Programming Interfaces for Realtime and Cloud-based Computing

Gregory Katsaros  
National Technical University of Athens, Greece  
Tommaso Cucinotta  
Scuola Superiore Sant'Anna, Pisa, Italy

ABSTRACT
Research in the fields of Grid Computing, Service Oriented Architectures (SOA) as well as Virtualization technologies has driven the emergence of Cloud service models such as Software-as-a-Service (SaaS), Platform-as-a-Service (PaaS) and Infrastructure-as-a-Service (IaaS). The appearance of different business roles according to this classification, potentially with differing interests, introduces new challenges with regard to the tools and mechanisms put in place in order to enable the efficient provisioning of services. Security, Quality of Service (QoS) assurance and real-time capabilities are just a few issues that the providers are trying to tackle and integrate within the new products and services that they offer. In this chapter, we make an overview of the approaches that aim to APIs for real-time computing. In the first part of this chapter, several Real-Time Application Interfaces will be presented and compared. After that, we will document the state-of-the-art regarding the Cloud APIs available and analyze the architecture and the technologies that they support.

INTRODUCTION

New generation interactive distributed applications, such as various multimedia, virtual collaboration and e-learning applications, have significant demands on processing, storage and networking capabilities, as well as stringent timing requirements. For these applications, the time at which individual computations and data transfers are terminated is as important as their functional correctness. Contrarily to the traditional domain of hard real-time systems, where a single violation of the timing constraints is not acceptable because it would lead to potential overall system failures and/or life losses, the mentioned applications posses instead soft timing constraints, whose violation leads to degradation of the offered Quality of Service and interactivity level.
Soft real-time applications, and especially multimedia ones, would greatly benefit from a real-time run-time support like commonly available on Real-Time Operating Systems. In fact, this type of OS usually provides those features that allow for a well-known, predictable and analyzable timing behavior of hosted applications: all kernel segments are characterized with well-known worst-case durations, scheduling latencies and interrupt latencies which may be
controlled through the appropriate tuning of the interrupt and process schedulers. A set of real-time scheduling policies is available for the system designer and programmers. Time may be measured (and timers may fire) with a high precision (typically sub-millisecond) while there are tools available for WCET estimation and (off-line) analysis, as well as for schedulability analysis. Unfortunately, such OSes are designed for embedded control applications, thus they imply serious constraints on the supported hardware and available high-level software infrastructures.

Nowadays multimedia applications are increasingly complex and they tend to be distributed, thus their development on a hard real-time OS may be overly prohibitive, due to the lack of such OS functionality as: compression libraries, the support for a wide range of multimedia devices and protocols, including the possibility to handle various types of media, and the availability of a complete networking stack.

On the other hand, General-Purpose Operating Systems (GPOS) constitute the ideal development platform for multimedia applications. Unfortunately, GPOSes are not designed to provide the run-time support that is necessary for meeting timing requirements of individual applications. Therefore, in recent years, various efforts have been done towards the integration of real-time technologies within GPOSes, particularly within Linux, for the availability of its kernel as open-source and its worldwide diffusion. Various drawbacks make the Linux kernel, as well as most of the GPOSes, particularly unsuitable for running real-time applications: the monolithic structure of the kernel and the wide variety of drivers that may be loaded within, the impossibility to keep under control all the non-preemptable sections possibly added by such drivers, the general structure of the interrupt management core framework that privileges portability with respect to latencies, and others. As a result, the latency experienced by time-sensitive activities can be as large as tens or hundreds of milliseconds. This makes Linux non-suitable for hard real-time applications with tight timing constraints. Though, soft real-time applications may run quite well within such an environment, especially when the original kernel is modified so to integrate the necessary real-time capabilities.

Furthermore, research in the fields of Grid Computing, Service Oriented Architectures (SOA) as well as Virtualization technologies has driven the emergence of Cloud service models such as Software-as-a-Service (SaaS), Platform-as-a-Service (PaaS), and Infrastructure-as-a-Service (IaaS). Moreover, the emergence of different business roles according to this classification, potentially with differing interests, introduces new challenges with regard to the tools and mechanisms put in place in order to enable the efficient provisioning of services. Security, Quality of Service (QoS) and efficiency are just a few issues that the providers are trying to tackle and integrate within the new products and services that they offer. In general, the SaaS provider offers an application as a service over a distributed environment to the end users. The PaaS provider offers a development platform and environment consisting of services and storage units hosted on the cloud, while the IaaS provider offers resources on demand depending on each deployed instance. The PaaS offerings include services for binding all the involved parties, in addition to the end user of the infrastructure. This includes adapting the application in order to be executed on a service oriented and distributed infrastructure, describing it in a machine understandable way and enabling it in order to be executed on virtualized environments. To this end, several Application Programming Interfaces have been published serving different layer of the Cloud architecture. In the following table (Table 1) we try to map the important ones regarding their characteristics.
Amazon and Vmware provide commercial solutions for IaaS layer (EC2 and vCloud). The first one offers packages for accessing and utilizing Amazon’s servers while with the vCloud you can build your own Cloud infrastructure. OCCI is a specification that is being developed for IaaS providing supported by the Open Grid Forum (OGF) community while Sun Cloud is the solution that Sun Microsystems published in the same context. From the other hand, there are APIs that are trying to cover the intermediate layer of the platform (PaaS) such as Microsoft Azure and Google App Engine. The first offers components and tools for developing and deploying applications within Microsoft’s data centers. It is commercial product that allows you to deploy Windows applications on Microsoft’s data centers. Google’s App Engine on the contrary, provides an API for developing and executing Cloud-based applications for free, with specific utilization boundaries and limits. Extension of those quotas can be achieved by joining the billing mode of that service. Finally, there are several cross platform Cloud APIs like jClouds, libCloud, deltacloud and others, that allow you to develop applications using various programming languages (e.g. Java, Python) and deploy them onto proprietary as well as open Cloud infrastructures.

In what follows we make an overview of the approaches that aim to APIs for real-time computing. In the first part of this chapter, several Real-Time Application Interfaces will be presented and compared. After that, we will document the state-of-the-art regarding the Cloud APIs available and analyze the architecture and the technologies that they support.

### Table 1: Taxonomy of the major Cloud solutions and APIs

<table>
<thead>
<tr>
<th>Providers/APIs</th>
<th>SaaS</th>
<th>PaaS</th>
<th>IaaS</th>
<th>Open Source</th>
<th>Commercial</th>
</tr>
</thead>
<tbody>
<tr>
<td>Amazon EC2</td>
<td></td>
<td></td>
<td>x</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>Vmware vCloud</td>
<td>x</td>
<td></td>
<td></td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>Sun Cloud</td>
<td></td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OCCI</td>
<td></td>
<td></td>
<td>x</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>Google AppEngine</td>
<td>x</td>
<td></td>
<td></td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>Microsoft Azure</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td>x</td>
</tr>
<tr>
<td>jCloud</td>
<td></td>
<td>x</td>
<td></td>
<td></td>
<td>x</td>
</tr>
<tr>
<td>libCloud</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td>x</td>
</tr>
<tr>
<td>deltacloud</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td>x</td>
</tr>
</tbody>
</table>
**Approaches, Implementations & Comparisons**

**POSIX**

POSIX stands for Portable Operating System Interface. Reference source not found and is the collective name of a family of standards, also referred as IEEE 1003 or ISO/IEC 9945, jointly developed by the IEEE Portable Application Standards Committee (PASC) and the Austin Common Standards Revision Group (CSRG) of The Open Group.

In 1998, the first real-time profile, IEEE Std 1003.13-1998, was published enabling POSIX to address real-time applications, even for embedded systems and small footprint devices. It must be said that support for most of these functionalities is not mandatory in a POSIX-conforming implementation and the 1003.1-2001 standard defines the X/Open System Interface (XSI) extensions that groups together several of these optional features in the so-called XSI Option Groups. A compliant XSI implementation has to support at least the following options: file synchronization, memory mapped files, memory protection, threads, thread synchronization, thread stack address attribute and size, and may also support a bunch of other option groups among whom: Realtime (grouping together asynchronous, synchronized and prioritized I/O, shared memory objects, process and range based memory locking, semaphores, timers, realtime signals, message passing and process scheduling), Advanced Realtime (grouping together clock selection, process CPU-time clocks, monotonic clock, timeouts and typed memory objects), Realtime Threads (grouping together thread priority inheritance and protection, and thread scheduling), Advanced Realtime Threads (grouping together thread CPU-time clocks, thread sporadic server, spin locks and barriers).

By means of the just listed features, the standard allows for a precise timing of real-time processes and threads on a system, and for a certain control over the scheduler configuration for individual processes and threads. This last aspect is critical: a real-time application should not compete for the access to the shared physical resources, first of all the CPU, with all the other non real-time processes on the system. In fact, POSIX defines a multi-queue priority-based regulated access to the CPU, where a set of system calls can be used in order to set the scheduling policy of each process to SCHED_FIFO, SCHED_RR or SCHED_OTHER:

- int sched_setscheduler(pid_t pid, int policy, const struct sched_param *param)
- int sched_setscheduler(pid_t pid)
- int sched_rr_get_interval(pid_t pid, struct timespec * tp)
- int sched_get_priority_max(int policy)
- int sched_get_priority_min(int policy)

The SCHED_FIFO and SCHED_RR policies allow the programmer to specify the real-time priority at which a process will be scheduled (by setting the priority field of the sched_param structure). The available priorities range for a given policy may be retrieved by using the sched_get_priority_min() and sched_get_priority_max() system calls. In case multiple processes are assigned the same priority, SCHED_FIFO simply schedules them in FIFO order, whilst SCHED_RR applies a Round-Robin policy with a fixed time-slice (whose duration may be retrieved using the sched_rr_get_interval() system call).
All other non real-time processes in the system have the default SCHED_OTHER policy, meaning that they are scheduled in the background of the real-time processes, allowing real-time processes to exhibit a far better responsiveness than non real-time ones.

In the context of cloud-computing and virtualized infrastructures, the POSIX API for real-time computing is relevant in those contexts in which a host Operating System (OS) is exploited as hypervisor for hosting multiple Virtual Machines (VM), along with the guest OSes running inside. In fact, in this scenario, each VM is seen by the host OS as a process, thus its scheduling policy and real-time scheduling priority may be manipulated by leveraging the POSIX API. For example, this kind of scenario is realized by the Kernel Virtual Machine (KVM) hypervisor on the Linux OS.

With the mentioned POSIX priority-based scheduling policies, the actual guarantees a real-time process receives depend basically on what other real-time processes run into the system at a higher real-time priority. The SCHED_FIFO and SCHED_RR POSIX scheduling classes have the heavy drawback that a higher priority process can indefinitely delay the execution of all other lower priority ones, and there is no run-time mechanism that ensures temporal encapsulation among real-time processes.

Such problem is mitigated by the SCHED_SPORADIC optional POSIX real-time scheduling class. In this model, the developer may reserve the CPU for a real-time task by specifying a budget, a period, a real-time priority and a low priority. The kernel runs the process at the specified real-time priority for an overall time duration not exceeding the budget in each time window as wide as the period. When the budget is exhausted, the process is downgraded to the low priority. The advantage of such a scheme is that, with proper refinements (Mark Stanovich et al., 2009), (Dario Faggioli et al., 2010) it is possible to rely on classical Fixed-Priority analysis for checking schedulability of the real-time processes on the system. Also, if an application tries to consume more CPU than the maximum value theoretically foreseen in the analysis, i.e., the reservation budget, it is downgraded to the low real-time priority. Therefore, it is possible to keep under control the maximum interference that a higher priority real-time process may have on lower priority ones.

Unfortunately, being optional, the SCHED_SPORADIC scheduling class is not widely implemented on POSIX compliant OSes. On Linux, an implementation of SCHED_SPORADIC has been developed in the context of the IRMOS European Project by Faggioli et al. Error: Reference source not found Error: Reference source not found Error: Reference source not found and made available as a kernel patch (Faggioli).

When using Fixed-priority scheduling, the optimum assignment of priorities is the well-known Rate-Monotonic, which needs knowledge of the entire set of real-time tasks running on the system, in order to set each one priority. Interfaces that are more suitable to open, dynamic real-time systems are based on EDF scheduling and they allow each application to request to the OS scheduling guarantees in terms of a minimum budget to be granted every application period. This model is also capable of theoretically saturating a single-processor system with real-time tasks, conversely to what happens with Fixed-priority scheduling. For example, such an API has been developed for Linux in the context of various academic projects, such as the ones summarized in what follows. (Mark Stanovich et al., 2009) (Dario Faggioli et al., 2010) (Faggioli)
The AQuoSA API

The AQuoSA framework (Luigi Palopoli et al., 2008), (OCERA – Open Components for Embedded Real-time Applications), (OCERA Project Deliverable D1.1 - RTOS), (S.Oikawa et al., 1999), Error: Reference source not foundenhances a standard GNU/Linux system with scheduling strategies based on the Resource Reservation techniques (Tommaso Cucinotta, 2008). AQuoSA features a well-designed C API allowing for an easy use of the EDF-based real-time scheduling capabilities available on the platform. This is exposed to applications in form of a set of header files and a dynamically loadable library that needs to be linked to the executables. The most important API calls, available after inclusion of the <aquosa/qreslib.h> header file, are the following:

- **creation and destruction** of resource-reservation “servers”:
  - `qos_rv qres_create_server(qres_params_t * p_params, qres_sid_t *p_sid)`
  - `qos_rv qres_destroy_server(qres_sid_t sid)`
  where parameters to be provided via the qres_params_t structure are a minimum guaranteed budget (Q_min) that is always granted once the reservation is accepted, a desired budget (Q) that is granted only if available, and the reservation period (P); once created, a server is identified via a qres_sid_t identifier;

- **attach and detach** of threads to the created server:
  - `qos_rv qres_attach_thread(qres_sid_t server_id, pid_t pid, tid_t tid)`
  - `qos_rv qres_detach_thread(qres_sid_t sid, pid_t pid, tid_t tid)`
  where the thread to be attached is identified by its POSIX process identifier (pid) and the Linux-specific thread identifier (tid), which can be obtained via the Linux gettid() system call;

- **dynamic setting and retrieval of scheduling parameters** for a server:
  - `qos_rv qres_get_params(qres_sid_t sid, qres_params_t *p_params)`
  - `qos_rv qres_set_params(qres_sid_t sid, qres_params_t *p_params)`

The FRSH API

The FRSH API Error: Reference source not found has been developed in the context of the FRESCOR project for the purpose of allowing writing complex embedded real-time applications which may be portable across multiple hard and soft real-time platforms and OSes. In fact, the FRSH API is available on the MarteOS, Partikle and Enea OSE OS Hard Real-Time OSes, and on the Linux General-Purpose OS. From an API and programmability perspective, the portability has been achieved mainly by realizing a POSIX-like common API layer, the FRESCOR Operating System Abstraction Layer (FOSA), which subsumes a minimum set of POSIX capabilities that are needed in order to realize complex, distributed real-time embedded applications.
When considering complex distributed Cloud Computing applications with real-time requirements, the FRSH API may be used in order to provide scheduling guarantees to Virtual Machines hosted on a Linux-based machine by means of the KVM hypervisor.

The FRSH API is divided into multiple modules. The most important ones are the following:

- **the Core Module** provides applications with fundamental real-time scheduling services;
- **the Shared Objects Module** allows for realizing atomic operations where the access to the shared resources is governed according to well-designed real-time protocols;
- **the Hierarchical Scheduling Module** provides applications with hierarchical scheduling capabilities, allowing for example to nest Fixed-Priority scheduling inside Resource-Reservation scheduling, etc.
- **the Distribution Module** provides services for distributed real-time applications, such as the capability to negotiate the allocation of resources atomically on a set of distributed physical nodes;
- **the Feedback Control Module** provides applications with adaptive reservation strategies, where the resource allocation can be varied dynamically following the actual instantaneous application workload;
- **the Energy Management Module** allows applications to specify their resource requirements in presence of dynamic voltage scaling capabilities of the CPU.

In what follows, we focus on FRSH Core Module. The reader may refer to Error: Reference source not found for a complete description of the FRSH API.

The most important functions of the FRSH Core Module are the following:

- **preparation** of the **scheduling parameters**, referred to as **contracts**, to be negotiated with the OS:
  - `int frsh_contract_init( frsh_contract_t *contract)`
  - `int frsh_contract_set_basic_params( frsh_contract_t*contract, const frsh_rel_time_t *budget_min, const frsh_rel_time_t *period_max, ...)`
  - `int frsh_contract_set_timing_reqs( frsh_contract_t*contract, const bool d_equals_t, const frsh_rel_time_t*deadline, ...)`
- **negotiation** of the **contract** with the OS:
  - `int frsh_contract_negotiate( const frsh_contract_t*contract, frsh_vres_id_t *vres)`
  - `int frsh_contract_cancel(const frsh_vres_id_t vres)`
int frsh_contract_renegotiate_sync(const frsh_contract_t*new_contract, const frsh_vres_id_t vres)

where, after negotiation, the granted reservation is referred to by means of a frsh_vres_t identifier, and its parameters may be also changed via the renegotiation function, if needed;

- **binding** and **unbinding of threads** to a negotiated contract:
  - int frsh_thread_bind(const frsh_vres_id_t vres, const frsh_thread_id_t thread)
  - int frsh_thread_unbind(const frsh_thread_id_t thread)

- **querying the run-time status** of the scheduler:
  - int frsh_vres_get_remaining_budget(const frsh_vres_id_t vres, frsh_rel_time_t *budget)

this function is useful for realizing anytime computing real-time algorithms, where some optional computations in the algorithm may be performed or suppressed depending on the availability of residual budget granted by the OS till the deadline.

**The IRMOS Real-Time Schedulers**

In the context of the IRMOS European Project, a new multi-processor real-time scheduler for Linux has been developed, with hierarchical scheduling capabilities. In this case, the real-time computing capabilities of the kernel are not exposed to applications through an API, but rather the Linux cgroup Error: Reference source not found virtual file-system is exploited. Therefore, the main operations related to real-time scheduling may be achieved as follows:

- **first of all**, the cgroup special file-system (and specifically the CPU controller) needs to be mounted somewhere on the system, e.g., for mounting it in /cg/ the following shell command can be used:
  - mount -t cgroup -o cpu cgroup /cg

- **for creating a new reservation**, a group must be created by creating a folder in /cg/, e.g.:
  - mkdir /cg/myresv1

- **for setting the scheduling parameters**, i.e., a given budget (a.k.a., runtime) and period, the corresponding quantities (in microseconds) need to be written into special entries available in the group folder, e.g., for creating a reservation of 10ms every 100ms:
  - echo 100000 > /cg/myresv1/cpu.rt_period_us
  - echo 10000 > /cg/myresv1/cpu.rt_runtime_us
  - echo 10000 > /cg/myresv1/cpu.rt_task_period_us
  - echo 10000 > /cg/myresv1/cpu.rt_task_runtime_us

where the cpu.rt_*_us entries decide the parameters for the entire group, comprising possible subgroups that one might want to create, whilst the cpu.rt_task_*_us entries decide the parameters dedicated to the tasks attached at this very level of the hierarchy

- **for attaching a task** to a group, its Linux TID (similar to the POSIX PID) needs to be written into the tasks special entry available in the group, e.g., if the TID of the task we're interested into is 1421:
- echo 1421 > /cg/myresv1/tasks
- for **detaching a task** from a group, its Linux TID needs to be written into the tasks special entry at the root cgroup level, e.g.:
  - echo 1421 > /cg/tasks
- finally, in order to **allow tasks to actually exploit real-time scheduling**, they need to be scheduled according to the needed POSIX real-time scheduling class (SCHED_RR or SCHED_FIFO), and a real-time priority needs to be assigned, e.g.:
  - chrt -r -p 20 1421

For the convenience of the programmer, the above operations are made available to C programs through an adaptation layer written in C implementing the AQuoSA API already detailed above, and to Python scripts via an adaptation layer written in Python realising a similar API.

**RTLinuxFree and RTAI**

- Error: Reference source not found
  RTLinux and RTAI (Setz) are projects mainly aimed to support real-time in the context of industrial control, and they do not generally target complex, distributed multimedia applications on Linux, nor general cloud-computing applications running in virtualized infrastructures. In fact, in these projects a real-time task is generally written as a kernel module for the Linux kernel, which may use a proper API in order to set its own real-time scheduling class and related parameters. Such an approach is perfectly viable for relatively simple control applications that need to periodically read information from a sensor, apply some control logic then drive an actuator, but it is not easy at all to realize complex real-time distributed applications in form of kernel modules.

**RTSJ**

To address the problems that Java has when used in real-time systems, the Real-Time Specification for Java has been developed. Among other features, RTSJ basically exposes at the Java programming level the POSIX real-time scheduling capabilities of the underlying OS. Also, in order to face with the well-known problem of the Garbage Collector that can interfere with Java applications in unpredictable ways, RTSJ defines immortal and scoped memory areas to supplement the standard Java heap. These areas allow applications to use memory without being required to block if the garbage collector needs to free memory in the heap. Objects allocated in the immortal memory area are accessible to all threads and are never collected. Because it is never collected, immortal memory is a limited resource that must be used carefully. Scoped memory areas can be created and destroyed under programmer control, through specific API available to the programmer.
Programming Interfaces in Cloud Computing

Amazon EC2 API

Amazon is considered to be the leader Cloud Provider in the current market. The Cloud solution offered by Amazon is called Amazon Web Services (AWS) and was initially launched in 2002. In principle AWS is a collection of Web services that together are forming the Amazon’s Cloud platform. The company offers a variety of Cloud related products such as Elastic Compute Cloud (Amazon EC2), Simple Storage Service (S3), Simple Queue Service (SQS), CloudFront, Simple DB and more. It is not in the context of this chapter to elaborate on the services and products offered by Amazon so we will only mention the basic characteristics of the two basic services: EC2 and S3, before elaborating on the Amazon EC2 API.

EC2 is a Web service that allows you to deploy and manage server instances with a variety of operating systems in Amazon’s Data Centers. It offers specific API and utilities to control the computing resources and manage the network’s access permission and therefore launch a custom application environment. Apart from the complete control that you have on the instances, scalability is another key feature of EC2 service. As the name of the service implies (Elastic), the computing capacity of each resource can be adjusted manually or automatically by the application through the provided API.

Amazon S3 service offers an on-line storage space that can be accessible by any individual or application on the Web. It appears as an alternative of local storage systems and is one of the major services of the whole AWS collection. Similar with EC2, Amazon provides an S3 API for storing and retrieving data from their Data Centers. In addition, Amazon is committed to provide 99.99% durability and availability of the stored data as described in the Amazon S3 Service Level Agreement.

For better understanding of EC2 service we will explain the terminology that Amazon uses for the specification of the related API. An instance is a virtual server that runs over the physical host within Amazon’s Data Centers. The templates from which the instances are been created are called Amazon Machine Images (AMIs) and include options of operating system or other properties than you can select in order to define the individual instance that you want to utilize. To access your instances you can use Amazon’s Elastic IP Addresses that are actually static IP addresses designed for dynamic Cloud computing. That IP address is associated with your account in Amazon and you can reconfigure to point to a different instance. Elastic Block Storage (EBS) is the persistent storage of each EC2 instance that keeps the state of the instance. In addition, you can create a snapshot of the state that will be stored within the S3 service for long term durability. Like presented in Figure 1, the basic flow when using the EC2 starts by finding an existing public AMI and customizing to our needs or building one from scratch. The next step is to create the instance for the AMI using the provided EC2 API. The result of this
bundling process is an AMI ID that you can use in order to launch as many instances of the AMI as you want. Finally, through the available tools of the API you can administer and use the instances as you want with any servers.

Amazon EC2 API provides access to EC2 Web service either using the SOAP API or the Query API. When using the SOAP Web Services messaging protocol, the interfaces are defined by a Web Service Description Language (WSDL) xml document. While the SOAP requests and responses in Amazon EC2 follow current standards, any programming language supporting those standards (e.g. Java, C++, C#, Python, Perl and Ruby) could be used within Amazon’s Cloud. The Query API interface is a REST-based interface that supports GET and POST method to perform any request. The Query interface is seems to be preferred by the Amazon’s developers and is provided for almost all AWS. Moreover, many browsers do not support the full range of HTTP methods, while all Web browsers can handle GET and POST requests.

VMware vCloud Computing Interface

VMware as a leading company in the field of virtualization released in 2009 the vCloud API (VMware vCloud), an OVF 1.0 standards-based Programming Interface for providing and consuming virtual resources from the cloud. The vCloud API was a result of a combined effort from VMware and its partners in order to deliver an easy to use Cloud Interface, extensible and based on various established open standards such as XML, HTTP, OVF etc. In more details, the vCloud API can be distinguished in two parts: the Administrative API which is used for creating, managing and monitoring resources, users and roles within a vCloud and the Users API which provides browsing and discovering resources operations as well as creating, modifying and deploying operating virtual appliances. The VMware vCloud API enables application developers to create clients of vCloud services using a RESTful application development style. vCloud API clients and servers communicate over HTTP, exchanging XML representations of vCloud entities.

The vCloud defines several resources and entities illustrated in Figure 2. All kind of objects (resources and other entities) are being described through XML. In addition, the vCloud API
elaborates and defines in detail with links the context of a resource, list other relevant resources, and define how they relate to the current resource. These links are the primary way that a server delivers information to a client about how to access and operate a resource.

**Figure 2: Resources entities in vCloud**

**Organization**: a vCloud can include more than one organization each one of them is a administrative superset of users, groups and resources.

**vDC**: A vCloud virtual datacenter (vDC) is a mechanism for allocating computing resources such as networks, storage, CPU, and memory. Within a vDC the resources are fully virtualized and either are reserved on demand or pre-defined through the SLA. There are Provider vDCs, containing the available resources of a vCloud Service Provider and Organization vDCs (Org. vDCs) that provide an environment where the virtual systems can be stored, deployed and operated.

**Catalogs**: Catalogs contain references to the organization’s virtual datacenters and media images. A catalog can be visible only to its creators or could be published and become visible to other members of the organization as well.

**Network**: Network represents the allocated network capacity of a vDC provider within the organization.
**User & Groups:** An organization can contain several Users or Groups that are being created from the Organization Administrator or imported from the Directory Service (LDAP). Roles and specific rights are being assigned from the Administrator for setting the users permissions.

**TasksList:** The tasks that are being created by the execution of long-running operations are kept within the TaskList of the Organization.

**Virtual Systems and Media Images:** A vDC can include several Virtual Systems and Media Images. References of those can be listed within the catalog, while the Media Images are stored in their native representation (ISO, floppy) and the Virtual Systems are stored as template based on the OVF 1.0 standard format. Those templates can then be retrieved from the catalog and be instantiated to become vApps. A vApp resource can contain even more than one Virtual Machine (VM) including the operational parameters such as: connection links between the VMs, the hierarchy of powering on and off the VMs, the End-user license agreements for each VM, the deployment terms and conditions that constrain the vDC resources consumption, access control and user permissions etc.

The vCloud API supports multiple operations for enabling the two-way transfer of Media Images and OVF packages between the client and the cloud. Those upload and download operations are implemented by POST and GET requests respectively. In this context, the vApp Lifecycle includes three major steps:

- Uploading the OVF package
- Instantiating the vApp template
- Deploying the vApp

![Figure 3: vApp lifecycle](image)
In order to create a vApp template an OVF package must be uploaded to the cloud through a vCloud API client. The steps that must be fulfilled are:

1. The client POSTs an initial request that specifies a name for the template, a transfer format for the data, and an optional description.
2. The server returns an unresolved (status="0") vAppTemplate document that includes an upload URL for the OVF package.
3. The client uses an HTTP PUT request to upload the OVF package descriptor (the .ovf file) to the upload URL.
4. The server reads the descriptor and constructs a complete vAppTemplate document (one that includes an upload URL for each file listed in the References section of the descriptor). While the server is constructing this document, the client makes periodic requests for it and examines the response for additional upload URLs. When the response contains any upload URLs beyond the one returned in Step 2, the template is complete.
5. The client uses HTTP PUT requests to upload each of the files.
6. If the OVF package includes a manifest file, the entire upload is validated against the contents of the manifest file.

The Instantiation process is a prerequisite to deployment and is based on the uploaded vApp Template. The template includes specific details of the vApp such as virtual disks that the cApp requires, CPU, memory and network connections that must be allocated by the vDC etc. A client can trigger the instantiation either using the default parameters of a vDC or he can override the vDC parameters with his own set. Instantiation parameters allow you to specify certain properties of a vApp, including: details of its vApp network, lease settings for the vApp, startup and shutdown parameters for the vApp. Instantiation parameters also include a way to indicate that any terms and conditions (such as license agreements) contained in the vApp have been accepted.

For the deployment and powering on a vApp a POST request is again being used. The client makes the request to the respective action/deploys URL and all VMs that are included within the vApp are being deployed. Similar requests for the operation of undeploy, power-off, reset, suspend and other are supported by the API.

**Google App Engine**

The introduction of Google into the Cloud computing technologies and the related marketplace was realized through Google’s App Engine. In contrast with other providers and solutions that implement IaaS (e.g. Amazon AWS), the App Engine is a PaaS system. In principle, Google App Engine is a platform through which a developer can deploy and execute web application to Google’s Data Centers. Currently the platform supports application written in Java and Python but can also serve other JVM related languages like Groovy, JRuby and Scala. Apart from the SDK that is available for both languages (Java and Python) there is also an Eclipse plugin provided for development. Moreover, the data handling at Google’s Cloud-enabled platform is provided with the Datastore API. In the case of Java development, the Datastore API stores and performs queries over data objects, known as entities. Each entity has a unique
identifier (key) and one or more properties, as values of specific data types. The Datastore supports Java Data Objects (JDO) 2.3 and Java Persistent API (JPA) 1.0 standard interfaces. When building Python applications the Datastore is implemented through an SQL-like language called QGL. It does not support Join statements while it is inefficient when queries span over multiple machines. In this context, GQL is not a relational database in the common SQL sense but such a modeling can be accomplished through specific mechanism provided by the API (e.g. ReferenceProperty()).

When it comes to build an application, Google App Engine API provides a development environment either for Java (Eclipse plugin) or for Python. This environment simulates Google App Engine (including local Datastore, Google accounts etc.) and gives you the ability to deploy (upload) the application directly to Google App Engine. As a result, the whole process includes the following steps:

- Develop the application through the provided tools and APIs
- Register an Application ID on Google App Engine through the Administration Console
- Uploading the actual application files
- Accesses the Web application using a specific URL based on the Application ID

Google offers usage quotas for deploying application for free with certain limits. Developers can always enable billing in order to extend their usage on CPU, bandwidth, storage and e-mails. An indication of the free provided usage is presented on the table below (Table 2):

<table>
<thead>
<tr>
<th>Quota</th>
<th>Limit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Apps per developer</td>
<td>10</td>
</tr>
<tr>
<td>Time per request</td>
<td>30 sec</td>
</tr>
<tr>
<td>Blobstore size (total file size per app)</td>
<td>1 GB</td>
</tr>
<tr>
<td>HTTP response size</td>
<td>10 MB</td>
</tr>
<tr>
<td>Datastore item size</td>
<td>1 MB</td>
</tr>
<tr>
<td>Application code size</td>
<td>150 MB</td>
</tr>
</tbody>
</table>

Table 2: Google App Engine free quota

Open Cloud Computing Interface (OCCI)

During the OGF25 in March 2009 Ignacio M. Llorente (UCM – OpenNebula) and Thijs Metsch (Sun Microsystems - RESERVOIR project) founded the Open Cloud Computing Interface Working Group (OCCI). At the following OGF26 and OGF27 the group was renamed to the current OCCI and the first results were presented. The goal of the OCCI-wg is to develop a clean, open API for 'Infrastructure as a Service' (IaaS) based Clouds. It has active membership of over 200 individuals and is led by four chairs from industry, academia, service providers and end-users. Several members are from commercial service providers that are committed to implementing the OGF-OCCI specification. OCCI will provide a slim and extensible RESTful based API. Every resource defined through OCCI will hold a
unique address using a Uniform Resource Identifier (URI). The API implements CRUD operations: Create, Retrieve, Update and Delete, each one mapped to HTTP verbs POST, GET, PUT and DELETE respectively. The types of resources that are currently supported are storage, network and compute resources and can be linked together to form a virtual machine with assigned attributes.

![Figure 4: Alignment of OCCI URI to IaaS Resources](image)

The Specification of OCCI is designed to be modular and is described to individual document deliverables:
- The OCCI Core & Models
- The OCCI Infrastructure Models
- OCCI XHTML5 rendering
- OCCI HTTP Header rendering

Currently the group is working on finalizing the specification while draft documents for all previous modules are available on SourceForge.

**Azure (.NET)**

The rapid evolution of Cloud Computing and the related technologies could not leave Microsoft out of this field. As a result, Microsoft published the Windows Azure solution as a Cloud service operating system. In principle, Azure is a service platform that allows the developer to deploy and run Windows applications and store data on Microsoft’s data centers. The developers could write applications with the common Windows languages (C#, C++, VB etc) through the Microsoft Visual Studio suite but it also supports Java, Ruby, PHP and Python. The Azure platform is consisted by three major parts: Compute Service, Storage Service and Application Fabric. The first, provides the appropriate interfaces and support for the applications developed with the pre-mentioned technologies and can have multiple instances deployed. All kind of applications can utilize the storage service and have access to data.
resources throughout a RESTful approach. The storage service provides BLOBs for storing binary large objects, tables and queues for managing data. For applications that are more demanding on data management, Windows offers the SQL Azure Database, a cloud-based data management system (DBMS). The system is based on the Microsoft SQL Server and offers a similar management environment within the cloud. The data can be then accessed using ADO.NET or other Windows access interfaces. The cloud-based infrastructure service of the Azure solution is implemented by the Application Fabric mechanism. To this end, each application can expose endpoints using the Service Bus component of the Application Fabric in order to be accessed by other application of the cloud or on-premise applications. The connection of a RESTful client towards an application is managed by the Access Control component of the Application Fabric mechanism. The developers can create applications with either Web role or Worker role, and define how many instances he wants to execute within the Windows Virtual Machines (VMs). Those VMs are not created by the developer but provided by the hypervisor that is specifically designed for use in the cloud. Applications with Web role are usually implemented with ASP.NET and are intended to accept and process HTTP requests using IIS. Apart from the Web applications, the Worker instances are batch jobs that interact with the Web role instances through the Storage Service.

**Sun Cloud API**

Sun Microsystems got involved into the Cloud technologies in 2009 by presenting their cloud computing infrastructure and API with the name Sun Open Cloud. Like the words imply, Sun’s Cloud system is an open source solution with the API published under the Creative Common license which in principle allows anyone to use it in any way. SUN’s Open Cloud is consisted by two core components: Sun Cloud Storage Service and Sun Cloud Compute Service. The first is a set of web services and WebDAV protocols that provide the functionality for accessing and storing data in various formats. It is also compatible with Amazon’s S3 API. The Sun Cloud Compute Service provides to the developer all the tools and interfaces to build and operate a data center in the cloud or as Sun names, a Virtual Data Center (VDC). The VDC offers an easy to use integrated graphical interface, accessible via any browser through which you can design an application running on various operating systems within the cloud (Windows, Solaris, Linux etc). This user friendly interface supports drag-and-drop features as well as specific APIs and command-line-client for provisioning compute, storage and network resources.

SUN’s Cloud API is a RESTful programming interface, with every entity be represented as cloud resources (compute, storage, networking components etc). The usage of the API is realized via HTTP protocol through the common GET, POST, PUT and DELETE requests. The API operates in the context of various resource types:

- **Cloud**: A top-level construct which groups all the Virtual Data Centers to which an API user has access.
- **Virtual Data Center (VDC)**: An isolated container which is populated with Clusters, Private Virtual Networks, Public Addresses, Storage Volumes, Volume Snapshots.
- **Cluster**: An administrative grouping of Virtual Machines, useful for access control,
copying or cloning, geographic isolation, and scripting automation.

- **Virtual Machine (VM):** A server.
- **Private Virtual Network (VNet):** A subnet, not connected to the Internet, which may be used to connect Virtual Machines within a VDC.
- **Public Address:** A connection to the Internet.
- **Storage Volume:** A storage resource which may be accessed via WebDAV and other storage protocols.
- **Volume Snapshot:** A snapshot of the state of a Storage Volume.

More information about the API specification can be found at the Kenai project wiki, where documentation as well as the actual source code and binaries are provided.

**Cross Platform Cloud APIs**

Apart from the Cloud Provider specific APIs there are several platform independent programming interfaces for Cloud Computing. Jclouds is a Java-enabled open source framework for developing applications, supporting many cloud-aware features. Using the framework you can develop applications for various Cloud Providers such as Amazon, VMWare, Azure and others. Deltacloud is an open source RESTful API, provided by Red Hat and is compatible with EC2, Rackspace, GoGRID and other Cloud Providers. Deltacloud offers drivers for every supporting Cloud as a translation layer between the client and the provider’s native API. In the same context, libcloud is a Python client library that allows your application to interact with various providers. It is distributed under the Apache Software License and serves through specific drivers provided by the API the most common Clouds in the market.

**CONCLUSION**

Through this chapter we documented the basic Application Programming Interfaces for integrating Real-Time capabilities to Linux kernels. Several approaches were presented and compared in the context of executing soft as well as hard real-time applications. From the other hand, the emergence of Cloud computing brought a revolution and a new perception on application development and execution. All major Service Providers of the global marketplace, sooner or later got involved into Cloud-enabled technologies and related products. In the previous section, we presented various solutions and APIs covering all three layers of Cloud service model (SaaS, PaaS and IaaS). After investigating all most of the APIs available, either commercial and open source, we concluded that while the last three years there is a vast explosion of Cloud solutions published, still many aspects should be further explored. One important drawback detected was that the Cloud technology available lacks in standardization. The field of Cloud Computing is quite immature and still under development. Every Service Provider publishes its own API to serve their Data Centers and there only a few open source
Cloud APIs. As a result, this affects interoperability in a very important way and the SaaS APIs (jCloud, libcloud etc) have to implement specific libraries and plugins for every individual Cloud (IaaS). When it comes to real-time capabilities in Cloud not much can be found in literature. The technology is relatively young and the Cloud-enabled APIs are focusing into the virtualization of resources, the deployment of applications and the control of the virtual environment. From the other hand, the integration of real-time capabilities into the Linux kernel is an active topic for more than a decade with several really promising results. In total, Cloud Computing, regardless the current deficiencies will remain the buzzword the following years and its capabilities will evolve to meet the requirements of the contemporary interactive and distributed applications.

BIBLIOGRAPHY


Mark Stanovich et al. (2009). *Defects of the POSIX Sporadic Server and How to Correct Them.*


Michael González Harbour et al. (2008). *FRESCOR Deliverable D-AC2v2 – Architecture and contract model for integrated resources II.*


Setz, J. Inter-Process Communication in RTAI and RTLinux. Saarland University.


