MANJRASOFT PTY LTD

Aneka Dynamic Provisioning

Aneka 5.0

Manjrasoft

This document describes the dynamic provisioning features implemented in Aneka and how it is possible to leverage dynamic resources for scaling up and down Aneka based Clouds. A general overview of the infrastructure is given with a focus on the interaction with the supported Infrastructure-as-a-Service (IaaS) providers or virtual machine managers.

Table of Contents

1 Introduction

1.1 What is Dynamic Provisioning?

Dynamic provisioning refers to the ability of dynamically acquiring new resources and integrating them into the existing infrastructure and software system. Resources can be of different nature: hardware or software. In the most common cases resources are virtual machines of software services and they identify two very well known segments in the Cloud Computing market: Infrastructure-as-a-Service and Software-as-a-Service.

Dynamic provisioning in Aneka is mostly focused on provisioning and controlling the lifetime of virtual nodes where to deploy the Aneka daemon and the container software. Hence, it specifically refers to the provisioning of hardware in the form of virtual machines whether they are provided by an IaaS provider such as Amazon EC2 or GoGrid or a virtual machine manager such as Xen Server.

1.2 Provisioning Scenario

Figure 1 describes the common scenario in which dynamic provisioning constitute an effective solution for real life deployments.

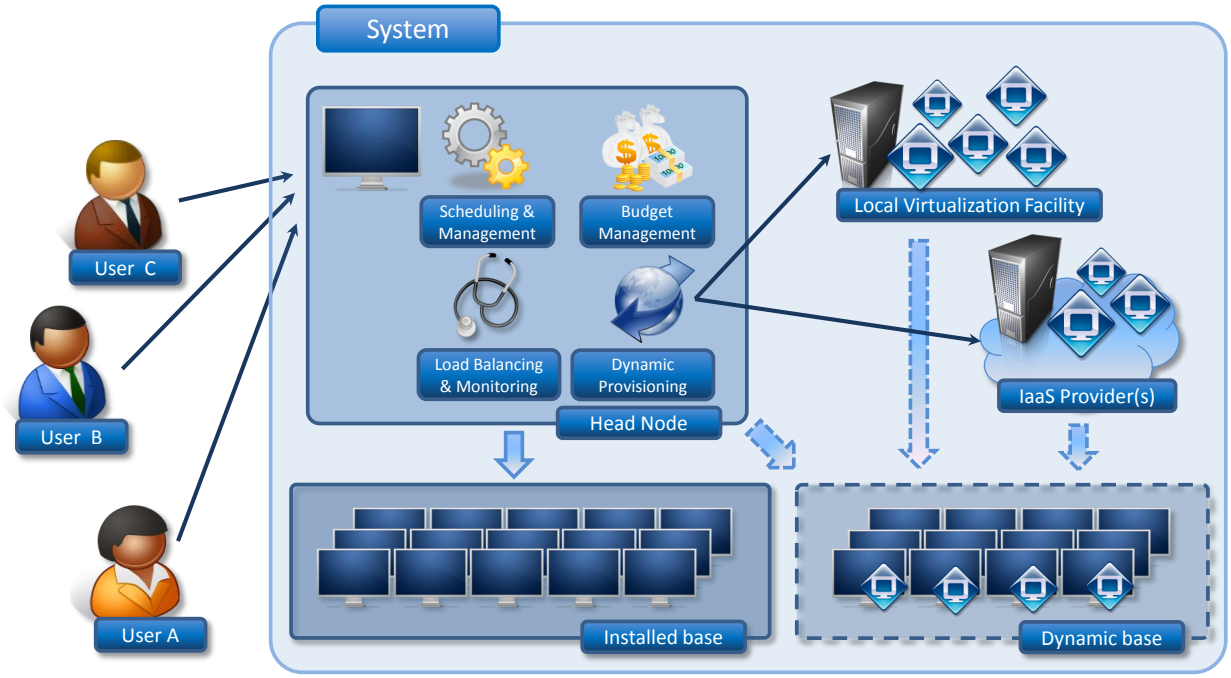

Figure 1 - Dynamic provisioning scenario.

A computing system, such as web application or platform for high performance computing such as Aneka, is normally configured with a reasonable installed base of computing nodes that in most of the cases are sufficient to address the performance and application

requirements of associated to the usual workload. There could be cases in which the installed base is not enough to cover the demand of users. These cases are normally identified by peak load conditions or very specific and stringent Quality of Service requirements for a subset of applications or users. Here is where dynamic provisioning can constitute an effective solution: by leveraging a dynamic base, commonly composed by a pool of virtual machines, the capacity and the throughput of the system can be temporarily increased in order to address the specific needs of users.

To make this happen, a collection of system components coordinate their activities:

- *Scheduling and management* facilities identify the need of additional resources and ultimately decide to provision them.

- *Load balancing and monitoring* facilities currently monitor the system load and the status of current installed base (physical and virtual nodes under management).

- *Budget management* facilities provide information about whether the system can afford additional resources given the requirements of users, their budget, and the system policies.

- *Dynamic provisioning* facilities actually deal with the technical issues of issuing the provisioning request and keep under control the lease of the managed virtual instances.

It is expected that the virtual nodes will be properly configured with the required software stack and virtual hardware to be integrated in the existing system. Whereas the virtual hardware properties can be generally defined while issuing the provisioning request, the software stack installed in the virtual machine can be statically configured in the template used to create virtual instances or dynamically deployed by the system once the node has become available.

Obviously, this is just a reference architecture but it is general enough to cover a huge variety of cases in which dynamic provisioning is used to enhance the performance and the throughput of the system. For example there could be cases in which the system only relies on a local virtualization facility that is freely available; in this case the provisioning process will be simpler and not conditioned by any consideration about the budget. But, since dynamic provisioning mostly relies on Infrastructure-as-a-Service providers that make available resources on a pay-as-you-go model, the common case includes an evaluation of the economical aspect of provisioning resources and whether it is of advantage to leverage them.

The dynamic provisioning infrastructure designed in Aneka has been designed to cover both of the two cases and its flexibility provides way to address new provisioning scenarios that might occur in specific deployments.

1.3 Common Terms

In this section we will briefly introduce some of the terms that will be used in the entire document and that are related specifically to dynamic provisioning and virtualization.

- **Virtual Machine (VM).** A virtual machine identifies a virtual node, where by node we refer to a physical machine, or computer, which has proper software stack installed on it including at least an operating system. In general a virtual machine only represents an abstraction of the computer hardware that is suitable for the installation of an operative system, but common uses of virtual machines require proper software stack already packaged and installed on the VM.

- **Virtual Machine Image, Virtual Image, or Virtual Template**. A virtual machine image or template represents the set of information that are used to create a specific virtual machine featuring a specific hardware configuration and software stack. This information is normally maintained as a binary file stored somewhere and it is used as a template that is cloned and completed with dynamic information whenever a virtual machine is created.

- **Virtual Machine Manager (VMM) or Hypervisor.** The virtual machine manager is the software infrastructure that is in charge of managing a virtual infrastructure. The tasks that are normally performed by a VMM are: creating and managing templates, creating and controlling the life cycle of virtual machines.

- **Local Virtualization Facility.** A local virtualization facility is a local installation of one or more virtual machine manager technology that is able to provide virtual infrastructure mostly constituted by virtual machines. The virtual infrastructure is generally available for free and subject to the physical capacity of the computing system backing the virtual infrastructure. In some cases the local virtualization facility is also referred as Private Cloud.

- **Pay-as-you-go (PAYG) or pay-per-use model.** PAYG identifies a specific pricing scheme in which an item, generally a service, is consumed under payment and the payment is proportional to the usage that the customer makes of the service. PAYG models generally bill customers according to some specific parameters that vary as the service gets more used. Examples are rented time for a virtual machine, data transfer volume for virtual storage, number of transaction for a database service.

- **Infrastructure-as-a-Service (IaaS).** Infrastructure-as-a-Service refers to a specific market segment in Cloud Computing where a virtual infrastructure including storage, computing, and networking can be consumed as a service according to specific pricing scheme and defined service level agreement. In the majority of the cases, the pricing scheme is characterized by a pay-as-you-go model. One of the key advantages of IaaS is the ability of dynamically providing virtual infrastructure on demand.

- **IaaS Provider.** An IaaS provider is a specific organization or company operating in the IaaS market segment that leverages some virtual machine manager technology for providing virtual infrastructure and making it available as a service to customer. IaaS providers generally introduce added value on top of the provision of virtual infrastructure by offering advanced services for the management of such infrastructure and ease of use.

- **Private Cloud.** A Private Cloud identifies a on-premise installation of virtualization facilities that are able to provide virtual infrastructure on demand and make it available as a service. The term private mostly refers to the fact that such facility is generally not publicly accessible and resides within the premises of the same organization that leverages its services. A Private Cloud can also be composed by several virtualization facilities as long as they are not publicly available.

- **Public Cloud.** A Public Cloud refers to the virtual infrastructure made accessible by an IaaS provider or a combination of them. What characterizes the Public Cloud is the fact that the virtual infrastructure provisioned does not reside within the premises of the organizations that use it. A Public Cloud generally includes a pricing scheme and charge users, there could be few cases in which the virtual resources is freely available.

- **Hybrid Cloud.** A Hybrid Cloud is the resulting distributed system obtained from the combination of at least a Public and a Private Cloud or a Public Cloud and the existing local infrastructure. By the term combination we refer to the fact that the resulting managed infrastructure is partially provisioned and/or the services made available within such infrastructure can be partially offered by third party organizations.

These terms cover the minimum knowledge that is required to understand the basics of dynamic provisioning in general and in the case of Aneka. This document is not intended to be a complete guide on dynamic provisioning, but focuses on the design and the architecture of the provisioning system in Aneka.

2 Architecture

Aneka integrates dynamic provision capabilities as part of its service model and dynamic provisioning is the result of the collaboration of several services and components that allow:

- Requesting virtual machine instances from a variety of virtual machine managers or IaaS providers;

- Dynamically detecting the need of additional resources to maintain the desired quality of service for application execution;

- Controlling the lease of dynamic resources to optimize their usage;

All these operations are transparently performed by the middleware without the user intervention and with a minimal configuration process during the Cloud setup.

In this section we will briefly recall the Aneka service model implemented in the Aneka Container and describe the architecture of the dynamic provisioning infrastructure.

2.1 Aneka Service Model

Aneka provides its services for application execution by deploying on each managed computing node a software container, called Aneka Container, which hosts a collection of services. The Aneka Container is in charge of managing the basic interaction with the hosting environment and operating system and provides accessibility to the services that constitute the "*intelligence*" and the "*horse power*" of the Aneka Cloud.

Aneka provides a large collection of services that cover all the needs for managing a distributed computing platform. Some of these services are part of the infrastructure and not of direct interest for end users others are primarily designed for interacting with users and applications. Among these services we can consider:

- *Scheduling and Execution services*: they are configured and managed as a pair and control the execution of applications.
- *Storage services*: they are in charge of the management of data and provide an internal storage for applications.
- *Reporting, Logging and Monitoring services*: they are configured and managed for collecting information about users, applications, and system components.
- *Security services*: they enforce a specific level of security in the Aneka Cloud.

Figure 2 provides a reference model of the Aneka Container together with some of the services that characterize a possible configuration. Aneka containers can be configured and deployed by using the Aneka Cloud Management Studio, which provides the basic configuration templates (*Master* and *Worker*) that are commonly used to setup Aneka Clouds as well as advanced capabilities for customizing these templates.

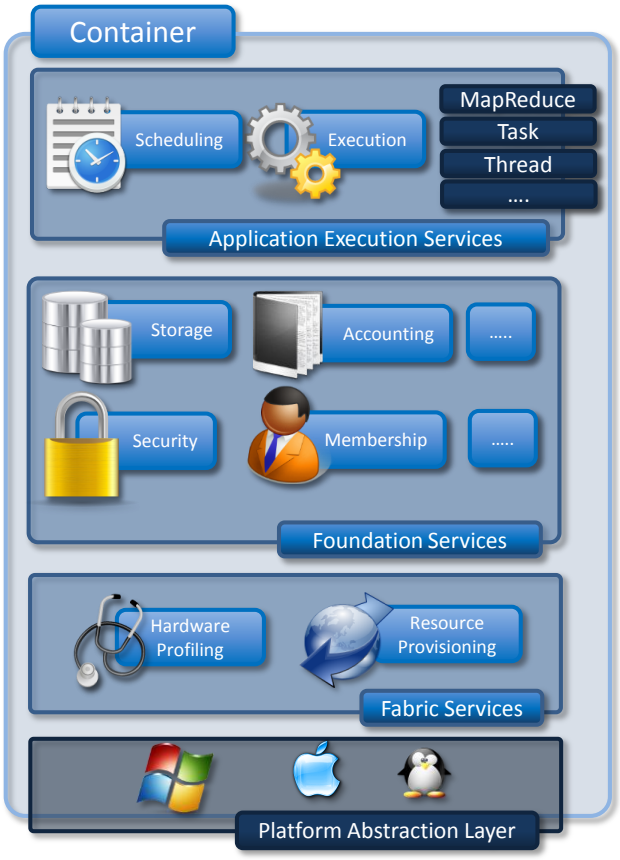

Figure 2 - Container architecture.

The installation and the configuration of the dynamic provisioning infrastructure mostly refer to the configuration of the Master template, and of particular interest are the *Scheduling Services* (*Task* and *Thread Programming Models*) and the Resource Provisioning Service. There are also other services involved in the provisioning of dynamic resources, but these services are not under the control of the user and operate as part of infrastructure. Hence, these services will not be discussed in this document.

NOTE: The MapReduce Programming Model has a different and separate collection of Scheduling, Execution, and Storage services. Such services, because of the specific implementation of MapReduce are not integrated with the dynamic provisioning infrastructure. Hence, the rest of this document applies for the case of provisioning virtual resources for executing applications that are based on the Task or Thread programming models.

2.2 Dynamic Provisioning Infrastructure Overview

Figure 3 shows an overview of the dynamic provisioning infrastructure implemented in Aneka.

The dynamic provisioning infrastructure is mainly the result of the collaboration between two services: the *Scheduling Service* and the *Resource Provisioning Service*. The former triggers the provisioning request according to the status of the system and the requirements of applications while the latter is in charge of making the provisioning happen by interacting with IaaS providers and local virtualization facilities.

The system is completely modular and can be fully customized to activate and control the behavior of dynamic provisioning. In Figure 3 the major components of the architecture have been pointed out.

- **Scheduling Service**: this service manages the execution of applications for a specific programming model. The *Task* and *Thread* programming models feature a scheduling service which is composed by a *Scheduling Context* and a configurable *Scheduling Algorithm*. While the scheduling context cannot be changed and deals with the internals of service implementation, the scheduling algorithm controls how applications are executed.

- **Scheduling Algorithm**: the scheduling algorithm is responsible for scheduling the execution of applications. More precisely, it is in charge of allocating a collection of tasks to the available set of resources in a dynamic fashion. There is one scheduling algorithm for all the applications developed for a specific programming model and it is possible to select different algorithms while configuring the service. Among the available options, there are two scheduling algorithms that support dynamic provisioning: *FixedQueueProvisioningAlgorithm* and *DeadlinePriorityProvisioning-Algorithm*. These two algorithms are the only two that leverage dynamic provisioning to cope with the application or system requirements.

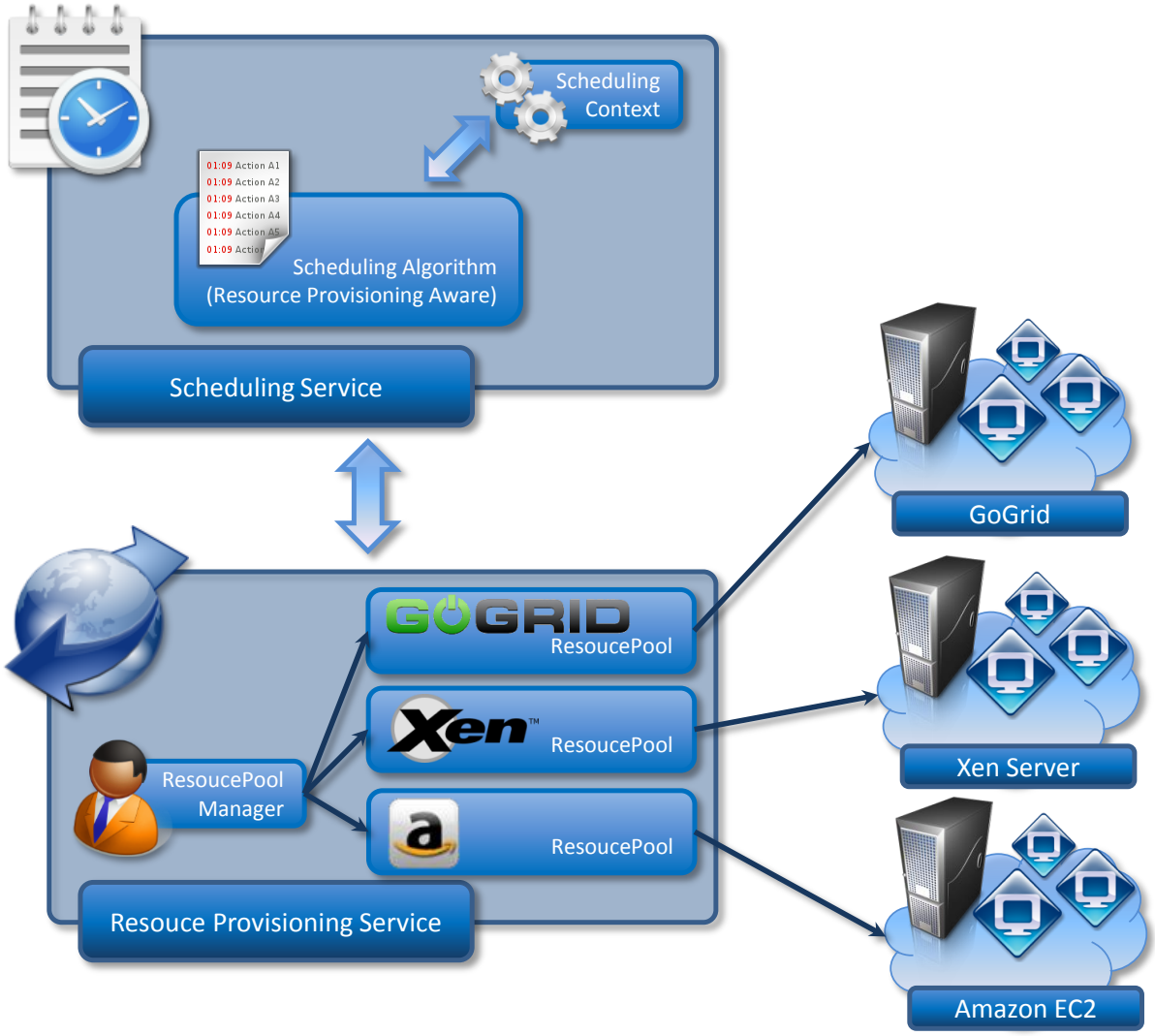

Figure 3 - Dynamic provisioning infrastructure overview.

- **Scheduling Context**: this component interfaces the scheduling algorithm with the service implementation. The algorithm uses the context interface to be notified about the events occurring in Aneka and to communicate scheduling decisions to the system. This interface has been designed to support dynamic provisioning and provides hooks that are used by the scheduling algorithm to ask for dynamic resources.

- **Resource Provisioning Service**: this service is responsible for satisfying a provisioning request. It can be considered as a black box that makes the provisioning magic happens. The service mainly performs the following operations: resource provision, resource release, resource status query, and resource pool status query. The current implementation of this service breaks down all the functionalities of provisioning into the following components: *Resource Pool Manager* and *Resource Pools*. The former is in charge of managing the pools by

delegating to them the specific technicalities required for dealing with a specific virtual machine manager or IaaS provider.

- **Resource Pool Manager**: the pool manager performs most of all the management tasks of the pools and is the place where dynamic provisioning strategies can be implemented. The main responsibility of the pool manager is controlling the life cycle of pools and redirecting a provisioning request, release, or query to the appropriate pool. The pool manager also notifies the provisioning service when a dynamic resource is activated and terminated.

- **Resource Pool**: the resource represents the last component in the provisioning chain and encapsulates all the technical details and specific protocols that have to be used to create virtual instances and manage them for a specific resource provider. The main operations performed by this component are: translating the provisioning request into a provider specific request; controlling the life cycle of virtual instances; and shutting them down when they are no longer needed. Aneka features three different implementations that allow the interaction with three different resource providers: *EC2ResourcePool*, *GoGridResourcePool*, and *XenResourcePool*. They respectively integrate *Amazon Elastic Computer Cloud (EC2)*, *GoGrid*, and *XenServer*.

This brief discussion provides an overview of the main components involved in the dynamic provisioning chain. The elements that are of major interest from an administrative point of view are the scheduling algorithm and resource pools, since are those that actually available for administrators to control the behavior of dynamic provisioning. The other components are mostly concerned with the development and the integration of new features within the dynamic provisioning infrastructure of Aneka.

3 Configuration and Management

Aneka provides a completely configurable and extensible architecture for what concerns dynamic provisioning. The previous section has highlighted the crucial components that are involved in the provisioning chain, in this section we will concentrate on the support given for configuring those components and managing them.

The configuration of the dynamic provisioning infrastructure can be performed by the Aneka Cloud Management Studio while configuring the services of the Master container. Figure 4 shows the steps that lead to the configuration of dynamic provisioning. In the Services page the administrator has to properly select and configure three services:

- **TaskScheduler**: this service controls the scheduling of applications based on the Task Programming Model and needs to be properly configured to leverage dynamic provisioning.

- **ThreadScheduler**: this service controls the scheduling of applications based on the Thread Programming Model and needs to be properly configured to leverage dynamic provisioning.

- **ResourceProvisioningService**: this service is generally selected by default but no resource pool is configured. Hence, dynamic provisioning is turned off.

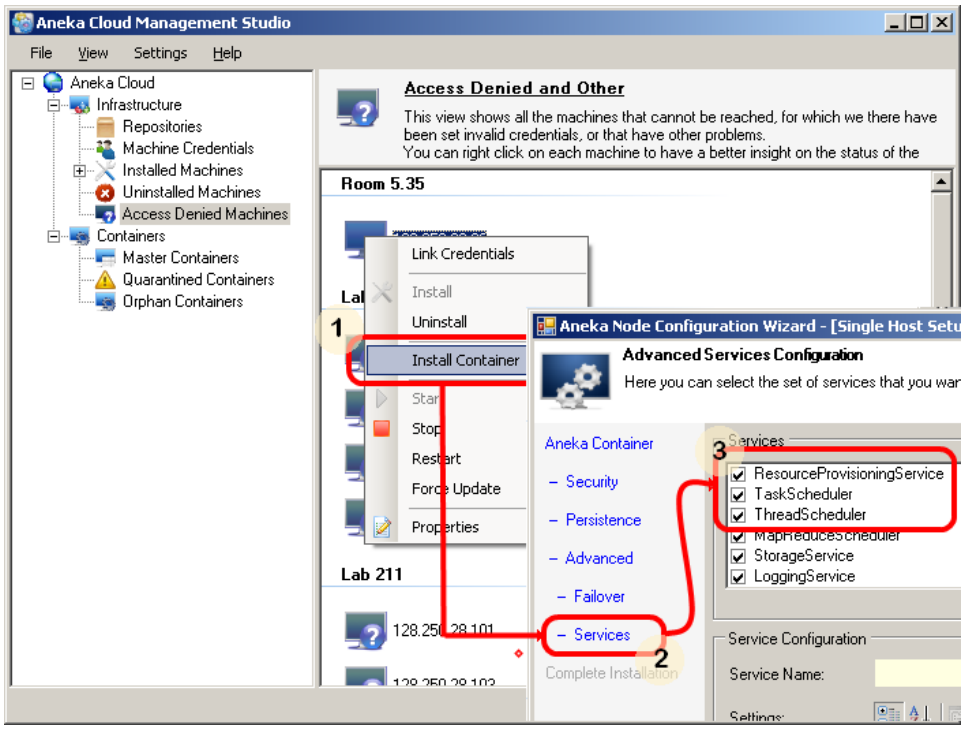

Figure 4 - Dynamic provisioning configuration.

The configuration of the two scheduling services is exactly the same and it is mostly concerned with selecting the appropriate scheduling algorithm while configuring the service. The configuration of the resource provisioning service instead allows the user to activate resource provisioning and select the collection of pools that service should leverage to provision virtual instances.

3.1 Scheduling Service Configuration

The configuration of the scheduling services only requires the selection of the proper scheduling algorithm. Two algorithms are available for using dynamic provisioning:

- **FixedQueueProvisioningAlgorithm**: this algorithm provides a programming model wide management for dynamic provisioning and it ensures that the queue of tasks or threads that are waiting to be executed never exceeds a threshold value (*GrowThreshold*). Also the algorithm allows for dynamic resource release when the size of the queue goes below a specific threshold value (*ShrinkThreshold*). This algorithm works across all the applications and makes use of provisioning for controlling the throughput and the performance of the entire system. Figure 5 shows how to select and configure this algorithm for the *TaskScheduler* service.

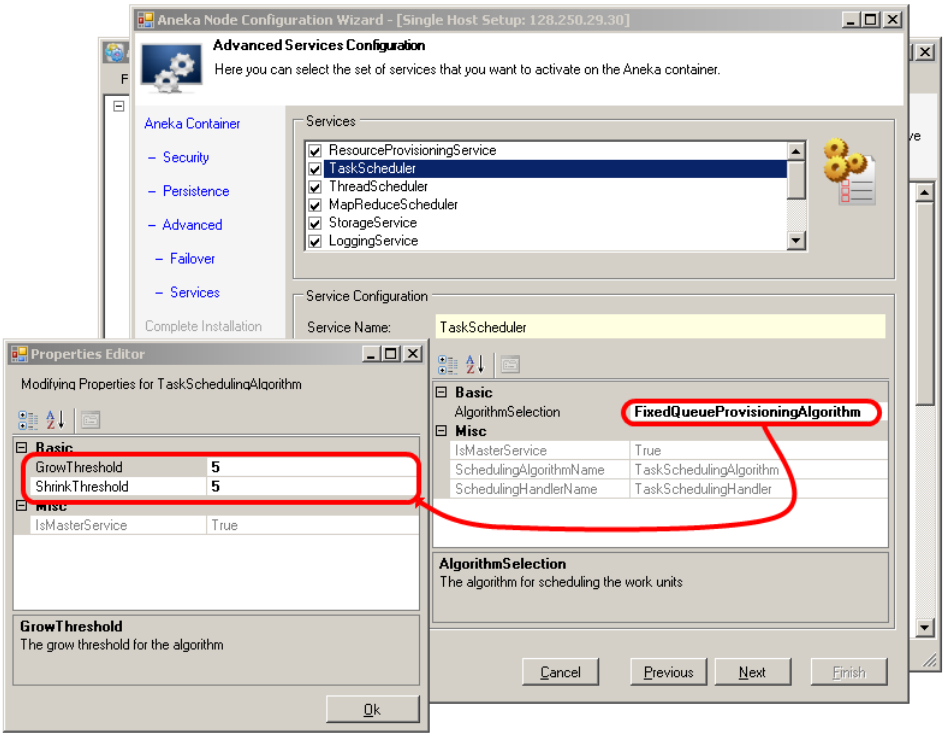

Figure 5 - Fixed queue provisioning algorithm configuration.

- **DeadlinePriorityProvisioningAlgorithm**: this algorithm is designed to leverage dynamic resources when the Quality of Service requirements of a specific application cannot be met. More specifically, the algorithm makes an estimation of the expected completion time of the application with currently available resources and if the expected completion time is later the deadline defined in the Quality of Service parameters of the application requests the required number of resources to complete the application in time. This algorithm works per application and provides a best effort strategy[1] for meeting the required deadline. Figure 6 shows how to set the algorithm for the *ThreadScheduler* service.

As shown by the figure this algorithm does not have any further parameter that needs to be configured.

[1] The strategy is best effort because if the provisioning fails or no resource pool is configured the deadline cannot be met.

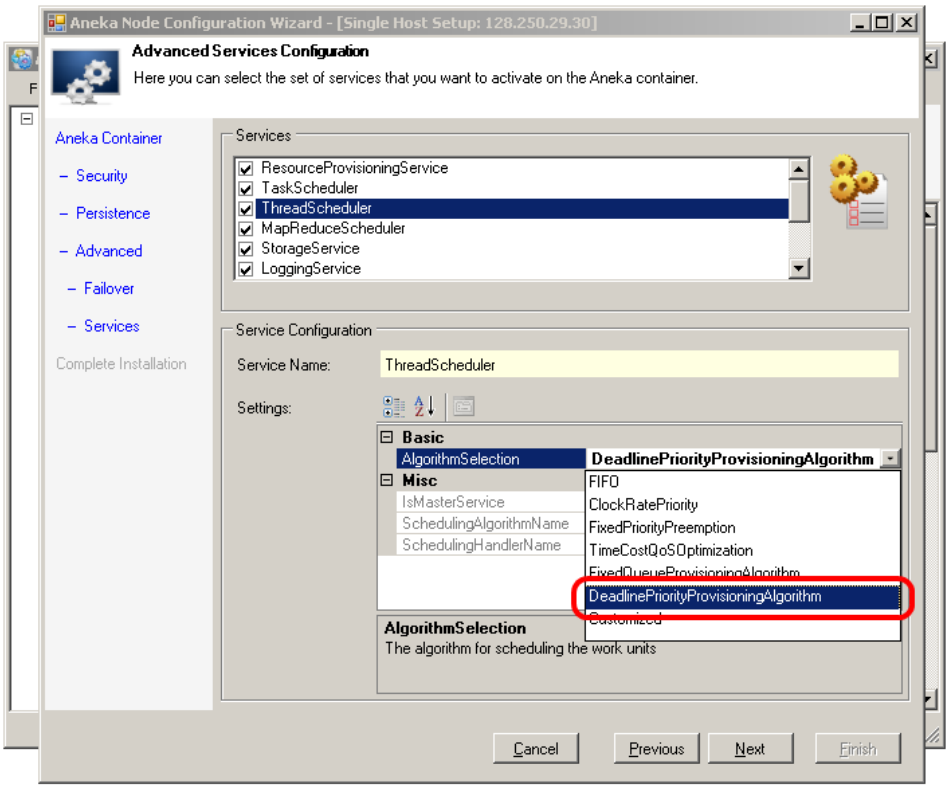

Figure 6 - Deadline priority provisioning algorithm selection.

As shown it is possible to configure the two different scheduling services with two different provisioning algorithms if the requirements for the two different programming models are different. There is no single scheduling algorithm that can leverage dynamic provisioning from a system wide point of view, this is because the management of the different programming models is separate and the two schedulers do not share the information about themselves.

3.2 Resource Provisioning Service Configuration

The configuration of the resource provisioning service relates to the selection and the configuration of the specific resource pools that the service will leverage while asking for dynamic resources. It is possible to configure the service with a collection of pools and to include multiple instances of the same pool with different configuration parameters. At the moment only three resource pools are supported:

- **XenResourcePool**: supports the provisioning and management of resources from XenServer.
- **EC2ResourcePool**: supports the provisioning and management of resources from Amazon EC2.
- **GoGridResourcePool**: supports the provisioning and management of resources from GoGrid.

In order to leverage such capabilities it is necessary to turn on the dynamic provisioning for the *ResourceProvisioningService* as indicated in Figure 7 and making sure that the service is selected.

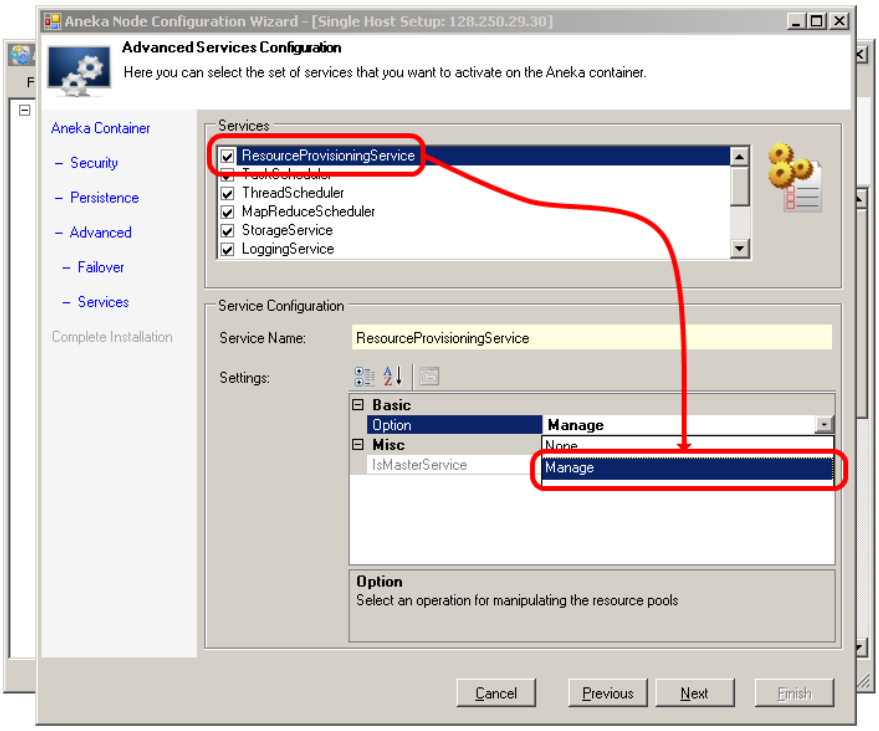

Figure 7 - Resource provisioning service configuration.

As shown in the figure, there is only one parameter for the configuration of the service: *Option*. This parameter is by default set to *None* that turns off dynamic provisioning. In order to activate it, it is necessary to select *Manage*. By selecting this option a new configuration editor is shown and it is possible to add resource pools to the service and configuring them with the proper information for interacting with the corresponding IaaS provider or virtual machine managers. As shown in Figure 8, in order to add a pool to the service it is necessary to select the type of pool in the drop down list, provide a name of the pool, and click the Add button. This operation will add the resource pool to the list of managed pools and give access to the configuration setting of the pool itself.

Once the pool has been added it is possible to change its configuration parameters by clicking the corresponding reference on the pool list on the left of the dialog. The pool will be referenced by using a string that is composed by the pool type followed by a colon and the pool name chosen by the user. In the case shown in Figure 8 the pool will be listed as: **XenPool:MyXenPool**. It is also possible to delete a pool by selecting and clicking the Delete button at the bottom of the dialog.

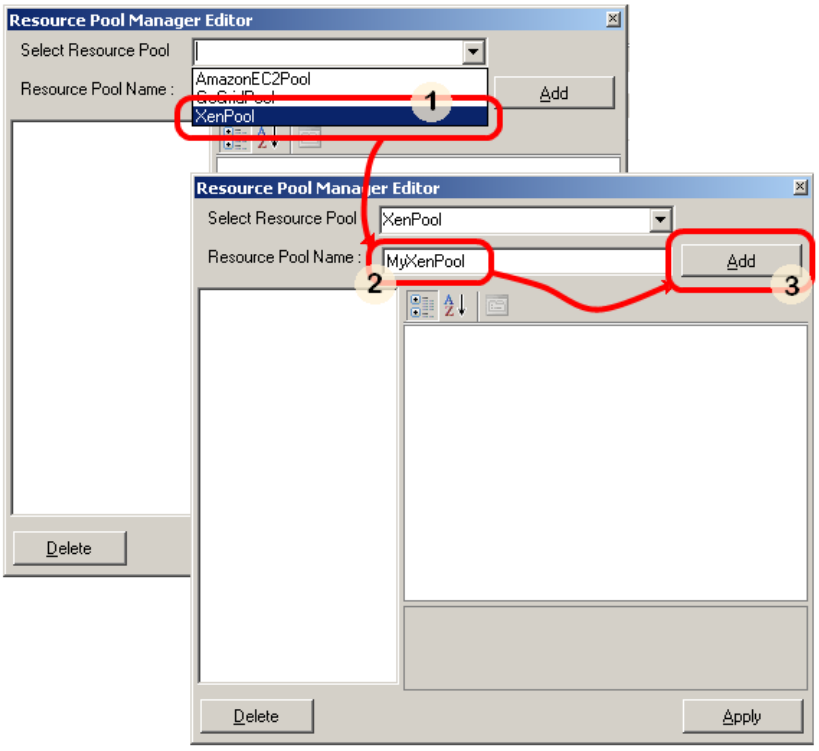

Figure 8 - Resource pool creation and configuration.

For each of the parameter settings of the pool at the bottom of the parameter list a small help text will be showed to help the administrator to understand the use of the parameter and properly configure it.

Once the creation and configuration off all the pools are completed it is possible to save the data by clicking the *Apply* button, otherwise if the user wants to cancel the operation it can simply close the editor by clicking at the close (top right corner) icon of the dialog.

3.2.1 Xen Resource Pool Configuration

Figure 9 shows the parameters that are required for properly configuring and adding the Xen Resource Pool. The parameters that can be configured are the following:

- **Capacity**: this value provides a limit to the number of virtual instances that can be requested and managed by the pool. By default this value is equal to 1 but it is up to the administrator to select a proper value.

- **Pool Master**: this parameter identifies the DNS name or the IP address of the Xen Pool Master. The pool master is the entry point to Xen-based managed system and it represents the virtual machine manager that will control all the requests for virtual machine creation.

- **Port**: the parameter identifies the port number on which the *XenServer* on the Pool Master is listening. By default this value is set to 80, which is the default port on which *XenServer* is listening, if the secure connection is enable an alternative port to select might be 443.

- **Template Name**: the parameter identifies the name of the virtual machine template that will be used by *XenServer* to create virtual machine instances. **This value does not have to be set to template id.**

- **MAC Address Pool**: the parameter allows the administrator to enter a collection of MAC addresses that the pool should use in order to create virtual machines. If this value is not set *XenServer* will automatically create MAC addresses, otherwise will use the ones provided. It is important to notice that the content of this list actually limits the number of virtual machines that can be managed at the same time by the pool. **If the pool is not empty the size of the pool will override the value set for the Capacity parameter.**

- **User Name**: the parameter identifies the user name used to issue the login to *XenServer*.

- **User Password**: the parameter identifies the user password that is used to authenticate the user with *XenServer*.

This operation completes the configuration of the resource pool for what concerns *Cytrix XenServer*.

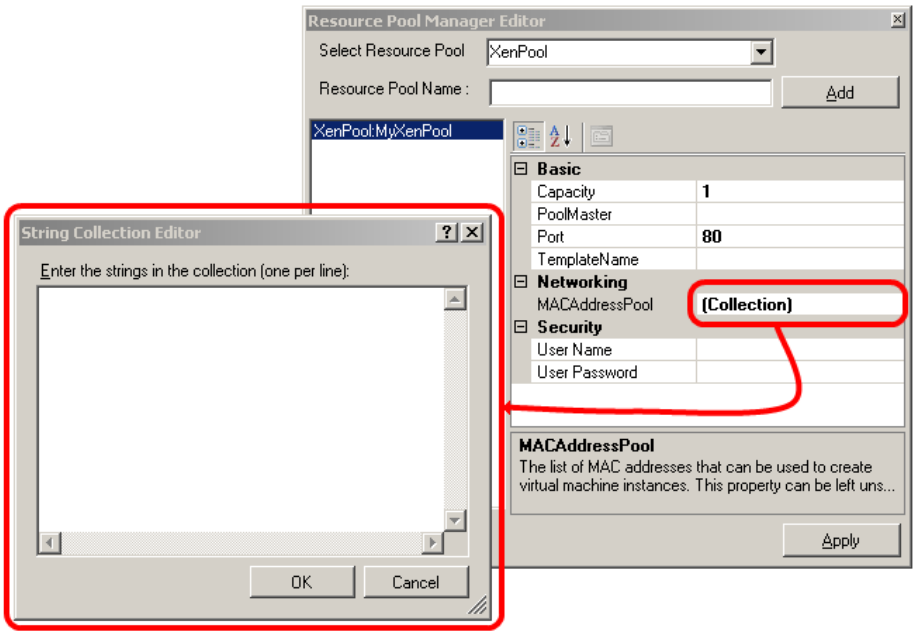

Figure 9 - Xen resource pool configuration.

3.2.2 Amazon EC2 Resource Pool Configuration

Figure 10 shows the set of parameters that can be configured for the Amazon EC2 resource pool. The pool is designed to work on behalf of one account and using a specific virtual machine image.

The parameters that can be configured are the following:

- **Capacity**: this parameter identifies the number of virtual resources that can be managed at the same time by the pool. This number has to be properly set and can be overridden by the constraints of the account that is used to interact with Amazon EC2. For example the default limit of concurrent instances that can be created on Amazon EC2 is 20. Hence, values bigger than 20 will automatically end up in a provision failure once the limit of twenty instances is reached.

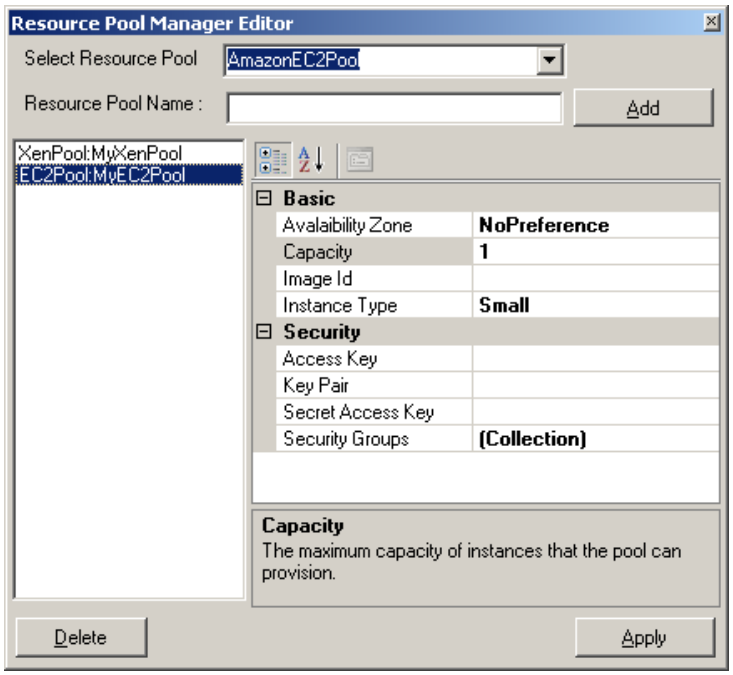

Figure 10 - Amazon EC2 resource pool configuration.

- **Image Id**: this parameter identifies the unique identifier of the virtual image template that Amazon EC2 should clone in order to create virtual machines. This value can be looked up in the *AWS Management Console* under *EC2* tab by clicking the *AMIs* button on the left side. The console will show the list of images available to the currently logged in account. The *AMI Id* column contains the required value.

- **Availability Zone**: this parameter allows the administrator to select the specific availability zone where to deploy the virtual instance. At the moment the only zones that can be requested are the following: *US East 1a, US East 1b, US East 1c, US East 1d.*

- **Instance Type**: this parameter identifies the hardware configuration required for the virtual machine that will be created. The range of possible values directly maps the instance types defined by Amazon EC2. More precisely it is possible to choose from: *Micro*, *Small*, *Medium*, *Large*, *HighMemXL*, *HighMem2XL*, *HighMem4XL*, *HighCPUSmall*, *HighCPUMedium*, and *HighCPULarge*. The mapping between the available options and the corresponding Amazon EC2 instance types is the following:
    - **Micro** → Micro Instance (API name: t1.micro)
    - **Small** → Small Instance (API name: m1.small)
    - **Medium** → Large Instance (API name: m1.large)
    - **Large** → Extra Large Instance (API name: m1.xlarge)
    - **HighMemXL** → High-Memory Extra Large Instance (API name: m2.xlarge)
    - **HighMem2XL** → High-Memory Double Extra Large Instance (API name: m2.2xlarge)
    - **HighMem4XL** → High-Memory Quadruple Extra Large Instance (API name: m2.4xlarge)
    - **HighCPUMedium** → High-CPU Medium Instance (API name: c1.medium)
    - **HighCPULarge** → High-CPU Extra Large Instance (API name: c1.xlarge)

Cluster instances are not supported at the moment. Please refer to the Amazon EC2 documentation for the characteristics of each of these instance types.

Moreover, the successful request of an instance of the specified type is subject to the constraints that are applied to the account on behalf of which the resource pool is issuing the request.

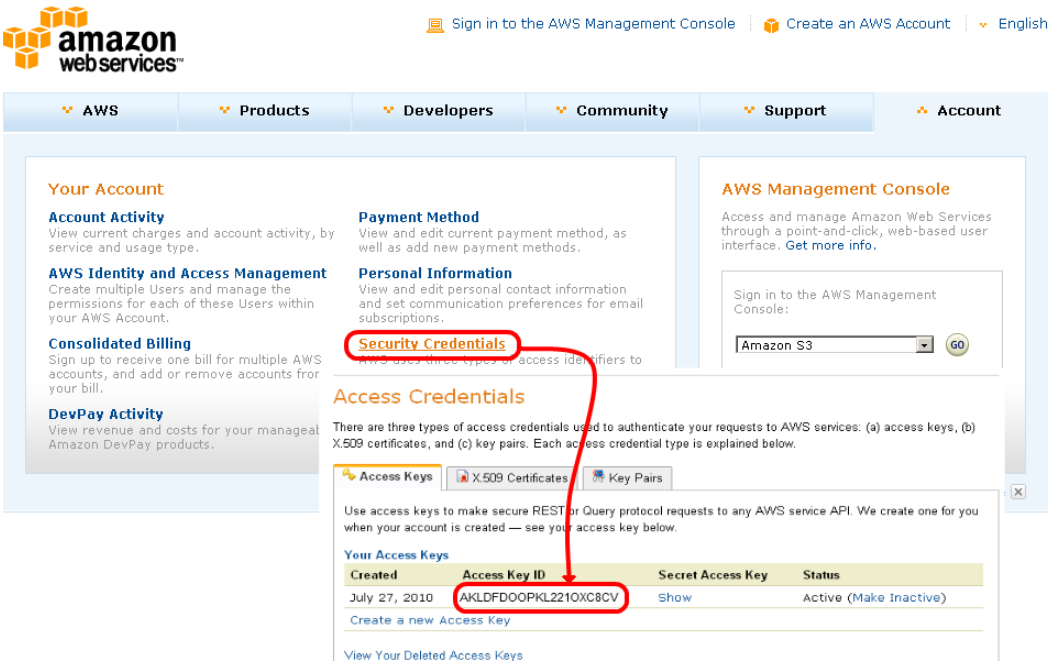

Figure 11 - Amazon AWS Access Key.

- **Access Key**: the parameter identifies the account on behalf of which the pool will be issuing the requests and interacting with Amazon EC2. This value can be obtained from the Amazon Web Service portal, under the *Security Credentials* area, once logged in. Figure 11 shows where to collect the value.

- **Secret Access Key**: this parameter identifies the authentication password that is used to validate the account represented by the *Access Key*. With reference to Figure 11, in order to get the value of the *Secret Access Key* it is necessary to click the Show link on following the selected *Access Key*.

- **Key Pair**: the key pair identifies the asymmetric key pair that is used to make a connection to the remote virtual instance once it has been launched. This value is not necessary unless the user tries to connect to the remote instance through SSH, but it is required in order to issue a proper request to Amazon EC2. This parameter collects the name of the key pair, which can be looked up in *AWS Management Console* as shown in Figure 12. If the account under use has no key pairs stored it is possible to create one from by clicking on the *Create Key Pair* button.

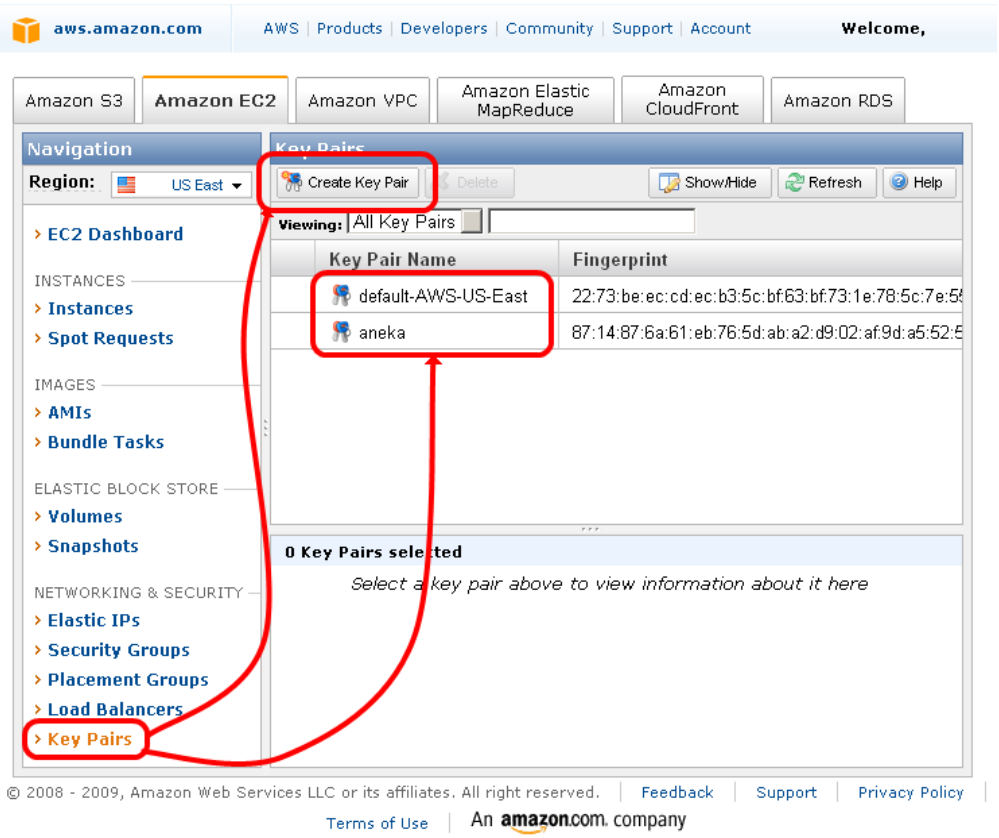

Figure 12 - Amazon AWS Key Pair management.

Key pairs can also be managed from the Amazon Web Services portal. As shown in Figure 11, under the Security Credentials, section there is a Key Pair tab that allows for managing the Key Pairs of both Amazon CloudFrount and Amazon EC2. If you are planning to use this feature, please ensure to create an Amazon EC2 key pair. For further information about the role of the key pair in using Amazon EC2 please refer to the Amazon EC2 Documentation.

- **Security Groups**: this parameter identifies the collection of security groups that the virtual instance that is created will belong to. The dialog opens up an editor where the administrator can enter the names of the security groups that will be associated to the virtual instance (one per line).

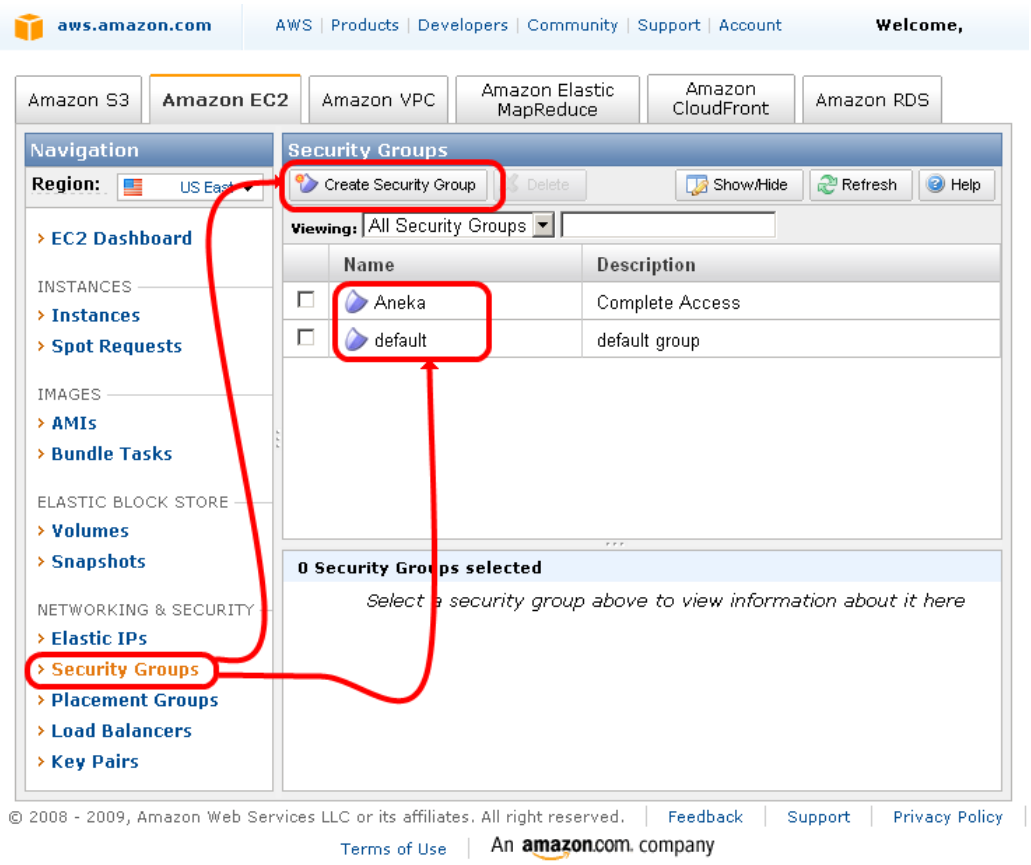

Figure 13 - Amazon AWS Security Group management.

As shown in Figure 13, the available security group can be looked up in the AWS Management Console, under the Security Groups section. If the current account is not configured with any security group the AWS Management Console allows the creation of a new security group by clicking the Create Security Group button. In order how to configure and use Security Group in Amazon EC2 please refer to the Amazon EC2 Documentation.

These parameters complete the configuration of the Amazon EC2 Resource Pool. As it is generally known, the default limit of instances for one single account is 20 instances (if no exceptional conditions apply); it is possible to leverage a major number of instances by using multiple accounts and configuring one Amazon EC2 Pool for each of the account used.

3.2.3 GoGrid Resource Pool Configuration

Figure 14 shows the set of parameters that can be configured for the GoGrid resource pool. The pool is designed to work on behalf of one account and using a specific virtual machine image.

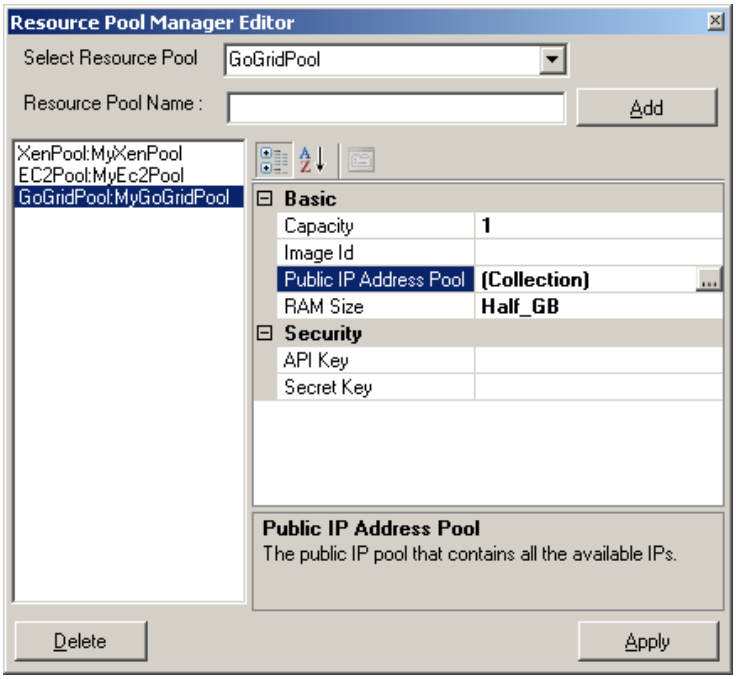

Figure 14 - GoGrid resource pool configuration.

The parameters that can be configured are the following:

- **Capacity**: this parameter identifies the number of virtual resources that can be managed at the same time by the pool. **This number cannot be set and it is automatically inferred from the size of the Public IP Address Pool**.

- **Image Id**: this parameter identifies the unique identifier of the virtual image template that GoGrid should clone in order to create virtual machines. This value can be looked up in the *GoGrid Management Console* by simulating the creation of a new Image sandbox and selecting the specific image from which we want to clone the sandbox.

What is shown in Figure 15 is the final window that is presented to the user once the right virtual machine image has been selected and the user has pressed the next button. The entire sequence of steps to get to this screen is the following:

1. Click on Add new image sandbox.
2. Select Image Sandbox from the list Available Objects.
3. Click on Yes, continue.
4. Select the image that you want to look the Image-id for.
5. Press next at the top of the window.

The sequence of steps described before will lead you to the screen depicted in Figure 15.

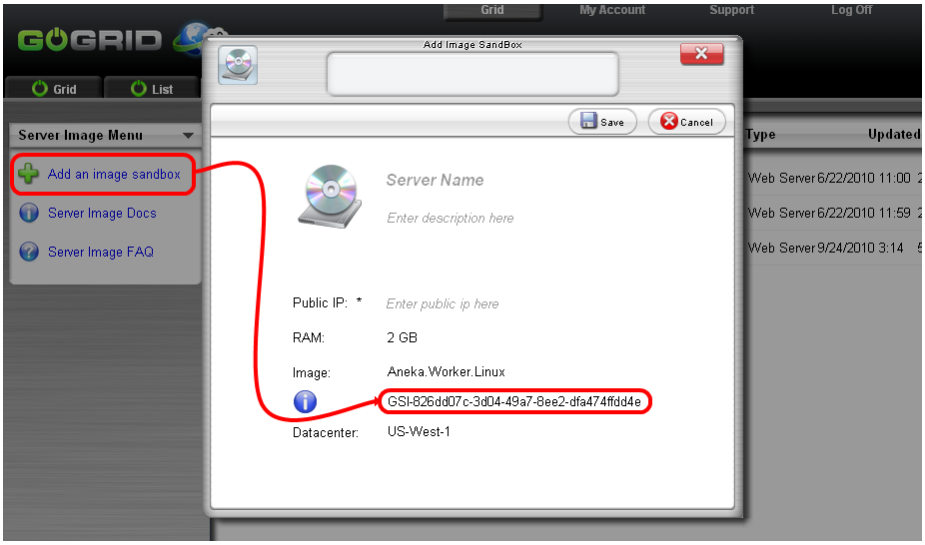

Figure 15 - GoGrid image GSI.

- **Public IP Address Pool**: this parameter allows you to select a list of public IP addresses that the resource pool can use to create virtual instances. In order to create a virtual machine instance **it is necessary** to provide a public IP address otherwise the instance will not be reachable. The possible IP available for use can be collected from the GoGrid account on behalf of which the resource pool operates. Figure 16 shows where to locate the available public IP addresses for use for a specific account. The user interface of the GoGrid management console also offers the possibility of requesting more IP if the current number is not sufficient.

- **RAM Size**: this parameter allows you to set the memory size of the virtual instance. The available options are the following:

  Half_GB → 500 MB

  One_GB → 1 GB

  Two_GB → 2 GB

  Four_GB → 4 GB

- **Secret Key**: the secret key identifies the user account on behalf of which the resource pool is interacting with GoGrid. The value of the secret key can be collected from the GoGrid portal under the My Account section as shown in Figure 17.

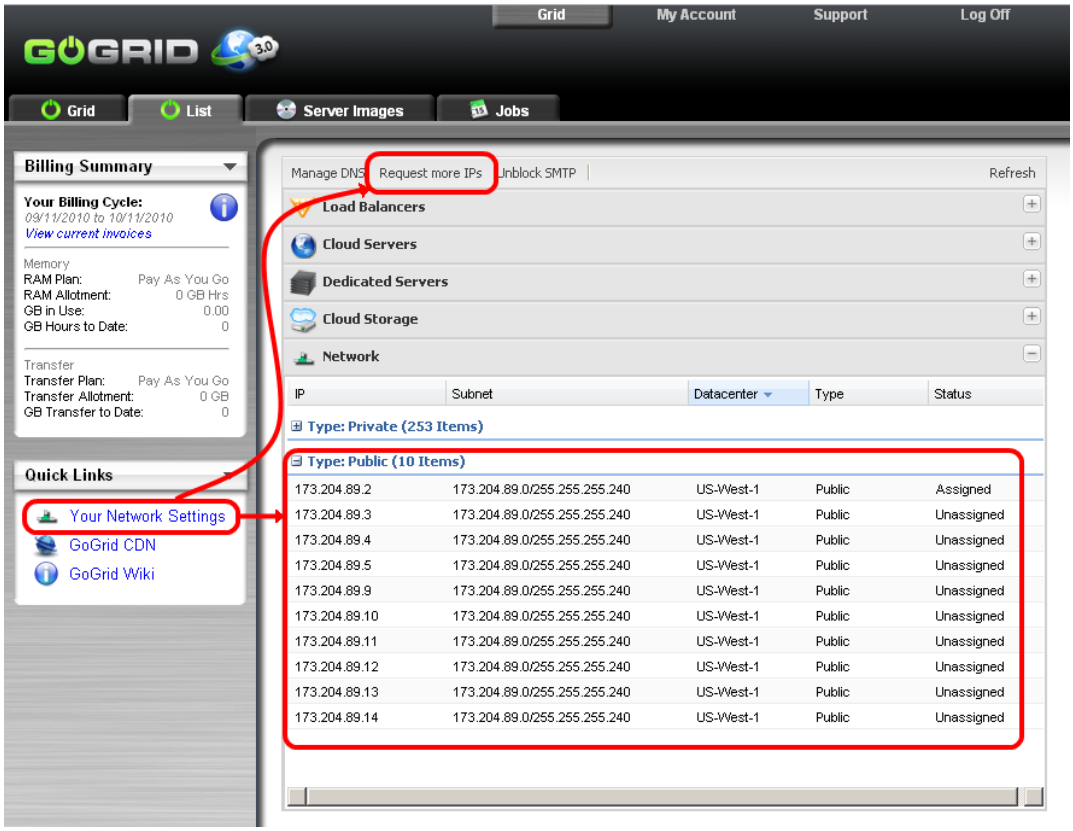

Figure 16 - GoGrid public IP pool configuration.

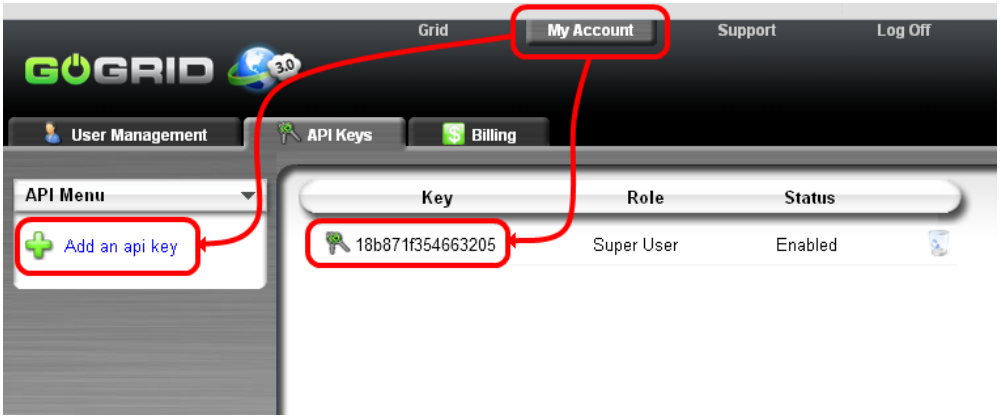

Figure 17 - GoGrid API key.

As shown in Figure 17, it is also possible to create new secret keys if needed.

- **API Key**: this parameter identifies the authentication code that is used to validate the Secret Key. With reference to Figure 17, this value can be collected by clicking on the desired secret key and copying the value that is shown in the Secret Key field of the window that pops up.

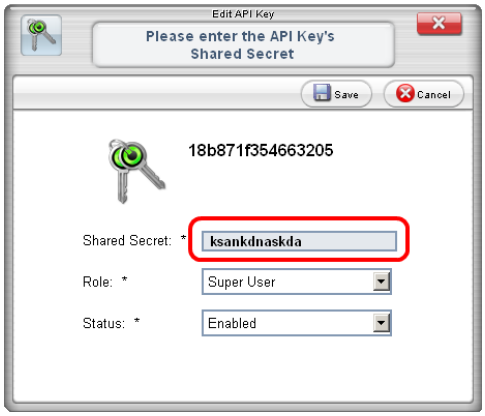

Figure 18 - GoGrid secret key.

These operations complete the configuration of the GoGrid resource pool. Should multiple accounts be used it is possible to configure multiple instances of the GoGrid resource pool (one per account).

NOTE: All the discussed resource pools do not perform any deployment and advanced configuration of the Aneka Container in the virtual machine. It is up to the administrator to properly prepare a virtual image that is compliant with the current installation settings of the Cloud. In particular, the URI to the index server (Master container: tcp://[host]:[port]/Aneka) and the necessary authentication keys. Also, it is not possible to perform any specific container configuration through the wizard for virtual instance, but the Spring.config.xml and the Aneka.cfg.xml files need to be already prepared and set inside the virtual image.

4 Programming

From a user point of view Aneka provides the capability of configuring several pools and leveraging different resource providers even by using multiple accounts and settings for each of them. These features allow for a high degree of flexibility in using the dynamic provisioning infrastructure. When these options are not enough to cover a specific scenario, developers can easily develop and integrate their own solutions which extend the current set of capabilities of the dynamic provisioning infrastructure.

4.1 Extending the Dynamic Resource Provisioning Infrastructure

The Dynamic Resource Provisioning Object Model of Aneka allows is completely extendable and customizable and offers different extension points that can be exploited to provide

new solutions for dynamic provisioning. As described in Figure 3 there are several components that compose the dynamic provisioning infrastructure and almost all of them are replaceable with a custom implementation. The object model actually allows developers to:

- Develop new scheduling algorithms
- Develop new provisioning strategies
- Develop new resource pool managers
- Develop new resource pools
- Develop new a resource provisioning service
- Develop a new scheduling service

Among all the options presented before only the first four options can be easily exploited to extend and to innovate the dynamic provisioning infrastructure. The latest two options allow developers to completely change the architecture of the dynamic provisioning and implement their own infrastructure.

In this document we will discuss the first three options, and provide few hints on how to develop a new resource provisioning service that can be integrated into the existing infrastructure.

4.1.1 Developing New Scheduling Algorithms

Scheduling algorithm control the execution of distributed application in the Aneka Cloud and the allocation of tasks to the resources available in the infrastructure. By using dynamic provisioning it is possible to integrate new resources into the infrastructure for a period of time that is subject to the needs of applications. Scheduling algorithms are generally the components that have the knowledge of when new resources are needed and for how long. Indeed, they are the enabler for the resource provisioning since in most of the cases they trigger resource provisioning requests and command the resource provisioning service to release virtual resources when they are no longer needed.

Support for scheduling in Aneka differentiates the logic with which tasks are allocated to resources from all the machinery required to interact with the Aneka runtime. This architecture makes possible to easily plug into the existing scheduling services new scheduling algorithms with a minimal knowledge of the internals of the Aneka runtime. It does so by defining to interfaces — namely *ISchedulerContext* and *ISchedulingAlgorithm* — that represent the interface with the Aneka runtime and the scheduling algorithm respectively. These interfaces and bases classes from which implement the components are part of the *Aneka.Scheduling* library.

```
using Aneka.Provisioning;
using Aneka.Scheduling.Entity;
using Aneka.Scheduling.Event;

namespace Aneka.Scheduling
{
   public interface ISchedulerContext
   {
        bool SupportsProvisioning { get; }

        event EventHandler<SchedulingEventArgs> AssignTask;

        event EventHandler<ProvisionResourcesArgs> ProvisionResources;

        event EventHandler<ReleaseResourcesArgs> ReleaseResources;

        void Start();

        void Stop();

        void AddTasks(params Task[] tasks);

        void AddResources(params Resource[] resources);

        void SetScheduler(ISchedulerContext scheduler);
   }
}
```

Listing 1 - ISchedulingAlgorithm interface

Listing 1 shows the *ISchedulingAlgorithm* interface, which represents the contract that each scheduling algorithm has to fulfill. The interface is minimal and provides some methods that are used by the Aneka runtime for starting, stopping the algorithm, adding tasks and resources and providing a reference to the scheduler context. The algorithm provides a feedback to the Aneka runtime about its scheduling decisions through the events exposed by the interface. For what concerns dynamic provisioning there are two members of interest:

- *SupportsProvisioning*: this is a boolean property that is set to true if the algorithm supports dynamic provisioning.

- *ProvisionResources*: this event is triggered when the scheduling algorithm issue a request for additional resources.

The algorithm works collaboratively with the Aneka runtime and interfaces with it by means of the ISchedulerContext interface reported in Listing 2.

```
using Aneka.Provisioning;
using Aneka.Scheduling.Entity;
using Aneka.Scheduling.Event;
using Aneka.Scheduling.Runtime;

namespace Aneka.Scheduling
{
   public interface ISchedulerContext
   {
      ISchedulingAlgorithm SchedulerAlgorithm { get; set; }

      ISchedulingHandler SchedulingHandler { get; set; }


      event EventHandler<ResourceEventArgs> ResourceDisconnected;

      event EventHandler<ResourceEventArgs> ResourceReConnected;

      event EventHandler<ProvisionEventArgs> ResourceProvisionProcessed;

      event EventHandler<ProvisionResourcesArgs> ResourceProvisionRequested;

      event EventHandler<ReleaseResourcesArgs> ResourceReleaseRequested;


      event EventHandler<TaskEventArgs> TaskFinished;

      event EventHandler<TaskEventArgs> TaskFailed;

      event EventHandler<TaskEventArgs> TaskAborted;

      event EventHandler<TaskEventArgs> TaskRequeued;
   }
}
```

Listing 2 - ISchedulerContext interface

The properties and events of the interface only a subset of them are of interest for the scheduling algorithm. In particular for what concerns the normal operations of the scheduling algorithm:

- *ResourceDisconnected* and *ResourceReconnected*: notify the scheduling algorithm that a resource has disconnected or reconnected from a temporary disconnection.

- *TaskAborted*, *TaskFinished*, *TaskFailed*, and *TaskRequeued*: these events notify scheduling algorithm of the status of the tasks.

For what concerns the dynamic provisioning infrastructure only one event is of interest for the scheduling algorithm: *ResourceProvisionProcessed*. This event provides information about the outcome of a resource provisioning request made earlier by the scheduling algorithm.

The general execution flow of the scheduling algorithm is managed in an internal thread that is controlled by the Start and Stop methods by the Aneka runtime. The scheduling decisions taken by the algorithm are triggered by the events exposed by the *ISchedulerContext* interface and by the *AddTasks* and *AddResources* methods.

Aneka features several built-in scheduling algorithms that can be used and some of them are designed to support dynamic provisioning of virtual resources by leveraging the resource provisioning service. These are all defined in the namespace *Aneka.Scheduling.Algorithm.Independent*, which can be found in the same library. Among all the algorithms the interesting algorithms for provisioning are:

- *DeadlinePriorityProvisioningAlgorithm*: this algorithm leverages dynamic provisioning in order to schedule the execution of the tasks within the expected deadline. If the local resources are not enough to execute all the tasks in time, a request for additional resources is issued.

- *FixedQueueProvisioningAlgorithm*: this algorithm leverages dynamic provisioning in order to maintain the size of the scheduling queue within a specific range. It is possible to define an upper and a lower threshold.

Other base classes are provided, from which to start the implementation of new scheduling algorithms that support dynamic provisioning:

- *ProvisioningAlgorithmBase*: this class provides a base abstract class for all dynamic provisioning algorithms. The algorithm provides a basic management of the provisioning request that have been issued.

- *ApplicationAwareProvisioningAlgorithm*: this algorithm is specialized for the scheduling a collection of tasks as a whole in order to ensure that some specific QoS parameters that are defined for the application are met.

Developers can start designing their new scheduling algorithms and new strategies for triggering resource provisioning by extending one of these two classes or specializing the previous two algorithms.

4.1.2 Developing New Provisioning Strategies

Provisioning strategies are useful for implementing provisioning policies. They constitute the best solution when it is necessary to change only the logic in which the pools are selected in order to satisfy a provision request. The implementation of new provisioning strategy frees developers from implementing all the machinery that is needed to:

- Manage resource pools
- Interact with resource providers
- Interact with the Aneka runtime

These strategies can be easily plugged into the existing infrastructure since the default resource pool manager that comes with the Aneka installation provides the capability of being configured with provisioning strategies.

The Aneka APIs provide developers with the *IProvisionStrategy* interface and two implementation classes, namely *ProvisionStrategyBase* and *DefaultProvisionStrategy*, for creating new provisioning policies.

```
using Aneka.Provisioning;

namespace Aneka.Runtime.Provisioning
{
   public interface IProvisionStrategy
   {
      void Init(IDictionary<string, string> properties);

      IResourcePool Select(List<IResourcePool> pools, ProvisionRequest request);

      void Update(ProvisionResponse response);

      void Shutdown();
   }
}
```

Listing 3 - IProvisionStrategy interface

Listing 3 describes the operations that are exposed by the IProvisionStrategy interface. These operations basically allow strategy initialization and finalization, pool selection, and strategy update. More precisely:

- The *Init* method initializes the strategy. It accepts a dictionary of string properties that are passed by the default resource pool manager.
- The *Select* method constitutes the core of the strategy, and it is called by the default pool manager when it is necessary to identify the best pool that is available for satisfying the given provisioning request.
- The *Update* method is called by the default resource pool manager to give feedback on provisioning request.
- The *Shutdown* method is used to release all the internal resources allocated by the strategy and it is called by the default pool manager when it is shut down.

A skeleton implementation of this interface can be found in the *ProvisionStrategyBase* class. It provides the basic locking logic to ensure that the methods of the strategy are called in a thread-safe context and template method that needs to be implemented by the

inherited classes. An extension to this class is the DefaultProvisionStrategy that implements the simple provisioning logic:

- If a provisioning request has a suggested pool name, it selects the pool named
- If a provisioning request does not have a pool name indicated it select the first available pool

This is the strategy that is currently used in the default resource pool manager.

4.1.3 Developing New Resource Pool Managers

The primary purpose of resource pool managers is to manage different resource pools and providing specific resource provisioning and allocation strategies. Implementing a new resource pool manager if the best solution whether the reason for customizing the resource provisioning infrastructure is:

- To develop new advanced strategies for resource provisioning
- To provide a better and advanced pool management services

Resource pool managers are completely separated by the Aneka runtime and the development of a new pool manager can be easily integrated into the existing runtime for dynamic provisioning. The implementation of a new resource pool manager is mostly concerned with providing new methods and algorithms for pool management rather than resource provisioning policies. These can be more easily implemented by implementing resource provisioning strategies as described in the previous section. The only reason for implementing a pool manager in order to develop new provisioning policies is because the options given with the solution discussed in the previous section are not enough and a finer control on the infrastructure is required.

The Aneka APIs provide useful extension points for implementing a new resource pool manager:

- Interface *IResourcePoolManager*: this interface represents the contract that all the resource pool manager need to fulfill in order to be plugged into the existing infrastructure and manage resource pools.
- Class *ResourcePoolManagerBase*: this is a base abstract class that can be used to implement a new resource pool manager. This class provides most of the functionalities that are required to implement a basic resource pool and provides a set of template methods that can be further specialized to implement resource provisioning policies.
- Class *ResourcePoolManager*: this class is the default resource pool manager used within Aneka and constitutes a working implementation of a pool manager. The

manager abstracts the pool selection process for a given provisioning request by using resource provisioning strategies that can be configured. This class is useful when it is necessary to provide small changes to the existing pool manager in order to address the needs of the provisioning scenario.

Listing 4 shows the *IResourcePoolManager* methods, events, and properties. This interface provides minimal functionalities:

- The *Init* method is used to initialize the pool manager and it takes as a parameter a dictionary of string properties. At the moment the only property that it is passed inside the dictionary is the URI of the Aneka master container (property name: *IndexServerUri*) and the security key that is used for communications (property name: *SharedAuthenticationKey)*. This method is called by the hosting resource provisioning service when the service starts.

- The *Provision* method is used to issue a resource provisioning request and it is called by the hosting provisioning service to forward to the pool manager the provisioning request.

- The *Release* method is used to release a list of virtual resources that have been previously provisioned. This method is called by the hosting provisioning service when some virtual resources are no longer needed.

- The *Shutdown* method is called when the service is stopped and it is expected that the pool releases all its resources.

```
using Aneka.Provisioning;

namespace Aneka.Runtime.Provisioning
{
   public interface IResourcePoolManager : IProvisionEventSource
   {
      List<IResourcePool> ResourcePools { get; set; }

      event EventHandler<ResourceActiveEventArgs> ResourceActive;

      void Init(IDictionary<string, string> properties);

      void Provision(ProvisionRequest request);

      void Release(IList<string> resources);

      void Shutdown();
   }
}
```

Listing 4 - IResourcePoolManager interface

- The *ResourcePools* list provides access to the list of *IResourcePool* instances that are configured with the resource pool manager.

The interface designed for the resource pool manager uses event notification for providing information about the provisioning requests and the status of virtual resources. In particular three different events are provided:

- The *ResourceActive* event fires when a virtual instance has become reachable.
- The *IProvisionEventSource.ProvisionCompleted* fires when a request for resource provisioning has been successfully submitted.
- The *IProvisionEventSource.ProvisionFailed* fires when a resource provisioning request fails for any reason. The information in the event argument instance can help the event handlers to identify the reason of the failure.

It is responsibility of the resource pool manager implementation to appropriately fire these events.

Listing 5 shows the *IProvisionEventSource* interface from which the *IResource-PoolManager* interface inherits.

```
using Aneka.Provisioning;

namespace Aneka.Runtime.Provisioning
{
   public interface IProvisionEventSource
   {
      event EventHandler<ProvisionEventArgs> ProvisionCompleted;

      event EventHandler<ProvisionEventArgs> ProvisionFailed;
   }
}
```

Listing 5 - IProvisionEventSource interface

4.1.4 Developing New Resource Pools

Developers might want to implement a new resource pool when it is necessary to integrate a new resource provider that is not actually supported by Aneka or when it is necessary to provide a different way in which Aneka should talk with the resource provider.

The Aneka APIs provide the *IResourcePool* interface and the *ResourcePoolBase* class as extension points for integrating new resource pools. The *ResourcePoolBase* class is a simple skeleton with a minimal set of functionalities implemented, while most of the

methods exposed by the *IResourcePool* interface have to be implemented. This class is a useful starting point for implementing a new resource pool.

Listing 6 describes the *IResourcePool* interface. The interface extends the *IProvisionEventSource* and uses the same mechanism of the resource pool manager implementation to notify client components (i.e. the resource pool manager or any other code section registered to these events) about a successful provision request submission or a failure. Moreover, the interface exposes a set of properties that provide some information about the pool and simplify the interaction with the pool manager:

```
using Aneka.Provisioning;

namespace Aneka.Runtime.Provisioning
{
   public interface IResourcePool : IProvisionEventSource
   {
      string Name { get; set; }

      ResourcePoolView PoolDescription { get; }

      string IndexServerUri { set; }

      bool SupportPushMode { get; }

      IResourcePoolManager PoolManager { set; }

      void Init(IDictionary<string, string> properties);

      void Provision(ProvisionRequest request);

      string PingResource(string resource);

      void Release(IList<string> resources);

      List<ResourcePoolItem> List();

      void Terminate();
   }
}
```

Listing 6 - IResourcePool interface

- *Name*: defines the name of the pool. The name of the pool should be a unique identifier and can be used by the pool manager or other components to identify a specific pool.

- *PoolDescription*: provides information about the pool and its current status.

- *PoolManager*: is a reference to the pool manager instance that is controlling the pool. **This property has to be set by the pool manager implementation**.

- *IndexServerUri*: is the address of the Aneka master container that manages the Cloud in which the resource provisioning service operates. **This property has to be set by the pool manager implementation**.

- *SupportPushMode*: this is a property that provides information about the behavior of the resource provider managed by the pool. In particular, if the value of such a property is true this means that the pool can automatically configure the virtual instances that are requested with an instance of the Aneka container running. Hence, when the provisioning request is successfully submitted it is possible to ping the virtual resource to get the URI of the Aneka container instance running on it. **This operation has to be performed by the pool manager implementation by calling *IResourcePool.PingResource***.

The list of operations supported by the pool manager interface is described as follows:

- *Init*: this method passes a dictionary of string properties to the resource pool. These properties are *runtime* properties that are set by the Aneka runtime and provide information about the runtime context. This method is used to initialize the pool.

- *Provision*: this method is used to issue a resource provisioning request and it is called by the pool manager to forward to the pool the provisioning request.

- *Release*: this method is used to release a list of virtual resources that have been previously provisioned. This method is called by the resource pool manager.

- *Terminate*: this method is called when the pool manager is shutting down and the pool should release all the resources that have been acquired as a side effect of this call.

- *PingResource*: this method returns the URI of the Aneka container instance that is running on the virtual resource represented by the resource identifier passed as parameter. This method is useful when the pool supports the push mode and automatically configures the virtual machine requested with an instance of the Aneka container.

The *ResourcePoolBase* class, as already noticed, provides a skeleton implementation of a resource pool. It provides an implementation of all the properties exposed by the interface and an extensible infrastructure managing the list of resources that are currently provisioned. An interesting feature of this class is that the infrastructure supports the management of virtual machine instances that have been released by the pool manager and whose time slice has not been expired yet. Most of the IaaS providers deliver virtual machine resources on a time slice basis and this feature is an add-on to the skeleton implementation that simplifies the implementation of resource providers.

4.1.5 Developing a New Resource Provisioning Service

The implementation of a resource provisioning service is the most radical and powerful solution for customizing the resource provisioning infrastructure. The implementation of the resource provisioning service provides developers with a complete control on the provisioning mechanics of the infrastructure. In particular allows controlling the behavior of the following aspects:

- Resource pool management
- Interaction with the Aneka runtime environment
- Definition of the provisioning strategies and policies
- Management of virtual resources and monitoring

The resource provisioning services groups all the features that are exposed by the other provisioning components and presents them as a whole to the Aneka runtime. It can be considered as a black box that manages provisioning of virtual resources. It is up to the developers to choose how to organize its internal structure and how to perform the provisioning operations required by the Aneka runtime.

The Aneka resource provisioning infrastructure is the result of the coordinated activity of the scheduling services and the provisioning service. In order to integrate a new resource provisioning service with the existing scheduling services it is necessary to implement handlers to some specific messages that are used by scheduling services to control provision and control virtual resources. The Aneka APIs provide skeleton classes from which where to start the development of a resource provisioning service. These are:

- *ResourceProvisioningServiceBase*: this is an abstract base class that can be used to create new provisioning services. The class provides the template methods for the basic resource provisioning message handling and implements the basic logic of the service implementation and interaction with the Aneka runtime. The template methods that are left to inherited classes are the following:
    - *HandleProvisioningRequestMessage*: this method is called when the provision of a set of virtual resources is triggered by some component in the Aneka runtime (most likely another service such as the scheduling services). As a parameter to the method an instance of the class *ProvisionRequestMessage* is passed. This class contains the information about the provisioning request made and exposes it with the *Request* property (type: *Aneka.Provisioning.ProvisionRequest*).
    - *HandleReleaseRequestMessage*: this method is called when the Aneka runtime is signaling the provisioning service to release a list of resources that are currently provisioned. As a parameter to the method an instance of the class ReleaseRequestMessage is passed. This class contains the information about virtual resources to release.

- *HandleProvisionQueryMessage*: this method is called whenever a client or component of the Aneka runtime wants to query the status of the resource provisioning infrastructure. It is possible to query for one of the following elements: virtual resources, provision requests, resource pools. This method takes as a parameter an instance of the *ProvisionQueryMessage*, which specifies the query that needs to be performed.

Other possible extension points in the same class are the following:

- *GetClient*: this method is called by the infrastructure whenever the resource provisioning service is asked to provide a client that is able to interact with the service and dealing with all the message-based protocol required by the resource provisioning service. The current implementation of the service return the default provisioning client which allows any software component to interact with the resource provisioning service in order to perform the operations described above. These operations are made available through the interface *Aneka.Provisioning.IProvisioningClient*. If the specific resource provisioning service implemented provides a more complex set of services it is possible to return a service that extends this interface with additional features by overriding this method.

NOTE: By design it is expected that each provisioning infrastructure will support the basic operations exposed by the interface. Hence, all the provisioning client implementations have to extend this interface if they want to be plugged within the provisioning infrastructure. Aneka provides a default implementation for this client which is sufficient in most of the cases and if no additional operations have to be exposed. In case additional operations have to be made available the best practice is to define an interface that extends the IProvisioningClient interface and expose the additional methods.

- *ProcessMessage*: this method implements the method dispatcher that handles the incoming message and calls the appropriate template method. In case of implementations providing additional features the behavior of this method needs to be extended to manage the additional features implemented. This method takes as a parameter an instance of the class *ProvisionMessage* that is the base class for all the resource provisioning messages. Hence, it does not manage the *ClientMessage*, which is a general Aneka message used to obtain a client for a service. Should a more radical implementation of the message handling be implemented, it is possible to override the *HandleMessage* method that is the main entry point in each service for message handling.

The other methods defined simply provide an implementation of the template methods defined in the *ServiceBase* class and do not perform any operation.

- *ResourceProvisioningService*: this class defines the reference implementation of the resource provisioning service and the one that is actually used inside Aneka. It extends the *ResourceProvisioningBase* class and integrates a resource pool manager and a provisioning store that can be configured by the user. The resource pool manager is configured with the default implementation and the store with an in memory volatile storage for the provisioning requests.

Developers can use these two classes as a starting point for defining a new resource provisioning service. The extension of the *ResourceProvisioningServiceBase* class provides more freedom for what concerns the logic and the implementation of the service but also less built-in features. For example it is possible to implement a service that does not use pools for virtual resource management. On the other hand, the *ResourceProvisioningService* class provides developer with a working implementation of a service and ready to use components, but limits the extension points and it has a more rigid logic.

4.2 Programming Provisioning From Client Applications

The resource provisioning infrastructure has been designed serve both the internal needs of the Aneka middleware and the external needs of specific client applications. As a result, client applications can easily interact with the resource provisioning service and provision virtual resources, query the resource provisioning system, and release resources. The set of classes, interfaces, and components that used to interact with the resource provisioning service are defined in the *Aneka.Provisioning* namespace inside the *Aneka.dll* library and there is no need to reference any other library - except than *Aneka.Data.dll* and *Aneka.Util.dll* – for interacting with the provisioning infrastructure.

The main component of the support for client based programming is the resource provisioning client, which abstracts the interaction with the installed resource provisioning service and exposes a set of basic operations. These operations are defined in the *IProvisioningClient* interface.

```
using Aneka.Runtime;

namespace Aneka.Provisioning
{
   public interface IProvisioningClient : IServiceClient
   {
      ProvisionResponse Provision(ProvisionRequest request);

      Exception Release(string resourceId);

      Exception Release(IList<string> resources);
```

```
        IList<ResourcePoolView> ListPools();

        IList<ResourcePoolItemView> QueryPool(string poolName);

        ResourcePoolItemView QueryResource(string resourceId);

        IList<ResourcePoolItemView>QueryResources(IList<string> resources);

        ProvisionResponse QueryRequest(string requestId);

        IList<ProvisionResponse> QueryRequests(IList<string> requests);
    }
}
```

Listing 7 - IProvisioningClient interface

The interface exposes the basic set of operations that all the resource provisioning services should support. Since Aneka allows for the integration of third party and custom solutions for resource provisioning one client will not fit all the possible scenarios. In particular, specific implementation of resource provisioning service might expose additional operations not available through this interface. This is the reason why Aneka allows for a dynamic retrieval of the appropriate client instance that is able to properly interact with the resource provisioning service.

In the previous section we have discussed the extension point provided by the infrastructure for implementing a new client, in this section we will illustrate how to retrieve and use these client.

It is possible to obtain a client for the resource provisioning service by using the *ProvisioningClient* class and invoking the *GetClient(Aneka.Entity.Configuration)* method. This method returns an instance implementing the IProvisioningClient interface and takes as a parameter a configuration object containing all the information that are required by the client libraries to:

- Contact the Aneka middleware
- Authenticate the request
- Provide additional parameters that might be of use to retrieve the client

Clients, in order to make use of the additional operations exposed necessarily have to cast the instance returned by the *GetClient* method to a well known interface or class type that defines the additional method.

The operations exposed by the *IProvisioningClient* interface are the following:

- *Provision(ProvisionRequest …)*: allows client to provision a collection of resources. This method takes as a parameter an instance of the class *ProvisionRequest* containing the information about the number of resource to provision, the specific pool to use (if required), and additional metadata that can help the selection of

virtual resources. The method returns an instance of the ProvisionResponse class that contains all the information about the provisioning request issued.

- *Release(string …)* and *Release(IList<string> …)*: these two methods allows to release a virtual resource and a list of virtual resources respectively. They take as input the identifiers of the resources to release and return an exception if some error occurred while releasing the resources. The list version of the method returns a list of exceptions (or null values) each of them mapping the outcome of the release operation of the corresponding identifier in the input list.

- *ListPools()*: this method returns all the available pools that are managed by the resource provisioning service and their status. It has been discussed that a specific implementation could not leverage resource pools. Even in that case the resource provisioning service should still expose the logical concept of resource pool to the client. The method returns a list of *ResourcePoolView* instances containing the information about the managed pools.

- *QueryPool(string …)*: this method provides the information about a specific resource pool identifier by the name passed as input parameter (or null if no pool with that name is found). The method returns an instance of the ResourcePoolView class containing the information about the selected pool.

- *QueryResource(string …)* and *QueryResources(IList<string> …)*: these methods provide information about one and a list of virtual resources respectively. Both of the two version takes as input parameters the resource identifiers and return instances of the *ResourcePoolItemView* class that contain information about the corresponding resources.

- *QueryRequest(string …)* and *QueryRequests(IList<string> …)*: these methods provide information about one and a list of resource provisioning requests. Both of the two version takes as input parameters the request identifiers and return instances of the *ProvisionResponse* class that contain information about the corresponding requests.

The set of methods that are exposed are enough to provide a basic management of the resource provisioning infrastructure. Additional properties are exposed by the IServiceClient interface that helps in identifying the component that is being used:

- *ServiceUrl*: URI of the Aneka container that is hosting the provisioning service.

- *ClientName*: name of the provisioning client.

- *Service*: name of the resource provisioning service that the client is interacting with.

- *Configuration*: configuration object that has been used to retrieve the client.

These properties are common to all the dynamic clients that are retrieved with the Aneka APIs.

As previously discussed the main features of resource provisioning are accessible through the *IProvisioningClient* interface. Additional classes are involved in the client management of resource provisioning; these classes are:

- *Aneka.Provisioning.ProvisionRequest*: this class encapsulates the information about a provisioning request. The important bits exposed by this class are the following:
    - *RequestId*: unique identifier of the request (automatically generated).
    - *Pool*: name of the pool to use for the provisioning of resources (if any).
    - *Resources*: number of resources to provision.
    - *Metadata*: a dictionary of string properties that can be used to pass any additional information to the resource provisioning service.

It is important to notice that the information entered in the metadata dictionary can be of any nature, and it is up to the resource provisioning service whether to use it or not. Client applications should also know the name and the meaning of the properties that are entered in this dictionary, which depend on the specific implementation of the resource provisioning service.

- *Aneka.Provisioning.ProvisionResponse*: this class encapsulates the response to a provisioning request. The most important properties are: *RequestId* and *Status*. The first one identifies the provisioning request that is mapped to this response, while the second identifies the status of the request and it is of type ProvisionRequestStatus. The additional information contained in this class is exposed through the following properties:
    - *ResourcesProvisioned*: number of resources provisioned.
    - *Resources*: is an array of strings containing the identifiers of the virtual resources if the provisioning request has been satisfied.
    - *Failure*: it is a value from the ProvisionFailure enumeration that identifies the nature of the failure of the request (in case of failure).
    - *Exception*: it is of type Exception and provides additional information about the error condition associated with the failure. It might be null.

All these information help the client application to track the provisioning request all over its life cycle.

- *Aneka.Provisioning.ProvisionFailure*: enumerates all the possible reasons of a failure. The current available values are the following: *Temporary*, *Permanent*, *CapacityExceeded*, and *Other*.

- *Aneka.Provisioning.ProvisionRequestStatus*: enumerated all the possible states of resource provisioning request. The current available values are the following: *Pending*, *Succeeded*, and *Failed*.

- *Aneka.Provisioning.ResourcePoolView*: this class provides a view over a resource pool. It is merely a container of information about the properties of the pool such as the name, the type, the number of virtual instances currently allocated and managed by the pool, and maximum capacity of the pool. Additional information can be retrieved from the *Metadata* property that is dictionary of string based properties.

- *Aneka.Provisioning.ResourcePoolItemView*: this class provides a view over a virtual instance managed by a resource pool. It is merely a container of information about the properties of the virtual resource such as the id, the state, the corresponding master container URI, and the start time of the instance. Additional information can be retrieved from the *Metadata* property that is dictionary of string based properties.

This set of classes completes the overview of the client API for interacting with the resource provisioning service. A more extensive treatment of how to program and interact with the resource provisioning service can be found in the Aneka documentation APIs.

4.3 Observations

This section has covered the basics of the object model and of the software components that compose the Aneka Resource Provisioning Infrastructure. The object model allows for a high degree of customization of the infrastructure and encapsulates in specific components different aspects of the resource provisioning. It is possible to extend the system by implementing new strategies for provisioning, new scheduling algorithms, new resource pools and pool managers, or new provisioning services.

The discussion held in this section is a starting point for further exploring the Aneka Provisioning APIs, which provide a more detailed description of each class and its role inside the framework.

5 Conclusions

In this document we have described the architecture of the dynamic provisioning infrastructure in Aneka. This document has also covered how to configure and manage the supported resource pools by the *ResourceProvisioningService*, which together with the scheduling services enable dynamic provisioning in Aneka.

The specific steps that are required to set up the dynamic provisioning infrastructure through the Aneka Cloud Management Studio have been discussed together on how to

collect the required information from the two supported IaaS provider: Amazon EC2 and GoGrid.

As a final note for developers, a brief insight on the Dynamic Resource Provisioning Object Model designed in Aneka and its extensibility points has been presented. This section can be used as a starting point for integrating new resource pools and devising advanced provisioning and scheduling strategies in Aneka.