Serpentine: Adaptive Middleware for Complex Heterogeneous Distributed Systems

M. Matos
U. Minho

A. Correia Jr.
U. Minho

J. Pereira
U. Minho

R. Oliveira
U. Minho

ABSTRACT

Adaptation of system parameters is acknowledged as a requirement to scalable and dependable distributed systems. Unfortunately, adaptation cannot be effective when provided solely by individual system components as the correct decision is often tied to the composition itself and the system as a whole. In fact, proper adaption is a cross-cutting issue: Diagnostic and feedback operations must target multiple components and do it at different abstraction levels.

We address this problem with the SERPENTINE middleware platform. By relying on the industry standard JMX as a service interface, it can monitor and operate on a wide range of distributed middleware and application components. By building on a JMX-enabled OSGi runtime, SERPENTINE is able to control the life-cycle of components themselves. The scriptable stateless server and cascading architecture allow for increased dependability and flexibility.

1. INTRODUCTION

The pervasiveness of computer systems and applications in daily lives is making computer dependability an issue of increasing relevance for the common citizen. Current applications tend also to be highly modular, composable and heterogeneous, leading to a steep increase of their complexity. This poses a great challenge to those that build them, often by integrating multiple cooperating services, but also to those who manage them daily and is in itself a menace to system dependability: It is hard to exclude human error while ensuring reaction to adversity within an adequate time-frame. Multiple studies have found human error to be comparable to hardware malfunction as a cause of downtime.

This challenge can be tackled in a cost effective way by mimicking the way the autonomic nervous system works in the human body, constantly adapting vital signals in order to reflect body needs in a decentralized fashion. Likewise, one should be able to build computer systems that adapt to changing needs and environment conditions without operator intervention [13]. Besides reducing the likelihood of human error, this should increase dependability by allowing faster response to different fault scenarios.

In this paper, we propose a middleware component, SERPENTINE, capable of addressing those issues by adapting the underlying system/service to changes in the production environment and/or user requirements without requiring the human intervention of a system administrator. The control techniques used are established and well known [12] allowing simple reactive control as well as feedback loop systems that constantly probe the state of the running services and apply pre-defined policies found adequate to the current environment and needs.

The strengths of SERPENTINE are the manifold. First, the fundamental design aspect of SERPENTINE is its statelessness which allows for seamless recovery after a management node failure or a broken connection because all the knowledge about a managed element can be rebuilt.

The choice of the *Java Management eXtensions* (JMX) standard [14] as the interface to managed elements provides direct and seamless support for a large number of middleware and application components built on the Java platform. To support the management logic SERPENTINE, uses the recent Java scripting engine [11] which provides a common framework for the execution of a multitude of scripting languages such as JavaScript, Python or Ruby. This technology provides an increased flexibility allowing SERPENTINE to transparently execute arbitrary scripts that define the policies to which the managed element must abide.

Furthermore, scripts can export information and operations themselves as JMX Managed Beans (MBeans). As these exported objects are just seen as common MBeans, it is possible to use another instance of SERPENTINE to monitor and actuate on them thus obtaining a cascading hierarchical setup. This hierarchical architecture is key both to build large and complex systems, but also as component wrappers when native semantics is not entirely adequate.

Finally, by relying on an JMX-enabled OSGi runtime [18], SERPENTINE can control component life-cycle as part of the adaptation process, thus being able to deploy, move, and undeploy software components.

The rest of the paper is structured as follows. Section 2 introduces the SERPENTINE architecture. Section 3 illustrates its usage with a concrete example. Section 4 compares the SERPENTINE approach with previous related work. Finally, Section 5 concludes the paper.

SAC'08 March 16-20, 2008, Fortaleza, Ceará, Brazil

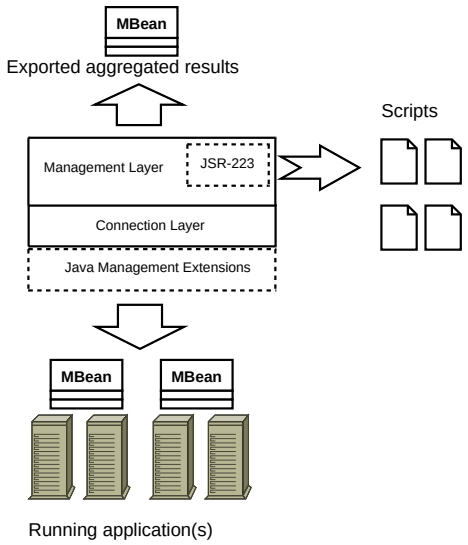

Figure 1: Architecture overview.

2. ARCHITECTURE

An overview of the SERPENTINE system architecture is presented in Figure 2. It is possible to observe three major components: The management targets, e.g. JMX-enabled application and middleware components; the core of SERPENTINE itself; and finally the pluggable policy scripts containing the *rules* to be applied. It also shows where the two key technologies come into play.

In the next section we describe the SERPENTINE stack in a bottom-up approach going from the lower tier of the communication with the targets to the logic and semantics necessary to write the scripts.

2.1 Management Targets and JMX

Management targets are required to be JMX-enabled. Java Management eXtensions (JMX) [14] provides standard interfaces and protocols to monitor and manage applications based on the Java platform. In short, JMX builds on Java Beans technology and allows the implementer of a software module to easily expose (read-only or read-write) attributes, operations and events. These features are grouped and exposed as Managed Beans, or MBeans for short.

The main advantage of this approach is that general purpose monitoring and management tools can use standardized introspection capabilities to learn the exposed interface and then interact with it. JMX is now a standard feature of all major Java middleware components. As an example, JBoss offers an extensive representation of the container state and configuration parameters using JMX. The Java runtime itself extensively supports JMX. In fact, JMX-enabled components integrate in the wider enterprise IT ecosystem by easing interoperation with other standards such as SNMP [17] and Web Services [15].

2.2 Connection Layer

The next layer up in the stack is the *Connection Layer*. Its responsibility is to maintain a pool of MBean proxies for usage by the Management Layer.

In detail, for each configured target, a connection is established and associated with an object that will be available to the next layer allowing it to read and change the state of the underlying application. A target may have several exported MBeans and each one may be accessed multiple times at nay given time thus the pool is responsible for maintaining only one connection to each target and forwarding all requests through it. Considering that the requests are normally very small (in size) this is important to minimize resources used by JMX connections and thus the impact on managed targets and the overall overhead of SERPENTINE.

The Connection Layer must also behave properly when the managed target is unavailable either due to a network problem, because the service was explicitly turned off or for some other reason. This is accomplished by checking the connection(s) as soon as a script is run and flagging an internal variable as appropriate. The script can check this variable and decide either to continue its execution or to stop (until the next execution). This of course does not prevent the case in which a connection failure occurs after the aforementioned check is done, thus bullet-proof scripts must always check the return values of its invocations because this layer will always return something to the upper one, either a value gathered from the MBean or a default value for that invocation (normally null).

With that approach the details and state of the connection are completely hidden from the Management Layer (and from the scripts) allowing it to assume that the monitored service is always up and running.

2.3 Management Layer

The Management Layer defines policies by loading and running user configuration scripts. Each script in SERPENTINE is a unique, self-contained entity that will define policies to manage/monitor an arbitrary number of MBeans. SERPENTINE will export an MBean for each running script with arbitrary operations and attributes, thus further expanding applicability scenarios.

2.3.1 Loading the scripts

Each script is treated individually by SERPENTINE and so each one must specify a set of properties that will allow SERPENTINE to properly handle them. Those properties are currently:

- the address(es) of the machine(s) to connect to;
- the JMX URL of the MBeans to manage in the above machines;
- the amount of time that will elapse before the script is re-executed.

and must be specified in every script in the form of a commentary on the scripting language being used. With this metadata SERPENTINE will create the connection(s) to the desired machine(s) (if possible), inject the MBean(s) into the script environment and schedule it for execution appropriately. Additionally two more objects are injected into the script, one is the dynamic MBean used to store arbitrary operations and attributes, the other is a simple object containing helper methods to use within the script. More details on the metadata can be found in Section 3 and in particular in Figure 3.

After this preliminary step the script is pre-compiled, if supported by the script engine implementation, and stored internally by the script engine thus avoiding to reparse it

before each execution and allowing for faster execution due to the fact that it is stored as Java classes or scripting language opcodes depending on the underlying implementation of the scripting engine.

Although the scripts may run many times during the entire life-cycle of SERPENTINE this step in only done once. All the information collected here is kept internally by SERPENTINE permitting a substantial increase in performance.

2.3.2 Running the scripts

After the script is properly set up it is time to run it. By default any loaded script will run forever (assuming that it is syntactical correct) unless explicitly killed by the administrator.

As said before, SERPENTINE supports multiple scripts running at any given time. To avoid conflicts between scripts each one will run in a unique thread. Because the scripts will run periodically and that period would normally be much longer that their running time, SERPENTINE will try to optimize resources by avoiding to run more than one script at a time.

2.3.3 Exported MBean and Cascading

Each script can store key/value pairs that will be automatically available as attributes along with its getter and setter methods in a JMX MBean. The exported operations will also be seen as standard JMX invocations by other components thus providing all the features needed to have an hierarchical cascading setup. It is interesting to note that the operation is defined in the script and will be called transparently from within the script when invoked in the exported MBean.

With this capabilities SERPENTINE can act as an aggregator, collecting information from various systems and resuming it in its exported MBean. Additionally it could export some operations to control the state of those systems but with the benefit of hiding how this control is enforced in the local scope. Thus it is possible to hide complex systems with complex policies under an instance of SERPENTINE that only presents to the *outer world* a few methods and attributes achieving true hierarchization capabilities.

2.4 Life-cycle Tools

The OSGi Service Platform [18] specification is an open, reference architecture to deploy, manage and deliver services in a coordinated and transparent manner. Its framework provides a general-purpose Java environment that supports the deployment of extensible and downloadable applications known as bundles. An OSGi-compliant implementation is capable of managing the bundles and its dependencies and control their life-cycle.

SERPENTINE builds on OSGi for two different purposes. First, it is able to run as an OSGi bundle and can thus be deployed remotely and in a fully automated fashion. Second, by using a JMX interface to the Apache Felix OSGi runtime, it can deploy and control arbitrary system components packaged as bundles, i.e. by user scripts and in reaction to monitoring events. That allows services to be started, stopped, installed and uninstalled at user demand or, with the adequate policies, according to the environment.

3. CASE STUDY

This section considers a realistic scenario to illustrate our system and its capabilities. SERPENTINE has been used successfully within the GORDA project [10], to monitor and control a cluster of replicated databases.

Roughly speaking, the GORDA project fosters database replication as a means to address the challenges of integration, performance and cost in current database systems underlying the information society. To achieve such goals, it proposes to standardize an architecture, a set of interfaces and sparks their usage.

This architecture has four main components: a load balancer, an augmented database with reflective interfaces [7], a replication engine [4] named ESCADA and a distributed management service.

The load balancer is responsible for spreading clients among replicas. ESCADA is a replication engine that uses group-based protocols [16]. Besides, it is also a JMX-enabled application and exports, among other things, database statistics that are used to achieve decisions on the life-cycle of the cluster based on desired policies as we shall explain.

The resulting distributed is then managed by SERPENTINE. In particular, it controls the life-cycle of all components by installing, upgrading, starting, stopping and pausing them when requested.

3.1 Scenario description

In this scenario, we have four machines each one with Apache Felix, ESCADA, and a database (e.g. PostgreSQL-G [2], Derby-G [1]), constituting a replica. Then we want by means of SERPENTINE to enforce the following policies:

- at least two replicas are running;
- if one or more replicas are overloaded then start a new replica (database);
- if one or more replicas are idle then stop them.

Additionally, SERPENTINE must also export an aggregated set of statistics about the replicas in order to be pretty-printed by another component used in-house: an adapted version of JManage [3]. In this case, these results are a simple arithmetic mean.

The life-cycle of the ESCADA engine is controlled using the `start` and `stop` operations of the Felix JMX enabled shell. All the handling of the replicas is of exclusive responsibility of the ESCADA engine and thus SERPENTINE does not need to have concerns about recovery and consistency when invoking the aforementioned operations.

When the ESCADA is up and running, SERPENTINE starts gathering statistics periodically and adapting the overall system taking into account the state of each replica. The scripting language chosen was in this case JavaScript.

The first step is to write the metadata as stated in Section 2.3.1. A small snippet of the metadata for this scenario is shown in Figure 3.

The first line instructs SERPENTINE to establish a connection with the specified machines (i.e. replica). The next one indicates that the MBean `TabUI:name=jmxshell` exported by the machine `s1` defined in the first line will be accessible from the script as an object named `shell1`. The last line indicates that the script will run every 60 second. The other machines and MBeans should be specified in the same way but we will left that out to save space.

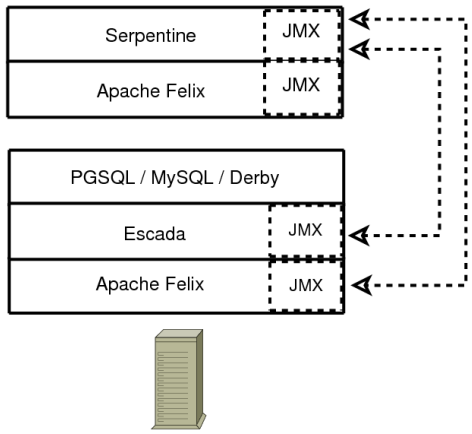

Figure 2: GORDA architecture overview.

```
//Machine 192.168.180.81:1100/coreShell s1
//MBean TabUI:name=jmxshell s1 shell1

//Period 60000
```

Figure 3: Script metadata.

The next step is to write the management logic necessary to enforce the desired policies. This is accomplished by determining the global state of the system and then adapting its components (replicas) accordingly.

The global state is determined by traversing all the machines and adjusting some variables that reflect the running system as a whole These variables are show in Figure 4.

Each variable represents a property of the overall system and the `machineState` variable, in particular, represents the state of each replica that can take one of the following values:

- 0 when the machine is disconnected/unreachable,
- 1 when the machine is in its *normal* state,
- 2 when the machine is idle,
- 3 when the machine is overloaded.

The other variable worth mentioning is `machinesToStart` and represents the number of machines SERPENTINE must (try to) start/stop in order to enforce the policy. Next we write the script logic to which the system must abide.

In Figure 5 is the portion of code responsible for building the overview of the system by consulting each machine and defining its state according to the values aforementioned. This is done in the `getStatus()` function by checking the connection status and getting the relevant attributes if connected. Additional checks are done to ensure the integrity of the read values by using the `getValue()` function and rechecking the state when exiting. Some low level functions like `attachEscada()` or `getValue()` are omitted because they are not relevant for the presentation of our case study.

After having a global overview of the system it is very simple to enforce the desired policies by calling the adequate actions on a per machine basis attending to the global state of the system. As seen in Figure 6 each machine state is tested

```
maxConnsPerReplica = 20;
machinesAvailable = 4;

var machinesToStart = 0, machinesUp = 0, machinesIdle = 0;
var totStarted = 0, totError = 0, totSuccess = 0;
var machineState = new Array(machinesAvailable);
```

Figure 4: Global variables.

```
//gets the state of a replica and adjust the global
//variables as appropriate
function getStatus(index){
var sensor = attachEscada(index);
if (sensor == null) return ;

if (sensor.connected()) {
 //get statistics from the database
 totStarted += getValue(sensor.getAttrib("StartedMessages"));
 totError += getValue(sensor.getAttrib("ErrorMessages"));
 totSuccess += getValue(sensor.getAttrib("SuccessfulMessages"));

 machineState[index] = getState(sensor.getAttrib("Connections"));
 //this condition is to prevent errors in the situation that
 //the machine may be disconnected while executing this function
 if (machineState > 0 ) {
  machinesUp++;
 }
}
else { machineState[index] = 0; }
}

//sets overview of system by consulting each machine
for(i = 0; i < 4 ; i++) {
 getStatus(i);
}
```

Figure 5: Building the overview of the system.

against the global state and if appropriate it is stopped or started.

Finally, in Figure 7, the gathered statistics are resumed using simple arithmetic mean for each relevant value and stored into the exported MBean for use by JManage.

Although some details have been suppressed due to lack of space, this case study demonstrates the usage of SERPENTINE in real environment. As expected this approach simplifies and reduces the management concerns and efforts needed to keep the running system healthy just by writing a simple management script.

4. RELATED WORK

The automated management of configuration parameters using a JMX interface and scripts has been already proposed [9] by adding scripting to the standard interactive JMX console, the `jconsole`. Such simplistic approaches fail however to cope with complex large-scale systems, as they do not control the life-cycle of system components and are restricted to a single managed component at each specific time. They are also strictly client approaches, thus precluding complex cascaded deployments.

The JADE system [5] is also able to control component life-cycle by building on an Fractal component model runtime [6].The architecture however is less flexible than SERPENTINE's, since it requires a stateful central server which cannot be restarted or even temporarily disconnected. This has a major impact in the availability of the service. Furthermore, diagnostics and actions are performed using a custom protocol, thus reducing its applicability. Finally, adaptation rules are not scriptable and thus more hardly changeable by end-users.

SERPENTINE abstracts policies using an action-policy approach. Action-policies are defined in a way that specifies which actions to take in a given state typically using a if-then clause. In a more abstract level are goal-policies that

```
//start or stops machines as needed
for(i = 0; i < 4 ; i++) {
 if (machinesToStart > 0) {
  //machine is down and no machines are idle
  if ( (machineState[i] == 0) && ( machinesIdle == 0)) {
   start(i);
  }
 }
 //guarantee that at least two machines are left running
 if ((machinesToStart < 0) && (machinesUp >=2) ) {
  //machine is idle and isn't needed
  if ( machineState[i] ==  2) {
  stop(i);
  }
 }
}
```

Figure 6: Enforcing of the required policies.

```
//calculate overall system statistics based on gathered data
var startMean = 0, errorMean = 0, successMean = 0;

if (machinesUp > 0 ) {
 startMean = totStarted / machinesUp;
 errorMean = totError / machinesUp;
 successMean = totSuccess / machinesUp;
}
//retrieve the hashtable from the exported MBean
table = state.getAttrib("table");
//update it
table.put("startMessages",startMean);
table.put("errorMessages",errorMean);
table.put("successMessages",successMean);
//and send it to the Mbean
state.setAttrib("table",table);
```

Figure 7: Computing and exporting summary statistics.

by observing the running system and with a set of high-level goals can infer the actions to take to achieve those goals,[8]. This has not been considered in SERPENTINE yet, which so far has emphasized the architectural aspects, interoperability, and configuration by end-users in a familiar scripting language.

5. CONCLUSIONS

In this paper we have proposed a simple architecture for adaptive management middleware. This architecture builds on the standard JMX interfaces that are already available in a large number of middleware and application components making it easily deployable on virtually any Java system. Adaptation is done through established control techniques and its rules defined using a variety of well known scripting languages thus minimizing the learning curve. Finally, it uses a JMX-enabled but otherwise standard OSGi runtime to manage component life-cycle. SERPENTINE is stateless, it's meant to be used without any redundancy concerns as its recovery, in case of failure, is almost instantaneous. It does not therefore hinder the dependability of the managed system.

Finally, the SERPENTINE middleware is here illustrated using a real example in the context of the GORDA[10] project.

6. REFERENCES

[1] Implementation of the gorda interface in apache derby. http://gorda.di.uminho.pt/community/derbyg/.

[2] Implementation of the gorda interface in postgresql. http://gorda.di.uminho.pt/community/pgsqlg/.

[3] JManage: Open source application management. http://jmanage.org.

[4] ESCADA Replication Server. http://sourceforge.net/projects/escada/, 2007.

[5] S. Bouchenak, N. D. Palma, D. Hagimont, S. Krakowiak, and C. Taton. Autonomic management of internet services: Experience with self-optimization. In *Third International Conference on Autonomic Computing (ICAC 2006)*, 2006.

[6] E. Bruneton, T. Coupaye, M. Leclercq, V. Quéma, and J.-B. Stefani. The FRACTAL component model and its support in java. *Software Practice and Experience*, 36(11-12):1257–1284, 2006.

[7] N. Carvalho, A. C. Jr., J. Pereira, L. Rodrigues, R. Oliveira, and S. Guedes. On the use of a reflective architecture to augment Database Management Systems. In *Journal of Universal Computer Science*, 2007.

[8] Y. Diao, J. Hellerstein, S. Parekh, and J. Bigus. Managing web server performance with AutoTune agents. *IBM Systems Journal*, 42(1):136–149, 2003.

[9] D. Fuchs. A BeanShell plugin for JConsole, 2006.

[10] GORDA Consortium. GORDA: Open Replication of Databases. http://gorda.di.uminho.pt/, Oct. 2004.

[11] M. Grogan. Scripting for the Java Platform. Technical report, Java Community Process (JSR-223), 2006.

[12] C. Karamanolis, M. Karlsson, and X. Zhu. Designing controllable computer systems. In *HOTOS'05: Proceedings of the 10th conference on Hot Topics in Operating Systems*, pages 9–9, Berkeley, CA, USA, 2005. USENIX Association.

[13] J. Kephart and D. Chess. The Vision of Autonomic Computing. *IEEE Computer*, 36:41–50, 2003.

[14] E. McManus. Java Management Extensions. Technical report, Java Community Process (JSR-003), 2006.

[15] E. McManus. Web Services Connector for Java Management Extensions Agents. Technical report, Java Community Process (JSR-262), 2007.

[16] F. Pedone, R. Guerraoui, and A. Schiper. The Database State Machine Approach. In *Journal of Distributed and Parallel Databases and Technology*, 2003.

[17] Sun Microsystems. Java dynamic management kit. http://java.sun.com/products/jdmk/index.jsp, 2007.

[18] The OSGi Alliance. OSGi service platform — core specification. http://osgi.org/osgi_technology/download_specs.asp, Aug. 2005. Release 4.