bareon-dynamic-allocator Documentation

Release

OpenStack Foundation

September 01, 2016

Contents

Contents:

CHAPTER 1

bareon-dynamic-allocator

A driver for Bareon for dynamic allocation of volumes

Please feel here a long description which must be at least 3 lines wrapped on 80 cols, so that distribution package maintainers can use it in their packages. Note that this is a hard requirement.

- Free software: Apache license
- Documentation: http://docs.openstack.org/developer/bareon-dynamic-allocator
- Source: http://git.openstack.org/cgit/openstack/bareon-dynamic-allocator
- Bugs: http://bugs.launchpad.net/bareon

1.1 Future Improvments

- create special types, like lv_mirror with special policy to allocate volume of the same size over several disks
- implement less or equal instead of equal for disk size constraint in this case artificial Unallocated space is not going to be required

CHAPTER 2

Architecture

2.1 Problem description

User may have a variety of bare-metal nodes configuration, with different amount of disks, types of disks and their sizes, there should be a way to store best practises on what is the best way to do partitioning, so they can be applied for the most configuration cases without asking the end user to manually adjust the configuration of partitioning, with posibility to do that, if user wants to.

2.2 History

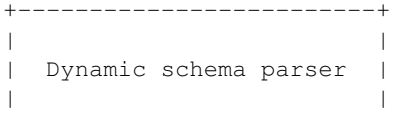
First (and second) attempts to solve the problem has begun during development of Fuel project, special module VolumeManager was created to solve the problem, it consumes hardware information and partitioning schema, as result it generates sizes of spaces which should be allocated on the disks.

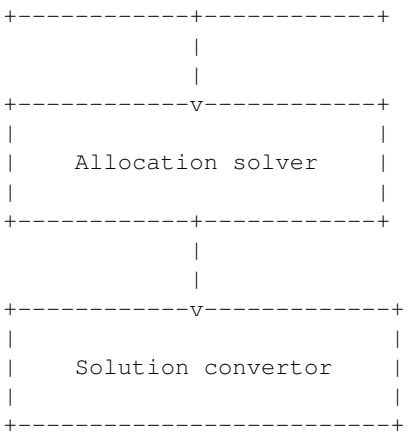
Current solution has plenty of problems, it's hard and expensive to solve these problems in terms of old VolumeManager, because trivial algorithms and schema format don't allow us to extend it easily, handle all complex cases is not a trivial task to do if we try to solve the problem using brute-force.

2.3 List of terms

- **Disk** - a place where space can be allocated.
- **Space** - an entity which can be allocated on several disks at once, a good example of a space is a logical volume for lvm, another one is partition.
- **Dynamic schema** - a schema without specific sizes, it's a schema which is used by user to specify partitioning schema without details.
- **Static schema** - a schema for Bareon which requires exact space, i.e. disk mapping with exact sizes for each space.

2.4 High level architecture





- **Dynamic schema parser** - parses an input from the user and prepares the data which can be consumed by Allocation solver.
- **Allocation solver** - algorithm which takes dynamic schema and produces a static schema.
- **Solution convertor** - a result which is produced by solver, should be parsed and converted into Bareon consumable format, for example for Logical Volume Solution convertor should generate a physical volume for each disk, where it's allocated.

2.5 Dynamic schema parser

In the current version we use flat schema, it's a list which consists of dictionaries.

2.5.1 Basic syntax

- **id** - id of a space.
- **type** - type of a space, for example Volume Group or Logical Volume.
- **max_size** - maximum size which is allowed for the space.
- **min_size** - minimal size which is allowed for the space.
- **size** - a static size, it's similar as to set for **min_size** and **max_size** the same value.
- **contains** - is required for hierarchical spaces such as Volume Group.

Also there are couple of different attributes, such as **mount**, **fs_type**, which are self-explanatory. A list of such attributes is not complete and may be easily extended in the future.

```
- id: os
  type: vg
  contains:
    - id: swap
    - id: root

- id: root
  type: lv
  max_size: 10000
  min_size: 5000
  mount: /
  fs_type: ext4
```

```
- id: swap
  type: lv
  size: 2000
  fs_type: swap
```

2.5.2 Dynamic parameters

What if user wants to allocate a size of space based on some different parameter? As an example lets consider a size of **swap** which has to be based on amount of RAM the node has.

```
ram: 2048
disks:
  - id: /dev/disk/by-id/id-for-sda
    path: /dev/disk/by-path/path-for-sda
    dev: /dev/sda
    type: hdd
    vendor: Hitachi
    size: 5000
```

From Hardware Information example we can see that the node has **2048** megabytes of RAM, according to best practises on swap size allocation swap size has to be twice bigger than current RAM.

```
- id: swap
  type: lv
  fs_type: swap
  size: |
    yaql=let(ram => $.get(ram, 1024)) ->
    selectCase(
      $ram <= 2048,
      $ram > 2048 and $ram < 8192,
      $ram > 8192 and $ram < 65536).
    switchCase(
      $ram * 2,
      $ram,
      $ram / 2,
      4096)
```

In order to implement algorithm of swap size calculation we use YAQL, which is a small but powerful enough query language. Any value of the parameter which matches to **yaql=yaql expression** will be evaluated using YAQL, execution result will be passed as is to the solver.

2.6 Allocation solver

Lets try to generalize the problem of spaces allocation:

- There are constraints, for example sizes of a spaces cannot be bigger than size of all disks, or size of swap space cannot be bigger or smaller than **size** of the space.
- There exists "the best allocation static schema", it's almost impossible to find out what "the best" is, what we can do is to parse all constraint and find such an allocation which fits all the constraints, and at the same time uses given resources (disks) by maximum.

Lets consider an example with two spaces and a single disk. Parameters which don't affect allocation problem were removed to reduce the amount of unnecessary information.

Two spaces **root** and **swap**, for **swap** there is static size which is **10**, the size of **root** space must be **50** or greater.

```
- id: root
  min_size: 50

- id: swap
  size: 10
```

A single disk with size **100**.

```
disks:
  - id: sda
    size: 100
```

Also we can describe the same problem as

$$\begin{cases} root + swap \leq 100 \\ root \geq 50 \\ swap = 10 \end{cases}$$

On disks with bigger sizes we can get a lot of solutions.

Lets consider two corner case solutions

$$root = 50, \quad swap = 10$$

and

$$root = 90, \quad swap = 10$$

Second one is better since it uses more disks resources and doesn't leave unallocated space. So we should find a way to describe that second one is better.

It can be described with the next function.

$$Maximize : root + swap$$

2.6.1 Solver description

The problem is described in terms of Linear programming (note that "programming" is being used in not computer-programming sense). The method is being widely used to solve optimal resources allocation problem which is exactly what we are trying to achieve during the allocation.

$$max \left\{ c^T x : Ax \geq b \right\}$$

- $\mathbf{c^T x}$ - is an objective function for maximization
- **c** - a vector of coefficients for the values to be found
- **x** - a vector of result values
- **A** - coefficients matrix
- **b** - a vector, when combined with a row from matrix **A** gives a constraint

Description of previous example in terms of Linear programming, is going to be pretty similar to what we did in previous section.

$$x_1 = root$$
$$x_2 = swap$$

Coefficients for objective function.

$$c = \begin{bmatrix} 1 & 1 \end{bmatrix}^T$$

A vector of values to be found, i.e. sizes of spaces.

$$x = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$$

System of linear inequalities. Inequalities which are "less or equal" multiplied by -1 to make them "greater or equal".

$$Ax \geq b = \begin{cases} -x_1 - x_2 & \geq -100 \\ x_1 & \geq 50 \\ -x_2 & \geq -10 \\ x_2 & \geq 10 \\ x_1 & \geq 0 \\ x_2 & \geq 0 \end{cases}$$

A and **b** written in matrix and vector form respectively.

$$A = \begin{bmatrix} -1 & -1 \\ 1 & 0 \\ 0 & -1 \\ 0 & 1 \\ 1 & 0 \\ 0 & 1 \end{bmatrix}$$

$$b = \begin{bmatrix} -100 \\ 50 \\ -10 \\ 10 \\ 0 \\ 0 \end{bmatrix}$$

In order to solve the problem Scipy linprog module is being used. It uses Simplex algorithm to find the most feasible solution.

So what allocator does, is builds a matrix and couple of vectors and using Simplex algorithm gets the result.

2.6.2 Two disks

If there are two spaces and two disks, there are going to be 4 unknown variables:

1. 1st space size for 1st disk.
2. 2nd space size for 1st disk.
3. 1st space size for 2nd disk.
4. 2nd space size for 2nd disk.

Lets take spaces definition which was used previously.

```
- id: root
  min_size: 50

- id: swap
  size: 10
```

And two disks.

```
disks:
  - id: sda
    size: 100

  - id: sdb
    size: 200
```

Resulting system of linear inequalities.

$$\begin{cases} x_1 + x_2 \leq 100 \\ x_3 + x_4 \leq 200 \\ x_1 + x_3 \geq 50 \\ x_2 + x_4 = 10 \end{cases}$$

- $x_1 + x_2 \leq 100$ inequality for root and swap on the 1st disk
- $x_3 + x_4 \leq 200$ inequality for root and swap on the 2nd disk
- $x_1 + x_3 \geq 50$ inequality for root space
- $x_2 + x_4 = 10$ equality for swap space

2.6.3 Integer solution

By default result vector provides rational number vector solution. Very naive way is being used to get integer soluton, we round the number down, this solution may have problems because some of the constraints may be violated with respect to one megabyte. Another side effect is we may get **N** megabytes unallocated in the worst case, where **N** is an amount of spaces. For our application purposes all above drawbacks are not so big, considering a complexity of proper solution.

Mixed integer programming can be used to get integer result, but solution for problems described in terms of Integer programming may be NP-hard. So it should be considered carefully if it's worth to be used.

2.6.4 Ordering

It would be really nice to have volumes allocated on disks in the order which was specified by the user.

Lets consider two spaces example.

```
- id: root
  size: 100

- id: var
  size: 100
```

With two disks.

```
disks:
  - id: sda
    size: 100

  - id: sdb
    size: 100
```

Which can be represented as next inequality.

$$\begin{cases} x_1 + x_2 \leq 100 \\ x_3 + x_4 \leq 100 \\ x_1 + x_3 = 100 \\ x_2 + x_4 = 100 \end{cases}$$

And objective function.

$$Maximize : x_1 + x_2 + x_3 + x_4$$

So we may have two obvious solutions here:

1. **var** for 1st disk, **root** for 2nd.
2. **root** for 1st disk, **var** for 2nd.

Objective function is being used by the algorithm to decide, which solution is "better". Currently all elements in coefficients vector are equal to 1

$$c = \begin{bmatrix} 1 & 1 & 1 & 1 \end{bmatrix}^T$$

We can change coefficients in a way that first volume has higher coefficient than the last one.

$$c = \begin{bmatrix} 4 & 3 & 2 & 1 \end{bmatrix}^T$$

Now Linear Programming solver will try to maximize the solution with respect to specified order of spaces.

But that is not so simple, if we take a closer look at the results we may get. Lets consider two solutions and calculate the results of objective function.

$$c^T x = \begin{bmatrix} 4 & 3 & 2 & 1 \end{bmatrix} \begin{bmatrix} 100 \\ 0 \\ 0 \\ 100 \end{bmatrix} = sum \begin{bmatrix} 400 \\ 0 \\ 0 \\ 100 \end{bmatrix} = 500$$

The result that objective function provides is **500**, if **root** is allocated on the first disk and **var** on second one.

$$c^T x = \begin{bmatrix} 4 & 3 & 2 & 1 \end{bmatrix} \begin{bmatrix} 50 \\ 50 \\ 50 \\ 50 \end{bmatrix} = sum \begin{bmatrix} 200 \\ 150 \\ 100 \\ 50 \end{bmatrix} = 500$$

The result that objective function provides is **500**, if **root** and **var** are allocated equally on both disks.

So we need a different monolitically increasing sequence of integers, which is increasing as slow as possible.

Also sequence must not violate next requirements.

$$n_{i+1} n_i \tag{2.1}$$

$$n_i + n_{j+1} n_{i+1} + n_j \quad \text{where} \quad i + 1 (2.2)$$

If we apply it to our example with **4** coefficients, it means that a sum of **bold** elements must not be equal.

$$\begin{matrix} \mathbf{c_1} & c_2 \\ c_3 & \mathbf{c_4} \end{matrix} \neq \begin{matrix} c_1 & \mathbf{c_2} \\ \mathbf{c_3} & c_4 \end{matrix}$$

In the example this requirement is violated

$$i = 1 \tag{2.3}$$
$$j(2.4)$$
$$1 + 4 = 2(2.5)$$

A sequence which doesn't not violate these requirements has been found

$$1, 2, 4, 6, 9, 12, 16, 20, 25, 30, 36, 42 \cdots$$

there are many ways to caculate such sequence, in our implementation next one is being used

$$a_n = \lfloor \frac{n+1}{2} \rfloor \lfloor \frac{n+2}{2} \rfloor$$

Lets use this sequence in our examples

$$c^T x = \begin{bmatrix} 6 & 4 & 2 & 1 \end{bmatrix} \begin{bmatrix} 100 \\ 0 \\ 0 \\ 100 \end{bmatrix} = sum \begin{bmatrix} 600 \\ 0 \\ 0 \\ 100 \end{bmatrix} = 700$$

And when **root** and **var** are allocated on both disks equally

$$c^T x = \begin{bmatrix} 6 & 4 & 2 & 1 \end{bmatrix} \begin{bmatrix} 50 \\ 50 \\ 50 \\ 50 \end{bmatrix} = sum \begin{bmatrix} 300 \\ 200 \\ 100 \\ 50 \end{bmatrix} = 650$$

So $700 > 650$, first function has greater maximization value, that is exactly what we needed.

2.6.5 Weight

Two spaces, no exact size specified.

```
- id: root
  min_size: 10

- id: var
  min_size: 10
```

A single disk.

```
disks:
  - id: sda
    size: 100
```

According to coefficients of objective funciton with respect to ordering, we will have the next allocation.

- **root** - 90
- **var** - 10

Which is not so obvious result for the user, the expected result would be to have the next allocation.

- **root** - 50
- **var** - 50

So for those spaces, which have the same **min_size**, **max_size** (and **best_with_disks** see next section), allocator adds special equality to make sure that there is a fair allocation between spaces with same requirements.

Each space can have **weight** variable specified (**1** by default), which is used to make additional equality.

$$\begin{cases} x_1 + x_2 \leq 100 \\ x_3 + x_4 \leq 200 \\ x_1 + x_3 = 100 \\ x_2 + x_4 = 100 \\ x_2 * (1/weight) - x_4 * (1/weight) = 0 \end{cases}$$

To satisfy last equality, spaces have to be equal in size. If it's required to have one space twice smaller than the other one, it can be done by setting the weight variable.

```
- id: root
  size: 10
  weight: 1

- id: var
  size: 10
  weight: 0.5
```

As result for **var** will be allocated twice smaller space on the disk.

2.6.6 Best with disks

User may want a space to be allocated on specific disk according to any attribute of a disk.

For example lets consider an example with **ceph-journal** which is better to allocate on **ssd** disks.

From user's perspective each space can have a new parameter **best_with_disks**, in order to fill in this parameter YAQL can be used.

```
- id: ceph-journal
  best_with_disks: |
    yaql=$.disks.where($.type = "ssd")

- id: root
  min_size: 10

disks:
  - id: sda
    size: 100
    type: hdd

  - id: sdb
    size: 10
    type: ssd
```

So in solver we get a list of **ssd** disks, if there are any.

Lets adjust coefficients to make ceph-journal to be allocated on ssd, as a second priority ordering should be respected.

In order to do that lets make order coefficient $0 <$ order coefficient < 1.

$$c = \begin{bmatrix} 0 + (1/2) \\ 1 + (1/4) \\ 1 + (1/6) \\ 0 + (1/9) \end{bmatrix}$$

or

$$c^T x = x_1 \cdot (0 + 1/2) + x_2 \cdot (1 + 1/4) + x_3 \cdot (1 + 1/6) + x_4 \cdot (0 + 1/9)$$

1. Build sets according to selected disks, in our case we have two sets, **hdd** and **ssd** disks.
2. For spaces which belong to specific set of disks add **1** to a coefficient which represents this space on a disk from the set.
3. Spaces which do not belong to any disks sets are assigned to set of disks which is left, in our case it is **hdd** disks set.

To make sure that spaces are always (unless size constraints are not violated) allocated on the disks which they best suited with, we automatically add a special artificial volume **unallocated**, whose coefficient is always **1**, and in this case we should change coefficient of space which belongs to the set of disks to **2**.

$$c = \begin{bmatrix} 0 + (1/2) \\ 2 + (1/4) \\ 1 \\ 2 + (1/9) \\ 0 + (1/12) \\ 1 \end{bmatrix}$$

As the result if space has one or more **best_with_disks**, it will be allocated on specified disks only.

CHAPTER 3

Allocation Examples

3.1 ceph_ds_multiple_disk

3.1.1 Hardware information

```
ram: 1024

disks:

  - id: sda
    path: /dev/disk/by-path/path-for-sda
    dev: /dev/sda
    type: hdd
    vendor: Hitachi
    size: 10000

  - id: sdb
    path: /dev/disk/by-path/path-for-sdb
    dev: /dev/sdb
    type: hdd
    vendor: Hitachi
    size: 10000

  - id: sdc
    path: /dev/disk/by-path/path-for-sdc
    dev: /dev/sdc
    type: hdd
    vendor: Hitachi
    size: 10000

  - id: sde
    path: /dev/disk/by-path/path-for-sde
    dev: /dev/sde
    type: ssd
    vendor: Hitachi
    size: 2048
```

3.1.2 Dynamic schema

```
- id: ceph
  type: partition
  fs_type: ext4
  min_size: 1000
  best_with_disks: |
    yaql=$.disks.where($.type = "hdd").skip(1)

- id: ceph-journal
  type: partition
  best_with_disks: |
    yaql=$.disks.where($.type = "ssd")

- id: os
  type: vg
  contains:
    - id: swap
    - id: root

- id: swap
  type: lv
  size: |
    yaql=let(ram => $.get(ram, 1024)) ->
    selectCase(
      $ram <= 2048,
      $ram > 2048 and $ram < 8192,
      $ram > 8192 and $ram < 65536).
    switchCase(
      $ram * 2,
      $ram,
      $ram / 2,
      4096)
  fs_type: swap

- id: root
  type: lv
  min_size: 5000
  mount: /
  fs_type: ext4
```

3.1.3 Allocation result

3.2 ceph_ds_single_disk

3.2.1 Hardware information

```
ram: 1024

disks:
  - id: sda
    path: /dev/disk/by-path/path-for-sda
    dev: /dev/sda
    type: hdd
```

```
    vendor: Hitachi
    size: 10000
```

3.2.2 Dynamic schema

```
- id: ceph
  type: partition
  fs_type: ext4
  min_size: 1000
  best_with_disks: |
    yaql=$.disks.where($.type = "hdd").skip(1)

- id: ceph-journal
  type: partition
  best_with_disks: |
    yaql=$.disks.where($.type = "ssd")

- id: os
  type: vg
  contains:
    - id: swap
    - id: root

- id: swap
  type: lv
  size: |
    yaql=let(ram => $.get(ram, 1024)) ->
    selectCase(
      $ram <= 2048,
      $ram > 2048 and $ram < 8192,
      $ram > 8192 and $ram < 65536).
    switchCase(
      $ram * 2,
      $ram,
      $ram / 2,
      4096)
  fs_type: swap

- id: root
  type: lv
  min_size: 5000
  mount: /
  fs_type: ext4
```

3.2.3 Allocation result

3.3 simple_os_ds_multiple_disk

3.3.1 Hardware information

```
ram: 1024

disks:
```

```
  - id: sda
    path: /dev/disk/by-path/path-for-sda
    dev: /dev/sda
    type: hdd
    vendor: Hitachi
    size: 10000

  - id: sdb
    path: /dev/disk/by-path/path-for-sdb
    dev: /dev/sdb
    type: hdd
    vendor: Hitachi
    size: 10000

  - id: sdc
    path: /dev/disk/by-path/path-for-sdc
    dev: /dev/sdc
    type: hdd
    vendor: Hitachi
    size: 10000

  - id: sde
    path: /dev/disk/by-path/path-for-sde
    dev: /dev/sde
    type: ssd
    vendor: Hitachi
    size: 2048
```

3.3.2 Dynamic schema

```
- id: os
  type: vg
  contains:
    - id: swap
    - id: root

- id: root
  type: lv
  size: 5000
  mount: /
  fs_type: ext4

- id: swap
  type: lv
  size: |
    yaql=let(ram => $.get(ram, 1024)) ->
    selectCase(
      $ram <= 2048,
      $ram > 2048 and $ram < 8192,
      $ram > 8192 and $ram < 65536).
    switchCase(
      $ram * 2,
      $ram,
      $ram / 2,
      4096)
  best_with_disks: |
```

```
    yaql=$.disks.where($.type = "ssd")
  fs_type: swap

- id: logs
  type: vg
  contains:
    - id: log

- id: log
  type: lv
  mount: /var/log
  fs_type: ext4
  size: 1000

- id: data
  type: vg
  contains:
    - id: data_var_lib

- id: data_var
  type: lv
  fs_type: ext4
  min: 1000
  mount: /var
```

3.3.3 Allocation result

3.4 simple_os_ds_single_disk

3.4.1 Hardware information

```
ram: 1024

disks:
  - id: sda
    path: /dev/disk/by-path/path-for-sda
    dev: /dev/sda
    type: hdd
    vendor: Hitachi
    size: 10000
```

3.4.2 Dynamic schema

```
- id: os
  type: vg
  contains:
    - id: swap
    - id: root

- id: root
  type: lv
  size: 5000
  mount: /
```

```
  fs_type: ext4

- id: swap
  type: lv
  size: |
    yaql=let(ram => $.get(ram, 1024)) ->
    selectCase(
      $ram <= 2048,
      $ram > 2048 and $ram < 8192,
      $ram > 8192 and $ram < 65536).
    switchCase(
      $ram * 2,
      $ram,
      $ram / 2,
      4096)
  best_with_disks: |
    yaql=$.disks.where($.type = "ssd")
  fs_type: swap

- id: logs
  type: vg
  contains:
    - id: log

- id: log
  type: lv
  mount: /var/log
  fs_type: ext4
  size: 1000

- id: data
  type: vg
  contains:
    - id: data_var_lib

- id: data_var
  type: lv
  fs_type: ext4
  min: 1000
  mount: /var
```

3.4.3 Allocation result

CHAPTER 4

Installation

At the command line:

```
$ pip install bareon-dynamic-allocator
```

Or, if you have virtualenvwrapper installed:

```
$ mkvirtualenv bareon-dynamic-allocator
$ pip install bareon-dynamic-allocator
```

CHAPTER 5

Usage

Has not been released yet, see example of usage in **run.sh** file.

CHAPTER 6

Contributing

If you would like to contribute to the development of OpenStack, you must follow the steps in this page:

> http://docs.openstack.org/infra/manual/developers.html

If you already have a good understanding of how the system works and your OpenStack accounts are set up, you can skip to the development workflow section of this documentation to learn how changes to OpenStack should be submitted for review via the Gerrit tool:

> http://docs.openstack.org/infra/manual/developers.html#development-workflow

Pull requests submitted through GitHub will be ignored.

Bugs should be filed on Launchpad, not GitHub:

> https://bugs.launchpad.net/bareon-dynamic-allocator

CHAPTER 7

Indices and tables

- *genindex*
- *modindex*
- *search*