Conference on Electronics, Telecommunications and Computers – CETC 2013

Rules-based self support operation in complex infrastructures

Eduardo Villoslada de la Torre, Andres Sedano Frade

*Telefonica Investigacion y Desarrollo, Parque Tecnologico de Boecillo, Valladolid, 47151, Spain

Abstract

High availability in complex distributed systems is a challenge that has to be considered not only at design time, but also at execution time. This paper proposes an automatized monitoring architecture, event driven, based in rules with the goal of minimizing systems unavailability. Translating the expertise that operators have to rules, and providing an appropriate interface for services operation, allow us to execute self-healing actions to anticipate or correct fault cases as quickly as possible.

© 2014 The Authors. Published by Elsevier Ltd. This is an open access article under the CC BY-NC-ND license (http://creativecommons.org/licenses/by-nc-nd/3.0/).

Peer-review under responsibility of ISEL – Instituto Superior de Engenharia de Lisboa, Lisbon, PORTUGAL.

Keywords: Rules engine; self management; distributed systems; event processing

1. Introduction

Actual applications and services achieve high complexity and scale complicating their deployment and infrastructure [1], and consequently their management. The need for all-time available services, without interruptions, makes monitoring, detection and failures recovery crucial in their operation.

A usual strategy is to use commercial monitoring tools, that either through general controls (% CPU, disk usage, delays, running services,…), either through specific controls, activate alarms when occurs a fact that affects to the service availability [2]. These alarms, through a control panel, or sending an alert as a sms or a support email, notify the problem to be solved. In a normal case, the problem is attended in a manual way (provisioning new machines, more disk, restarting services,…), but this solution can be high time-consuming and error-prone.

Although actions to perform require deep knowledge, most of the solutions usually can be represented as a set of rules that contains this expert knowledge (if X and Y are facts detected, then have to execute A, B and C actions). The solution proposed here applies a rules engine to the alarms in order to automatize operation tasks in such kind of distributed environment.

E-mail address: evdlt@tid.es, asedano@tid.es
2. Solution Architecture

Regardless of the deployment architecture of a service, both in simple cases and more in more complex solutions, the need to monitor and real-time controlling the service is very important [3], because at any time an unexpected behavior that compromises the service availability can appear [4]. The early resolution of problems in operation raises the quality of service offered. Thus, in this case, the goal tries to automatize the following tasks:

1. Detect operative failures.
2. Find a healing strategy.
3. Fix the failure

For this, the proposed solution applies the classical Control System Model to the operation scenario (figure 1). So, using several sensors (B), the state of the system is retrieved (C), and in the case some conditions coincide in time or a failure is detected, a system controller (A) acts on it. In the next iteration, sensors retrieve actual state info, and the process is repeated, again and again.

Translating and enriching this scheme to our solution, the process follows the next steps:

1. **Receive basic events:** The monitoring system receives basic events from multiple external sources using a common API. For instance, it could receive several events of different distributed components of the same service at the same time.
2. **Process Events:** The monitoring system, using different event processing strategies, generates complex events, with aggregate info.
3. **Analysis and Decision:** With all the available information about the state of the system, all expert knowledge decision rules are evaluated, and those fulfilled determines the actions to execute.
4. **Planning Actions:** Next, it is necessary to schedule the actions in the proper time.
5. **Performing Actions:** Finally, the planned actions are executed through a service operation API; this generic API expose common operations for any service, but it is possible to use for sending particular operations for a concrete service.

Figure 2 shows the **proposed architecture** to perform all these tasks. Describing starting from the bottom up, as input for the Monitoring system, through an API REST endpoint, basic events are received, for instance:

- **Alarms:** Active monitoring tools (nagios[5]/icinga[6]), detect unusual states, and in addition to the actual treatment, they send the info to the automatization system.
- **Incidents inserted by final users or by support staff** can be an information source to identify possible problems; this information is usually managed by Trouble Tickets systems.
- **Trouble Shooting systems** [7] launch a periodic services diagnosis detecting execution conflicts.
- **In a high level,** each service can self-monitored its operation, retrieve and share own information.
Once different facts are obtained, the following step is to process them: in real time, with all the received events, actual and pass, **CEP techniques** (Complex Event Processing) are applied, such as filters, absence, patterns, correlation, hierarchy, aggregation, compose, enrich,… removing non relevant events and generating additional info, or through received event inference, either through external service requests.

Next, on the consolidated events, **decision rules** are applied, evaluating if, in the actual state represented by those events, some corrective action should be executed. These rules are easily defined in a declarative way, but it must be remarked that the proper rules definition, containing all the expert knowledge, is the core of the model. For instance, as active rules are allocated in memory, and it is necessary to maintain both actual and previous states, a bad definition of the rules, can cause performance problems. On the other hand, the volume of rules to process can overgrow, and in these cases probably have to implement segmentation strategies by set of services. This is a compromise between flexibility and efficiency: by segmenting gain in efficiency, but cannot be created inter-domain cross rules.

In conclusion, at this stage, based on the known facts (and as they change) continuously evaluates rule conditions to identify those that are met and, therefore, the tasks to execute.

Then, the different tasks to execute are **planned**, deciding the order and moment when these management requests are executed. Usually, it is interesting to run them immediately, but there are cases when it is necessary to differ in the time of execution, or some of the tasks are executed repeatedly several times.

Finally, requests to the services are sent through a predefined Operation REST API. The actions to do can be against the monitored service or against other common service, such as Trouble Shooting or Trouble Ticketing tools. In general, the complete model for the Monitoring system has an “effectors” catalog, allowing it to act on external services. This API allows requests both common operations, like those of a concrete service. For instance, in the first case, the API enables change service logging level, the verbose of outputs, open an urgent ticket, stop and start a server or replicate a node and reconfigure its corresponding load balancer.

The definition of a wide and effective service operation API is out of the scope of this paper, and there is plenty of related references [8–12], but it is another crucial element in the proposed model: the rules must be well defined, based on the maintenance operations that have the services monitored; without the ability to operate the services automatically, the model cannot be used with all its possibilities: if the rules established that under certain conditions have to reset the service, but it is not possible to be done automatically, the model is not fully exploited. The point here is that the API is more generic, more scenarios can be addressed.
3. Scenario example

There has been a proof of concept of the proposed model supported by the following technologies:

- The **events processing** are supported by Drools Fussion [13], and the history event persistence is management with a MongoDB [14].
- The **decision rules** environment uses Drools Expert.
- For **planning** the different tasks to execute and the flows management OptaPlanner is used (Drools module, too); it decides the order and moment when these management requests are executed.

Figure 3 shows an easy example to understand how the monitoring system can work:

- Service S is running on Server X.
- Icinga is supervising Server X.
- The Service Monitor API returns the average number of transactions per second executed by the Service in the last minute. The Monitoring System asks for this information every minute.
- The Operation API let, among other actions, a **clear cache operation** of the Service through an invocation.
- The Trouble Shooted System, can perform a diagnosis, obtaining general information of Server X and particular information of Service S.
- The Monitoring system has a rules catalog (in pseudo-language), such as can be seen in Table 1:

<table>
<thead>
<tr>
<th>Rule id</th>
<th>Rule description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RULE 1</td>
<td>WHEN Alarm ON in Server X AND TPS in Service S &lt; 700 THEN REQUEST DIAGNOSIS Service S NOW</td>
</tr>
<tr>
<td>RULE 2</td>
<td>WHEN TPS in Service S &lt; 400 THEN REQUEST OPEN TICKET Service S NOW</td>
</tr>
<tr>
<td>RULE 3</td>
<td>WHEN TPS in Service S &lt; 400 AND CDRs in Service S &gt; 1000 THEN REQUEST Service S (Clear Cache) NOW</td>
</tr>
<tr>
<td>RULE 4</td>
<td>WHEN TPS in Service S &lt; 200 THEN REQUEST Service S (Reset) NOW</td>
</tr>
<tr>
<td>RULE 5</td>
<td>WHEN Alarm ON in Server X (T-1) AND Alarm OFF in Server X (T) THEN REQUEST DIAGNOSIS Service S NOW AND REQUEST DIAGNOSIS Service S (T+1)</td>
</tr>
<tr>
<td>RULE 6</td>
<td>WHEN TPS in Service S &gt; 1000 AND EXIST TICKET THEN REQUEST CLOSE TICKET Service S NOW</td>
</tr>
</tbody>
</table>
Following the sequence diagram of figure 4 it is possible to observe the evolution of the events in the scenario environment. For instance, in one moment, the running monitoring done by Icinga, detects that the CPU from Server X has raised up to 95%, activating an alarm; this information is sent to the Monitoring system as an event using the REST API.

Every minute, the Monitoring system asks to the service the number of transactions per second, and in the following message receives that its value has fallen below 500 transactions per second, when the average in that slot is usually 1000 transactions per second.

Under these circumstances, the Monitoring system, when the state of the service changes, applies the rule 1 and enrich the information asking through the Trouble Shooting system a diagnosis of the service. The received response said that the memory allocated by the process has an excessive value, and the number of objects of CDR type is higher than 50000.

Next minute, the service returns again the number of transactions done, and has down below 300 transactions per second. This makes the rules 2 and 3 are triggered, associated with the service cache cleaning, that should be executed immediately, and opens an incident in the Trouble Ticketing system to keep track of it.

Following this, monitoring done by Icinga detects that the CPU from server x has fallen below 5%, so the alarm changes to inactive, and it sends this information to the Monitoring system. As the alarm state has changed from previous state, the rule 5 is triggered, and the Monitoring system asks for a new service diagnosis, using the Trouble Shooting system: now it obtains that the service is OK, and the number of CDR objects allocated in memory is below 500. The same rule set to request the Trouble Shooting system in a while.

Next minute, the service informs that the number of transactions done has increased over 1200 transactions per second. Consequently, the rule 6 is triggered to send a request to the Trouble Ticketing system to close the previous incident ticket. Finally, the previous planned diagnosis request is sent, and the response is OK again.

The example shows a simple case, but it is easy to extrapolate to more complex scenarios, with hundreds of physical and virtual server, distributed services and other event sources, such as automatic planned diagnosis or reactive healing treatment after final user incidence creation. Also, example rules are very concrete to facilitate the understanding, but in the complex case they can be generalized; for instance, the rule shown in Table 2.
Table 2. General Rule.

<table>
<thead>
<tr>
<th>Rule id</th>
<th>Rule description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RULE i</td>
<td>WHEN ALARM ON in Element A THEN REQUEST DIAGNOSIS Element A NOW</td>
</tr>
</tbody>
</table>

In this case, the rule can be interpreted as every time an alarm is activated over any monitored element, then a diagnosis is requested over that element.

4. Conclusions

The proof of concept done has served to validate the proposed rule-based architecture for automatic monitoring of systems. Applying all these steps (receive events, process them, apply rules, planning and executing task) in an iterative way, as in the Control System Model (figure 1), together with a proper operation rules definition and a complete Operation API, increases the availability of monitored distributed systems. In addition, as rules are defined in a declarative way, the maintenance and correction are quick and easy to perform.

The key elements of the model are the correct definition of the rules of expert knowledge and the availability of a complete operating API for the services. Therefore, further work will be strengthening these two elements: on the one hand, to establish a rules model and efficient configuration that provides good performance with a large number of rules; on the other hand, establishing a minimum operating API but complete enough to exploit all the required capabilities.

References