Reconfigurable Applications Using GCMScript

Matías Ibáñez, Universidad de Chile
Cristian Ruz, Pontificia Universidad Católica de Chile
Ludovic Henrio, French National Centre for Scientific Research
Javier Bustos-Jiménez, Universidad de Chile

A component-based framework for building reconfigurable and distributed applications facilitates the programming of autonomic behavior. A distributed master-slave application that self-adjusts its load demonstrates the approach.

Today's applications are usually deployed in distributed environments such as grid or cloud infrastructures, where it's easy to acquire elastic computational resources over which different application components can be deployed and run in parallel. Developers of these distributed architectures must consider concerns such as network traffic, request rate, cost, and security requirements, which can vary for different cloud providers and affect the application at deployment or even runtime, sometimes unexpectedly. Cloud-based applications must be dynamically reconfigured to quickly adapt to new conditions to meet the required quality of service (QoS) and avoid lengthy downtimes. The need for rapid reaction and the complex management requirements governing cloud services reach a level that goes beyond the capabilities of human administrators. Autonomic behavior allows applications to manage themselves.

 2325-6095/16/$33.00 © 2016 IEEE

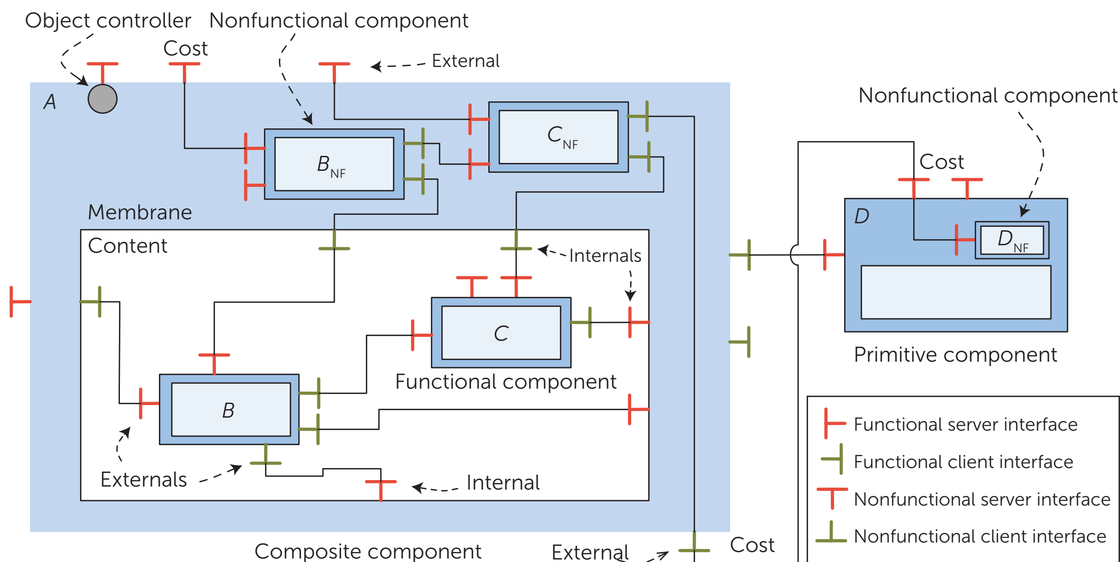

FIGURE 1. Grid Component Model (GCM) notation. The membrane of a composite component *A* contains two components, B_{NF} and C_{NF}; and the membrane of primitive component *D* contains one component, D_{NF}. (NF: nonfunctional, F: functional)

Cloud-based applications' distributed nature makes the existence of a centralized knowledge of the whole application architecture impractical. In this context, component-based software development appears to be a suitable model to build applications through the composition of loosely coupled entities that can be deployed and run on different virtual resources. Component-based applications are also designed to be reconfigurable and adaptable through simple binding modifications, and several component models have been proposed. Component-based systems, such as those built using Fractal[1] or the Grid Component Model (GCM),[2] aim to facilitate reusability and naturally adapt to their execution context. Adaptation, however, isn't always easy to achieve, even less so to program, and some developments (such as FScript[3]) are specifically designed for a less error-prone reconfiguration.

Here, we show how we program autonomic reconfigurations of distributed component-based applications using the GCMScript language. GCMScript is an extension of FScript that allows distributed execution of reconfiguration actions through the collaboration of several GCMScript interpreters embedded in GCM applications running in one or more cloud environments. Using GCMScript actions and autonomic control loops inside GCM applications, we can build effective autonomic behavior into an application. We can then program this autonomic behavior to dynamically modify the application's architecture, deploy or remove components using the infrastructure's elastic nature, or modify runtime parameters to scale and improve QoS management. We also present an example GCM application in which self-optimization is triggered either through an autonomic control loop that continually evaluates the application's performance and modifies its parameters to improve itself or through the autonomic adaptation triggered by a modification in the environment.

Model and Language

We present our work in the context of the GCM and the GCMScript language for programming distributed reconfigurations.

Grid Component Model

GCM is a hierarchical component model,[2] adapted from Fractal,[1] for large-scale distributed computing. Figure 1 shows the GCM assembly structure and terminology. Services are offered through server interfaces, with required services bound to client interfaces. GCM components provide collective

interfaces, which can be one to many (multicast), many to one (gathercast), or many to many (MxN). One-to-many interfaces transform a single invocation into many invocations, potentially distributing the invocation parameters. Many-to-one interfaces wait for several invocations and transform them into a single invocation, potentially aggregating the parameters received. MxN interfaces feature both aspects. Another architectural addition in the GCM is the possibility of structuring the nonfunctional (NF) part (the membrane) through nonfunctional components that share the same features as regular functional (F) components: they can be assembled, composed, and introspected. The main advantages of using such a membrane are the separation of nonfunctional concerns from the functional part and the possibility of designing and implementing complex and adaptable nonfunctional behaviors.

GCM/ProActive

GCM/ProActive is the implementation of the GCM using active objects. In GCM/ProActive, an active object—that is, an object with a single thread of execution and a service queue—implements each component. Active objects can have their own execution policy and implement asynchronous requests using futures (that is, placeholders for the results of an invocation). One of the main advantages of using active objects to implement components is their adaptation to distribution. Active objects provide a natural way to build loosely coupled components—that is, components responsible for their own state and execution—that only communicate via asynchronous messages. Figure 1 shows the GCM/ProActive implementation of the membrane.

By separating concerns between the functional and nonfunctional parts, we can design each behavior separately and delay integration of the two aspects until deployment time. Even after the component has been instantiated, it's possible, through introspection, to analyze and dynamically modify both the functional and nonfunctional components without affecting the rest of the architecture. Because of this decoupling, an expert can design component management concerns (such as composition, bindings, monitoring, reconfiguration, security policies, load balancing, and general self-management activities) without knowing about the functional behavior, and, conversely, the functional part's programmer need not focus on the nonfunctional concerns. At execution time, however, the functional and nonfunctional parts need to interact. In particular, a nonfunctional concern related to self-management might need to know the composition or observe the behavior of the functional part, and it might need to modify some attribute or carry on an action that affects that part.

Self-management behaviors might require a complex implementation, and separating the application control into different components becomes reasonable. By employing a componentized membrane, we can implement complex tasks using components that, similarly to the functional part, can be assembled, composed, and replaced. For example, in Figure 1, components A (with subcomponents B and C) and D might run in different cloud providers with different pricing models. The cloud provider of A charges a fixed amount per day, while the cloud provider of D charges depending on incoming traffic on D. To compute a unified cost metric, A can use a nonfunctional component, C_{NF}, that tracks the uptime days of A, while D_{NF} tracks requests received by D. We can program a component B_{NF} to query both C_{NF} and D_{NF} (using an intercloud nonfunctional binding) and determine the cost of running the whole application. This metric is exposed to a nonfunctional interface for further composition. Also note that we can modify the pricing model by changing only the nonfunctional components without modifying the functional components. By inserting nonfunctional components, we can introduce autonomic behavior in an application.

GCMScript and Reconfiguration Controller

Similarly to Fractal components, GCM components expose control interfaces that let us modify, at runtime, the application structure. Using these interfaces, we can add or remove components or bindings at runtime to implement dynamic reconfigurations.

To simplify the programming of reconfiguration procedures, GCMScript is dedicated to reconfiguration of distributed components—in particular, the reconfiguration of GCM components.[2] GCMScript is based on an extension of FScript, the reconfiguration scripting language used by Fractal components.[3] FScript is executed by a centralized interpreter located in a global component that has knowledge of all the architecture. GCM applications, however, don't have that restriction, and GCMScript can interpret reconfiguration scripts in a distributed manner. In this way, composite components can be responsible for reconfiguring their inner components independently, allowing reconfiguration to take place in parallel and facilitating their execution in large-scale infrastructures. Access to GCMScript engines is achieved through a controller interface—the *reconfiguration controller*—shown in Figure 2.

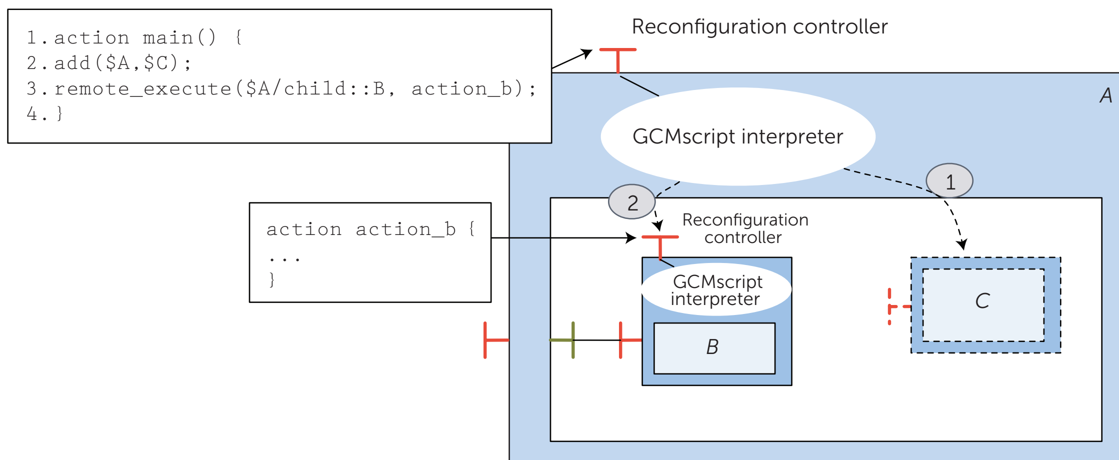

FIGURE 2. GCMScript interpreter delegation. The GCMScript interpreter of *A* executes a local action (adding component *C* to *A*), and a remote action that is delegated to the GCMScript interpreter of *B*, which is a child of *A*. Component *B* is located using FPath axes.

The reconfiguration controller embeds an instance of an FScript interpreter and provides methods `load` and `execute` for loading and triggering reconfiguration scripts and actions. To ensure consistency and avoid unpredictable interplay between a component system's execution and reconfiguration actions, we must ensure that a component's functional behavior is stopped while reconfiguration takes place. Because we're dealing with an asynchronous setting, we rely on a protocol to be able to safely stop an asynchronous composite component together with all its subcomponents.[4] The protocol's objective is to simplify the design of safe adaptation procedures: when a subsystem is entirely stopped, it can go through a reconfiguration phase before being restarted.

Autonomic Component System

Our autonomic component system (ACS) relies on nonfunctional components and GCMScript to facilitate the creation of GCM applications that contain autonomic behavior and can be deployed and applied over GCM applications running in cloud environments. The design relies on a set of nonfunctional components based on the stages of the monitor/analyze/plan/execute (MAPE) control loop, and a set of GCMScript actions that let the ACS activate or modify these components' behavior at runtime. For this purpose, one GCMScript interpreter is located in each GCM component and accessed through a reconfiguration controller.

Configuration Elements

Following the MAPE autonomic control loop, we define four components: *monitor*, *analyze*, *planner*, and *execute*. In previous work, we developed GCM/ProActive autonomic applications following this scheme (see the sidebar for a discussion of this and related work). However, the insertion and configuration of nonfunctional components is quite cumbersome and error prone as it requires several calls to the component API, and we must take additional precautions to stop and start membranes and components in the appropriate order. Here, we provide a set of GCMScript actions that simplifies the construction of the self-adaptive behavior.

ACS aims to make the application compliant with certain high-level objectives. These objectives are defined by three configuration elements:

- *metric* in the monitor component,
- *rule* in the analyze component, and
- *plan* in the planner component.

In our implementation, each of these elements is a Java class, defined by the user as an extension of a provided parent class. Their instances operate together to define the self-adaptive behavior. ACS provides actions to configure the monitor, analyze, and planner components using these elements. The execute component has a fixed configuration embedding the GCMScript interpreter.

Metric. The *metric* element performs measurements over the application at runtime. The GCM/ProActive implementation generates a set of general-purpose events at runtime, which are caught and stored by the monitor component. A *metric* that's loaded into the monitor component can use the stored data to

BACKGROUND AND RELATED WORK IN COMPONENT MODELS FOR AUTONOMIC COMPUTING

In 2009, at the "Grids Meet Autonomic Computing" workshop, researchers focused on the key challenges in grid and cloud computing that autonomic computing techniques support.[1] In the workshop panel, researchers and practitioners observed that "existing cloud computing infrastructure already supports some autonomic concepts, such as determining the number of virtual machines (VMs) to support, providing dynamically expandable storage, and migrating workloads across computational platforms." In this scenario, the convergence of cloud computing with autonomic components seems to be natural and highly synergic.

One of our first attempts in autonomic computing for grid/cloud computing was the use of coupling contracts for ProActive's active objects (the base of the GCM). Using these contracts, we achieved autonomic behaviors such as dynamic load balancing.[2]

Component models aim to increase code reusability and software adaptability by introducing well-delimited software entities with clearly defined interfaces corresponding to either the offered services or those explicitly required to fulfill them.[3] Component-based applications differ from other kind of applications in that they make the resulting software architecture explicit. As in the ProActive contracts, we describe the component architecture using an assembly language (ADL). Associated to an ADL, a factory instantiates all the components that constitute the application using a parser and component generators. This step deploys base components from their source implementation and binds component instances according to the dependencies defined in the ADL.

Hierarchical composition simplifies the creation of large applications. Gluing together components to form another component that can be used in the composition makes it easier to design larger applications and also to use different deployment concerns over them. For example, a cloud provider could deploy groups of components by using a single composite component that contains them all.

In addition to using an API directly, modifications over the application can be expressed through scripting languages associated with the ADL, as in ArchJava,[4] or built on an API as in FScript for Fractal components.[5] Allowed modifications can also be the result of some component-associated constraints[6] or reconfiguration rules (such as event condition action rules). As in our case, most of these approaches rely on the monitor/analyze/plan/execute (MAPE) model for autonomic computing[7]; however, our framework allows reconfiguration to take place in a distributed manner instead of relying on a centralized control. To enable the dynamic reconfiguration of the control layer itself, this layer should be programmed yielding a componentized membrane concept.[8] Other

measure nonfunctional aspects, such as how many times a component receives requests from other components through a server interface or the average response time for a particular interface. Measurement can be done either manually by calling a method of the monitor component that gets or computes the value for the *metric* or automatically by using event subscription, so whenever a GCM/ProActive event is caught, the metric value is updated and sent to all subscribers.

Additionally, if a component *A* depends on the services of a component *B* (that is, *A* has a functional binding to *B*), the monitor of component *A* has access (through a nonfunctional binding) to the metrics of component *B*. This property is transitive and lets us build more complex and varied metrics into the application.

Rule. The *rule* element represents a service-level objective that must be enforced during runtime. A *rule* relies on the value of a subset of *metrics* that must be available in the monitor to verify the state of the execution and determine the state of the *rule* compliance. If the rule isn't in compliance, the *rule* object automatically triggers an alarm.

approaches include the SOFA component model, whose control part is composed of microcomponents,[9] and Dynaco, which uses full-fledged components.[10] The GCM achieves such flexibility because the control level is also made of components.

To provide autonomic behavior to cloud applications, some approaches use MAPE loops in the application architecture and organize managers hierarchically.[11,12] Although they effectively collect the required metrics, our scheme lets us introduce the autonomic behavior as part of the application itself, instead of a separate architecture. Plus, our componentized membrane model together with the scripting language lets us program the autonomic behavior in a more generic and less error-prone manner.

References

1. C. Germain-Renaud and O.F. Rana, "The Convergence of Clouds, Grids, and Autonomics," *IEEE Internet Computing*, vol. 13, no. 6, 2009, p. 9.
2. J. Bustos-Jiménez et al., "Coupling Contracts for Deployment on Alien Grids," *Proc. CoreGRID 2006, UNICORE Summit 2006, Petascale Computational Biology and Bioinformatics Conf. on Parallel Processing* (Euro-Par'06), LNCS 4375, Springer, 2007, pp. 61–73.
3. C. Szyperski, *Component Software: Beyond Object-Oriented Programming*, Addison-Wesley Longman Publishing Co., 2002.
4. J. Aldrich, C. Chambers, and D. Notkin, "ArchJava: Connecting Software Architecture to Implementation," *Proc. 24th Int'l Conf. Software Eng.* (ICSE 02), 2002, pp. 187–197.
5. P.-C. David et al., "FPath and Fscript: Language Support for Navigation and Reliable Reconfiguration of Fractal Architectures," *Annals Telecomm.*, vol. 64, no. 1, 2009, pp. 45–63.
6. C. Tibermacine, D. Hoareau, and R. Kadri, "Enforcing Architecture and Deployment Constraints of Distributed Component-Based Software," *Fundamental Approaches to Software Eng.*, LNCS 4422, Springer, 2007, pp. 140–154.
7. IBM, *An Architectural Blueprint for Autonomic Computing*, white paper, June 2006; www-03.ibm.com/autonomic/pdfs/AC%20Blueprint%20White%20Paper%20V7.pdf.
8. L. Seinturier et al., "A Component-Based Middleware Platform for Reconfigurable Service-Oriented Architectures," *Software: Practice and Experience*, vol. 42, no. 5, 2012, pp. 559–583.
9. T. Bures, P. Hnetynka, and F. Plasil, "SOFA 2.0: Balancing Advanced Features in a Hierarchical Component Model," *Proc. 4th Int'l Conf. Software Eng. Research, Management and Applications*, 2006, pp. 40–48.
10. J. Buisson, F. André, and J.-L. Pazat, "A Framework for Dynamic Adaptation of Parallel Components," *Proc. Int'l Conf. Parallel Computing: Current & Future Issues of High-End Computing* (ParCo), vol. 33, 2005, pp. 65.
11. P. Martin et al., "Autonomic Management of Elastic Services in the Cloud," *IEEE Symp. Computers and Comm.* (ISCC 11), 2011, pp. 135–140.
12. F. de Oliveira, T. Ledoux, and R. Sharrock, "A Framework for the Coordination of Multiple Autonomic Managers in Cloud Environments," *Proc. IEEE 7th Int'l Conf. Self-Adaptive and Self-Organizing Systems* (SASO 13), 2013, pp. 179–188.

After a rule has been added to the analysis component, we can verify its state either manually by calling a method on the analysis component or setting an update period or automatically by subscribing to *metric* elements, so the *rule* is checked whenever the metric value changes.

Plan. The *plan* element handles one or more *rule* alarms and determines the necessary changes that must be applied to the application to restore the breached rule. In our implementation, a *plan* has access to the reconfiguration controller and, through it, can execute any necessary GCMScript commands.

Plans are inserted into the planner component. Execution of the *plan* can be requested either manually by invoking a method on the planner or automatically by subscribing to *rules*, so that whenever the rule triggers its alarm, the *plan* is executed.

GCMScript Extension

To facilitate the building of reconfiguration actions, we explicitly incorporate the elements described in the GCMScript model. These extensions let us

FIGURE 3. The GCMScript model with the extension for metric, rule, and planner. In addition to the FScript model, GCMScript extensions allow the components to include *metric*, *rule*, and *plan* objects. Furthermore, *plans* can subscribe to *rules*, and *rules* can subscribe to *metrics*.

browse, using FPath queries, the elements that are part of the reconfiguration architecture, and to introspect the subscription relationships between reconfiguration elements.

The goal is to manage the reconfiguration process using unique GCMScript commands and, keeping these concerns detached from the main application, facilitate the programming of autonomic behavior. We added a node for each configuration element to the model, as Figure 3 shows. Keeping the nodes separate lets queries easily reach them using FPath expressions, avoiding explicit API calls.

FPath queries use the model to navigate through the component architecture, where nodes are elements that can be queried and edge labels allow queries to navigate through the nodes. For example, the expression `$c/child::d` returns a reference on component `d`, which is a subcomponent of `c`; and `$c/metric::avgResponseTime` returns a reference to the metric called *avgResponseTime* that's located inside the monitor component of `c`.

We introduce several procedures to GCMScript. To add an element to a component, we use

- `add-metric($hostComponent, "fooName", "cl.example.FooMetric");`
- `add-rule($ruleNode, "fooRule", "cl.examples.rules.MyRule");`
- `add-plan($planNode, "fooPlan", "cl.examples.plans.MyPlan");`

To remove an element from a component, we use

- `remove-metric($hostComponent/metric::fooName);`
- `remove-rule($hostComponent/rule::fooRule);`
- `remove-plan($hostComponent/plan::fooPlan);`

To get property values, we query the element state:

- `value($myComponent/metric::myMetricId);`
- `alarm($myComponent/rule::myRuleId);`

Finally, for subscription, we use

- `fooRule = $hostComponent/rule::myRuleId;`
- `add-subscription($hostComponent/plan:fooPlanId, $fooRule);`

FScript provides a console application that lets us interactively execute commands over an FScript interpreter. We extended this console to integrate

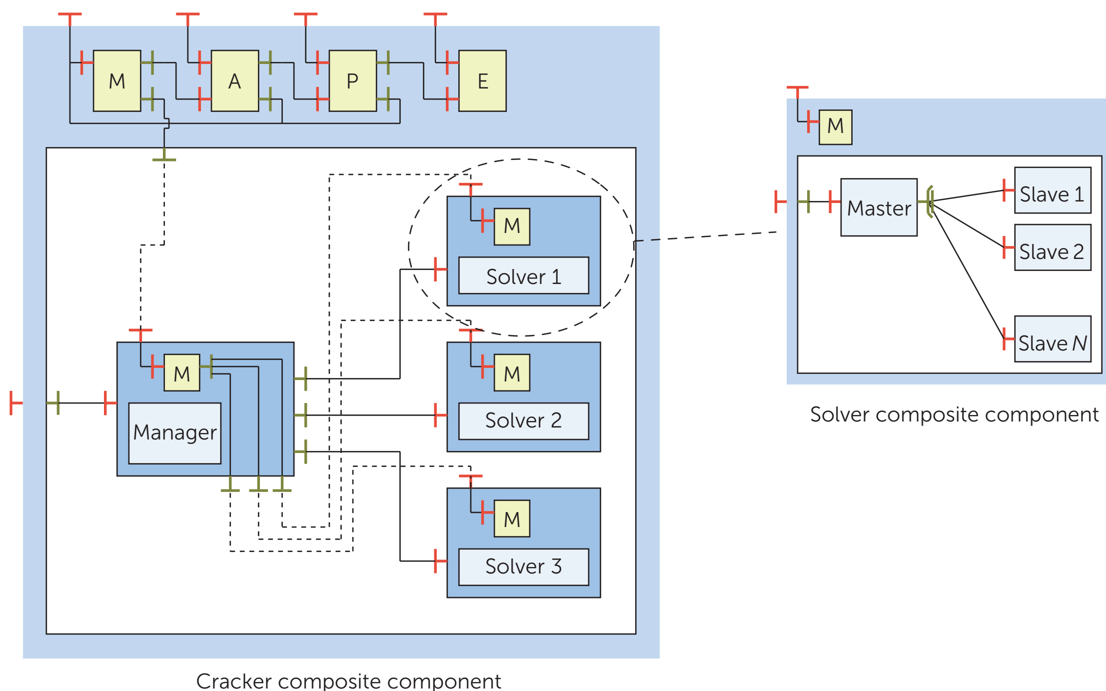

FIGURE 4. The MD5-hash cracker component. The manager splits the job into three subsets, and sends each to a different solver. Inside each solver, the set of words is evenly split among the slaves.

it with the GCMScript model. Since every component embeds its own GCMScript engine inside the reconfiguration controller, the GCMScript console consists of a user-component communication, instead of user-system communication. The console hides the notion of the GCMScript engine and lets users browse through components until they reach the desired one and send commands there. Figure 2 illustrates this interaction.

Use Case: Self-Balanced Distributed Brute-Force Cracker

A use case demonstrates how we can use our proposed ACS to build self-optimizing applications that can autonomically balance workloads. The application is a distributed brute-force cracker that uses a set of available machine resources, each handled by a GCM/ProActive component, hosted in different virtual instances, where each component can be located in different clouds.

Figure 4 shows the application, which works by testing a set of N words and computing its MD5 hash. The manager component distributes the load by splitting the job into three subsets J_1, J_2, J_3, $|J_1| + |J_2| + |J_3| = N$ of words, and sends each to a different solver. Each solver component is deployed on a different cloud resource and follows a master-slave model to provide local multicore parallelism to the cracking process. Inside, the master component evenly splits the set of words among the available slave components.

At the beginning of the execution, no autonomic behavior is enabled and the manager distributes an equal amount of work to each solver, so $|J_1| = |J_2| = |J_3|$. When the self-optimization capability is activated, the system evaluates the throughput of each solver component and tries to balance the load among them so they take similar amounts of time, thus avoiding the bottlenecks that occur when the slowest solver delays the answer (note there is no expropriation of tasks in this experiment).

We can manually change the amount of work distributed to each solver at any moment using the manager's *AttributeController*. In our experiment, we use this capability to introduce an autonomic behavior that chooses the optimal load distribution.

To set up the autonomic cracker application, we follow several steps:

1. Obtain computational resources for each component and deploy a GCM/ProActive component in a different resource.
2. Add a $avgRT_i$ metric on each solver. This metric calculates the average response time for solver i and will serve as a performance metric for the solver.

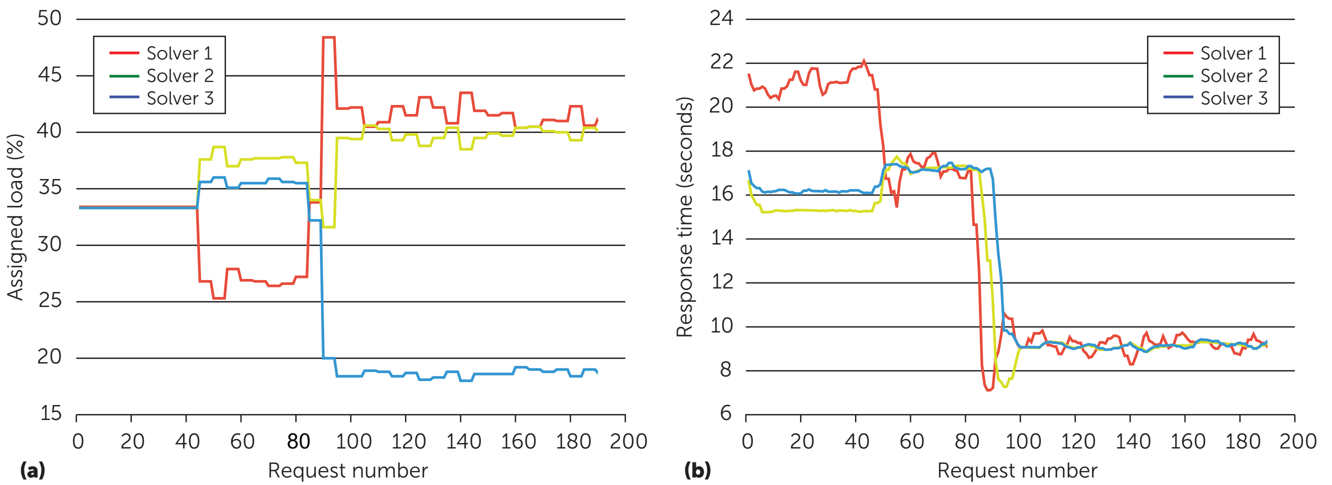

FIGURE 5. Use case performance results: (a) load distribution among solvers at each phase of computation; (b) solvers' response time for the cracking request. After the first 40 requests, self-optimization adjusts the load distributed to each slave. After request 80, a manual configuration change is introduced, and self-optimization behavior reacts to redistributed the load and obtain similar response times on each slave.

3. Add a *optimalBal* metric on the cracker to calculate an appropriate balance for the load given to the solvers. We compute this metric using the value of each $avgRT_i$.
4. Add a *balanceStatus* rule on the cracker component to check that *optimalBal* doesn't change by more than a tolerance *d* and subscribe the rule to the "receive a new cracking request" event.
5. Add a plan on the cracker component that replaces the current load distribution on the manager. Subscribe this plan to the rule *balanceStatus*.

We deploy the application in a cloud using two slaves on each solver. The solvers and the manager are deployed on different instances. Solver 1 is deployed on a machine with 8 × 1.2 GHz cores, solver 2 on a machine with 8 × 1.6 GHz cores, and solver 3 on an instance with 8 × 2.2 GHz cores.

The execution consists of three phases:

- In phase 1 (first 40 requests), the cracker runs without any autonomic behavior enabled. The default load balance $|J_1| = |J_2| = |J_3|$ is used for every request.
- In phase 2 (requests 41 to 80), self-optimization is activated and self-optimization elements are enabled.
- In phase 3 (requests 81 to last), environmental change occurs. Before the 81st request, four additional slaves are added to solver 1 and two additional slaves are added to solver 2. These modifications are made manually using the GCMScript console.

Figure 5a shows the load assigned to each solver at each phase of computation. Figure 5b shows the performance as measured by the metric *avgRT* for each solver. In this case, the *y*-axis represents response time, noting that the performance of the whole system as it is seen by a client is $\max\{S_i\}$. In fact, a user of this application isn't interested in the individual times taken by each solver. Even more, the user doesn't need to know about the existence of several solvers or the distributed cloud nature of the application, even if more than one cloud provider is being used. The user only perceives the time the application, as a whole, takes to respond to requests. However, it's possible that a modification requiring deployment of a new instance on multiple clouds will take longer to adjust than one deployed in a local cloud.

During phase 1, each solver receives equals amounts of work. Since the machines have different hardware capabilities, this distribution (same load for each solver, according to Figure 5a) isn't optimal, as evidenced in the first part of Figure 5b, where solver 1 seems to be the slowest. During phase 2, the self-balancing system is turned on, and the application autonomically determines that the current balance is far from *optimalBal*. Consequently, it computes a new load distribution. After the plan updates the new load distribution on the manager, the modification takes some iterations to reach a stable situation, and more work is assigned to solvers 2 and

3, and less to solver 1 (see Figure 5a), reaching a situation in which all solvers take a similar response time (Figure 5b).

Finally, after the architectural change in phase 3, solver 1 becomes more powerful than the others. The self-balancing system detects a deviation from the last computed *optimalBal* and recalculates the optimal distribution, leading again to a situation in which all solvers receive different amounts of work such that they all take similar time to execute. Notice that through each phase, the application's performance varies until it reaches a stable situation, effectively improving the throughput, and the response time perceived by the client, which is max(*s1, s2, s3*), decreases.

Our model can provide a basis for building more complex self-adaptation capabilities and facilitate the construction of autonomic cloud applications. Future work will consider development of self-repair capabilities, such as when a slave component is randomly taken out of line or deployed in a cloud that suddenly becomes unreachable. Although this is also an architectural change in the application, additional precautions must be taken to avoid possible inconsistencies or to perform necessary rollbacks as part of the recovery process. We plan to experiment with self-protecting capabilities that consider a prediction process that lets us anticipate degradations in the application or environment.

Acknowledgments

This work was partially funded by Communication and Information Research and Innovation Center (CIRIC)-INRIA, Chile, and the Safe Compositions of Autonomic Distributed Applications (SCADA) associated team.

References

1. E. Bruneton, T. Coupaye, and J.-B. Stefani, *The Fractal Component Model Specification*, OW2 Consortium, Feb. 2004; http://fractal.objectweb.org/specification/index.html.
2. F. Baude, L. Henrio, and C. Ruz, "Programming Distributed and Adaptable Autonomous Components: The GCM/ProActive Framework," *Software: Practice and Experience*, vol. 45, no. 9, 2015, pp. 1189–1227.
3. P.-C. David et al., "FPath and Fscript: Language Support for Navigation and Reliable Reconfiguration of Fractal Architectures," *Annals Telecomm.*, vol. 64, no. 1, 2009, pp. 45–63.
4. L. Henrio and M. Rivera, "Stopping Safely Hierarchical Distributed Components," *Proc. Component-Based High Performance Workshop* (CBHPC 08), 2008, article no. 8.

MATÍAS IBÁÑEZ *is a software developer at PostCenter, and formerly a student in the Computer Science Department at NIC Labs, Universidad de Chile. His research interests include distributed systems and component models. Ibáñez has a computer engineering diploma from the Universidad de Chile. Contact him at mnip91@gmail.com.*

CRISTIAN RUZ *is an assistant professor in the Computer Science Department at Pontificia Universidad Católica de Chile. His research interests include distributed systems, in particular parallel computing and the use of high-performance computing architectures, heterogeneous systems, and component-based software development. Ruz has PhD in computer science from the Université de Nice Sophia-Antipolis. Contact him at cruz@ing.puc.cl.*

LUDOVIC HENRIO *is the scientific leader of the SCALE team and a researcher (CR1) in the I3S lab at the French National Centre for Scientific Research (CNRS). His research interests include semantics, object calculi, components, concurrency and distribution, confluence and determinacy, and distributed systems. Henrio has a PhD in computer science from the Université de Nice Sophia-Antipolis. Contact him at ludovic.henrio@cnrs.fr.*

JAVIER BUSTOS-JIMÉNEZ *is the head of NIC Chile Research Labs, an institution affiliated with the Universidad de Chile. His research interests include networking, Internet protocols, and mobile and distributed computing. Bustos-Jiménez has a PhD in computer science from the Université de Nice Sophia-Antipolis, France. Contact him at jbustos@niclabs.cl.*

Selected CS articles and columns are also available for free at http://ComputingNow.computer.org.