MIRRA: Rule-Based Resource Management for Heterogeneous Real-Time Applications Running in Cloud Computing Infrastructures*$\S$

Yong woon Ahn, Albert Mo Kim Cheng
Department of Computer Science
University of Houston
4800 Calhoun Road, Houston, Texas, U.S.A.
+1-713-743-3350
{yahn, cheng}@cs.uh.edu

ABSTRACT
Real-time software and hardware applications are attracting more attentions from many different areas of industry and academia due to exponentially growing markets of Cyber Physical System (CPS) and Internet of Things (IoT) devices. In order to satisfy high scalability requirements of data processing, storage, and network bandwidth from these applications, using cloud computing technologies has become one of the most cost-efficient and practical options. However, currently serviced public cloud computing technologies are originally designed for best-effort applications such as web services, and the cloud vendors’ service level agreements (SLAs) do not provide any application level Quality of Service (QoS) guarantees. In this paper, we propose a novel middleware platform, MIRRA, running between existing cloud computing technologies and real-time application servers. MIRRA provides multiple software layers to schedule real-time tasks by automatically scaling up and down virtual resources using a knowledge base with various rules, and its internal architecture consists of multiple subcomponents based on the autonomic computing architecture principles to implement the self-resource-adjustment design.

Keywords

1. INTRODUCTION
Cloud computing is not a new concept anymore. It has introduced new software development and deployment paradigms for operating virtualized compute, storage, and network communication resources elastically. These virtualized resources can be used for various application requirements to maintain their desirable Quality of Service (QoS) levels. Currently most public cloud computing solutions are designed to support best-effort applications such as web, database, and file storage services. These services provide auto-scaling solutions [1, 2, 3, 4] by monitoring system performance such as the amount of virtual CPU and memory usage. Since preserving compute resources is the most important requirement to process all tasks properly, these services are useful to implement cost-efficient server-side solutions for most best-effort applications. However, these auto-scaling solutions are not appropriate to support real-time applications because of two reasons. First, only limited types of resource condition can be determined by system administrators and monitored for scaling virtual resources up and down. However, there can be numerous conditions with different types of system metrics as well as application-specific performance requirements such as task deadlines and the maximum computation time to maintain desirable QoS levels for various real-time applications. Second, even though compute resources can be resized on demand, a real-time application can be entirely or partially halted by the inevitable service-down time for reconfiguring and booting new Virtual Machines (VMs) during the execution of the auto-scaling procedures. This unexpected system halt caused by resizing compute resources can be a serious issue for real-time hardware and software applications which require processing tasks at remote application servers.

In this paper, we focus on Internet-connected real-time applications which request the processing of real-time tasks in remote application servers running in the public cloud with metric-based auto-scaling solutions. Our target applications include remote patient monitoring systems [5, 6], real-time traffic control systems [7], drone navigate cloud platforms [8], and Internet of Things (IoT) devices requiring transmission of deadline-sensitive data and periodic task executions. We assume that the public cloud infrastructure provides proper security and data backup solutions with a Service Level Agreement (SLA) and mechanisms to fairly share its virtual resources among all its running VMs.

There are five important requirements in the design and implementation of these systems. First, our solution for these applications must provide automatic performance monitoring by identifying QoS issues. Second, our system must be capable to monitor real-time application servers to determine whether they miss task deadlines or not. Third, hard-real-time application servers must process all scheduled tasks. For soft-real-time applications, a limited number of tasks can be dropped, but the system must reconfigure its virtual resources to prevent failing tasks for the next service iteration. Fourth, an unknown number of heterogeneous real-time applications can be connected to their destined application servers after their tasks are registered and scheduled by our system. Fifth, our solution must provide a capability to monitor application-specific QoS conditions such as erroneous data communication and software bugs.

One good example with these five requirements is the real-time IoT, one of the big trends in information technology areas for industry and academia with wearable devices currently. Gartner’s report [9] expects that over 26 billion IoT devices will be connected by the year 2020. The amount of data produced by these devices can be immeasurable, and conventional data centers using physical server infrastructures will not be a cost-efficient solution for processing fluctuating data traffic from heterogeneous IoT devices. Internet-connected Cyber Physical Systems (CPSs) are also good examples, where each component has sensors and actuators operated by one or more real-time software or hardware applications communicating with remote application servers. In order to operate its physical actuators correctly and smoothly, transferred real-time tasks must be processed within their specific deadlines.

In this paper, we propose Middleware for Rule-based Resource

$^*$ Supported in part by the National Science Foundation under Awards No. 0720856 and No. 1219082.
$^\S$ Supported in part by Amerra Inc. for the Amerra Connect cloud platform.
Auto-scaling (MIRRA) which is implemented as a system service between a non-real-time guest operating system and application servers running in each VM. In order to satisfy these five requirements, MIRRA is designed to support four steps (monitoring, analyzing, planning, and executing) of the autonomic computing concept [10] by operating a knowledge base storing facts and rules. Our rule-based approach is more scalable and precise to determine the future and current resource requirement than existing system-metric-based auto-scaling mechanisms. MIRRA can monitor registered VMs and use rules to identify performance issues to provide proper solutions for the next service iteration. These rules are stored in a knowledge base using the modified RuleML [11] data type.

The remainder of this paper is organized as follows: In Section II, we introduce some previous research results for satisfying similar requirements in real-time applications and cloud computing. In Section III, we model a real-time task, and show a server-side middleware architecture with our proposed rule-based resource manager. In Section IV, we present our approach for implementing a practical system. In Section V, we show simulation results to evaluate our approach.

2. RELATED WORK

Liu et al [12] proposed an on-line scheduling algorithm for real-time services in cloud computing. Their algorithm modifies the traditional utility accrual approach [13, 14] to have two different time utility functions (TUFs) of profits and penalties on task executions. One important assumption for this research work is the timeliness using relative task deadlines. Although this research does not provide a proper solution for resource scaling, we use this timeliness concept to model a real-time task.

Xiao et al [15] introduced a new approach to design and implement auto-scaling using the Class Constrained Bin Packing problem. They developed a color set algorithm to deploy application servers in multiple VMs. Although their approach is reasonable and practical, it is only suitable for supporting non-real-time applications since it ignores task deadline constraints.

As we stated in the previous section, the most feasible solution for real-time applications in the cloud is the auto-scaling mechanism, which scales up virtual resources to handle real-time tasks to meet their deadlines and scales down to achieve cost-efficiency and high resource utilization. Mao et al [16] proposed an auto-scaling mechanism considering task deadlines and budget constraints. Although their idea is based on deadline constraints to overcome the downsides of system-metric based auto-scaling mechanisms, a system administrator still has to adjust the configuration file manually when a new real-time task needs to be scheduled.

In our previous paper [17], we introduced an autonomic computing approach for medial CPS devices running in private cloud infrastructures. In this paper, we mature this architecture to support more real-time application types and to design a rule-based virtual resource manager running in public cloud infrastructures.

3. SYSTEM DESIGN

As we briefly introduced in Section I, real-time applications running in the public cloud must be managed by specialized methods to schedule newly connected real-time tasks with protecting already scheduled tasks. If our system detects resource-scarce issues, it requires to increase the size of the corresponding resource type to schedule new tasks. Otherwise, it requires to reduce it to optimize virtual resource utilization. Figure 1 shows our system overview. Each real-time application consists of three general components: a software or hardware controller, sensors, and actuators. This controller has a role of transmitting real-time tasks to its application server and receiving processed operations to control its physical actuators if it has them. Sensors sample subject’s status periodically. Each VM runs one or more real-time application servers. A new task with its associated data is transmitted after MIRRA checks its task schedulability. Our current design uses the Earliest Deadline First (EDF) algorithm for task scheduling. Processed tasks can be responded to its originated real-time application if it is requested. We assume that each VM can reserve its assigned virtual resource fairly and surely using physical resource virtualization technologies.

Figure 1. The system overview: (1) registering a new task, (2) scheduling the task by checking the current resource size, and (3) the application starts sending tasks to the assigned application server

MIRRA consists of four subcomponents: Task Scheduler, Load Balancer, Autonomic Manager, and Message Parser.

(a) Task Scheduler checks task schedulability by EDF.

(b) Load Balancer relays received tasks to child VMs running the same application servers. We will discuss ours hierarchical VM structure with the four autonomic computing concepts to describe this child VM in the next subsection.

(c) Autonomic Manager manages the virtualized computing resources such as VMs, network bandwidth, and disk storages. In order to manage VMs, MIRRA can execute VM control operations include Launch, Suspend, and Terminate. We use several VM templates and guest operating system images when launching a new VM.

(d) Message Parser parses XML/JSON format messages delivered from real-time applications. This message includes instructions and raw data for scheduling a new real-time task, reporting deadline violations, and adjusting scheduled task properties.

In this paper, we focus on discussing Autonomic Manager due to the page limitations. Other components will be briefly introduced to support it in the following sections.

3.1 Real-Time Periodic Tasks

In this paper, we focus on a real-time periodic task which is a data stream from a single source real-time application. In our system, all real-time applications must submit its task properties to MIRRA scheduling their tasks with other already registered tasks. Each task
Each MIRRA runs an autonomic manager with four steps:

(a) In the monitoring step, an autonomic manager checks the states of processing assigned tasks such as task drop rates, CPU utilization, and memory usage. These states are pushed to its parent autonomic manager to let it reschedule tasks if it is required.

(b) In the analyzing step, an autonomic manager checks QoS issues by searching its knowledge base having multiple rules of previous resource usage histories and task drop rates, and it sends the policy change request message to the planning step.

(c) In the planning step, an autonomic manager searches its event-action rules to modify its resource management plan for the next service iteration.

(d) In the executing step, an autonomic manager executes the plan by calling VM management operations. We will discuss more about our knowledge base and rules in the following subsections.

Although every VM has MIRRA, only the parent VMs run it to manage its VMs. MIRRA in the child VMs can be activated when it receives a new task from a new App and requires launching a new child VM. This new VM also can be forwarded by its parent VM.

### 3.3 The Service Interval

In our system design, we define the service interval, $I$, for four steps of our autonomic manager. $I$ must satisfy

$$I = \{\max(d_i) \mid 0 < i < n\},$$

where $n$ is the total number of registered Apps. $I$ must be updated when a new App tries to send real-time tasks.

### 3.4 A Knowledge Base for the Planning

Each knowledge base stores various rules to manage VMs and their managed resource sizes such as the number of cores, the size of RAM, operating system types and versions, allowable network bandwidth, VM launch templates, etc. We can represent the current managed resource state, $R$, for the current service iteration, $j$, as

$$R_j = \{V_j, \Sigma C_j, \Sigma M_j, \Sigma B_j, T_j, META_j\},$$

where $\Sigma C_j$ is the total available computation time of child VMs, $\Sigma M_j$ is the total available memory size of child VMs, and $\Sigma B_j$ is the total network bandwidth. $\Sigma C_j$, $\Sigma M_j$, and $\Sigma B_j$ are used for making an immediate decision to check task schedulability used in the executing step shown in Figure 2. In the next subsection, we will discuss how to determine and use these values for applying rules for the planning. $V_j$ is the set of child VMs and a parent VM controlled by the same autonomic manager. $T_j$ is the set of already scheduled tasks. $META_j$ is the set of metadata not related to scheduling tasks. Each $V_j$ can be represented as

$$V_j = \{VM_k \mid k > 0, k \in \mathbb{N}\},$$

$$VM_k = (uCPU_k, uMEM_k, uB_k, uS_k, DR_k, AID_k, CF_k),$$

$$AID_k = \{App_m \mid m > 0, m \in \mathbb{N}\},$$

$$CF_k = (CPU_k, MEM_k, B_k, S_k, CL_k).$$

Each VM consists of the real-time CPU usage, $uCPU_k$, the memory usage, $uMEM_k$, the bandwidth usage, $uB_k$, and the storage usage, $uS_k$.
The text is about managing reasoning rules in a system. It discusses how an autonomic manager can determine if the current resource usage or performance is adequate or needs to be adjusted. The manager uses reasoning rules to decide whether to launch a new VM, resize the current VM, or do nothing. The rules are defined in a way that they can be dynamically generated and modified as needed.
reactions cannot be executed simultaneously, an autonomic manager chooses a higher priority rule to be executed for the next service iteration.

\[
\begin{align*}
\alpha R_e[OID: "2"]: & \quad \alpha R_e[OID: "3"]: \\
E & : "need\_more\_cores" & E & : "need\_more\_cores" \\
VA & : \{w:vm\_t\_1\} & VA & : \{u:resize\_t\_1\} \\
RE & : \{q:"available"\} & RE & : \{g:"available"\} \\
IF & : \{u\perp q\} & IF & : \{u\perp g\} \\
DO & : "launch\_vm" & DO & : "reboot\_vm" \\
L & : "launch\_for\_more\_cores" & L & : "reboot\_for\_more\_cores" \\
PR & : "\theta" & SC & : \{App0\} & SC & : \{App0\} & PR & : "\theta" \\
& & & & & & \\
\end{align*}
\]

Figure 5. Examples of the reaction rules for an event of "need\_more\_cores"

![Software Components Running in a Single VM](image)

3.7 Rule Conflict Resolution

Although each rule’s priority is manually assigned by a system designer, an autonomic manager possibly must select one or more rules from the enabled rules with the same priority. The same priority can be detected more frequently if the system designer assigns a small number of predetermined priorities. Moreover, it is sometimes very hard to assign well-proven priorities to numerous rules if our system is a distributed and clustered system. For example, rebooting with two different VM templates, VM\_t\_1 and VM\_t\_2, can have the same rule execution time and the same priority to reserve more compute resources or to cancel reserved resources.

To choose more efficient and proper rules to achieve higher resource utilization without missing deadlines, we provide two rule-conflict-resolution methods. For the first method, in a knowledge base, all rules are categorized by different types for different reasoning rules and by the resource type for the reaction rules. SC in (8) and (9) is used for specifying these categories. For example, \(r R_e[OID: "\theta"]\) and \(r R_e[OID: "1"]\) in Figure 3 are searched by the same event. However, \(r R_e[OID: "\theta"]\) is in the CPU category, and \(r R_e[OID: "1"]\) is in the Memory category. These categories are determined and sorted by a system administrator. If multiple rules are still chosen and make conflicts in the same category, we use the second method. For the second method, rules in the same category are sorted by the matching success score between reasoning and reaction rules. If a reaction conclusion works properly for the next service iteration, an autonomic manager adds one point to this just executed rule and sort its rule list again by rule’s score. Rules with higher scores would be chosen and executed first for the next iterations. If there is no score data for all rules, an autonomic manager chooses the first reaction rule in the rule list.

4. SYSTEM IMPLEMENTATION

We use OpenStack [18] Grizzly version and Jclouds [19] to implement our design. Since Jclouds supports multiple cloud services, we can replace a cloud provider to another without rewriting our codes. We assume that each VM runs possibly multiple application servers and MIRRA as a system service in a non-real-time guest operating system.

Each server has multiple sub-components as shown in Figure 8. Each guest operating system runs three indispensable services for other system components. JavaVM runs Jclouds and our autonomic manager written in Java. A web server supports common HTTP and HTTPS protocols to exchange RESTful API [20] messages for a message parser in Figure 6, and MySQL runs as a main data repository for a knowledge base and a message parser. Reasoning rules and reaction rules are stored in a RuleML format which is an XML type representation of rules. We modified a few original tags and data formats to specify our variables and constants. Figure 7 shows a reaction rule, \(\alpha R_e[OID: "2"]\), used in Figure 5.

```xml
<rule style="reaction">
  <event>"need_more_cores"</event>
  <label>"launch for more cores"</label>
  <oid>"2"</oid>
  <priority>"3"</priority>
  <if>
    <atom>
      <rel0>"available"</rel0><var>W:vm_t_1</var>
    </atom>
  </if>
  <do>
    <atom>"launch_vm"</atom>
  </do>
</rule>
```

Figure 7. An example of converting \(\alpha R_e[OID: "2"]\) into our modified RuleML

5. PERFORMANCE EVALUATION

In order to develop and evaluate our system, we setup our indoor test environment with a single access point supporting IEEE 802.11a/b/g/n protocols. For this evaluation, we only use the OpenStack Nova Compute architecture and the simplified MIRRA simulator to avoid any unknown overhead caused by numerous other subcomponents not related to task scheduling and processing. Each VM fairly shares the same physical CPU and memory using the OpenStack’s round-robin scheduler. We use one physical server machine to evaluate our approach to emulate the limited size of physical computing resources. Each VM is launched with 128 Mbyte virtual RAM, on core virtual CPU, and no local storage as a default VM template. We use the CirrOS cloud image [21] as a non-real-time guest operating system to run the MIRRA simulator. Each VM runs Netcat [22] shell commands to emulate the MySQL and the Apache web server accepting RESTful HTTP messages. The MIRRA simulator has multiple shell scripts using Linux’s cURL and Wget commands to send VM control messages and to retrieve its knowledge base.

In order to emulate real-time applications, we have written a Java application to send a task consisting of data and operation commands via HTTP. Task’s data amount is fixed, and tasks are delivered periodically. The test application server spends the fixed amount of time for one service iteration.
Table I shows a periodic real-time task used in this evaluation. Here, we only consider the computation time and CPU usages and skip other properties due to page limitations. Other task properties can be evaluated by the same method used for the CPU property. Each application server can reserve 1000 ms of the CPU time per one service iteration. Therefore, if we run five real-time applications simultaneously sending tasks to one application server, no task would be dropped due to a deadline miss. Figure 8 shows the total computation time of tasks from the real-time applications and the ideal number of VMs to process it theoretically. Every five seconds, one more real-time application (App) is added to send more tasks.

### Table I. A periodic real-time task used for the evaluation

<table>
<thead>
<tr>
<th>Task Properties</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Start Time: S</td>
<td>0 ms</td>
</tr>
<tr>
<td>Computation Time: C</td>
<td>400 ms</td>
</tr>
<tr>
<td>Task Deadline: d</td>
<td>2000 ms</td>
</tr>
<tr>
<td>Task Period: p = d</td>
<td>2000 ms</td>
</tr>
<tr>
<td>Number of cores: O</td>
<td>1</td>
</tr>
<tr>
<td>Required memory: M</td>
<td>10 Mbyte</td>
</tr>
<tr>
<td>Network bandwidth: B</td>
<td>10 Mbps</td>
</tr>
</tbody>
</table>

![Figure 8. The total computation time to process real-time tasks successfully according to variation of the number of real-time applications](image)

![Figure 9. The number of VMs changed by adding new apps every five seconds without MIRRA](image)

![Figure 10. The number of VMs changed by adding new devices every five seconds with MIRRA](image)

Even though we tried to remove most expected overhead or QoS downgrade sources from our simulation environment, this ideal number of VMs could not be realized because we still need to use a non-real-time host operating system for OpenStack. Since this non-real-time operating system runs multiple system services such as disk I/O and network management daemons, possibly these services can interrupt OpenStack processes.

First, we ran test tasks without MIRRA. All VMs accept tasks by using their first-come-first-serve policy. Our test system without MIRRA uses a metric-based auto-scaling mechanism to operate VMs just like other public cloud service providers. In our setup, this auto-scaling mechanism simply launches a new VM to accept the new tasks if the virtual CPU usage is over 70%. There is no admission control by checking task schedulability. If its cloud resource monitor detects 0% VM CPU usage, it terminates it and empties its incoming task queue by assigning the remaining tasks to other VMs. Currently, this approach is commonly used for most commercial cloud service providers and open source cloud solutions.

Figure 9 shows the number of VMs without MIRRA. Since its auto-scaling mechanism only uses the CPU usage of each VM, it launches at most three times more VMs to process the same task amount than the ideal usage of virtual resources shown in Figure 8.

Figure 10 shows simulation results with MIRRA. All applications must send their task registration messages to MIRRA before getting scheduled. From this simulation, the number of VMs are almost identical to the ideal case shown in Figure 8 because of checking task schedulability and launching VMs before receiving tasks.

However, an autonomic manager used for this case could make a delay to launch a new VM due to its knowledge base accesses of reasoning and reaction rules. For example, at ten seconds in Figure 10, our test system requires one more VM to have more computing power. However, due to this delay of checking task schedulability and booting up a new VM, the additional VM is launched after three seconds. We assume that dropping out a few tasks is acceptable before starting a new real-time application server.

### 6. CONCLUSIONS

Real-time software and hardware solutions are becoming much more popular technologies due to explosively growing demands of IoT and CPS devices. Since these devices commonly generate a huge amount of data and request compute-intensive real-time tasks at corresponding remote application servers, it has become necessary to use cloud computing infrastructures because of their cost-efficiency and easy-maintenance. However, most cloud computing technologies are designed to support only best-effort applications such as web services and cannot provide proper solutions for deadline-constrained real-time systems. In this paper, we focus on real-time applications with periodic tasks which are deployed in public cloud infrastructures.

As a solution, we introduced a middleware platform, MIRRA, which can process and monitor real-time tasks delivered to remote real-time application servers. MIRRA follows four steps of an autonomic computing architecture to implement a proper auto-scaling mechanism for real-time applications, and it can be installed on any VM with non-real-time guest operating systems working independently. Also, we introduced a concept of the reasoning role for identifying potential reasons for the QoS downgrade and a concept of the reaction rule updating a resource management plan to accept more tasks or change the size of managed resources. These rules are stored at a knowledge base maintained by each MIRRA.

To evaluate our approach, we simulate MIRRA under controlled conditions. From the simulation, we confirm that our approach works better for real-time tasks than other existing system-metric-based auto-scaling mechanisms used in most cloud infrastructures.
7. REFERENCES