution from software intensive systems to socio-technical ecosystems, users and businesses are demanding the provisioning of distributed resources from different and independent infrastructure providers (IP) (e.g., Cloud, Grid or Cluster) in a reliable way, despite the uncertain, heterogeneous, transient and volatile environmental conditions that can affect the behaviour of these systems [1]. Thus, socio-technical ecosystems require effective mechanisms to manage the provisioning of resources while releasing the user from having to manually orchestrate providers to achieve system objectives. Regardless whether infrastructure providers are called *Grid-based Virtual Organizations* or now *Cloud providers*, their objective is to offer reliable, secure and efficient computational resources ensuring that the underlying systems are fault tolerant by exposing *self-healing* capabilities under dynamic environmental conditions [2]. Moreover, providers need to maintain their *autonomy* by keeping their own scheduling, security and negotiation policies. Therefore, under these complex dynamics, self-healing and autonomous behaviour are essential in any multi-provider Resource Management System (RMS).

Most current solutions for RMS have been developed for cluster or intra-provider usage where access policies are subject to the local authority decisions [3]–[6]. However, when considering inter-provider environments, these solutions need be adapted to facilitate task migration under certain access and sharing rules. Autonomy can be implemented using Multi Agent Systems (MAS). They rely on autonomous entities called agents to interact and to make decisions based on internal logic [7]. MAS offer a natural extension to RMS as they allow multi-providers to interoperate through negotiation [8], [9]. The foundations of multi-site MAS scheduling are briefly presented by Sauer et al. [10]. They present the problem as a hierarchical two-level structure that reflects the typical layout found in inter-site systems. The upper level consists of the global scheduler responsible for coordinating the lower level comprised of local schedulers working on individual locations.

As shown in Sect. 2, several approaches have been proposed to create MAS for task scheduling. However, most approaches focus on individual aspects such as negotiation, SLA management, scheduling or self-healing. Providing a completely distributed, self-healing and customizable scheduling solution for inter-provider usage is a challenge from both technical and scientific points of view. Technical issues arise due to compatibility problems when binding together various software for distributed storage or communication. Another reason could be that current solutions are still at an early stage of development and do not yet meet the required expectations. From a scientific perspective we require a model which allows us to represent the system components as autonomous, self-healing, fully customizable entities. Section 2 also outlines some work in the area of self-healing MAS for RMS. However results are scarce and do not offer complete self-healing and autonomous solutions to the problem.

To face the challenge of building an autonomous self-healing RMS platform, we tackle both technical and sci-

entific aspects and propose an adaptive inter-provider MAS scheduling platform able to (i) offer fully distributed storage and communication mechanisms; (ii) self-heal to support fault-tolerance by means of implementing agents as recoverable modules of feedback control loops; (iii) support autonomy among providers by separating the dynamically changing scheduling policy from the application by means of an inference engine; and (iv) adapt/change the negotiation policy by implementing negotiators as plugable modules. To minimize the impact of rescheduling tasks on the system, we reduce the number of agents needed during the negotiation phase without affecting the scheduling objective function. Moreover, we validate our approach by testing the time needed to heal the platform after module failures. Finally, we performe a study on finding the optimal set-up platform parameters that would not produce false healing events.

The remainder of this paper is organized as follows. Section 2 discusses related work on MAS for task scheduling. Section 3 gives an overview of the proposed RMS by presenting the platform model. The main modules are presented in Sect. 3.1. Details on the distributed solutions for managed resources and touch-points are shown in Sect. 3.2. The two types of agents, for scheduling and for self-healing are presented in Sects. 3.3 and 3.4. Insight on the platform is given from a self-healing perspective with details on how the control loops are implemented inside agents. Section 4 presents how we optimize the number of agents involved during the negotiation phase to minimize the impact of scheduling tasks on the system. Section 5 presents empirical tests on platform recovery times, as well as a method for determining the optimal parameters for the platform setup. Section 6 outlines the main conclusions of the paper.

2. Related Work

Well known MAS for RMS include Nimrod/G, which uses agents for handling the setup of the running environment, transporting the task to the site, executing it and returning the result to the client [11]; TRACE, which dynamically allocates resources and agents based on the demand [12]; ARMS [13], which uses PACE [14] for application performance predictions; and AppLeS (Application-Level Scheduling) that also implements adaptive capabilities [15].

Tang and Zang proposed a service-oriented peer-to-peer MAS [16]. However, their system relies on a simplistic scheduling mechanism and does not implement self-adaptation. The MAS proposed by Amoon et al. uses a single fixed scheduling agent responsible for planning tasks on resources [17]. Tasks are then moved to and from these resources by using mobile agents. While the approach offers an advantage by using mobile agents for migrating and executing tasks, its main drawback is the use of a single static scheduling agent. This compromises the robustness of the system by introducing a bottleneck as well as a single point of failure for the entire scheduling mechanism. Cao et al. proposed a load balancing method for MAS aimed at Grids [18]. Although their system implements scheduling algorithms based on artificial intelligence (AI) techniques, it lacks support both for overcoming security issues when multiple providers are used and for a customizable negotiation module. The approach proposed by Ouelhad et al. focuses on Service Level Agreement (SLA) protocols for MAS and presents a protocol for negotiation as well as for renegotiation in the presence of uncertainty [9]. The SLA is designed following the two level inter-site approach [10] and thus could be easily adapted to multi-provider scenarios. Shen et al. [8] present a negotiation protocol based on a case-based learning and reasoning mechanism. The four negotiation models included in the experiments involve the Contract Net Protocol, the Auction Model, the Game Theory Based Model and Discrete Optimal Control Model. This work could be used as a starting point for building adaptive negotiation agents, but it lacks the experiments for validating the benefits of such an approach.

Self-healing mechanisms can be implemented with feedback control loops [19]. When applied to MAS, feedback control loops allow agents to recover from expected failures (e.g., scheduling and security policy modifications) and more importantly from unexpected ones (e.g., network or resource failures, agent availability). Page et al. proposed a distributed self-adaptive and failure tolerant system based on an *n*-ary tree [20]. Each tree node represents a scheduler while leaves stand for processing nodes. However, the system has the following problems: it lacks a negotiation protocol between schedulers, communication is not truly decentralized as scheduling nodes can only communicate with their parents or children, and adaptivity is restricted to reassigning tasks to parent schedulers in case a scheduling node fails.

3. Platform Overview

An MAS intended for task scheduling usually consists of several intelligent agents working together toward efficient task scheduling. Intelligent agents differ from regular agents by supporting the following three important characteristics [10]: *reactivity*—enables agents with capabilities to react to events from the environment; *pro-activeness*—allows agents to take initiatives driven by goals; and *social ability*—enables agents to interact with other agents.

In this paper we propose a self-adaptive distributed scheduling platform composed of intelligent agents enhanced with custom modules (cf. Fig. 1). Agents are defined as intelligent control loops composed of one or more modules. Every control loop supports scheduling activities by implementing the foundational steps performed by an autonomic MAPE (Monitor-Analyse-Plan-Execute) loop [21]. As depicted in Fig. 1, modules that define an agent correspond to the monitor, analyser, planner and executor. Anal-

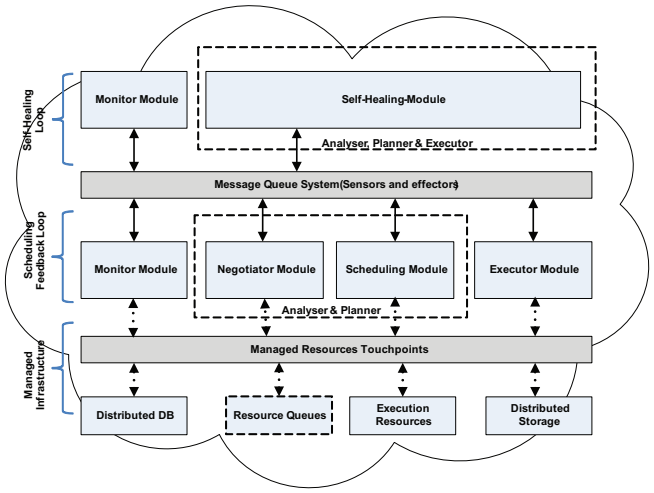

Figure 1. Self-adaptive MAS scheduling platform: high level architecture. Agents are implemented as intelligent feedback contriol loop composed of one or more modules.

ysis and planning activities are performed by the negotiator and the scheduler modules. Thus, an intelligent feedback loop supports not only scheduling but also re-scheduling activities. Environment data received through the monitor module is analysed by the negotiator and the scheduler modules that act accordingly to scheduling plans. Tasks are allocated among available resources by the executor module. The monitoring of resources and tasks related conditions feeds the system back to ensure the rescheduling process. Finally, a self-healing module built on top of the MAS enables the infrastructure with self-healing capabilities that allow module recovery. Agent distribution across multiple providers is supported by communication mechanisms implemented by the message queue system. This queue system not only enables the communication among agents and modules but also acts as the sensors and effectors required to support self-healing capabilities. As proposed in the Autonomic Computing Reference Architecture (ACRA) [22], the queue system implements a level of indirection that supports the gathering of contextual events related to the scheduling infrastructure behaviour and the execution of module recovery tasks. Depending on the case and to increase failure tolerance, agents can be made up of single modules spread across the system. Following the modular approach an agent can comprise all the system functionality (self-healing, negotiating and scheduling/executing tasks) or be a specialized agent (e.g., only for task scheduling).

The proposed MAS scheduling platform allows each provider to keep its own internal scheduling policies. Agents can also use their own scheduling policies at the meta-scheduling level. Hence, each agent is autonomous in deciding when to request/send tasks from/to partner agents.

3.1. Agent Modules

Agent modules are designed as autonomous entities that continue working even when the connection with their partners has been severed. To increase the platform's fault tolerance, we propose an approach where agents are implemented as intelligent control loops composed of several distributed modules that inter-communicate using asynchronous message queues. These control loops enable the scheduling infrastructure to self-adapt to both changes in the platform characteristics and to module failures.

Based on these feedback control loops we classify agents as either scheduling or self-healing. Both types are defined as a set of modules that implement the intelligent feedback loops' phases (cf. Sects. 3.3 and 3.4).

The distributed and autonomous approach allows agents to spread across several resource nodes and to continue functioning when some of their modules fail. Moreover, by being modular every provider can create custom agents tailored to its own requirements.

3.2. Managed Resources and Touch-points

Managed resources help the platform store and handle relevant data including task, resource and module information. Managed resources include storage, databases and execution resources (cf. Fig. 1). A special case is represented by the resource queues (i.e., dotted box in Fig. 1) which are not actual resources but are strongly linked to the execution resources as they provide the order in which tasks execute. In the proposed MAS they are implemented as simple ordered lists that can be accessed directly by means of a service.

Choosing a correct storage environment is essential to any distributed system as data needs to be stored in a reliable manner. As a distributed system can be quite volatile in terms of node availability and network traffic, we have opted for a distributed file system where failures of single nodes do not affect the overall system behaviour. The Hadoop Distributed File System (HDFS) is a suitable choice as it offers fault tolerance, scalability up to several thousands nodes, allows data re-balancing on cluster nodes and can take into consideration the physical location of the node when allocating storage [23]. HDFS was initially developed for clusters. However it can be expanded to inter-cluster usage as long as the same trust policies exist between peers. In a multi-provider environment this is somewhat difficult to implement as partners usually want to keep their internal policies. To address this problem, we have used an FTP server that works over HDFS and allows agents to connect to any remote HDFS.

For storing information related to task to resource mappings and task ordering in resource queues, the MAS also relies on a key-value access mode distributed database

system called HBase [24]. This database is primarily used by the scheduler, the executor and the monitor modules.

Execution resources represent resources on which agent modules run or where tasks execute. Usually resources are represented but not restricted to single machines. The choice of the actual task deployment platform is left to the IP.

Platform touch-points represent the means of binding the agents modules to the managed resources (cf. dotted arrows in Fig. 1). Usually they are specific to the software solution used for implementing the managed resource. For instance, the touch-point between HDFS and the MAS is achieved through FTP, the touch-point between HBase and the MAS is achieved with the HBase query language. A distinct case of touch-points is represented by the inter-module communication.

Communication is essential in any distributed system and facilitates the dialogue between the system's components. Because communication needs to be resilient when faced with failures, and to provide asynchronous read/write functionality, standard solutions such as HTTP, RMI or SOAP are improper. Distributed Message Queuing Systems seem to provide a viable solution. Our implementation uses RabbitMQ [25] which implements the AMQP (Advanced Message Queuing Protocol) standard [26]. RabbitMQ has the advantage of not requiring to know a priori the number of registered exchanges (stateless routing tables). This built-in feature is especially useful when issuing negotiation bids, checking the availability of existing modules or when attempting to activate modules. It also acts as the de facto Yellow Page directory where all agents are published. The procedure only requires the agent modules to bind to an exchanged and implement the corresponding listeners.

As RabbitMQ exchanges are persistent, any unprocessed message is safely stored when failures—either in the communication infrastructure or in the modules themselves—occur.

Data is sent across communication partners using JavaScript Object Notation [27]. It consists of four main elements: sender ID, recipient ID, message content and message type. This email-like message format represents the de facto SLA used by the platform and allows for modules to easily recognize whether the message is intended for them and where it originated. Any messages not conforming to this format are automatically ignored.

3.3. Scheduling Agents

Each scheduling agent implements a *scheduling feedback loop* composed of at least one monitor, one negotiator, one scheduler, and one executor module (cf. Fig. 2). To accomplish their business objectives, the scheduling systems must be able to adjust their behaviour according to environmental conditions that affect the state of computational resources. Feedback-loops are important models

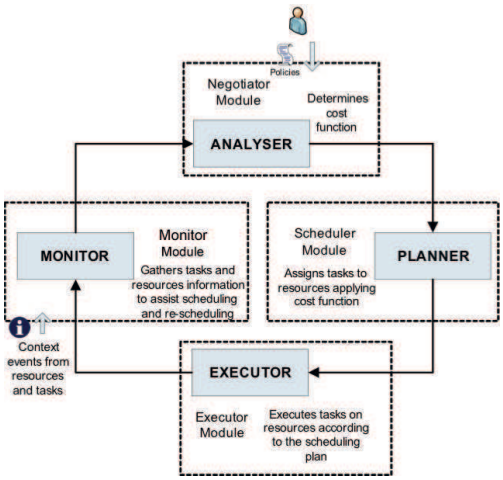

Figure 2. The feedback loop for the task rescheduling process

in the engineering of adaptive scheduling systems. They define how the interactions among agent modules ensure successful task scheduling according to SLAs and desired properties. Moreover, context information that characterizes the situation of resources and tasks must be monitored to control the rescheduling process. The following subsections detail the modules that compose every scheduling agent.

3.3.1. The Monitor Module. From a dynamic scheduling perspective, context can be defined as any information that characterizes the state of entities that can affect the behaviour of computational resources and tasks. The context information must be modelled so that it can be discovered, pre-processed after its acquisition from the environment, and handled to be provisioned based on the system's requirements [2]. Instances of context entities relevant for RMS are services, processing nodes and SLAs.

Monitor modules in scheduling agents are responsible for gathering information regarding the current situation of tasks and computational resources. Context acquisitors deployed as context probes gather at given events or time intervals information related to resource load, system heterogeneity, task size in terms of megabytes; task execution requirements such as required processing power in flops for a particular system state; and task failures from logs. Monitoring actions are triggered by task arrivals/completion or changes in resource availability. Whenever a new task is ready to be scheduled, the monitor gathers the relevant information of the task and then sends this information to the negotiator module that acts as the analyser in the scheduling feedback loop. Monitored information about task failures includes data on wrong input files, invalid dependencies and resource failures. Resource availability is periodically monitored within each IP's domain by sending ping signals to the task executor module.

3.3.2. The Negotiator Module. It is part of the analyser in the scheduling feedback loop. The negotiation process starts with the selection of the desired policy to guide the intra-scheduling process. The selection is based on a best cost provider approach where every available policy is evaluated in the light of the current system's configuration. Once the policy that provides the best solution is selected, a message is sent to the targeted scheduler requesting the loading of the corresponding rule-base definition to apply the policy.

The negotiator module's main purpose is inter-provider task relocation. Relocation usually follows a scheduling policy that extends/complements the possibly different heuristics at the intra-provider level. To minimize the number of messages the negotiator uses the dynamic scheduling algorithm for heterogeneous environments with regular task input proposed by Frîncu [28]. This policy only relocates tasks that have exceeded a certain waiting threshold on the local agent's resources. The reason for trying to minimize the number of task relocation is a direct consequence of the results provided by Tumer [29]. Tumer demonstrated that an MAS obtains best scheduling results when the agents' impact on the system is minimized. We argue that in our system this goal is achieved when both the number of bid requests and the number of scheduling agents involved in the negotiation is reduced. The former can be adjusted by tuning the rate of bid requests inside the scheduling policy as proposed by Frîncu [28], while the latter can be adjusted by reducing the number of agents involved in the negotiation phase as explained in Sect. 4.

The current negotiation policy is comprised of a one step action based negotiation phase [8]. Task relocation commences when the task exceeds a certain waiting time threshold on the currently assigned resource queue. The agent overseeing the task requests costs bids from all partner agents by broadcasting a message containing information related to the task at hand. The broadcast is received by all scheduling modules including the initiator of the request.

Each module adds the task to its list, marks it as temporary, executes the scheduling algorithm and assigns it to a resource queue. The estimate is then sent back to the negotiating module that handles the bid request. The negotiator waits within a predefined time interval for bid responses. Once this interval has expired the negotiator selects the agent module that provided the best bid (e.g., smallest estimated execution time) as the winner. The final scheduling decision is achieved by broadcasting the ID of the winner module to all registered scheduling modules. All agent modules that do not match the winner's ID will erase the temporary task from the list. This policy can be changed based on the needs of every provider by creating a custom negotiation module. In this way each organization can create and run its own policies for accepting/issuing bids.

To minimize transfer costs during negotiation only task meta-data is sent between agent modules. The data itself (the concrete task) is sent only when the executor module submits the task for execution.

3.3.3. Scheduling module. Scheduling modules are part of the analyser in the scheduling feedback loop and are in charge of the actual task to resource mapping. To maintain autonomy each provider is able to choose its own internal scheduling policy. Based on the policy, this module can decide on whether to relocate/accept tasks to/from other agents or to reschedule them on one of its resource queues. The scheduling module operates at the meta level. Once a task is submitted to a resource for execution, the scheduling module relinquishes any control over it and the scheduling proceeds according to the rules of the internal schedulers (e.g., Condor, Legion, or NetSolve).

A scheduling module is usually attached to one or more resources (e.g., that form a cluster) belonging to a single provider. Modules can even share the same resources for faster processing as proposed by Frîncu [30]. Rescheduling is accomplished periodically either at a given interval or when tasks get completed.

To facilitate the management and the separation of the scheduling policy from the scheduler itself, the algorithms are implemented as event-condition-action rules described using the SiLK formalism and executed using OSyRIS [31]. The rules can be easily loaded and unloaded from the rule base without restarting the module. Rule bases are usually changed when one of the following occurs: failures of the inference engine; changes in the scheduling policy as dictated by the organization; or adaptations to the new platform characteristics. Once the rule base has been loaded, the OSyRIS engine executes all the rules that have their conditions matched in the working memory. This is updated automatically after every successful rule execution. The cycle continues until either an error occurs or a special rule that makes the system idle is triggered. In either case the working memory is cleared and the cycle is restarted. When executing rules, remote service invocations are made for specific methods that provide information related to resources and tasks. The information is taken from the distributed database (cf. Sect. 3.2). The methods only offer access to atomic information [30]. In this way the building blocks for compound operations for a large range of scheduling heuristics are provided.

Rescheduling task issues (i.e., failures and optimization) are usually solved by reassigning tasks on a different resource. Task rescheduling is mandatory in any distributed system because of both the volatile nature of resource nodes and the on-line task arrival rate that needs to be considered. Therefore, the system needs to adapt to changes in resource availability and load rates as these two context facts influence the cost function directly.

Special cases are represented by already running tasks that failed to finish before the estimated execution time and

by resources that have gone offline during task execution. In both cases the tasks are considered to be aborted and the scheduler will attempt to reschedule them. Due to the difficulty of distinguishing dead resources from overloaded ones it is likely to end up with several instances of the same task running in parallel. In this case we followed an approach where only the first received result is considered valid.

3.3.4. The Executor Module. The purpose of the executor module is to take tasks from resource queues populated by the planner (i.e., scheduler module) and to start their execution. When the resources are exposed as services, the module's role is to submit the tasks' execution via the service's interface. Before execution, the task data, including executable/jar archive and dependencies, is grabbed from HDFS and copied to the resource.

To determine the set of tasks to be executed, the executor accesses the resource queue, determines the first *n* tasks ordered by execution number, and starts the tasks transfer.

Once the archive is copied to the corresponding resource, the executor extracts it and starts the execution. It is up to the provider to specify a means for executing the tasks. The executor is custom tailored to fit the provider and acts only as a liaison between the actual deployment platform and the MAS. As soon as the execution is completed, the results are archived and sent back to the HDFS. A completed execution does not necessarily mean a successful one. The concrete status can be queried by the monitor module from the execution logs to trigger a rescheduling process in case the execution has failed.

3.4. Self-Healing Agents

Self-healing agents, built on top of our MAS scheduling platform, enable the infrastructure with an intelligent feedback loop that supports module recovery capabilities. Self-healing agents are designed similar to the multi-agent feedback loop that is part of the autonomic system proposed by Caprarescu and Petcu [32]. Module failures represent failures in the execution of any of the MAS's modules.

As depicted in Fig. 3, the monitor module is responsible for gathering information about the status of modules. Gathering mechanisms are either supported actively by sending requests and/or pings, or passively by receiving messages from the monitored modules. Context sources are represented by the distributed database and message queues. The first one provides information for determining the load of the system and the topology configuration, while the second one offers data related to the existing modules. The monitored information is then sent to the analyser that determines the current state of the monitored modules.

Each time a module becomes active it sends a registration message to the monitor modules of the self-healing agents that belong to the same provider. Every module periodically

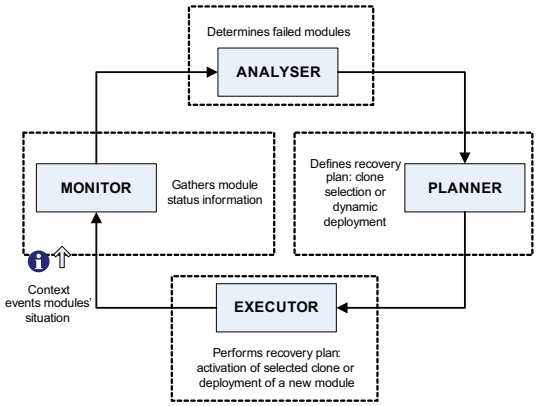

Figure 3. The feedback loop for the module recovery process that implements self-healing capabilities.

notifies through messages its availability. By using the monitored information, the analyser checks whether or not any of the registered modules has timed out.

Once a time out has been reached, the analyser sends the relevant details about the failure to the planner. The planner then searches for an inactive clone module (i.e., registered idle module) that can be used to replace the failed one. Clone modules receive messages from the environment and send notification pings but do not execute their logic.

After a clone has been identified the planner sends the details to the executor to activate it. In the case where no such clone exists, the system sends a multicast message searching for resources willing to deploy on demand a module. If such a resource is found, its endpoint identification (e.g., IP address) is sent to the executor in charge of transferring the proper module data and starting it. Newly deployed agents resemble mobile agents [17].

To avoid the redundancy of running several identical modules, every module ready to activate sends a broadcast message searching for one performing identical tasks. Depending on the broadcast outcome the module either activates or becomes idle.

Self-healing modules are also able to recover themselves from failures. The module recovery feedback loop implements several healing modules dealing with the MAS. As all modules are registered during their activation, healing modules are also monitored by partner modules and are subject to the same recovery mechanisms. A special category of failures consists of infrastructure failures. These are failures of components that cannot be handled explicitly by the self-healing mechanisms implemented on top of the MAS scheduling platform. They include most of the problems that arise from infrastructure related issues. In this paper the infrastructure is seen as the bundle of existing physical resources, network fabric, communication system, database storage and file system.

Our MAS can be viewed as a tree with the negotiator at

the root, the providers' schedulers at the second level and the executors as leafs. Failures at any level allow the system to continue functioning partially even without recovery mechanisms. Without a negotiator, the providers' schedulers continue scheduling tasks without inter-provider migration; without schedulers, the executors continue executing tasks in the order provided by the last schedule; without some of the executors, the scheduler reschedules tasks to resources still having executors attached to them. Other combination of failures lead to similar results. Errors in the HBase and the HDFS are dealt automatically by the systems themselves as they replicate data on several nodes for better fault tolerance.

Self-healing modules that have been isolated from the platform continue monitoring and healing reachable agent modules. Therefore, isolated islands of modules can continue to function even when separated from the platform.

4. Reducing the Number of Agents used in the Negotiation Phase

Reducing the number of involved scheduling agents minimizes the impact of the rescheduling tasks on the system [29]. However the reduction needs not to negatively influence the overall schedule cost function, which in this paper is set to makespan—time needed to complete the existing tasks. Selecting a proper agent subset from a suitable partition set is an essential step. In this paper we propose and test a partition method based on the fuzzy C-Mean algorithm for clustering agents [33]. Based on these partitions we select rescheduling candidates from the agents that match best the task requirements. We opted for this solution because clustering algorithms allow data to be partitioned based on similarities found in it.

In our tests we used the following similarity parameters when creating the clusters: *acceptTasks*, which specifies whether the agent module accepts (1) or not (0) tasks; *offerTasks*, which specifies whether or not the agent module offers (1) or not tasks (0); *noTasks*, which represents a normalized value of the number of tasks handled by this module at a given moment in time; and a list in which an element $operation_i$ indicates whether the agent module supports (1) or not (0) the respective operation for $i = \overline{1, n}$. Here n indicates the total number of operations supported by the platform and is used mostly when tasks are submitted for execution through a service interface that supports a restricted number of operations.

When trying to find a suitable cluster for a specific task its similarity parameters are set as follows: *acceptTasks=1*, *offerTasks=0.5*, *noTasks=0*, and $operation_i = 1$ for all operations required by the task.

Tests on the clustering algorithm used a fuzziness value of 1.25 and an algorithm stopping condition with a value of 10^{-10}. The number of agents was set to 100 and tests were repeated 20 times. The number of clusters increased with each test from one to 10 clusters and the pair (mean makespan, standard makespan deviation) was retained after each test. For each scheduling module a number of 250 tasks were generated. Each task had an *estimated execution time* (EET) following a Pareto distribution with a minimum EET value of 1,000ms and a shape parameter of 2. Task size was generated with a Pareto shape parameter of 1.3. This provided 25,000 tasks to be scheduled for the entire platform. Task arrival rate was modelled by using a 8 degree polynomial extrapolation [34]. The Grid'5000 network was used as topology for the tests [35]. At the meta level the DMECT algorithm proposed by Frîncu [28] was used while at provider level a FIFO policy was in place.

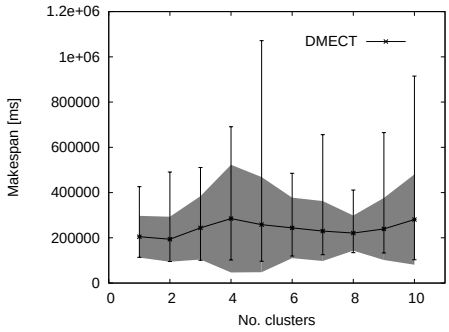

Figure 4. The influence of the number of clusters over the makespan produced by DMECT. The results suggest that the negotiator should use two clusters during the scheduling modules selection phase.

Figure 4 shows the main results of our tests. It shows the makespan evolution when clustering is used. The plot shows the mean makespan obtained from the DMECT heuristics, the standard deviation of the makespan (cf. grey polygon in Fig. 4) and the minimum and maximum values obtained in the tests. Tests reveal that the makespan obtained by applying the chosen clustering technique does not differ significantly for the 2 cluster case (-5%) while a maximum mean degradation (relative to the best case) is observed for 4 (+47%) respectively 10 (+44%) clusters. From these results we can infer that the safest way of reducing the number of involved schedulers without negatively influencing the makespan is to allow the negotiator to use 2 clusters during the scheduling modules selection phase.

5. Testing Scenario and Results

Stability and short healing times are important properties in our scheduling platform. On one hand, false healing events increase instability. On the other hand, short healing times are important for the platform's faster convergence to its stable state. Thus, our tests were aimed at determining the time needed to heal the platform after agent failures, and at finding the optimal platform parameters that

would not produce false healing events. False healing events usually happen when a running module's ping is not read inside the timeout interval. Consequently the module can be wrongly paused and another restarted. To test our MAS, we used a real life platform comprised of 12 distributed virtual machines (VM). The testing platform is based on the infrastructure services provided by the Cloud-related project mOSAIC [36]. The scenario used 12 agents represented by 2 healers, 1 negotiator, 2 schedulers and 8 executors.

5.1. Optimal Parameters Tests

Finding optimal parameters for the platform is crucial for ensuring its behaviour. This means not only that the platform should self-heal in the smallest amount of time but also not to provide false self-healing events due to a poor setting of the *time-out between two consecutive module pings* ($ping_i$) (interval in which a module *i* is considered active), *module idle times* (mit_i), *message receive time-out* ($msg_{timeout}$) or *batch size for the number of messages read* (msg_{batch}) during one healing iteration.

To avoid false healing events we need to properly adjust the reading rate to the provided parameters. For this we need to identify the length of every read beat (the time between two message reads). As events in the proposed platform can be mapped to a discrete time space, we can deduce the following relationship between two time intervals $\Delta t = hit + msg_{batch} \cdot msg_{timeout} + \tau_H$, where hit represents the healing's module idle time and τ_H represents the time taken by module operations distinct from message reads.

After every Δt we are left with a number of unread messages equal with $msg_{unread_{k+1}} = max(msg_{unread_k} - msg_{batch} + \sum_i^n \Delta t/mit_i, 0)$, where the sum provides the number of messages sent by module i during Δt.

In order to maintain $msg_{unread_k} = 0$ we need to have $msg_{batch} = \sum_i^n \Delta t/mit_i$. Because of the high number of variables involved in computing Δt we need to simplify the equation in order to solve it easily. Generally we need to ensure that there will be always at least one ping from every module read during each interval. The condition for obtaining this is: $\forall msg_k \mid arrival\_time\_msg_k \in [last\_ping\_time, next\_ping\_time] \Rightarrow \bigcap sender\_msg_k = n$ (*safe read condition*). To avoid violating it we need to find the minimal functioning configuration for the simplest case (i.e., all module send pings at the same moment) and to increase the ping until a sufficient value is reached.

For testing we used a variation of the previously listed parameters as follows: $mit_i = \overline{1,500}$ms, $msg_{batch} = \overline{1,20}$, $msg_{timeout} = 1$ s and $ping_i \in \{i \mid i = \overline{1,10}\}$s. Figures 5 and 6 show some of the results achieved during testing. The plots represent the average result obtained after 10 tests on each configuration. Results have shown that configurations that use a minimal $ping_i$ timeout of 0.9s have a low chance of providing false module healing events. The reason for this

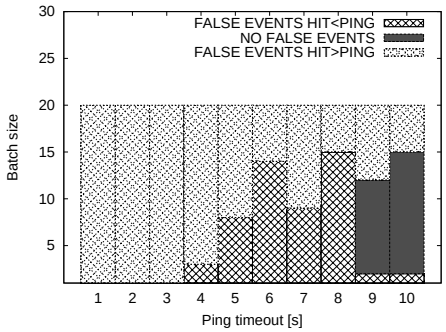

Figure 5. False recovery events related with the ping interval size for $hit = 1$, $mit_i = 1$, $msg_{timeout} = 1$

can be the fact that too small ping intervals cannot deal with cases when the *safe read condition* is not met. This is highly probable as modules are usually set with different idle times and started at different moments in time. The batch interval is maximized when hit and mit_i have both the smallest possible values (cf. Fig. 5). As shown by the results depicted in Figs. 5 and 6 the different module starting times become irrelevant for cases when the $ping_i$ timeout and the batch size become large enough as the healing module has enough time to process messages from all modules during a timeout.

5.2. Recovery Time Tests

Recovery time of the platform after a partial or total failure of the involved modules is crucial to its efficiency. Based on the tests from Sect. 5.1 the following optimal setup parameters have been used during experiments: $hit = 1$ms, $mit_i = 1$ms, $msg_{batch} = 20$, $msg_{timeout} = 1$ms and $ping = 20$s. Two scenarios were of interest. The first one involved one healing agent and only on demand deployment while the second one assumed one healing agent and only idle clones. Tests results are depicted in Fig. 7. As it can be noticed when using idle clones the recovery times are significantly improved as their are no other operations besides sending activation messages to the clones.

The average time to recover a module depends on its type. Executor and negotiator modules require an average of 4.2s and 3.6s while schedulers need around 15.3s to fully start. This difference is induced by the fact that schedulers also require to initialize the rule engine and load the rule base in memory before starting an operation that takes around 12s in the case of the DMECT algorithm. A complete step in a healer's loop takes around 1.7s from which 75.29% takes to process the messages, 1.17% is for sleeping and the rest of 23.54% is for executing other module logic. When starting remote modules an iteration can increase by up to 3.3 times due to transfer times. Tests also confirmed a dependency between the number of modules to be recovered and their recovery time. Given the long term running of the RMS

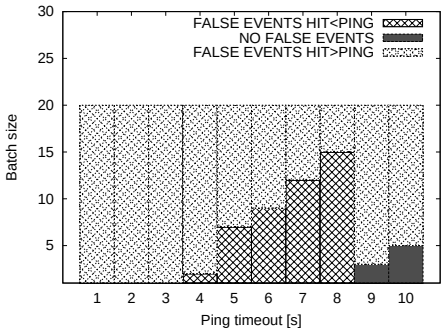

Figure 6. False recovery events related with the ping interval size for $hit = 1$, $mit_i = 500$, $msg_{timeout} = 1$

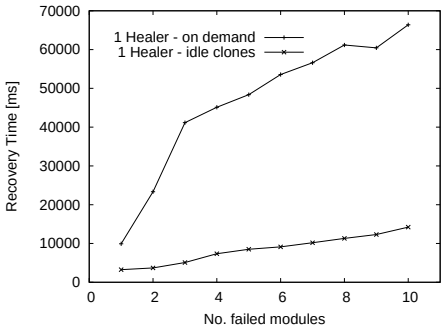

Figure 7. Recovery time vs. number of failed modules in the platform

a recovery time of around 60s for 10 modules using on demand deployment is considered to be reasonable. This recovery time can be diminished by attaching a healing module to every group of at most 10 modules.

6. Conclusion

In this paper we proposed a multi agent system for inter-provider task scheduling enhanced with self-healing capabilities. To achieve the goal of providing a distributed, self-healing scheduling platform several issues needed to be addressed as follows: (i) to provide fully distributed storage and communication mechanisms, we used distributed underlying platforms; (ii) as agents are required to be fault-tolerant and self adaptive, we implemented agents as modular intelligent control loops; (iii) to maintain the independence among multiple providers and easily switch scheduling policies, policy execution is based on an inference engine; and (iv) to support flexibility in changing the negotiation policy according to particular needs, we implemented a negotiator as a plug-in-able module. Simulated tests were run to minimize, without reducing the makespan, the number of agents involved in inter-provider scheduling. Finally, we tested our RMS on a real life environment in order to observe its healing capabilities. Tests have shown that the platform recovery times are inside reasonable limits. Future work includes integrating and testing the platform on the future cloud API provided by the mOSAIC project.

Acknowledgments

The work of the first and third authors has been funded by the European FP7-ICT project mOSAIC grant no. 256910. The work of the second and fourth authors is partially funded by the National Sciences and Engineering Research Council (NSERC) of Canada (CRDPJ 320529-04 and CRDPJ 356154-07), IBM Corporation, Icesi University, Cali, Colombia and University of Victoria, British Columbia, Canada.

References

[1] L. Northrop, P. Feiler, R. Gabriel, J. Goodenough, T. Longstaff, R. Kazman, M. Klein, D. Schmidt, K. Sullivan, and K. Wallnau, "Ultra-large-scale systems—The software challenge of the future," Carnegie Mellon University Software Engineering Institute, Tech. Rep., 2006. [Online]. Available: http://www.sei.cmu.edu/uls

[2] N. M. Villegas and H. A. Müller, "Managing dynamic context to optimize smart interactions and services," ser. LNCS, vol. 6400. Springer, 2010, pp. 289–318.

[3] D. Thain, T. Tannenbaum, and M. Livny, "Distributed computing in practice: the condor experience." *Concurrency - Practice and Experience*, vol. 17, no. 2-4, pp. 323–356, 2005.

[4] I. T. Foster, "Globus toolkit version 4: Software for service-oriented systems." in *NPC*, ser. LNCS, H. Jin, D. A. Reed, and W. Jiang, Eds., vol. 3779. Springer, 2005, pp. 2–13.

[5] S. J. Chapin, D. Katramatos, J. Karpovich, and A. Grimshaw, "Resource management in legion," *Future Generation Computer Systems*, vol. 15, no. 5, pp. 583–594, 1999.

[6] H. Casanova and J. Dongarra, "Netsolve: A network server for solving computational science problems," *Intl. Journal of Supercomputing Applications and High Performance Computing*, vol. 11, no. 3, pp. 212–223, 1997.

[7] M. Wooldridge, "Inteligent agents," in *Multiagent Systems: A modern Approach to Distributed Artificial Inteligence*, G. Weiss, Ed. MIT Press, 1999, pp. 27–77.

[8] W. Shen, Y. Li, H. Genniwa, and C. Wang, "Adaptive negotiation for agent-based grid computing," in *Proceedings of the Agentcities/AAMAS'02*, 2002, pp. 32–36.

[9] D. Ouelhadj, J. M. Garibaldi, J. MacLaren, R. Sakellariou, and K. Krishnakumar, "A multi-agent infrastructure and a service level agreement negotiation protocol for robust scheduling in grid computing," in *EGC*, 2005, pp. 651–660.

[10] J. Sauer, T. Freese, and T. Teschke, "Towards agent-based multi-site scheduling," in *Proceedings of the ECAI 2000 Workshop on New Results in Planning, Scheduling, and Design*, Berlin, 2000, pp. 123–130. [Online]. Available: citeseer.ist.psu.edu/sauer00towards.html

[11] D. Abramson, R. Buyya, and J. Giddy, "A computational economy for grid computing and its implementation in the nimrod-g resource broker," *Future Gener. Comput. Syst.*, vol. 18, no. 8, pp. 1061–1074, 2002.

[12] S. S. Fatima and M. Wooldridge, "Adaptive task resources allocation in multi-agent systems," in *AGENTS '01: Proceedings of the fifth international conference on Autonomous agents*. New York, NY, USA: ACM, 2001, pp. 537–544.

[13] J. Cao, S. A. Jarvis, S. Saini, D. J. Kerbyson, and G. R. Nudd, "Arms: An agent-based resource management system for grid computing," *Scientific Programming*, vol. 10, no. 2, pp. 135–148, 2002.

[14] J. Cao, D. J. Kerbyso, E. Papaefstathiou, and G. R. Nudd, "Performance modeling of parallel and distributed computing using pace," in *Proceedings of 19th IEEE International Performance, Computing and Communication Conference*, ser. Lecture Notes in Computer Science. IEEE Computer Press, 2000, pp. 485–492.

[15] F. Berman, R. Wolski, H. Casanova, W. Cirne, H. Dail, M. Faerman, S. M. Figueira, J. Hayes, G. Obertelli, J. M. Schopf, G. Shao, S. Smallen, N. T. Spring, A. Su, and D. Zagorodnov, "Adaptive computing on the grid using apples," *IEEE Trans. Parallel Distrib. Syst.*, vol. 14, no. 4, pp. 369–382, 2003.

[16] J. Tang and M. Zhang, "An agent-based peer-to-peer grid computing architecture: convergence of grid and peer-to-peer computing," in *ACSW Frontiers '06: Proceedings of the 2006 Australasian workshops on Grid computing and e-research*. Darlinghurst, Australia, Australia: Australian Computer Society, Inc., 2006, pp. 33–39.

[17] M. Amoon, M. Mowafy, and T. Altameem, "A multiagent-based system for scheduling jobs in computational grids," *ICGST International Journal on Artificial Intelligence and Machine Learning*, vol. 9, no. 2, pp. 19–27, 2009.

[18] J. Cao, D. P. Spooner, S. A. Jarvis, and G. R. Nudd, "Grid load balancing using intelligent agents," *Future Generation Computer Systems*, vol. 21, no. 1, pp. 135–149, 2005.

[19] H. A. Müller, H. M. Kienle, and U. Stege, "Autonomic computing: Now you see it, now you don't—design and evolution of autonomic software systems," in *International Summer School on Software Engineering (ISSE) 2006-2008*, ser. LNCS, A. D. Lucia and F. Ferrucci, Eds., vol. 5413. Springer, 2009, pp. 32–54. [Online]. Available: http://www.springerlink.com/content/0k008550k285gq9v/

[20] A. Page, T. Keane, and T. J. Naughton, "Adaptive scheduling across a distributed computation platform," in *Third International Symposium on Parallel and Distributed Computing*, J. P. Morrisson, Ed. Cork, Ireland: IEEE Computer Society, July 2004, pp. 141–149.

[21] J. O. Kephart and D. M. Chess, "The vision of autonomic computing," *Computer*, vol. 36, no. 1, pp. 41–50, 2003. [Online]. Available: http://www.computer.org/portal/web/csdl/doi/10.1109/MC.2003.1160055

[22] IBM Corporation, "An architectural blueprint for autonomic computing," IBM Corporation, Tech. Rep., 2006. [Online]. Available: http://www-03.ibm.com/autonomic/pdfs/AC_Blueprint_White_Paper_4th.pdf

[23] "Hadoop distributed file system," 2010, http://hadoop.apache.org/hdfs/docs/r0.21.0/ (accessed September 28, 2010).

[24] "Hbase," 2010, http://hbase.apache.org/ (accessed Octomber 14, 2010).

[25] "Rabbitmq," 2010, http://www.rabbitmq.com/documentation.html (accessed September 28, 2010).

[26] "Advanced message queuing protocol," 2010, http://www.amqp.org/confluence/download/attachments/720900/amqp-master.1-0r0.pdf (accessed September 28, 2010).

[27] "Java simple object notation," 2010, http://www.json.org/ (accessed September 28, 2010).

[28] M. E. Frîncu, "Dynamic scheduling algorithm for heterogeneous environments with regular task input from multiple requests," in *Procs. of the 4th Int. Conf. in Grid and Pervasive Computing GPC'09*, ser. Lecture Notes in Computer Science, vol. 5529. Springer-Verlag, 2009, pp. 199–210.

[29] K. Tumer and J. Lawson, "Multiagent coordination for multiple resource job scheduling," in *Adaptive and Learning Agents*, M. Taylor and K. Tuyls, Eds. Lecture notes in AI, Springer, 2009.

[30] M. Frincu, "A method for distributing scheduling heuristics inside service oriented environments using a nature-inspired approach," in *Proceedings of the 9th International Symposium on Parallel and Distributed Computing*, 2010, pp. 211–218.

[31] M. Frîncu and D. Petcu, "Osyris: a nature inspired workflow engine for service oriented environments," *Scalable Computing: Practice and Experience*, vol. 11, no. 1, pp. 81–97, 2010. [Online]. Available: http://www.scpe.org/vols/vol11/no1/SCPE_11_1_08.pdf

[32] B. A. Caprarescu and D. Petcu, "A self-organizing feedback loop for autonomic computing," in *Proceedings of the International Conference on Adaptive and Self-Adaptive Systems and Applications*, 2009, pp. 126–131.

[33] S. Chuai Aree, C. Lursinsap, P. Sophasathit, and S. Siripant, "Fuzzy c-mean: A statistical feature classification of text and image segmentation," *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems*, vol. 9, no. 6, pp. 661–671, 2001.

[34] D. G. Feitelson, "Workload modeling for computer systems performance evaluation," September 2010. [Online]. Available: http://www.cs.huji.ac.il/~feit/wlmod/

[35] "The grid'5000 experimental testbed," https://www.grid5000.fr (accessed Jun 23rd 2010).

[36] "The mosaic project," 2010, http://www.mosaic-project.eu.