Efficient Distribution of Virtual Machines

Miha Stopar, XLAB, March 24, 2011

1. Virtual Machine Management

Two modules for virtual machine management have been developed under the umbrella of the SLA@SOI project to better deliver a SLA-enabled infrastructure. Both are integrated in Apache Tashi and can be further integrated in other IaaS platforms. The SLA@SOI source code can be found on Source Forge and the documentation can be found here.

The first module is a scheduler that allocates the virtual machines to the physical machines. The allocations take into account the Infrastructure SLA specifications for the virtual machines (CPU, memory, location, isolation, HW redundancy level, auditability,...) as well as data center policies, e.g. server efficiency, energy consumption, user priorities and over-provisioning of the resources.

Virtual machine re-provisioning is the second extension and it provides the functionality for a dynamic resource management (CPU, memory, network bandwidth, disk bandwidth) of the virtual machines.

In this document we will describe a scheduler module. The diagram below shows the scheduler flow, which is executed every "scheduleDelay" seconds (scheduleDelay is part of the scheduler configuration). The flow is managed from the SchedulerManager, which is executing the methods from the scheduler implementation. The scheduler implementation needs to implement the following methods:

```python
def setVmsForMigration()
def handleMigration()
def determineHost()
def getMostSuitableHost()
def freeUpResources()
def checkAdaptedVms()
def reschedule()
def powerOffHosts()
def powerOnAppropriateHost()
def guaranteedServersOn()
```
Overall, the responsibility of the scheduler is to guarantee the Infrastructure SLA terms, for example the following part of the SLA specifies that changes to the virtual machine disk have to be persistent.

```xml
<slasoi:VariableDeclr>
  <slasoi:Text/>
  <slasoi:Properties/>
  <slasoi:Customisable>
    <slasoi:Var>VM_PERSISTENCE_VAR</slasoi:Var>
    <slasoi:Value>
      <slasoi:Value>true</slasoi:Value>
    </slasoi:Value>
    <slasoi:Datatype>http://www.w3.org/2001/XMLSchema#boolean</slasoi:Datatype>
  </slasoi:Customisable>
</slasoi:VariableDeclr>
```

Figure 1 - Scheduler flow
The scheduler has to take care for preparing enough images for a virtual machine to be started with a persistent disk and after this it has to start the virtual machine with the appropriate parameters (for a KVM that means to start a virtual machine with a properly set snapshot parameter). Other Infrastructure SLA terms are: VM_QUANTITY_VAR, VM_ISOLATION_VAR, VM_CORES_VAR, VM_CPU_SPEED_VAR, VM_MEMORY_SIZE_VAR, VM_IMAGE_VAR and others.

3. Scheduler demonstration

For a simpler demonstration of the scheduler we will assume that we have a micro data center with a 3 physical servers. A scheduler simulation will be used for a demo, which behaves exactly as the real scheduler, but instead of starting a real virtual machine, it just adds a virtual machine entry in the configuration file and it changes the states of the virtual machines and hosts accordingly. The simulation enables much faster and simpler testing. You can read more about SLA@SOI schedulers and how to run scheduler simulations on the SourceForge. The scheduler and its simulation are written in Python.

The following is the list of servers in our data center and their attributes:

<table>
<thead>
<tr>
<th>ID</th>
<th>Reserved Name</th>
<th>ID</th>
<th>Reserved Name</th>
<th>Memory</th>
<th>Usage</th>
<th>CPU</th>
<th>Usage</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>blade043</td>
<td>2</td>
<td>blade044</td>
<td>450</td>
<td>0</td>
<td>2200</td>
<td>3</td>
<td>None</td>
</tr>
<tr>
<td>2</td>
<td>blade044</td>
<td>3</td>
<td>blade045</td>
<td>350</td>
<td>0</td>
<td>3200</td>
<td>3</td>
<td>None</td>
</tr>
</tbody>
</table>

The scheduler configuration for this demo is:

```
[Scheduler]
loadExpression = memUsage + coresUsage * 512 + cpuUsage
rescheduleAfterStart = True
optimalConsolidation = False
cpuSpeedOverrate = 2
```
What does this configuration mean? We will explain it using the examples. First, we will create the virtual machine with the following parameters:

```python
tashi-client.py createVm --userId 1 --name vm1 --cpuShare 1500 --disks tashi.img
```

A user with id 1 has created a VM with a CPU share 1500, with a 128 MB of memory (default value) and 1 CPU core (default value). The virtual machines list now:

```
1 2   vm1  user1  Running  tashi.img  128   1   1500
```

For a demonstration we will have two users - the user with id 1 with priority 1 and the user with id 2 with priority 2. The servers are ordered (order needs to be specified in the scheduler configuration) by their efficiency and when choosing the appropriate host the most efficient servers are always checked first:

```
[Servers]
# hostId = priority
1 = 0
3 = 1
2 = 2
```

The server with id 2 is the most efficient in our center and this is why the VM was started on this host (see hostId in the table). List of the hosts shows now that host 2 is now on. It was powered on to enable the provisioning of the vm1.

```
1 0 [ ] blade043 0 450 1 Normal None 2200 3 None
2 1 [ ] blade044 0 350 1 Normal None 3200 3 None
3 0 [ ] blade045 0 350 1 Normal None 4200 6 None
```
Let’s create a new VM now:

```
python tashi-client.py createVm --userId 1 --name vm2 --cpuShare 1600 --disks tashi.img
```

```
Macintosh:simulation mihastopar$ python tashi-client.py getInstances
id hostId name user state disk memory cores cpuShare
-----------------------------------------------
  1  2  vm1 user1 Running tashi.img 128  1  1500
  2  2  vm2 user1 Running tashi.img 128  1  1600
Macintosh:simulation mihastopar$
```

And two more (user with id 2 – priority 2):

```
python tashi-client.py createVm --userId 2 --name vm3 --cores 3 --disks tashi.img
```

```
python tashi-client.py createVm --userId 2 --name vm4 --cpuShare 4000 --disks tashi.img
```

```
Macintosh:simulation mihastopar$ python tashi-client.py getInstances
id hostId name user state disk memory cores cpuShare
-----------------------------------------------
  1  2  vm1 user1 Running tashi.img 128  1  1500
  2  2  vm2 user1 Running tashi.img 128  1  1600
  3  3  vm3 user2 Pending tashi.img 128  1  1024
  4  3  vm4 user2 Pending tashi.img 128  1  4000
Macintosh:simulation mihastopar$
```

Both virtual machines were started on host with id 3. There were no problems for the first one. But note that the second one (which has a large CPU share) actually shouldn’t be started on the host 3, because it has only 4200 available CPU speed (and 1024 is already consumed by vm3).

```
Macintosh:simulation mihastopar$ python tashi-client.py getInstances
id hostId name user state disk memory cores cpuShare
-----------------------------------------------
  1  2  vm1 user1 Running tashi.img 128  1  1500
  2  2  vm2 user1 Running tashi.img 128  1  1600
  3  3  vm3 user2 Running tashi.img 128  1  1024
  4  3  vm4 user2 Running tashi.img 128  1  4000
Macintosh:simulation mihastopar$
```

What happened? Let’s see the scheduler log first:

```
2011-03-29 09:07:21,238 - INFO - #################### new provisioning request - memory: 128, CPU speed: 4000, CPU cores: 1
2011-03-29 09:07:21,239 - INFO - Host 2 would have 384 memory, 3 CPU cores and 7100 CPU speed usage.
```
The host 2 was checked first, but there was not enough resources for vm4. Then the host 3 was checked and there was not enough resources as well (see CPU speed). Then the number of guaranteed servers (see the guaranteedServers scheduler configuration option) was reached, which is 2 in our configuration. That means that two servers are used for provisioning of the VMs before starting using over-provisioning. That also means that over-provisioning of the CPU resources will be used on the guaranteed number of servers before powering on the remaining servers.

A search for the appropriate server for a new virtual machine request can be essentially viewed as a constraint satisfaction problem (CSB). Let $h_i^{\text{cores}}$ denote the number of CPU cores on a physical server $h^i$, $h_i^{\text{cpu}}$ denote the CPU speed in MHz and let $h_i^{\text{mem}}$ denote the amount of memory of $h^i$ in megabytes. The number of physical servers is denoted by $k$ and the number of virtual machines that are running on server $h^i$ as $n^i$. Request for a new virtual machine can be described as $v = \{v_i^{\text{cores}}, v_i^{\text{cpu}}, v_i^{\text{mem}}\}$, where CPU cores, CPU speed and memory are specified for a virtual machine request. Running virtual machines are described as $v_{ij} = \{v_{ij}^{\text{cores}}, v_{ij}^{\text{cpu}}, v_{ij}^{\text{mem}}\}$, where $v_{ij}$ denotes a running virtual machine on a server $h^i$ and $j < n_i$. Let $OP_{\text{cores}}^i$ denote the over-provisioning rate for CPU cores and $OP_{\text{cpu}}^i$ the over-provisioning rate for CPU speed (see cpuCoresOverrate and cpuSpeedOverrate configuration option). When over-provisioning variables are
set to 1, the over-provisioning is not allowed. A problem of finding the most suitable server for over-provisioning can be illustrated as follows:

\[
\minimize \left( f(0, a_i^{cores}, a_i^{cpu}) \right), \text{where } 1 \leq i \leq k \text{ and:}
\]

\[
\sum_{j=1}^{n_i} v_{cores}^{ij} + v_{cores} \leq h_{cores}^i \times op_{cores}^i
\]

\[
\sum_{j=1}^{n_i} v_{cpu}^{ij} + v_{cpu} \leq h_{cpu}^i \times op_{cpu}^i
\]

\[
\sum_{j=1}^{n_i} v_{mem}^{ij} + v_{mem} \leq h_{mem}^i
\]

Regarding the guaranteedServers option - there are two scheduler types - Consolidated and Balanced (see the scheduler configuration option). The configuration option guaranteedServers makes more sense for the Consolidated type. To fully exploit the Balanced scheduler type the guaranteedServers option should be set to the number of all available servers.

A short description about the two scheduler implementations would be:

**Balanced scheduler:** When VMs are provisioned, they are spread evenly across the number of available hosts. Hosts with highest efficiency rating are filled first. VMs priorities come into account when an unsuccessful provisioning request happens - in this case and if provisioning request is of a higher priority - resources are taken from the VMs with the lowest priority to provide resources for a request. When some resources are released the VMs with the highest priority are migrated to a better hosts if possible. When rescheduling is executed the scheduler tries to balance VMs distribution across the servers. In general too many migrations are avoided, because migrations besides other effects also require two VMs instead of one (source and destination VM) at the time of migration.

**Consolidated scheduler:** When VMs are provisioned, each physical host is filled to its capacity before moving to the next one. Hosts with highest efficiency rating are filled first. VMs priorities come into account when an unsuccessful provisioning request happens - in this case and if provisioning request is of a higher priority - resources are taken from the VMs with the lowest priority to provide resources for a request. When some resources are released the VMs with the highest priority are migrated to better hosts if possible. When rescheduling is executed the scheduler tries to consolidate VMs distribution across the servers. In general too many migrations are avoided, because migrations besides other effects also require two VMs instead of one (source and destination VM) at the time of migration.

**When do we want to switch the scheduler type?** For example from Consolidated to the Balanced when we predict a rise in the cloud utilisation. The other way around when we predict a fall in the cloud utilisation.

Back to the guaranteedServers option - if the number of guaranteed servers would be 3, the scheduler would go checking the third host (host with id 1), but in our case the scheduler started checking the over-provisioning possibilities of
the hosts. The hosts were ordered by their over-provisioning load (load is calculated using the formula given by loadExpression in the scheduler configuration), host 3 had smaller over-provisioning load than host 2, which means that scheduler checked it first and it was able to provision vm4 there, because the cpuSpeedOverRate is 2, so using the over-provisioning we assume there is $2 \times 4200$ GHz CPU speed available.

Back to the virtual machines:

```python
tashi-client.py createVm --userId 1 --name vm5 --cores 3 --disks tashi.img
```

```
Macintosh:simulation mihastopar$ python tashi-client.py getInstances
id hostId name user state disk memory cores cpuShare
-----------------------------------------------
1 2 vm1 user1 Running tashi.img 128 1 1500
2 2 vm2 user1 Running tashi.img 128 1 1600
3 3 vm3 user2 Running tashi.img 128 1 1024
4 3 vm4 user2 Running tashi.img 128 1 4000
5 1 vm5 user1 Running tashi.img 128 3 1024
```

Host with id 1 was previously powered off, host 2 and 3 didn’t have enough resources (also no over-provisioning resources). Thus host 1 was powered on and vm5 was provisioned there.

```
Macintosh:simulation mihastopar$ python tashi-client.py getHosts
id on reserved name decayed memory up state version cpuSpeed cores notes
-----------------------------------------------
1 1 [] blade043 0 450 1 Normal None 2200 3 None
2 1 [] blade044 0 350 1 Normal None 3200 3 None
3 1 [] blade045 0 350 1 Normal None 4200 6 None
```

Let’s create another VM:

```python
tashi-client.py createVm --userId 2 --name vm6 --memory 200 --disks tashi.img
```

```
And the log:

```
2011-03-29 11:14:46,524 - INFO - Host 2 would have 456 memory, 3 CPU cores and 4124 CPU speed usage.
2011-03-29 11:14:46,524 - INFO - Available resources on host 2: 350 memory, 3 CPU cores, 3200 CPU speed
2011-03-29 11:14:46,525 - INFO - Available resources on host 2 (using over-provisioning): 350 memory, 3 CPU cores, 6400 CPU speed
2011-03-29 11:14:46,526 - INFO - Host 3 would have 456 memory, 5 CPU cores and 6048 CPU speed usage.
2011-03-29 11:14:46,527 - INFO - Available resources on host 3: 450 memory, 6 CPU cores, 4200 CPU speed
2011-03-29 11:14:46,527 - INFO - Available resources on host 3 (using over-provisioning): 450 memory, 6 CPU cores, 8400 CPU speed
2011-03-29 11:14:46,528 - INFO - Host 1 would have 328 memory, 4 CPU cores and 2048 CPU speed usage.
2011-03-29 11:14:46,529 - INFO - Available resources on host 1: 350 memory, 3 CPU cores, 2200 CPU speed
2011-03-29 11:14:46,529 - INFO - Available resources on host 1 (using over-provisioning): 350 memory, 3 CPU cores, 4400 CPU speed
2011-03-29 11:14:46,530 - INFO - Hosts that are currently powered off: []
2011-03-29 11:14:46,531 - INFO - ##################### determineHost: no id returned
2011-03-29 11:14:46,532 - INFO - Hosts that are currently powered off: []
2011-03-29 11:14:46,532 - INFO - CPU over-provisioning possibilities will be checked
2011-03-29 11:14:46,534 - INFO - Hosts and their over-provisioning load would be: {1: 512, 2: 924, 3: 1848}
2011-03-29 11:14:46,534 - INFO - Host 1 would have 328 memory, 4 CPU cores and 2048 CPU speed usage.
2011-03-29 11:14:46,535 - INFO - Available resources on host 1: 350 memory, 3 CPU cores, 2200 CPU speed
2011-03-29 11:14:46,535 - INFO - Available resources on host 1 (using over-provisioning): 350 memory, 3 CPU cores, 4400 CPU speed
2011-03-29 11:14:46,536 - INFO - Host 2 would have 456 memory, 3 CPU cores and 4124 CPU speed usage.
2011-03-29 11:14:46,537 - INFO - Available resources on host 2: 350 memory, 4 CPU cores, 2048 CPU speed
2011-03-29 11:14:46,537 - INFO - Available resources on host 2 (using over-provisioning): 350 memory, 3 CPU cores, 6400 CPU speed
2011-03-29 11:14:46,538 - INFO - Host 3 would have 456 memory, 5 CPU cores and 6048 CPU speed usage.
```

```
Macintosh:~ mihostopar$ python tashi-client.py getInstances

<table>
<thead>
<tr>
<th>id</th>
<th>hostId</th>
<th>name</th>
<th>user</th>
<th>state</th>
<th>disk</th>
<th>memory</th>
<th>cores</th>
<th>cpuShare</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>vm1</td>
<td>user1</td>
<td>Running</td>
<td>tashi.img</td>
<td>64</td>
<td>1</td>
<td>1500</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>vm2</td>
<td>user1</td>
<td>Running</td>
<td>tashi.img</td>
<td>64</td>
<td>1</td>
<td>1600</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>vm3</td>
<td>user2</td>
<td>Running</td>
<td>tashi.img</td>
<td>128</td>
<td>1</td>
<td>1024</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>vm4</td>
<td>user2</td>
<td>Running</td>
<td>tashi.img</td>
<td>128</td>
<td>1</td>
<td>4000</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>vm5</td>
<td>user1</td>
<td>Running</td>
<td>tashi.img</td>
<td>128</td>
<td>3</td>
<td>1024</td>
</tr>
<tr>
<td>6</td>
<td>2</td>
<td>vm6</td>
<td>user2</td>
<td>Running</td>
<td>tashi.img</td>
<td>200</td>
<td>1</td>
<td>1024</td>
</tr>
</tbody>
</table>
```
The memory of the machines with priority 1 (on host 1) was adapted to enable the provisioning of the new virtual machine (which has the priority 2). As soon as some memory will be freed up, the virtual machine will get back the adapted memory amount. The minimalVmMemory configuration option specifies the minimal amount of memory for the virtual machine, memory cannot be adapted under this amount.
Let's create another VM:

```bash
python tashi-client.py createVm --userId 2 --name vm7 --memory 200 --disks tashi.img
```

```
Macintosh:simulation mihastopar$ python tashi-client.py getInstances
  id  hostId  name    user  state    disk       memory  cores  cpuShare
  1   2       vm1     user1  Running  tashi.img  64       1      1500
  2   2       vm2     user1  Running  tashi.img  64       1      1600
  3   3       vm3     user2  Running  tashi.img  128     1      1024
  4   3       vm4     user2  Running  tashi.img  128     1      4000
  5   1       vm5     user1  Suspended tashi.img  128     3      1024
  6   2       vm6     user2  Running  tashi.img  200     1      1024
  7   1       vm7     user2  Running  tashi.img  200     1      1024
Macintosh:simulation mihastopar$
```

The vm5 was suspended in order to enable the provisioning of the vm7 (there were not enough machines for adapting their memory in order to enable the provisioning of the vm7).

Now we will destroy vm6:

```bash
python tashi-client.py destroyVm --userId 1 --instance 6
```

```
Macintosh:simulation mihastopar$ python tashi-client.py getInstances
  id  hostId  name    user  state    disk       memory  cores  cpuShare
  1   2       vm1     user1  Running  tashi.img  128     1      1500
  2   2       vm2     user1  Running  tashi.img  128     1      1600
  3   3       vm3     user2  Running  tashi.img  128     1      1024
  4   3       vm4     user2  Running  tashi.img  128     1      4000
  5   1       vm5     user1  Suspended tashi.img  128     3      1024
  7   1       vm7     user2  Running  tashi.img  200     1      1024
Macintosh:simulation mihastopar$
```

The virtual machines on the host 2 got back the adapted amount of memory. We will destroy now vm7:

```bash
python tashi-client.py destroyVm --userId 1 --instance 7
```
The virtual machine vm5 was resumed now. Let's destroy vm1 now:
```
python tashi-client.py destroyVm --userId 1 --instance 1
```

When some VM is destroyed, the scheduler checks if some VMs with can be moved to a host where the VM expired. In this case vm3 was migrated to the host 2. Let's now destroy vm2:
```
python tashi-client.py destroyVm --userId 1 --instance 2
```

No VMs were migrated now, because there is no enough resources for vm4 on host 2 (see CPU share). The virtual machine vm5 was not considered to be migrated, because it has lower priority as vm3 (too many migrations are avoided).

Let's restart the scheduler now with rescheduleAfterStart configuration option enabled and with optimalConsolidation set to False. The result is:
Virtual machine vm3 was migrated to the host 3 to reduce the number of powered on servers. Host 2 was powered off. If optimalConsolidation is set to False, the hosts with the smallest usage / load are attempted to be released first (in this case host 2 has the smallest load – see the number of CPU cores and CPU share, memory usage is at this stage the same for all hosts). If optimalConsolidation is set to True, the hosts with the smallest efficiency rating are attempted to be released first. In some cases optimal consolidation means a lot of migrations, because some hosts with smallest efficiency can be heavily used and a lot of migrations are needed to achieve a better distribution across the hosts (to move the load to the servers with the highest efficiency rating and to power off the older, slower hosts).

Let’s set the configuration now to use the Balanced scheduler implementation and set the guaranteedServers option to be 3. Restart the scheduler:

```
Macintosh:simulation mihastopar$ python tashi-client.py getInstances
id hostId name user state disk memory cores cpuShare
-----------------------------------------------
 3  2  vm3  user2 Running tashi.img  128  1  1024
 4  3  vm4  user2 Running tashi.img  128  1  4000
 5  1  vm5  user1 Running tashi.img  128  3  1024
Macintosh:simulation mihastopar$  
```

Host 2 was powered on and vm3 was migrated from host 3 to host 2 to achieve a more balanced load.