The BtrPlace Constraints Catalog

Fabien Hermenier

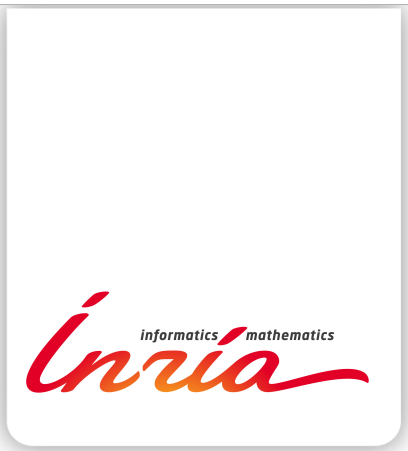
TECHNICAL REPORT

N° ????

08/19/2012

Project-Team OASIS

ISSN 0249-0803 ISRN INRIA/RT--????--FR+ENG



The BtrPlace Constraints Catalog

Fabien Hermenier*

Project-Team OASIS

Technical Report n° ???? — 08/19/2012 — 71 pages

Abstract: Datacenters provide through virtualization, platforms of choice to host a large range of applications. At submission, each tenant provides through a Service Level Agreement its concerns in terms of VMs management. These concerns are typically related to performance, reliability, or isolation criteria. In parallel, platform administrators have management criteria related to power management, resource allocation or security concerns.
All of these constraints have to be considered by the VM manager running on the hosting platform to make management decisions aligned with the tenants and the administrators expectations. Each existing VM manager provides its own constraints to address some of these criteria or allow to integrate external constraints. However, the semantic of a constraint varies with the VM manager while constraints' users lack of expertise when they have to select or develop new constraints matching their expectations.
This catalog gathers constraints that are of a practical interest to manage VMs and servers in a hosting platform such as a datacenter or a cloud. Each constraint is associated with practical use cases and classification criteria to help users at understanding its interest. In addition, a formal definition and a vendor neutral model allow to understand the constraint impact on a VM manager decision. Finally, the availability of each constraint in common VM manager is described while its formal model provides the fundaments for an implementation in flexible VM managers.

Key-words: Virtualization; Datacenter; Resource Management; Service Level Agreements; Reconfiguration

* INRIA - CNRS - I3S, Univ. Nice Sophia-Antipolis, –fabien.hermenier@inria.fr

RESEARCH CENTRE
SOPHIA ANTIPOLIS – MÉDITERRANÉE

2004 route des Lucioles - BP 93
06902 Sophia Antipolis Cedex

The BtrPlace Constraints Catalog

Résumé : Pas de résumé

Mots-clés : Virtualisation; Centre de données; Gestion des ressources; Qualité de service; Reconfiguration

Contents

Chapter 1

Preface

A joint effort between academics and industrial to homogenize hosting platform usage. Virtual Machine descriptor [36]

DMTF [15], OCCI [35] CDMI [11] Datacenters provide through virtualization, platforms of choice to host a large range of applications. To perform a deployment, each application administrator embeds his application into Virtual Machines (VMs) that are placed by the VM manager on the datacenter servers. In addition to the VMs, the application administrator provides through a Service Level Agreement (SLA), his concerns in terms of VMs management. The VM manager must then place every applications' VM, according to their SLA, to satisfy their expectations in terms of security, availability, or performance. Placement requirements are subjects to evolution over the time and new expectations emerge regularly depending on the applications' domain, computer science trends, or new technologies. The VM manager should then support these evolutions as soon as possible to keep being suitable to host a large range of applications with their particular placement requirements.

Traditional VM managers such as VMWare DRS [47], Amazon EC2 [2] are not designed for extensibility. When new needs emerge, they have to modify their placement algorithm and change their application interface. On the other side, Flexible VM managers [8, 24, 25, 31] argue for proactive approaches. In addition to a list of already supported placement constraints, they allow third party developers to implement their own placement constraints that can be plugged into the VM manager. As an example, BtrPlace provided a first library of 12 placement constraints to express dependability requirement and the Fit4Green projects [17] extended it internally to make BtrPlace able to address power saving and hardware requirements concerns.

In practice, the variety of the supported constraints

Required expertise -> several constraints may lead to the same result in terms of placement. However, depending on the context and their use, their efficiency may be different (unjustified scalability degradation,

proposed constraints and the development of new constraints require a certain expertise for being fully effective. As an example, several combination of constraints may lead to the same result however, each constraint may have a specific domain or scope which make it particularly suitable in a specific context. As a result, their usage may lead to unexpected results or may limit the VM manager scalability due to an inappropriate usage. The same situation

may occur when developers implement a new constraint to fulfill a lack in a VM manager. The lack of knowledge about the possible optimization techniques or a miss-understood of the real constraints expectation and behavior may result in a not fully effective and reusable constraint.

In this paper, we arguing for a catalog We provide through BtrPlace an flexible virtual machines manager for datacenters to address applications and datacenter administrators expectations in terms of placement constraints. At the first stage, it allows users to pick up among the available constraints in its library, those who match placement requirements.

Standard software documentation is not enough. Several solutions are possible to express the same requirements but they are not redundant as they differ in their performance in particular context. So the administrators must be able to understand relationships between constraints. Aside, they have to be able to choose the right constraints when a problem is not stated exactly in the documentation. In this situation, the administrator background may be misleading and the demonstration of the constraint usage in several contexts is required.

-> have to

Potential problems:

-> miss understanding of existing constraints characteristics. several constraints may appear at first sight to be suitable to cover a specific requirement but a lack of knowledge or informations may lead to a non-expected result or poor performance if the choosed constraints are not the most efficient approach to handle the requirements. ex: 1 globalCapacity vs. n singleCapacity ex: 1 among constraint with only 1 group of nodes -> choose a fence constraint

-> users have requirements that are thighly coupled with their own use case. This lack of perspective may mislead the users from the real requirements they have to address but also let them ignore existing solutions to problems having similar facets. - users will re-invent the wheel - users may develop useless constraints that have an unnecessary specialization, limiting its reusability

-> users may want to develop a constraint matching his requirements. The gap to be able to provide the first constraint may be big despite the few lines of code that may be required : context of constraint programming, specific model, scalability requirements. -> bad developments

A catalog of constraints, based on BtrPlace characteristics, would guide users at being able to express their requirements through: - combination of existing constraints - the development of new constraints

The catalog should provide: - description of the context to let users understand the VMs and servers management capabilities - categorization of constraints by concern / users' role / manipulated elements / ... - strong uses cases to let users discover the applicability range of each constraint - a neutral model to allow them to think about the constraint model rather than implementation details - relationships between constraints to indicate + constraints that may overlap other constraints in certain conditions, + constraints that are more powerful than other in certain conditions - description of each constraint model and methods to detect violating elements

The rest of the report is organized as follow.

In Chapter **??**, we describe the fundaments related to a hosting platform dedicated to virtualization. In Chapter 2 we present the principles of a reconfiguration process, the supported virtual machines and servers action. In

Chapter **??** we present a format model to represent a reconfiguration process. In Chapter 3 we detail each of the constraints available in the catalog

In Chapter 4, we present several notations that are used to describe configurations, reconfigurations and constraints in a textual or in an illustrated form.

Chapter 2

Virtualized hosting platforms

Over the last 20 years, the design of platforms dedicated to the execution of parallel and distributed applications has evolve from monolithic supercomputers to a flexible aggregation of interconnected servers made up with commodity hardware. In 1994, the NASA deploy a 16-nodes clusters called *Beowulf* [43] to perform physical simulation and data acquisition. Nodes, having one 100 MHz processor each, were connected through a 10 Mb ethernet network.

Contrary to supercomputers, clusters are made for being updated. It is then possible to increase its performance by adding additional nodes. They are also cheaper to build. As an example,Barroso *et al.* [7] indicate that in 2003, the cost of a 88 nodes cluster, each having two 2 GHz Intel Xeon processors, 2 Gb RAM and a 80 Gb of disk storage is 8 times cheaper than a single server having 8 processors, 64 GB RAM and 8 To of disk storage. Clusters appeared then to be less expensive and provide a better cost/performance ratio. Over the years, this design became a reference for high performance computing [44] and hosting platforms.

Orientation service, utilization variable, coût d'exploitation. Amazon on demand computing, [12]
cloud computing, testbeds

2.1 Infrastructure

2.1.1 Hosting platforms

A hosting platform are made up using multiple interconnected servers that are used to host virtual machines. In this section, we present the typical infrastructure of an hosting platform and its usage to run virtualized applications.

Working nodes

Working nodes are dedicated to the hosting of the users application. In a virtualized hosting platform, each working node runs an *hypervisor* to allow to run simultaneously multiple operating systems called *virtual machines*. In practice, an *hypervisor* hides to the VMs the physical characteristics of the server and provide to each VM the illusion of having a real dedicated physical machine. The VM administrator can run its own operating system and applications. The

hypervisor is then responsible of ensuring the isolation between the VMs and sharing amount them its resources.

It currently exists different hypervisors for hosting platforms, *e.g.*Xen [6], KVM [29], VMWare ESX [39], or Microsoft Hyper-V [32]. Each differs by its features, performance or management capabilities. Each server provides a part of its resources to the VMs it hosts. It typically shares among the VMs the CPU, the network, the disk and the memory resources. It is possible to control this sharing and dedicate specific resources such as CPUs or core to the VMs. It is also possible to bind to VMs, specific hardware that is available from the server. Typically PCI expansion cards or disk partition.

Rappeler ressource partage fixe, partage floue (cpu et mémoire)

partage mémoire [**?**, **?**]

The resource sharing policy is decided when the VM or the hypervisor is configured.

The heterogeneity between the servers in a datacenter exists but it is limited. In practice, at construction, the datacenter is designed around one specific type of server. However, when the datacenter is upgraded to increase its hosting capacity, new servers are acquired. In this situation, the new servers hardware differ from the previous one to provide a more competing performance.

Network

The current practice in network design is to rely on a hierarchical network called *fat tree* [30]. Servers are connected to *edge switches* using gigabit ethernet links [18]. These edge switches are then connected to *aggregation switches*. Finally, the aggregation switches are connected to the *core switches*. Links between the switches are usually an aggregation of multiple gigabit or 10-gigabit ethernet links [16] to provide a bisection bandwidth as high as possible. Usually, the network is fully replicated to support the loss of switches. Figure 2.1 depicts a replicated fat tree with one level of aggregation switches.

In a fat tree topology, the bisection bandwidth is usually not provisioned enough to support a traffic at full speed and the network latency between the nodes is not constant all over the datacenter. The bisection bandwidth limitation is explained in a fat tree by the unit cost of the network equipments but also the limitation of the current network technologies that can not provide to modular switches a non-blocking bandwidth. The non-constant latency is explained by the number of network hops between nodes that differ depending on their relative position. Recently, Al-Fares *et al.* [1] proposed a modification of the traditional fat tree design to provide a non-blocking bisection bandwidth while using commodity switches. In addition, Greenberg *et al.* [20] proposed an alternative network design which guarantee a non-blocking bisection bandwidth, while offering a constant latency between the servers.

Storage

SAN, or local disk

One or multiple DFS, available to part of the infrastructure

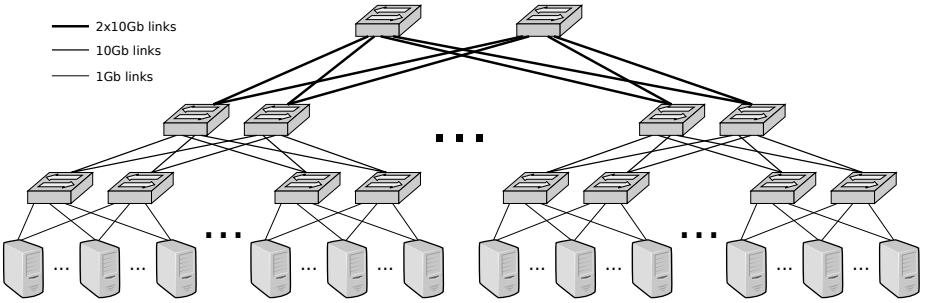


Figure 2.1: A fat-tree network topology with redundant links for fault tolerance

From server housing to container housing

A recent trend in datacenter design relies on the usage of shipping containers to embed the servers [23]. In practice, a container can be considered as a datacenter. It may embeds thousands of servers with their network [21], their storage and their cooling systems. Shipping containers of servers are then currently used as a building block for large scale datacenters. Containers are first very energy-efficient while they allow to expand the hosting capacity of a datacenter easily by acquiring and plugin new containers when needed.

Hosting platforms: clouds (amazon [2], rackspace [37], azure [5]), testbeds such as Emulab [28] or Grid'5000 [10].

2.1.2 Virtualized applications

User want to have its application running into the platform

Either instantiate an existing template that is already customized for the hosting platform. As an example Amazon EC2 has a list of pre-made instances

otherwise, create from scratch. Justify replication: elastic computing, absord load

Figure 2.2 illustrates a typical 3-tier HA Web application. The first tier is composed of 3 VMs, each running one Apache HTTP server [3]. The second tier is made up with 2 VMs, each running one Apache Tomcat instance [4]. The third tier is made up with 2 VMs, each running one instance of the MySQL database [33]. Each instance of Apache HTTP and Apache Tomcat includes a load balancer to spread out the requests associated to servlets or the database respectively.

The Apache services in tier 1 and the Tomcat services in tier 2 are stateless: all the handled requests are independent transactions and no synchronization of their state is required. On the other hand, tier 3 runs a replicated MySQL database, which is stateful: transactions that modify the data must be propagated from one VM hosting a replica of tier 3 to the others, to maintain a globally consistent state. In this setting, the application administrator has three expectations. First, the VMs must be running on servers having enough resources to make the services work at peak level. Second, the VM replicas must be running on distinct servers to provide the awaited fault tolerance to hardware failure. Finally, the VMs in the Tiers 3 must be running on servers connected by a low latency medium that reduce the overhead of the databases synchronization

protocol.

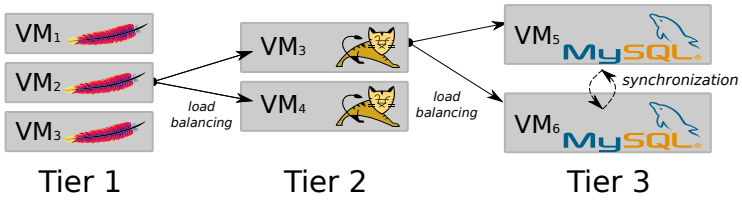


Figure 2.2: Sample architecture for a Highly-Available web service.

2.2 Hosting platform management

2.2.1 VM management

Through its lifetime on the platform, the state of a virtual machine is subject to change during a reconfiguration process. Figure 2.3 depicts as a finite state automaton the six possible states for a virtual machines and the action to apply to perform the transition between one state to another.

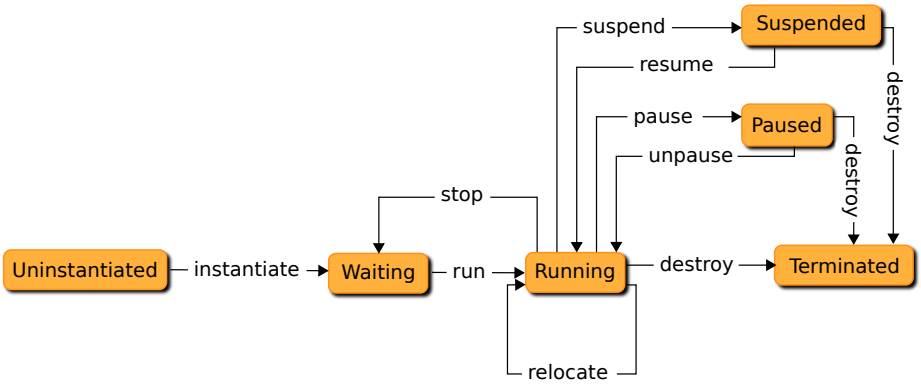


Figure 2.3: Lifecycle of a virtual machine

Uninstantiated

The `Uninstantiated` state is the initial state for any virtual machine. At this stage, the virtual machine is not prepared to be deployed on the servers. The action *instantiate* leads to prepare the virtual machine for the hosting platforms. This may imply several platform dependent operations such as the creation of configuration files to describe the virtual machine, but also the preparation of its disk image from a template or from scratch. At the end of the *instantiate* action, the virtual machine lies in the `Waiting` state.

Waiting

The `Waiting` state denotes a virtual machine that has been configured to be deployed on a server. The action *run* will then deploy the virtual machine on

a given server. The virtual machine will be booted and will start to consume resources. The action is not necessarily synchronous and may immediately end once the booting process of the virtual machine is engaged despite it is not available to its owner. At the end of the action, the virtual machine lies in the `Running` state.

Running

The `Running` state denotes a virtual machine that is running on a server. Several actions are then possible to manipulate the state or the location of the virtual machine.

The *relocate* action is responsible of relocating of the virtual machine to another server. Such an action can be achieved in different manners, depending on the hypervisors capabilities, the virtual machine specification and the software it embeds. Live migration [13, 34] is the most known technique to relocate a running virtual machine. It consists in moving the virtual machine to a given server with a negligible downtime. In practice, the virtual machine configuration, context and memory are copied in background to an inactive virtual machine on the destination server. Once the state of the running virtual machine is mostly coherent with the state of the inactive virtual machine on the destination server, the running virtual machine is suspended to finalize the transfer and the virtual machine on the destination server is activated. The original virtual machine is finally destroyed. *Cold* migration is another solution where the virtual machine on the source server is set inactive before performing the transfer. Finally, if the virtual machine hosts a replicated service, it is also possible to simply instantiate a new virtual machine on the destination server then stop the original virtual machine once the new one is booted. Such an operation is however more intrusive as it implies to know the embedded softwares.

The *pause* action blocks the virtual CPUs allocated to the virtual machine. At the end of this action, the virtual machine lies in the `Paused` state.

The *suspend* action suspends the virtual machine on a disk. The virtual CPUs allocated to the virtual machine are paused and its memory is copied on the disk. At the end of this action, the virtual machine lies in the `Suspended` state.

The *stop* action shutdown a virtual machine gracefully in a similar way a computer is turned off. At the end of the action, the virtual machine is back into the `Waiting` state and is supposed to be re-used.

Paused

The `Paused` state indicates the virtual machine is suspended into the server memory. The virtual machine memory is still lying on its hosting server memory but its virtual CPUs have been blocked. The virtual machine is then not available to its administrator. The *unpause* action moves the virtual machine back into the `Running` state by unblocking its virtual CPUs.

Suspended

The `Suspended` state indicates the virtual machine is suspended into a disk. A consistent image of the VM memory and state has been stored on a disk. The

virtual machine no longer consumes any physical resources except disk storage. The *resume* action moves the virtual machine back into the `Running` state by restoring its memory and its state from the disk image.

Terminated

The *destroy* action can be used from the states `Running`, `Suspended`, or `Paused` to set a virtual machine in the `Terminated` state. The action brutally stops the virtual machine. This corresponds to a hard shutdown on a computer when it is powered off instantly. As the virtual machine has been stopped brutally, its disk image may be corrupted. It is then not sure the virtual machine can be restarted. We then consider the `Terminated` state as a terminal state.

2.2.2 Server management

Similar to a virtual machine, a hosting server also has its state updated through actions during its lifecycle. Figure 2.4 depicts as a finite state automaton, the two possible states for a server and the action to apply to perform the transition between one state to another.

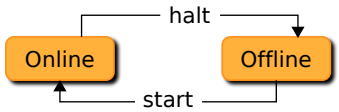


Figure 2.4: Lifecycle of a server

Online

The `Online` state indicates the server is available to the hosting platform and may host virtual machines. The *halt* action is responsible of setting the server in the `Offline` state.

Offline

The `Offline` state indicates the server is not directly available to the hosting platforms. It can be made available using the action *start*. The `Offline` state can be provided using different approaches. The traditional approach is to turn the server off. The action *start* will then boot it. It may also be convenient to suspend the server into disk or into RAM. Such a physical state has the same properties with regards to the external users as it is not directly available to them. However, restoring a server from a suspend-to-RAM or suspend-to-disk state may provide a faster response time with regards to a traditional unpowered state.

2.3 Reconfiguration

Occurs when actions have to be executed. Typically motivated by the arrival, departure of VMs, server maintenance or SLA violation fix or overal usage improvement.

A reconfiguration occurs when the current platform configuration does not meet the tenants or the platform administrator expectations. During a reconfiguration, several management operations are performed to put back the hosting platform into a viable configuration. These operations typically manipulate the virtual machines' state or placement, their resource allocation, or the servers' state.

In this chapter, we describe the principles of a reconfiguration process. In a first Section, we describe the management actions that can be performed during a reconfiguration and their impact on the servers and the virtual machines. In a second Section, we discuss about the necessity of scheduling the action execution to ensure the reliability of the reconfiguration process. Finally, we present a sample reconfiguration.

The action to execute during a reconfiguration process manipulate the virtual machines and the servers' states and the allocation of the servers resources.

First, any action on a virtual machine is exclusive as it impacts either it state or its location. In this setting, to operate on a virtual machine, the virtual machine must be in a state and no actions must already be pending. In addition, performing an action on a virtual machine alter the resource offered by its hosting server in two ways. In practice, an action may liberate or consume some resources on the hosting server. Table 2.1 details this impact per action.

Action	Liberating	Consuming
instantiate		
run		✓
relocate	✓	✓
destroy	✓	
suspend	✓	
resume		✓
pause	✓	
unpause		✓

Table 2.1: Impact on actions related to a virtual machine on its hosting server

The *run* and the *resume* actions are similar in terms of resource occupation. Indeed, in both situations, the manipulated virtual machine was not consuming any resources and once the action started, the virtual machine starts consuming resources on its hosting server. The *suspend*, the *stop*, and the *destroy* actions are also similar. Indeed, the manipulated virtual machine was running on the server. Once the action executed, the virtual machine is no longer running and all the allocated resources are freed. The *pause* action liberates only a part of the resources allocated to the manipulated virtual machine. In practice, every resources except the memory are liberated. The *unease* action makes then the virtual machine consumes all other resources again. Finally, the *relocate* action is both an action that liberate and consume resources. Indeed, once the action executed, the virtual machine is running on another server. In this situation, the resources that was consumed on the origin server were liberated while the virtual machine consumes now resources on its new host.

Actions related on the management of a server also impacts the resources available to the virtual machine. Table 2.2 details this impact for every action. The *start* actions makes the server online and potentially available to host vir-

tual machine. This action liberates then the resources that was busy due to the non-availability of the server. On the opposite, the action *halt* makes the server offline so, consumes every resources on the server.

Action	Liberating	Consuming
start	✓	
halt		✓

Table 2.2: Impact on actions related to a virtual machine on its hosting server

Consuming actions have then preconditions to be executed. These preconditions consists in having enough free resource on the target server to perform the action. Typically, enough memory, CPU, disk space, *etc.*. Aside, actions on virtual machines and servers are closely related as action related on a virtual machine hosted on a server requires to have the involved servers online. While turning off a server requires to have this server not hosting any virtual machine.

As a result, actions may have to be planned depending on their resource profile. Scheduling the So, in addition to the state of the virtual machines, the servers, the placement of the virtual machines on the servers, and the resource allocation, the temporality of the actions must be considered to ensure the reliability of the reconfiguration process.

2.4 A sample reconfiguration

Let's consider a hosting platform of 4 servers (N1 to N4) managing a total of 6 virtual machines (VM1 to VM6). Figure 2.5 depicts an initial and a destination configuration to reach. Initially, virtual machines VM1 and VM2 are in the `running` state on the server N1. The virtual machines VM3, VM5, and VM6 are in the `Running` station servers N2, N3, and N4, respectively. Finally, the virtual machine VM4 is in the `Waiting` state. In we consider that 1) every virtual machine must have an access to enough uCPU and memory capacity, 2) VM4 must be in the `Running` state, and 3) N3 must be in the `Offline` state, then this configuration is not viable.

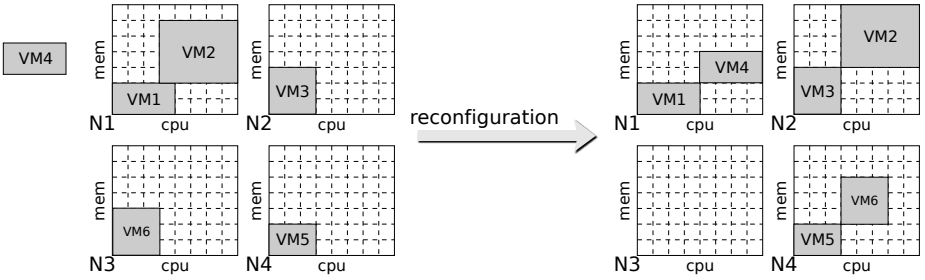


Figure 2.5: A reconfiguration between a non-viable configuration and a destination configuration

The destination configuration pictured on the right satisfies every prerequisite. One possible reconfiguration process to achieve the transition between the source and the destination configuration consists to execute 4 actions. N3 has

to be halted, VM4 must run on N1, VM2 and VM6 must be relocated to N2 and N4 respectively. However, it is not possible to execute these actions in any order. As an example, it is not possible to execute the *halt* action on N3 until VM6 is away or to run VM4 on N1 before relocating VM4 to N2 as their is no enough free resources at this point. Figure 2.6. depicts a possible event-oriented reconfiguration plan to ensure the termination of every actions. At startup, it is possible to relocate VM2 and VM6 in parallel. Once VM6 is running on N4, then it is possible to halt N3. Finally, once VM2 is running on N2, then VM4 can be deployed on N1.

Ø	→ relocate(VM2) & relocate(VM6)
!relocate(VM6)	→ halt(N3)
!relocate(VM2)	→ run(VM4)

Figure 2.6: Event-based reconfiguration plan that ensure the termination of the reconfiguration process depicted in Figure 2.5.

Another solution to express sequences of actions in a reconfiguration plan consists in providing a theoretical duration for each each then computing a actions schedule. Table 2.3 shows an estimated duration for each action composing the reconfiguration process in Figure 2.5. Table 2.4 depicts then a timer-based reconfiguration plan. **TODO: From a theoretical timer-based rp to a reliable practical one.**

relocate(VM2)	0'02
relocate(VM6)	0'06
run(VM4)	0'06
halt(N3)	0'06

Table 2.3: Estimated actions duration

Start	End	Action
0'00	0'02	relocate(VM2)
0'00	0'05	relocate(VM6)
0'02	0'08	run(VM4)
0'05	0'11	halt(N3)

Table 2.4: Timer-based reconfiguration plan that ensure the termination of the reconfiguration process depicted in Figure 2.5.

2.5 Conclusion

Reconfiguration process, act on the state of the virtual machine, its resource allocation, and its placement. Also acts on the servers state.

Several actions, to be feasible, have preconditions related to the state of the manipulated and the involved elements but also preconditions related to the available resources on the server.

The model describing a customizable reconfiguration problem must then provides these bases to be able to be customized enough. In the next chapter, we then present a mathematical model to specify a reconfiguration process.

Principles

Next: A formal model to represent a reconfiguration late amount of constraints, leaded by technology innovation, application type, platform capabilities, trends

Chapter 3

Constraints

This catalog aggregates constraints related to the management of VMs and servers on hosting platforms. Previous chapters present key elements that may be impacted by a reconfiguration : VM state and placement, server state, resource allocation. The present chapter enumerates numerous constraints that are of a practical interest for applications and datacenter administrators to express dependability or management requirements such as performance, availability or security. Such constraints may then be implemented in different VM managers to provide an initial deployment or a reconfiguration process that is consistent with regards to the expressed constraints.

Each section of this chapter details a specific constraint. First, a definition is provided to specify the guarantees provided by the constraint. A classification allows to categorize the constraint depending on the element it manipulates, the primary concern it addresses or the typical users. Several use cases illustrate then the practical interest of the constraint and examples depicts its impact on a reconfiguration process. Finally, references to closely related constraints are provided.

Several notations are used within this chapter to write the constraints signature or sample configuration. Chapter 4 details these notations.

3.1 Root

3.1.1 Definition

Signature

`root(s : set<VM>)`

- `s` : an non-empty set of VMs for a meaningful constraint. VMs not in the `Running` state are ignored.

The `root` constraint forces each running VM in `s` to not move from its current location. This constraint only restricts the placement of running VMs with regards to a previous placement. As a result, it is not possible to state for the satisfaction of one `root` constraint before a reconfiguration occurred.

Classification

- **Primary users** : application administrator,datacenter administrator
- **Manipulated elements** : VM placement
- **Concerns** : VM-to-server placement,Performance,Resource management

Usage

The `root` constraint is mostly used to disallow the relocation of VMs when it is not possible or tolerated. Typically, a running VM may be attached to a peculiar device such as a filesystem or a PCI device on its host. In this setting, the relocation of the VM may not maintain this link and should not be performed for reliability reasons. An application administrator may then use a `root` constraint on this constraint to prevent from relocation. In addition, an application administrator may disallow the relocation of his VMs to prevent from the temporary performance loss that occur during this action.

Another possible usage is to disallow VMs relocation when the infrastructure or the underlying hypervisor does not support it. In this setting, the datacenter administrator may use a `root` constraint on all the VMs to disallow their relocation.

Example

Figure 3.1 depicts a sample reconfiguration between a source and a destination configuration. In this example, the following `root` constraints were considered:

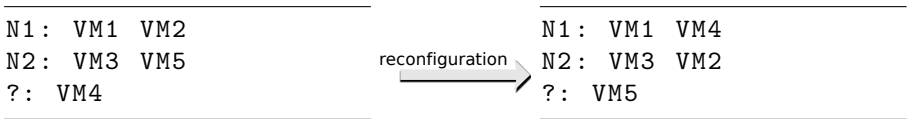


Figure 3.1: A reconfiguration motivated by `root` constraints.

- `root({VM1,VM3})`. This constraint is satisfied as none of the VMs were relocated during the reconfiguration
- `root({VM4, VM5})`. This constraint is satisfied as `VM4` is not running in the initial configuration while `VM5` is no longer running in the destination configuration. The constraint ignores then these VMs.
- `root({VM2})`. This constraint is not satisfied as `VM2` has been relocated from `N1` to `N2` during the reconfiguration process.

3.1.2 See also

Related Constraints

- `fence`: The `root` constraint can be emulated using a `fence` constraint when its user knows the current host of the specified VMs. For each VM, the list of servers given in the `fence` constraint is a singleton only composed of the current hosting server.

3.2 Ban

3.2.1 Definition

Signature

`ban(vs : set<VM>, ns : set<server>)`

- `vs` : an non-empty set of VMs for a meaningful constraint. VMs not in the `Running` state are ignored.
- `ns` : an non-empty set of servers for a meaningful constraint. Servers not in the `Online` state are ignored.

The `ban` constraint disallows each running VM in `vs` to be hosted on any of the online servers in `ns`.

Classification

- **Primary users** : datacenter administrator
- **Manipulated elements** : VM placement
- **Concerns** : Maintenance,Partitioning,VM-to-server placement

Usage

This constraint may be use by the datacenter administrator to prepare servers for a software maintenance. In this situation, the administrator must first be sure that the servers do not host any running VMs to prevent a misconfiguration from altering them. Every running VM on these servers must then be relocated elsewhere while the other VMs should not be relocated on the servers to put into maintenance. A datacenter administrator may rely on a `ban` constraint to achieve that purpose. In this setting, all the VMs in the datacenter are given in parameters in addition to the servers to put into maintenance. At the end of the reconfiguration, no VMs will be running on the servers. Once the maintenance operation is terminated, the constraint may be removed to put the servers back into the hosting pool.

For partitioning reasons, some VMs may be disallowed to be running on some servers. As an example, servers may be dedicated to run service VMs. In this setting, the client VMs must not be allowed to run on the servers dedicated to run service VMs. `Ban` constraints may then be used by the datacenter administrator for that purpose.

Example

Figure 3.2 depicts a sample reconfiguration between a source and a destination configuration. In this example, the following `ban` constraints were considered:

- `ban({VM1, VM2, VM4}, {N2})`. This constraint was not satisfied in the source configuration as `VM4` was running on `N2`. The reconfiguration fixed this violation by relocating `VM4` to `N3`.

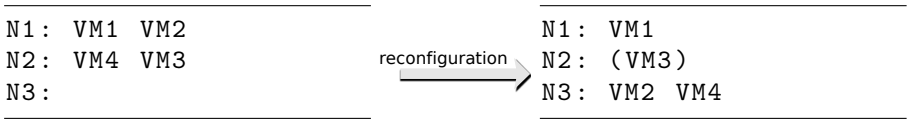


Figure 3.2: A reconfiguration motivated by `ban` constraints.

- `ban({VM3}, {N2})`. This constraint was not satisfied in the source configuration. However it is satisfied in the destination configuration as `VM3` is no longer running.
- `ban({VM1, VM2}, {N2})`. This constraint was satisfied in the source configuration. It is still satisfied in the destination configuration as the relocation of `VM2` is compatible with this constraint.

3.2.2 See also

Related Constraints

- `fence`: the opposite constraint of `ban`. One `ban` constraint can be emulated using a `fence` constraint by specifying to the `fence` constraint, the absolute complement of the set of servers specified in the `ban` constraint.
- `quarantine`. The `quarantine` constraint may also be used to prepare a software maintenance on servers when relocation is not possible. In this setting, the given servers will be ready for the maintenance once their VMs are terminated.

Reformulation(s)

- Using `fence`: `ban(vs1, ns1)` $\equiv$ `fence(vs1,`$\overline{\texttt{ns1}}$`)`
- Using `among`: `ban(vs1, ns1)` $\equiv$ `among(vs1,{`$\overline{\texttt{ns1}}$`})`

Specialization(s)

- To `fence`: `ban(vs1,`$\overline{\texttt{ns1}}$`)` $\equiv$ `fence(vs1, ns1)`

3.3 Fence

3.3.1 Definition

Signature

```
fence(s1 :  set<VM>, s2 :  set<server>)
```

- `s1` : an non-empty set of VMs for a meaningful constraint. VMs not in the `Running` state are ignored.
- `s2` : an non-empty set of servers or the constraint is sure of not being satisfiable. Servers not in the `Online` state are ignored.

The `fence` constraint forces each running VM in `s1` to be running on one of the online servers in `s2`.

Classification

- **Primary users** : datacenter administrator
- **Manipulated elements** : VM placement
- **Concerns** : Partitioning,VM-to-server placement

Usage

A `fence` constraint deserves partitioning purposes. First, it may be used by a datacenter administrator to part the infrastructure. Such a situation is typically motivated for security or administrative purposes. As an example, a datacenter may be the property of multiple organizations that have aggregated their servers. However, administrative policies may disallow to have specific VMs of one organization running on servers belonging to another organization. In this setting, `fence` constraints may be used to specify the list of allowed servers for these VMs.

Another possible usage of `fence` constraints consist in splitting the VMs when the infrastructure is designed using an aggregation of independent partitions of servers. As an example, a partition may consist of a standalone shipping container of servers or all the servers connected to a same storage space when VMs disk images are only available to these servers. In this setting, `fence` constraint allow the datacenter administrator to restrict the placement of the VMs to their assigned partition.

Finally, a `fence` constraint may be used as a backend for a resource matchmaking system [38] for non-cumulative resources. As an example, a VM may require a particular hypervisor, processor or GPU. `Fence` constraints may then be used to force the VMs to be running on compatible environments. The list of the compatible servers must be established before using the constraint. It has to be noticed that this use case is not suitable when the matchmaking is focusing shareable but finite resources such as memory or computing power.

Example

Figure 3.3 depicts a sample reconfiguration between a source and a destination configuration. In this example, the following `fence` constraints were considered.

- `fence({VM1,VM2}, {N1, N2})`. This constraint was already satisfied in the source configuration as both VMs were running on `N1`. The constraint is still satisfied in the destination configuration despite `VM2` has been relocated to `N2` as this action is allowed by the constraint.
- `fence({VM2, VM3}, {N2})`. This constraint was not satisfied in the source configuration as `VM2` was not running on `N2`. Its relocation to `N2` during the reconfiguration fixed this violation. During the reconfiguration process, `VM3` has been stopped and is now in the `Waiting` state. The VM was then ignored by the constraint.

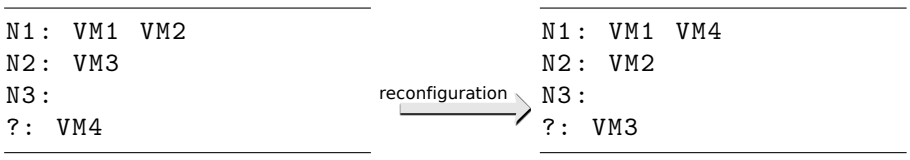


Figure 3.3: A reconfiguration motivated by `fence` constraints.

3.3.2 See also

Related Constraints

- `ban`: the opposite constraint of `fence`. One `fence` constraint can be emulated using a `ban` constraint by specifying to the `ban` constraint the absolute complement of the set of servers specified in the `fence` constraint.
- `quarantine`: This constraint encapsulates the `fence` constraint but also disallows the VMs outside the fence to be relocated to servers inside the fence.
- `among`: The `among` constraint encapsulates the `fence` constraint. The given VMs will necessarily be running on a single set of servers but multiple candidate set of servers may be passed as an argument to let the `among` constraint state about the set of servers to choose. A `fence` constraint can be emulated using one `among` constraint when only of set of servers is specified.

Reformulation(s)

- Using `ban`: `fence(vs1, ns1)` $\equiv$ `ban(vs1,`$\overline{\texttt{ns1}}$`)`

Specialization(s)

- To `ban`: `fence(vs1,`$\overline{\texttt{ns1}}$`)` $\equiv$ `ban(vs1, ns1)`

3.4 Quarantine

3.4.1 Definition

Signature

`quarantine(s : set<server>)`

- `s` : an non-empty set of servers for a meaningful constraint. Servers not in the `Online` state are ignored.

The `quarantine` constraint disallows any VM running on servers other than those in `s` to be relocated into a server in `s`. In addition, every VM running on a server in `s` cannot be relocated to another server. This constraint only restricts the placement of running VMs with regards to a previous placement. As a result, it is not possible to state for the satisfaction of one `quarantine` constraint before a reconfiguration occurred.

Classification

- **Primary users** : datacenter administrator
- **Manipulated elements** : VM placement
- **Concerns** : VM-to-server placement,Partitioning,Maintenance

Usage

A `quarantine` constraint may be use by the datacenter administrator for isolation purpose. When a server appears to be compromised, a first step to avoid to propagate the situation is to isolate it from the datacenter (put it into *quarantine*). Using one `quarantine` constraint, the datacenter administrator is ensured that no running VMs may enter or leave the quarantine zone.

This constraint may also be used to prepare the servers for a software maintenance operation when it is not possible to relocate the VMs it runs. To prepare a software maintenance, the datacenter administrator must be sure the server does not host any running VMs to prevent a misconfiguration from altering them. As their relocation is not possible in this setting, the only solution to tend to have a server ready for the maintenance is to wait for its VMs termination while disallowing other VMs to be running on the server. The datacenter administrator may then use a `quarantine` constraint for that purpose.

Example

Figure 3.4 depicts a sample reconfiguration between a source and a destination configuration. In this example, the following `quarantine` constraints were considered:

- `quarantine({N2})`. This constraint is satisfied as `VM3` was not relocated while no VMs were moved or launched to `N2`.
- `quarantine({N1,N4})`. This constraint is satisfied as `VM2` is terminated
- `quarantine({N3})`. This constraint is not satisfied as `VM5` was launched on `N3` which is in the quarantine zone.

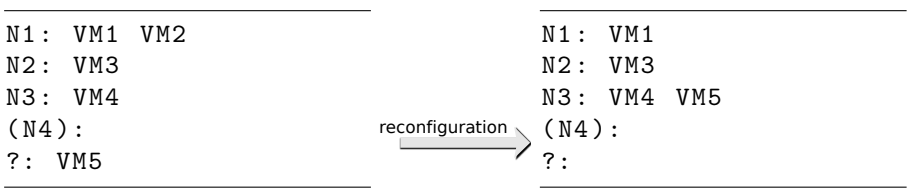


Figure 3.4: A reconfiguration motivated by `quarantine` constraints.

3.4.2 See also

Related Constraints

- `ban`: When it is possible to relocate the VMs, a `ban` constraint may be use to prepare servers for a software maintenance operation. In this setting, the server will be ready for the maintenance sooner as the VMs will be immediately relocated while the datacenter administrator has to wait for their termination when a `quarantine` constraint is used.
- `fence` + `root`. This composition of constraints emulates a `lonely` constraint. One `fence` constraint will disallow the VMs outside the quarantine zone to enter it while one `root` constraint will prevent the relocation of the VMs running into the quarantine zone.
- `lonely`. A partitioning constraint to isolate VMs rather than servers.

Reformulation(s)

- Using `fence`+`root`: `quarantine(s)` $\equiv$ `root(`$\uparrow$`s), fence(`$\overline{\uparrow s}$`, `$\overline{s}$`)`

3.5 Among

3.5.1 Definition

Signature

`among(vs : set<VM>, ns : set<set<server»)`

- `vs` : an non-empty set of VMs for a meaningful constraint. VMs not in the `Running` state are ignored.
- `ns` : an non-empty set of set of servers or the constraint is sure of not being satisfiable. Sets composing `ns` must be disjoint. Servers not in the `Online` state are ignored.

The `among` constraint forces each running VM in `vs` to be hosted on one of the set of servers in `ns`.

Classification

- **Primary users** : application administrator,datacenter administrator
- **Manipulated elements** : VM placement
- **Concerns** : Partitioning,VM-to-VM placement,VM-to-server placement,Performance

Usage

The `among` constraint may be used by a datacenter administrator or an application administrator to group closely related VMs with regards to specific criteria. A datacenter network is usually designed as a fat-tree [30, 1] that provides a non-uniform network latency and bandwidth between the servers. An application administrator having strongly communicating VMs may then require to have its VMs hosted on servers connected to a same edge switch to improve their communication. This can be achieved using one `among` constraint where each set of servers denotes the servers connected to a same edge switch.

A datacenter administrator may also rely on `among` constraints to part the datacenter and ease its management. First, the administrator reflects, using disjoint set of servers, the current physical partitioning (different geographical sites, administrative zones, shipping containers of servers [23], ...) of the datacenter. `Among` constraints are then use to force groups of VMs to stay inside a single physical partition.

Example

Figure 3.5 depicts a sample reconfiguration between a source and a destination configuration. In this example, the following `among` constraints were considered:

- `among({VM1, VM2, VM3}, {{N1, N2},{N3, N4}})`. This constraint was not satisfied in the source configuration as `VM1` and `VM2` were running on the partition `{N1, N2}` while `VM3` was running on the partition `{N3, N4}`. The reconfiguration fixed this violation by relocating `VM3` to `N2`. `VM2` was relocated to `N2` but this action does not contradict the constraint.

- `among({VM4, VM5}, {{N1},{N2, N3, N4}})`. This constraint was not satisfied in the source configuration as `VM4` and `VM5` were running on distinct partitions. This violation was fixed by suspending `VM5`.
- `among({VM6, VM7}, {{N1,N2},{N3, N4}})`. This constraint was already satisfied in the source configuration as both VMs were running on the partition `{N1,N2}`. The constraint is still satisfied in the destination configuration as both VMs have been relocated to the partition `{N3,N4}`.

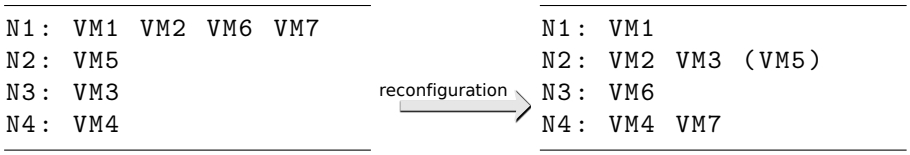


Figure 3.5: A reconfiguration motivated by `among` constraints.

3.5.2 See also

Related Constraints

- `fence`: The `fence` constraint is a specialization of the `among` constraint: only one set of servers is possible to restrict the VMs placement. A `among` constraint is then equivalent to a `fence` constraint when there is only one possible set of servers to host the VMs. This occurs when only one set of servers is specified, when other sets are offline, or when one of the given VMs is ensured to be hosted on a known server as the other VMs will necessarily be hosted on the set of servers the known server belong to.

Reformulation(s)

- Using `splitAmong`: `among(vs1, ns1)` $\equiv$ `splitAmong({vs1},ns1)`

Specialization(s)

- To `gather`: `among(s,`$\mathcal{N}$`/|`$\mathcal{N}$`|)` $\equiv$ `gather(s)`
- To `ban`: `among(vs1,{`$\overline{\texttt{ns1}}$`})` $\equiv$ `ban(vs1, ns1)`

3.6 Lonely

3.6.1 Definition

Signature

```
lonely(s :  set<VM>)
```

- `s` : an non-empty set of VMs for a meaningful constraint. VMs not in the `Running` state are ignored.

The `lonely` constraint forces all the running VMs in `s` to be running on dedicated servers. Each of the used servers can still host multiple VMs but they have to be in `s`.

Classification

- **Primary users** : application administrator
- **Manipulated elements** : VM placement
- **Concerns** : VM-to-VM placement,Partitioning

Usage

The `lonely` constraint deserves isolation purposes. Hypervisors are supposed to provide a strong isolation between the VMs. However various attacks, such as those based on VM escaping [50], allow to break this isolation to provide from a malicious VM, a non-legitimate access to the hypervisor or the other VMs. An application administrator may then want to have to prevent this situation by requiring to have its VMs hosted on servers that do not host unknown, potentially malicious VMs. A `lonely` constraint can then be used to indicate the VMs that must be running on dedicated servers.

Example

Figure 3.6 depicts a sample reconfiguration between a source and a destination configuration. In this example, the following `lonely` constraints were considered:

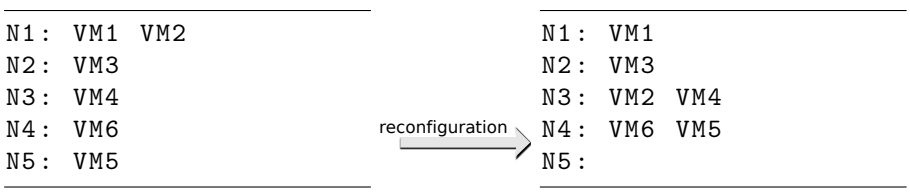


Figure 3.6: A reconfiguration motivated by `lonely` constraints.

- `lonely({VM1,VM3})`. This constraint was not satisfied in the source configuration as `VM2` was colocated with `VM1` despite `VM2` does not belong to the VMs given as parameter. This violation was fixed by relocating `VM2` to `N3`.
- `lonely({VM2, VM4})`. This constraint was not satisfied in the source configuration. It was fixed by colocating `VM2` with `VM4` on `N3`.
- `lonely({VM5, VM6})`. This constraint was satisfied in the source configuration. The constraint is still satisfied in the destination configuration despite the relocation of `VM5` to `N4` which is allowed by the constraint.

3.6.2 See also

Reformulation(s)

- Using `split`: `lonely(vs1)` $\equiv$ `split({vs1,`$\overline{\texttt{vs1}}$`})`

3.7 Split

3.7.1 Definition

Signature

`split(vs: set<set<VM»)`

- `vs` : a non-empty set of set of VMs for a meaningful constraint. VMs not in the `Running` state are ignored. Sets inside `vs` must be disjoint.
- `s2` : an non-empty set of VMs for a meaningful constraint, that is distinct from `s1`. VMs not in the `Running` state are ignored.

The `split` constraint forces the given sets of VMs in `vs` to not share hosting servers. Each of the used servers can still host multiple VMs but they have to be in the same set.

Classification

- **Primary users** : application administrator
- **Manipulated elements** : VM placement
- **Concerns** : VM-to-VM placement,Partitioning,Fault tolerance

Usage

The `split` constraint deserves isolation requirements. Hypervisors are supposed to provide a strong isolation between the VMs. However, various attacks such as those based on VM escaping [50], allow to break this isolation to provide from a malicious VM, a non-legitimate access to the hypervisor or the other VMs. An application administrator may then want to have its VMs hosted on servers that do not host potentially malicious VMs. A `split` constraint may then be used to indicate the VMs that must be running on servers other than the supposed malicious ones.

The `split` constraint deserves also fault tolerance requirements. For high-availability purposes, replicated applications are supposed to be running on distinct servers. In this setting, an application administrator may use one `split` constraint to ensure all the VMs of the application do not share any server with the replicated VMs.

Example

Figure 3.7 depicts a sample reconfiguration between a source and a destination configuration. In this example, the following `split` constraints were considered:

- `split({{VM1,VM3},{VM2, VM4}})`. This constraint was not satisfied in the source configuration as `VM2` and `VM1` were colocated despite they belong to different sets. This violation was fixed by relocating `VM2` to `N3`.

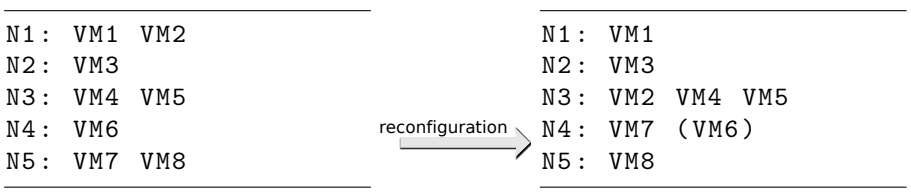


Figure 3.7: A reconfiguration motivated by `split` constraints.

- `split({{VM5,VM6},{VM7, VM8}})`. This constraint was satisfied in the source configuration as no set of VMs share hosts. The constraint is still satisfied in the destination configuration: despite `VM7` and `VM6` are on the same server, `VM6` is not in the `Running` state, so it is ignored by the constraint.

3.7.2 See also

Related Constraints

- `spread`, `lazySpread`: These constraints disallow the colocation between VMs rather than groups of VMs. A `split` constraint is equivalent to a `lazySpread` constraint when the `split` constraint is made up with two sets of one VM each.
- `splitAmong`: This constraint forces several set of VMs to be hosted on distinct groups of servers among those explicitly allowed.
- `lonely`. The `lonely` constraint is a specialization of the `split` constraint that isolate a set of VMs from all the others. It can then be emulated using a `split` constraint when the second set of VMs is the absolution complement of the first one.

Specialization(s)

- To `lonely`: `split({vs1,`$\overline{\texttt{vs1}}$`})` $\equiv$ `lonely(vs1)`
- To `lazySpread`: `split(s/|s|)` $\equiv$ `lazySpread(s)`

3.8 SplitAmong

3.8.1 Definition

Signature

`splitAmong(vs : set<set<VM», ns : set<set<server»)`

- `vs` : a non-empty set of set of VMs for a meaningful constraint. VMs not in the `Running` state are ignored. Sets inside `vs` must be disjoint.
- `ns` : a set of set of servers that is composed of more sets than `vs` or the constraint is sure of not being satisfiable. Sets composing `ns` must be disjoint. Servers not in the `Online` state are ignored.

The `splitAmong` constraint forces the sets of VMs inside `vs` to be hosted on distinct set of servers in `ns`. VMs inside a same set may still be collocated.

Classification

- **Primary users** : application administrator
- **Manipulated elements** : VM placement
- **Concerns** : VM-to-VM placement,Partitioning,Fault tolerance

Usage

The `splitAmong` constraint deserves isolation requirements. One solution to ensure disaster recovery for an application is to replicate it. When the master application fail, the replica is activated transparently to neglect the failure effect. In practice, the replication is a mechanism provided at the hypervisor level [14, 41]. The replicas are then placed to a distant server to make the application survive to a datacenter failure. One application administrator may obtain this fault tolerance using one `splitAmong` constraint. The sets of VMs given as parameters are the master then the slave VMs while the set of servers are the servers composing each datacenter.

Example

Figure 3.8 depicts a sample reconfiguration between a source and a destination configuration. In this example, the following `splitAmong` constraints were considered:

- `splitAmong({{VM1,VM3},{VM2,VM4}},{{N1,N2},{N3,N4}})`. This constraint was not satisfied in the source configuration as `VM2` and `VM1` were both running inside the set of servers `{N1,N2}` despite they belong to different sets of VMs. In addition, the set of VMs `{VM2,VM4}` was spread among the two set of servers while it should be running on only one. These violations were fixed by relocating `VM2` to `N4` to let the first set of VMs running on the first set of servers and the second set of VMs running on the second set of servers.

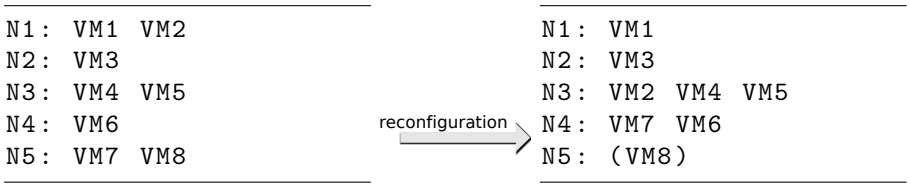


Figure 3.8: A reconfiguration motivated by `splitAmong` constraints.

- `splitAmong({{VM1,VM3},{VM5,VM6,VM7, VM8}},{{N1,N2},{N3,N4}})`. This constraint was not satisfied in the source configuration as `VM7` and `VM8` were running on `N5`, that does not belong to any of the allowed sets. This violation was fixed by relocating `VM7` to `N4` and by suspending `VM8` which is now ignored by the constraint.
- `splitAmong({{VM1,VM2,VM3},{VM7,VM8}},{{N1,N2,N3},{N4,N5}})`.This constraint was satisfied in the source configuration as the sets of VMs share do not share a group of servers. The constraint is still satisfied in the destination configuration despite the relocation of `VM2` and `VM7` to `N3` and `N4` respectively which let them running inside their dedicated group of servers.

3.8.2 See also

Related Constraints

- `split`: This constraint disallows two set of VMs to share servers.
- `spread`, `lazySpread`: These constraints disallow the colocation between VMs rather than groups of VMs.
- `fence`: `splitAmong` is equivalent to a `fence` constraint when only one set of VMs and one set of servers are given as arguments.

Specialization(s)

- To `lazySpread`: `splitAmong(s/|s|,`$\mathcal{N}$`/|`$\mathcal{N}$`|)` $\equiv$ `lazySpread(s1)`
- To `among`: `splitAmong({vs1},ns1)` $\equiv$ `among(vs1, ns1)`

3.9 Gather

3.9.1 Definition

Signature

`gather(s : set<VM>)`

- `s` : a set of at least 2 VMs for a meaningful constraint. VMs not in the `Running` state are ignored.

The `gather` constraint forces all the running VMs in the set `s` to be hosted on the same server.

Classification

- **Primary users** : application administrator
- **Manipulated elements** : VM placement
- **Concerns** : Performance,VM-to-VM placement

Usage

The `gather` constraint may first be used by an application administrator to force the colocation of strongly communicating VMs. Using one `gather` constraint, the VMs will be running on a same server and the virtual network between them will be embedded into the internal network provided by the hypervisor instead of the physical network, leading to better network performance.

The `gather` constraint may also be used by an application administrator to force the colocation of VMs that have to share a component such as a filesystem. Without any guarantee of colocation, it is a necessary to rely on a distributed filesystem or a file server which provide less performance than a raw and direct access to the data. Using the colocation guarantee provided by a `gather` constraint, the filesystem may be placed directly on the hosting server to achieve better performance. In this context, it is also often desirable to use one `root` constraint to prevent from VMs relocation.

It has however to be noticed that these usages ask explicitly for the VMs colocation. It is then not allowed to host the VMs on several servers when the colocation is not possible. The application administrator should then not over-estimate his needs to prevent him from having its VMs not running at all.

Example

Figure 3.9 depicts a sample reconfiguration between a source and a destination configuration. In this example, the following `gather` constraints were considered:

- `gather({VM1, VM4})`. This constraint was satisfied in the source configuration as only `VM1` was running, so considered in the constraint. The constraint is also satisfied in the destination configuration: during the reconfiguration, `VM4` was set in the `Running` state and deployed on `N1` according to the constraint requirement.

- `gather({VM2, VM5})`. This constraint was not satisfied in the source configuration as `VM4` and `VM5` were running on different servers. The reconfiguration fixed this violation by relocating `VM2` to `N3` which is also hosting `VM5`.
- `gather({VM3, VM6})`. This constraint was satisfied in the source configuration as both VMs were running on `N2`. Despite both VMs were relocated during the reconfiguration process, the constraint is still satisfied in the destination configuration as both VMs are running on `N4` and the end of reconfiguration.

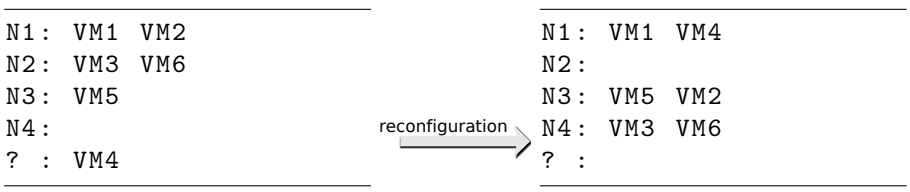


Figure 3.9: A reconfiguration motivated by `gather` constraints.

3.9.2 See also

Related Constraints

- `spread`, `lazySpread`: the opposite constraints of `gather` that force the VMs to be hosted on distinct servers.

Reformulation(s)

- Using `among`: `gather(s)` $\equiv$ `among(s,`$\mathcal{N}$`/|`$\mathcal{N}$`|)`

3.10 Spread

3.10.1 Definition

Signature

```
spread(s :  set<VM>)
```

- `s` : a set of at least 2 VMs for a meaningful constraint. VMs not in the `Running` state are ignored.

The `spread` constraint forces all the running VMs in `s` to be hosted on distinct servers at any time, even during the reconfiguration process.

Classification

- **Primary users** : application administrator
- **Manipulated elements** : VM placement,Actions schedule
- **Concerns** : Fault tolerance,VM-to-VM placement,Partitioning

Usage

The `spread` constraint may be used by an application administrator to provide to a replicated service, fault tolerance to hardware failures. By hosting each replicas on a distinct server, the service will be available while at least one server is still online. To achieve this purpose, one `spread` constraint can be used with the replicas provided as arguments.

Example

Figure 3.10 depicts a sample reconfiguration between a source and a destination configuration. In this example, the following `spread` constraints were considered:

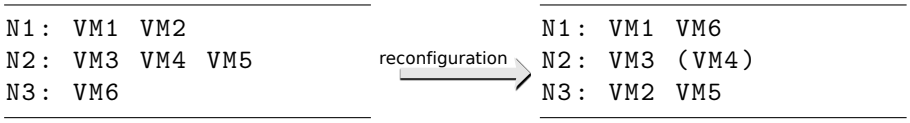


Figure 3.10: A reconfiguration motivated by `spread` constraints.

- `spread({VM1,VM2})`. This constraint was not satisfied in the source configuration as both VMs were colocated. The reconfiguration fixed this violation by relocating `VM2` to `N3`.
- `spread({VM3, VM4})`. This constraint was not satisfied in the source configuration. Putting `VM4` in the `Suspended` state fixed this violation without having to perform a relocation.

- `spread({VM5, VM6})`. This constraint was satisfied in the source configuration. However, let consider `VM5` must be running on `N3`, which was already hosting `VM6`. In this setting, `VM6` has been relocated to `N1` to disallow the colocation. Furthermore, to prevent from a temporary colocation on `N3`, it was a necessary to relocate `VM6` before `VM5`. Figure 3.11 depicts the associated event-based reconfiguration plan.

Ø	→ stop(VM4) & relocate(VM5)
!relocate(VM5)	→ relocate(VM6)
!relocate(VM6)	→ relocate(VM2)

Figure 3.11: Event-based reconfiguration plan associated to the reconfiguration depicted in Figure 3.10.

3.10.2 See also

Related Constraints

- `gather`: the opposite constraint of `spread`.
- `lazySpread`: a constraint similar to `spread` that does not guarantee the non-overlapping of the VMs during the reconfiguration process.
- `split`: a constraint to ensure two sets of VMs do not share servers.

3.11 LazySpread

3.11.1 Definition

Signature

`lazySpread(s : set<VM>)`

- `s` : a set of at least 2 VMs for a meaningful constraint. VMs not in the `Running` state are ignored.

The `lazySpread` constraint forces all the running VMs in `s` to be hosted on distinct servers at the end of a reconfiguration process.

Classification

- **Primary users** : application administrator
- **Manipulated elements** : VM placement
- **Concerns** : VM-to-VM placement,Fault tolerance,Partitioning

Usage

The `lazySpread` constraint may be used by an application administrator to provide to a replicated service, fault tolerance to hardware failures. By hosting each replicas on a distinct server, the service will be available while at least one server is still online. To achieve this purpose, one `lazySpread` constraint can be used with the replicas provided as arguments.

Example

Figure 3.12 depicts a sample reconfiguration between a source and a destination configuration. In this example, the following `lazySpread` constraints were considered:

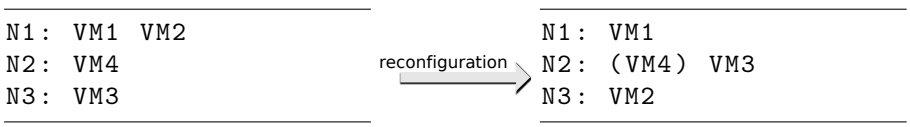


Figure 3.12: A reconfiguration motivated by `lazySpread` constraints.

- `lazySpread({VM1, VM3, VM4})`. This constraint was satisfied in the source configuration as each VM were running on distinct servers. The constraint is still satisfied in the destination configuration despite `VM3` and `VM4` are colocated as `VM4` has been suspended.
- `lazySpread({VM1,VM2})`. This constraint was not satisfied in the source configuration as both VMs where running on `N1`. The relocation of `VM2` to `N3` fixed this violation.

- `lazySpread({VM2, VM3})`. This constraint was satisfied in the source configuration as both VMs were running on distinct servers. The constraint is still satisfied in the destination configuration despite the relocation of `VM2` to `N3` that was running `VM3` initially as `VM3` has been relocated elsewhere to let both VMs be running on distinct servers at the end of the reconfiguration process.

3.11.2 See also

Related Constraints

- `gather`: the opposite constraint of `spread`
- `spread`: a constraint similar to `lazySpread` but which also guarantee that the VMs will never overlap on a same server, even during the reconfiguration process.

Reformulation(s)

- Using `mostlySpread`: `lazySpread(s)` $\equiv$ `mostlySpread(s,|s|)`
- Using `splitAmong`: `lazySpread(s1)` $\equiv$ `splitAmong(s/|s|,`$\mathcal{N}$`/|`$\mathcal{N}$`|)`
- Using `split`: `lazySpread(s)` $\equiv$ `split(s/|s|)`

3.12 MostlySpread

3.12.1 Definition

Signature

`mostlySpread(s : set<VM>, n : number)`

- `s` : a non-empty set of virtual machines for a meaningful constraint
- `n`: a positive number, inferior to the number of virtual machines in `s`

The `mostlySpread` constraint ensures the running virtual machines in `s` will be running on at least `n` distinct servers.

Classification

- **Primary users** : application administrator
- **Manipulated elements** : VM placement
- **Concerns** : VM-to-VM placement,Fault tolerance, Partitioning

Usage

The `mostlySpread` constraint may be used by an application administrator to provide to a replicated service, fault tolerance to hardware failures. By hosting replaces on distinct servers, the service will be available while at least one server is still online. When the number of replicas is important, it is however difficult to have a large amount of different servers. Furthermore, the chances of having all the hosting hervers but one failing simultaneously decrease when the number of replicas increases. It is then tolerable to use a number of servers that is smaller to the number of replicas. The application administrator can then use one `mostlySpread`ăconstraint to indicate the minimum number of distinct servers that must be used to host the replicas.

Example

Figure 3.13 depicts a sample reconfiguration between a source and a destination configuration. In this example, the following `mostlySpread` constraints were considered:

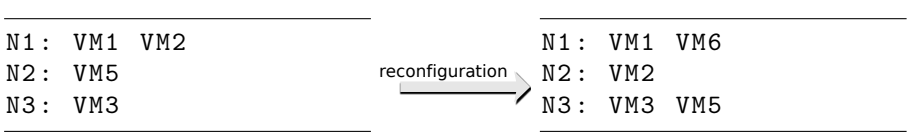


Figure 3.13: A reconfiguration motivated by `mostlySpread` constraints.

- `mostlySpread({VM1, VM2, VM5}, 2)`. This constraint was satisfied in the source configuration as all the VM were running on 2 distinct servers. The constraint is still satisfied in the destination configuration as all the VMs are running on 3 distinct servers.
- `mostlySpread({VM1,VM2}, 1)`. This constraint was not satisfied in the source configuration as the VMs where running on `N1`. The relocation of `VM2` to `N3` fixed this violation.

3.12.2 See also

Related Constraints

- `spread`: a constraint that guarantees the VMs will never overlap on a same server, even during the reconfiguration process.
- `lazySpread`: a constraint similar to `mostlySpread` but that guarantee every VMs are running on distinct servers.

Specialization(s)

- To `lazySpread`: `mostlySpread(s,|s|)` $\equiv$ `lazySpread(s)`

3.13 Preserve

3.13.1 Definition

Signature

```
preserve(s :  set<VM>, r:string, n :  number)
```

- `s`: a non-empty set of VMs for a meaningful constraint. VMs not in the `Running` state are ignored.
- `r` : a resource identifier such as `mem`, `ucpu`, `pcpu` to identify the physical memory, the computational capacity, the physical CPUs, respectively.
- `n`: a positive amount of resources

The `preserve` constraint ensures each running VM in `s` is hosted on a server having at minimum an amount of resource of type `r` equals to `n` dedicated to the VM.

Classification

- **Primary users** : application administrator
- **Manipulated elements** : Resource allocation,VM placement
- **Concerns** : VM-to-server placement,Resource management

Usage

The `preserve` constraint allows to control the resource allocated to the given VMs. This constraint may first be used inside a provisioning algorithm developed by an application administrator to indicate to the resource manager an ideal amount of resources to provide to the VMs to let them work at a peak level of performance.

Inside a non-conservative consolidated environment, a server may host several VMs that currently ask for a small amount of resources. In this situation, the server may become saturated if VMs ask suddenly for a little more resources. This situation is not idyllic as these micro variations may lead to numerous relocations. To prevent that situation and reduce the frequency of the reconfigurations, the datacenter administrator may use one `preserve` constraint to over-allocate a minimum amount of resources to the VMs that ask for a small amount of resources.

Example

Figure 3.14 depicts a sample reconfiguration between a source and a destination configuration where each server provides 8 unit of CPU and 7 unit of memory resources to VMs. Each VM is associated to a gray rectangle that denotes its resource requirement, expressed using `preserve` constraints. A rectangle overlapping another one on a dimension indicates the two associated VMs have to share resources. This reveals an overloaded server. Figure 3.15 depicts the associated event-based reconfiguration plan. The following `preserve` constraints were considered:

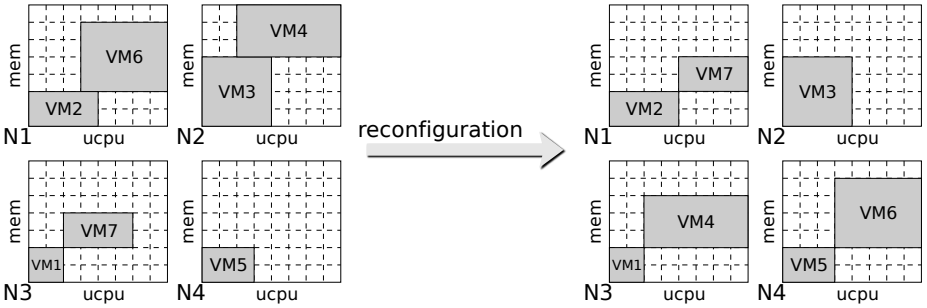


Figure 3.14: A reconfiguration motivated by `preserve` constraints.

Ø	→ relocate(VM6)
!relocate(VM6)	→ relocate(VM7)
!relocate(VM7)	→ relocate(VM4)

Figure 3.15: Event-based reconfiguration plan.

- `preserve({VM2, VM3},4,"ucpu")`. This constraint was not satisfied in the source configuration as the hosting servers of the given VMs do not provide enough resource to them: `N1` provides 8 unit of CPU resources but `VM6` requires 5 units and `VM2` requires 4 units . In addition, VMs on `N2` required 10 units of CPU resources. These violation were fixed by relocating `VM4` and `VM6` to `N3` and `N4`, respectively. However, `N4` does not initially provides enough resources to host simultaneously `VM4` and `VM7`. It has then be decided to relocate first `VM6` to `N4`Êto liberate enough resource on `N1` to host `VM7`. Once this relocation terminated, enough resources were available on `N3` to host `VM4`.
- `preserve({VM1, VM7, VM5},2,"mem")`. This constraint was satisfied in the source configuration as their hosting servers provided enough memory resources to meet their requirement. The constraint is still satisfied in the destination configuration despite the relocation of `VM7`.

3.13.2 See also

Related Constraints

- `oversubscription`: A constraint made available to the datacenter administrator to control the resource overbooking on the servers.

3.14 Oversubscription

3.14.1 Definition

Signature

```
oversubscription(s :  set<server>, r :  string, x :  number)
```

- `s` : a non-empty set of servers
- `r` : a resource identifier such as `mem`, `ucpu`, `pcpu` to identify the physical memory, the computational capacity, the physical CPUs, respectively.
- `x` : a positive percentage

The `oversubscription` constraint ensures the online servers in `s` have for each hosted VM, an amount of free resources at least equals to a given factor of a physical resource. Servers not in the `Online` state and VMs not in the `Running` state are ignored.

Classification

- **Primary users** : datacenter administrator
- **Manipulated elements** : Resource allocation
- **Concerns** : Resource management

Usage

The memory is usually considered as the bottleneck that limitooor the servers hosting capacity. Originally, the memory was not shareable between the VMs so their cumulated requirement shall not exceed the servers capacity. Hypervisors such as Xen [22] or VMWare [49] now provide sharing systems to oversubscribe the memory. The datacenter administrator can then use `oversubscription` constraint to control this sharing. With a factor greater than 100%, a server can host simultaneously VMs with a cumulated memory usage that exceed its capacity. A high factor increases the hosting capabilities of the servers but may alter the VMs performance due to the overhead of the memory sharing system.

It is also accepted to oversubscribe the physical CPUs (PCPUs) of a server. Each VM uses one or more virtual CPUs (VCPUs) and for the maximum performance, each VCPU should be mapped to a dedicated PCPU. In practice, multiple VCPU are mapped to a same PCPU when the performance overhead is acceptable. In 2010, The Virtual Management Index reported an oversubscription ratio of 200%: each PCPU is allocated to 2 VCPUs on average. [46]

Finally, a VM is usually an instance of a given template that define the maximum amount of computational resources (UCPU) it can consume. To increase the hosting capacity of the servers, the resource are often allocated dynamically, on demand, rather than statically[45, 40, 9, 26]. It is however not desirable to place too many VMs that currently consume a few uCPU on a single server as it increases the chances of having a saturated server if the VMs simultaneously increase their uCPU demand. One solution to control

this oversubscription is to ensure to each of the VMs a given percentage of its maximum uCPU resource usage. For example, the datacenter administrator may use one `oversubscription` constraint and an oversubscription ratio of 80% to ensure each server must be able to provide each of the VMs it hosts 80% of the VM's maximum uCPU usage, even if its current demand is inferior.

Example

Figure 3.16 depicts a sample reconfiguration between a source and a destination configuration where each server provides 8 unit of CPU and 7 unit of memory resources to VMs. The rectangles of the VMs denotes their maximum requirements while the grey part denotes their usage. During the reconfiguration, the following `oversubscription` constraints were considered:

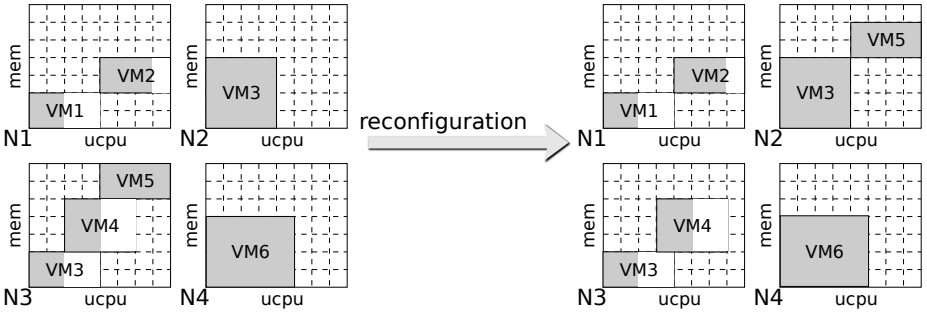


Figure 3.16: A reconfiguration motivated by `oversubscription` constraints.

- `oversubscription({N1,N2},"ucpu", 100%)`. This constraint was satisfied in the source configuration as each of the hosted VMs has enough UCPU resources to satisfy its maximum usage. The constraint is still satisfied in the destination configuration despite the relocation of `VM5` to `N2`.
- `oversubscription({N3, N4},"ucpu",66%)`. This constraint was not satisfied in the source configuration. `VM3` and `VM4` were consuming 50% of their maximum UCPU resources allowed but the presence of `VM5` disallows them to be able to consume the required 66%. The reconfiguration process fixed that violation by relocating `VM5` to `N3`. As a result, `VM3` and `VM4` are ensured to be able to consume at least 66% of their maximum allowed. In practice, there will be able to consume 100%.

3.14.2 See also

Related Constraints

- `preserve`: A constraint to control the resource allocation at the VM level.
- `singleCapacity`: A constraint to control the resource available on servers.

3.15 CumulatedCapacity

3.15.1 Definition

Signature

`cumulatedCapacity(s:set<server>, r:string, nb:number)`

- `s`: a non-empty set of servers for a meaningful constraint. Servers not in the `Online` state are ignored.
- `r` : a resource identifier such as `vm`, `mem`, `ucpu`, `pcpu` or `nodes` to identify the number of virtual machines, the physical memory, the computational capacity, the physical CPUs, respectively.
- `nb`: a positive amount of resources.

The `cumulatedCapacity` constraint restricts to a maximum of `nb`, the total amount of a specific resource of type `r` that can be used on the online servers in `s` to run VMs.

Classification

- **Primary users** : datacenter administrator
- **Manipulated elements** : VM placement,Resource allocation
- **Concerns** : VM-to-server placement,Resource management

Usage

The `cumulatedCapacity` constraint enables first the datacenter administrator to control the shared resources of a platform. As an example, each VM has at least one IP address to be accessible from the network. In practice, a datacenter has a finite pool of addresses to share among all the VMs. Such a datacenter has then a global hosting capacity limited by the size of the address pool. In this setting, one `cumulatedCapacity` constraint may be used to limit the hosting capacity of VMs according to the size of the address pool.

The `cumulatedCapacity` constraint may also be used to control license restrictions. As an example, on a datacenter running vSphere, the hosting capacity is limited by the cumulated amount of memory allotted to the running VMs. [42] In this setting, one `cumulatedCapacity` constraint may be use by the datacenter administrator to guarantee the overall consumption of memory used on the servers running VMWare is necessarily lesser to the maximum allowed by the acquired licenses.

Example

Figure 3.17 depicts a sample reconfiguration between a source and a destination configuration where each server provides 8 unit of CPU and 7 unit of memory resources to VMs. Each VM is associated to a gray rectangle that denotes its resource usage. In this setting, the following `cumulatedCapacity` constraints were considered :

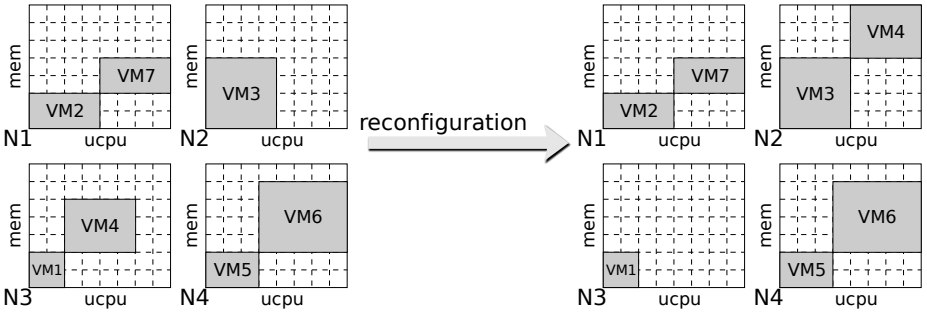


Figure 3.17: A reconfiguration motivated by `cumulatedCapacity` constraints.

- `cumulatedCapacity({N1, N3},7,"mem")`. This constraint was not satisfied in the source configuration as the cumulated memory usage on `N3` and `N1` was equals to 9. This violation was fixed by relocating `VM4` to `N2`.
- `cumulatedCapacity({N3, N4},3,"vm")`. This constraint was not satisfied in the source configuration as there was 4 VMs running on the two nodes while the constraint restricts this number to 3 at maximum. This violation was fixed with the relocation of `VM4` to `N2`.
- `cumulatedCapacity({N2, N4},16,"ucpu")`. This constraint was satisfied in the source configuration as the cumulated UCPU usage of `N2` and `N4` was equals to 12. The constraint is still satisfied in the destination configuration as the cumulated UCPU usage equals 16.

3.15.2 See also

Related Constraints

- `singleCapacity`: A constraint to restrict the resource capacity on each server. A `cumulatedCapacity` constraint is then equivalent to a `singleCapacity` constraint when the set of servers in `cumulatedCapacity` is a singleton.

Specialization(s)

- To `singleCapacity`: $\forall$`n` $\in$`ns, cumulatedCapacity({n}, nb, r)` $\equiv$ `singleCapacity(ns, nb, r)`

3.16 SingleCapacity

3.16.1 Definition

Signature

```
singleCapacity(s:set<server>, nb:number, r:string)
```

- `s`: a non-empty set of servers for a meaningful constraint. Servers not in the `Online` state are ignored.
- `r` : a resource identifier such as `mem`, `ucpu`, `pcpu` or `vm` to identify the physical memory, the computational capacity, the physical CPUs or the number of hosted VMs, respectively.
- `nb`: a positive amount of resources.

The `singleCapacity` constraint restricts to a maximum of `nb`, the amount of a specific resource of type `r` that can be used on each of the online servers in `s` to run VMs.

Classification

- **Primary users** : datacenter administrator
- **Manipulated elements** : VM placement,Resource allocation
- **Concerns** : Resource management,VM-to-server placement

Usage

The `singleCapacity` constraint is used by a datacenter administrator to indicate to the VM manager, the practical amount of resources on each server, that will be available to the VMs. As an example, a server having 4 GB of RAM running a Xen hypervisor [6] cannot offer this amount of memory to the VMs as the *Domain-0* requires memory resources to run. Using a `singleCapacity` constraint, the datacenter administrator may then declare the practical amount of memory that will be available to the VMs by removing the amount used by the *Domain-0*.

Management operations, such as migrations, requires CPU resources on the involved servers. When every resources are devoted to the running VMs, a migration will alter their performance as the hypervisor will use a significant amount of the resources that was allotted to the VMs, to manage the migration. Using `singleCapacity` constraints, the datacenter administrator may prevent this temporary performance loss by dedicating in advance some resources to the hypervisor. Typically a core or a CPU, to let it perform the management operations without impacting the running VMs.

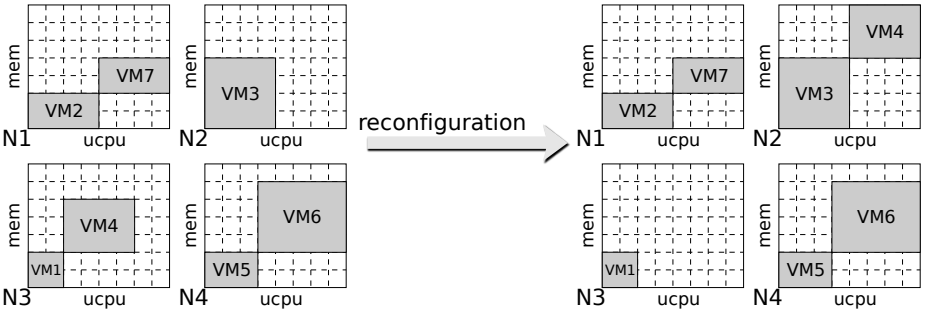


Figure 3.18: A reconfiguration motivated by `singleCapacity` constraints.

Example

Figure 3.18 depicts a sample reconfiguration between a source and a destination configuration where each server provides 8 unit of CPU and 7 unit of memory resources to VMs. Each VM is associated to a gray rectangle that denotes its resource usage. In this setting, the following `singleCapacity` constraints were considered :

- `singleCapacity({N1, N3},4,"mem")`. This constraint was not satisfied in the source configuration as the memory usage by the VMs on `N1` and `N3` equals 4 and 5, respectively. This violation was fixed by relocating `VM4` to `N2` to liberate some resources.
- `singleCapacity({N1, N2},8,"ucpu")`. This constraint was satisfied in the source configuration as the UCPU consumption of the running VMs was equals to 7 at maximum. The constraint is still satisfied in the destination configuration as the relocation of `VM4` to `N2` makes the UCPU resource usage of `N2` to 8, the maximum allowed.
- `singleCapacity({N3},1,"vm")`. This constraint was not satisfied in the source configuration as the number of VMs running on `N3` was 2. The reconfiguration process fixed this violation by relocating `VM4` to `N2`.

3.16.2 See also

Related Constraints

- `cumulatedCapacity`: This constraint can be used when the resource restriction is related to the aggregation of some servers' resources.
- `preserve`: This constraint can be used in addition of `singleCapacity` constraint to ensure every VM has a sufficient amount of resources to run at peak level, according to the resources made available by `singleCapacity` constraints.

Reformulation(s)

- Using `cumulatedCapacity`: `singleCapacity(ns, nb, r)` $\equiv \forall$`n` $\in$`ns, cumulatedCapacity({n}, nb, r)`

3.17 MinSpareResources

3.17.1 Definition

Signature

```
minSpareResources(s : set<server>, rc : string, n : number)
```

- `s` : a non-empty set of servers for a meaningful constraint.
- `rc` : a resource identifier such as `mem`, `ucpu`, `pcpu` or `nodes` to identify the physical memory, the computational capacity, the physical CPUs or the node itself, respectively.
- `n` : a positive number

The `minSpareResources` restricts to at least `n`, the number of free resources directly available for VMs on the online servers in `s`. Servers in the `Offline` state are ignored.

Classification

- **Primary users** : datacenter administrator
- **Manipulated elements** : Resource allocation
- **Concerns** : Resource management,Power saving,Capacity planning

Usage

This constraint deserves the control of a power saving strategy. The most effective solution to reduce the energy consumption of a non-saturated datacenter is to turn off unused servers and to turn them on and off depending on the load variation. When a load spike arise, it may be a necessary to put some servers online to make their resources available to the VMs. [27, 48] The time to boot the awaited servers may however be significant and alter the reactivity of the datacenter when it faces an emergency situations. One solution is to let online a controlled number of *spare* resources that can be used instantly to absorb the load increase. A datacenter administrator may then use `minSpareResources` constraints to control the minimum number of free resources to let directly available while the other unused resources are still manageable by the power saving strategy.

Example

Figure 3.19 depicts a sample reconfiguration between a source and a destination configuration where each server provides 8 unit of CPU and 7 unit of memory resources to VMs. During the reconfiguration, several relocations have been performed and the server `N3` has been turned off to save power. In this setting, the following `minSpareResources` constraints were considered:

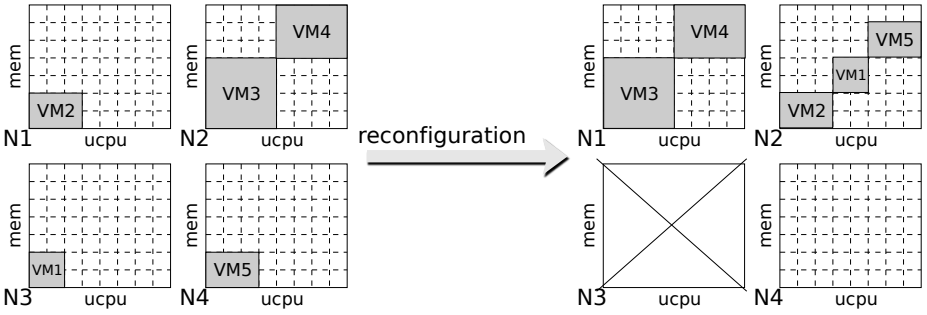


Figure 3.19: A reconfiguration motivated by `minSpareResources` constraints.

- `minSpareResouces({N1,N2,N3},"ucpu",0)`. This constraint was satisfied in the source configuration as 11 unit of CPU were directly available to the running VMs. The constraint is also satisfied in the destination configuration as 0 unit are available: resources on `N3` are not considered as it is in the `offline` state.
- `minSpareResources({N2},"mem",1)`. This constraint was not satisfied in the source configuration as no memory resources were available on `N2`. The reconfiguration process fixed that violation by relocating away `VM4` and `VM3` and host `VM2`,`VM1`, and `VM5` while keeping one unity of memory resources directly available.
- `minSpareResources({N2,N3,N4},"node",1)`. This constraint was not satisfied in the source configuration as no servers were idle. The reconfiguration fixed this violation by relocating all the VMs on `N4` to other servers while keeping it in the `Online` state.

3.17.2 See also

Related Constraints

- `maxSpareResources`: This constraint controls the maximum number of unused but available resources.

3.18 MaxSpareResources

3.18.1 Definition

Signature

```
maxSpareResources(s : set<server>, rc : string, n : number)
```

- `s` : a non-empty set of servers for a meaningful constraint.
- `rc` : a resource identifier such as `mem`, `ucpu`, `pcpu` or `nodes` to identify the physical memory, the computational capacity, the physical CPUs or the node itself, respectively.
- `n` : a positive number

The `maxSpareResources` restricts to at most `n`, the number of free resources directly available for VMs on the online servers in `s`. Servers in the `Offline` state are ignored.

Classification

- **Primary users** : datacenter administrator
- **Manipulated elements** : Resource allocation
- **Concerns** : Resource management,Power saving,Capacity planning

Usage

This constraint deserves a power saving concern. The most effective solution to reduce the energy consumption of a non-saturated datacenter is to turn off unused servers and to turn on and off servers depending on the load variation. When a load spike arise, it may then be a necessary to put some servers online to make their resources available to the VMs. [27, 48] The time to boot the awaited servers may however be significant and alter the reactivity of the datacenter when it faces an emergency situations. One solution is to let online a controlled number of *spare* resources that can be used instantly to absorb the load increase. A datacenter administrator may then use one `maxSpareResources` constraint to control the maximum number of unused servers to let online while the others will be turned off to save power.

It is worth noting that despite this constraint is applicable to a various number of resources, it is preferable to only focus the `nodes` resource. Indeed, a server provides resources at a coarse grain and it may not be possible to manage the resources according to the constraint by only managing the servers state.

Example

Figure 3.20 depicts a sample reconfiguration between a source and a destination configuration where each server provides 8 unit of CPU and 7 unit of memory resources to VMs. During the reconfiguration, several relocations have been performed and the server `N3` has been turned off to save power. In this setting, the following `maxSpareResources` constraints were considered:

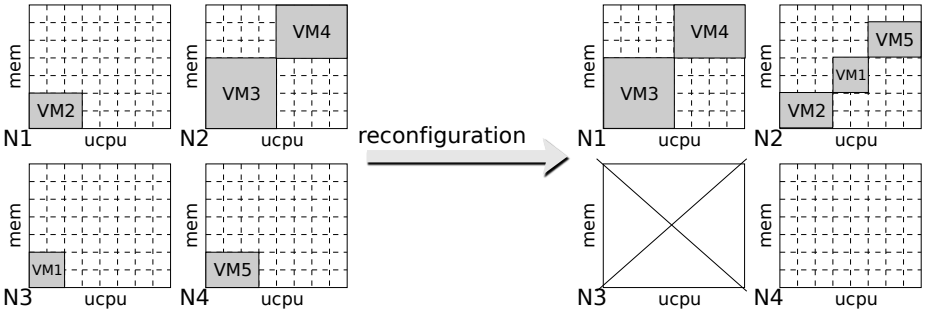


Figure 3.20: A reconfiguration motivated by `maxSpareResources` constraints.

- `maxSpareResources({N2,N3,N4},"node",1)`. This constraint was satisfied in the source configuration as no server was idle. The constraint is still satisfied in the destination configuration as only `N4` is in the `Online` state and idle.
- `maxSpareResources({N1,N2,N3},"ucpu",10)`. This constraint was not satisfied in the source configuration as 11 uCPU was directly available to the running VMs. The reconfiguration fixed this violation. With the shutdown of `N3`, 8 uCPU resources are directly available in the destination configuration, which is an amount allowed by the constraint.
- `maxSpareResources({N1, N3},"mem",3)`. This constraint was not satisfied in the source configuration as 10 unit of memory were directly available to the running VMs. The reconfiguration fixed this violation. With the shutdown of `N3`, and the saturation of `N1`, no memory resources are left available to the VMs on these servers.

3.18.2 See also

Related Constraints

- `minSpareResources`: This constraint restricts the number of unused online servers to a given minimum.

3.19 MaxOnlines

3.19.1 Definition

Signature

```
maxOnlines(s :  set<server>, n :  number)
```

- `s` : a non-empty set of servers for a meaningful constraint.
- `n` : a positive number, inferior to the number of servers in `s`.

The `maxOnlines` ensures the number of online servers in `s` is inferior or equals to `n`.

Classification

- **Primary users** : datacenter administrator
- **Manipulated elements** : Resource allocation
- **Concerns** : Resource management,Power saving,Capacity planning

Usage

This constraint deserves the control of the datacenter hosting capacity. A datacenter may be composed of servers that differ in their hardware capacity or performance. It may however not be possible to keep all the servers online simultaneously. As an example, the cooling system or the power distribution unit may restrict the number of online servers due to its delivering capacity. Licenses restrictions, such as the per-server license model of XenServer, may also limits the number of nodes that are online simultaneously. [**?**]

In this setting, once the maximum number of online servers reached, turning on one additional server to use its specificities is only allowed if an online server can be turned off in exchange. In this setting, a datacenter administrator may then use of `maxOnlines` constraints to control the number of online servers and automatically manage their state if needed.

Example

Figure 3.21 depicts a sample reconfiguration between a source and a destination configuration where only servers `N1`, `N2` and `N3` are online in the source configuration. Each VM is associated to a gray rectangle that denotes its resource usage in terms of memory and UCPU. In the source configuration, the server `N1` was saturated has `VM4` and `VM6` were competing for the same resources. The reconfiguration process fixed this violation but also consider the following `maxOnlines` constraints:

- `maxOnlines({N1,N2, N3, N4}, 3)`. This constraint was satisfied in the source configuration as three over the four servers were in the `Online` state. The constraint is still satisfied in the destination configuration. `N4`ăhas been turned on to host `VM6` but `N3` has been turned off in exchange

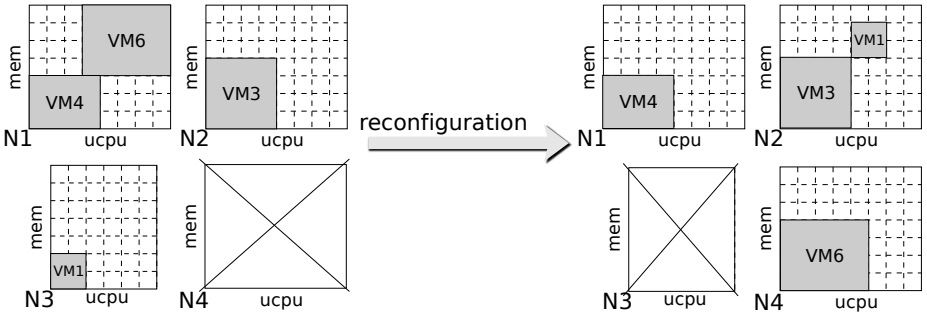


Figure 3.21: A reconfiguration motivated by `maxOnlines` constraints.

Ø	→ relocate(VM1)
!relocate(VM1)	→ halt(N3)
!halt(N3)	→ boot(N4)
!boot(N4)	→ relocate(VM6)

Figure 3.22: Event-based reconfiguration plan associated to the reconfiguration depicted in Figure 3.21.

according to the constraint specification. To be able to turned off `N3`, `VM1` has been relocated to `N2`. Figure 3.22 depicts the associated event-based reconfiguration plan.

- `maxOnlines({N1, N3},1)`. This constraint was not satisfied in the source configuration as the two servers were in the `Online`ăstate. The reconfiguration process fixed this violation by turning off `N3`.

3.19.2 See also

Related Constraints

- `maxSpareResources`: A constraint to restrict the number of unused but available resources to a given maximum.

3.20 Offline

3.20.1 Definition

Signature

`offline(s : set<server>)`

The `offline` constraint forces every server in `s` to be set in the `Offline` state.

Usage

This constraint deserves first hardware maintenance concerns. Using this constraint, one datacenter administrator can turn off a set of servers to perform maintenance operation on the hardware

Classification

- **Primary users** : datacenter administrator
- **Manipulated elements** : Servers state
- **Concerns** : Resource management

Example

3.20.2 See also

Related constraints

- `online`: The opposite constraint that is used to force servers to being set in the `Online` state.
- `maxOnlines`: To restrict the maximum number of servers that are simultaneously in the `Online` state.

3.21 Online

3.21.1 Definition

Signature

`online(s : set<server>)`

The `online` constraint forces every server in `s` to be set in the `Online` state.

Usage

This constraint deserves first the necessity of having servers available to host VMs. This constraint is also useful in a context where servers can not be managed, *i.e.*turned off.

Classification

- **Primary users** : datacenter administrator
- **Manipulated elements** : Servers state
- **Concerns** : Resource management

Example

3.21.2 See also

Related constraints

- `offline`: The opposite constraint that is used to force servers to being set in the `Offline` state.

Chapter 4

Notations

This chapter describes the notations that are used in the catalog to depict configurations or constraints

4.1 Describing a configuration

A configuration depicts the state of servers and virtual machines (see Chapter 2), and the current placement of virtual machines. For convenience, we propose here a textual, human readable, format to describe a configuration. Listing 4.1 describes the textual format for a configuration using the Extended Backus-Naur Form (EBNF) [19]. In addition, every identifier in a configuration is supposed to be unique.

```
configuration = server (endl+ server)* (endl+ waitings)? endl*;
on_id = id;
off_id = "(" id ")";
paused_id = "!" id
waitings = "?" ":" on_id+;
server = (server_off | server_on);
server_off = off_id;
server_on = on_id ":" vm*;
vm = on_id | off_id | paused_id;
endl = "\n";
letter = "a" ... "z" | "A" ... "Z";
digit = "0" ... "9";
id = letter (letter | digit)*;
```

Figure 4.1: EBNF definition of a configuration

Listing 4.2 depicts a sample configuration composed of 3 servers and 5 virtual machines. Server `N1` and `N2` are in the `Online` state. `N1` is hosting 3 virtual machines that are `VM1`, `VM2`, and `VM3`. `VM1` and `VM2` are in the `Running` state while `VM3` is in the `Suspended` state. The server `N2` is hosting the virtual machine

`VM4` that is in the `Paused` state. The server `N3` is in the `Offline` state. Finally, the virtual machine `VM6` is in the `Waiting` state.

```
N1 : VM1 VM2 (VM3)
N2 : !VM4
(N3)
?  : VM6
```

Figure 4.2: A sample well-formed configuration.

4.2 Describing a constraint

4.2.1 Declaration

Each constraint presented in the catalog has a specific signature. In practice, a constraint has an unique identifier and takes into account a variable amount of parameters. Listing reflet: cstr decl ebnf describes the textual format for a constraint signature using the EBNF.

```
constraint_decl = id "(" params ");
id = letter (letter | digit | "_")*;
letter = "a" ... "z" | "A" ... "Z";
digit = "0" ... "9";

params = param ("," param) *;
param = id ":" type;
type = (VM_t | server_t | number_t | set_t | string_t);
VM_t = "VM";
server_t = "server";
number_t = "number";
string_t = "string";
set_t = "set<" type ">";
```

Figure 4.3: EBNF definition of a constraint signature

The following signature declares a constraint named `foo`, that takes as parameters a set of VMs named `s1`, a set of servers named `s2`, a number `x`, and a string `y`:

```
foo(s1:set<VM>, s2:set<set<server», x:number, y:string)
```

4.2.2 Usage

Constraints deserves to be used to indicate management restrictions. Listing 4.4 describes the textual format for a constraint call using the EBNF.

```
constraint_ref & id "(" params ");
params & param ("," param)*;
param & id | set | string;
set & { params? };

string & """ (letter | digit)* """;
id & letter (letter | digit | "_")*;
letter & "a" ... "z" | "A" ... "Z";
digit & "0" ... "9";
```

Figure 4.4: EBNF definition of a constraint signature

Following example shows a sample usage of the constraint `foo` declared previously. The first argument is a set of 3 elements named `VM1`,`VM2`,`VM3`. From the constraint signature, each of these elements has to be a virtual machine. The second parameter is a set of 2 servers that are named `N1`, and `N2`. The third parameter is the number 5. The fourth parameter is the string "bar" :

```
foo({VM1, VM2, VM3}, {{N1, N2},{N3}}, 5, "bar")
```

Chapter 5

Constraint Reformulation

Each VM manager supports a limited set of constraints and a user may expressed some constraints that are not available on the VM manager that will host its VMs.

Using the semantic of each constraint, it is possible to establish an inheritance relationship between constraints. These relations allow to reformulate some constraints that are missing using a peculiar utilization of more generic constraints while preserving their semantics. It has however to be noticed that the semantical equivalence between a constraint and its reformulation does not ensure the two implementations will have an equivalent practical efficiency. Only an analysis of the constraints model and internals may reveal their relative performances.

This chapter summarizes the inheritance relationship between the constraints presented in the catalog. First, we present the inheritance graph between all the constraints to indicate the available reformulations We then present the associated rewriting rules.

5.1 Inheritance Graph

Figure 5.1 depicts the global inheritance graph between the constraints. An arrow between two constraints indicates a possible reformulation. As an example, a `gather` constraint can be reformulated using a `among` constraint. It is then possible to express the restriction provided by a `gather` constraint using a peculiar specialization of a `among` constraint. The inheritance relationship is also transitive. A `gather` constraint can then be expressed using a `splitAmong` constraint. A "+" vertex indicates a composition of constraints. As an example, the `quarantine` constraint can be reformulated using a peculiar composition of `root` and `ban` constraints.

5.2 Rewriting rules

In this section, we present the rewriting rules associated to each constraint reformulation.

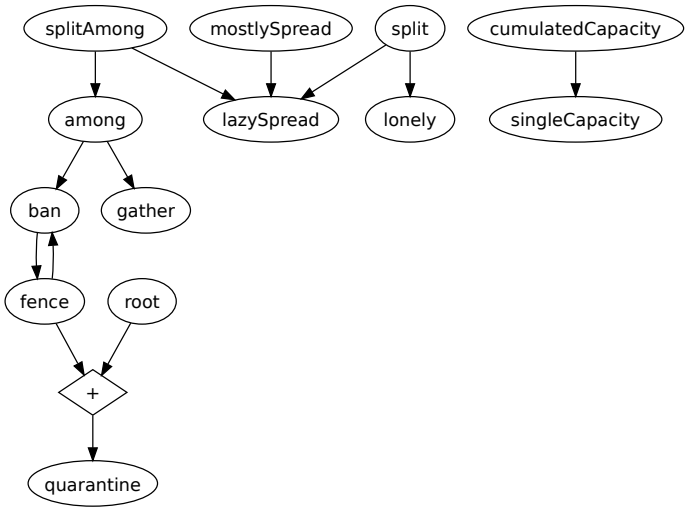


Figure 5.1: Inheritance graph between the constraints.

5.2.1 SplitAmong

Specialization(s)

- To `lazySpread`: `splitAmong(s/|s|,`$\mathcal{N}$`/|`$\mathcal{N}$`|)` $\equiv$ `lazySpread(s1)`
- To `among`: `splitAmong({vs1},ns1)` $\equiv$ `among(vs1, ns1)`

5.2.2 Lonely

Reformulation(s)

- Using `split`: `lonely(vs1)` $\equiv$ `split({vs1,`$\overline{\texttt{vs1}}$`})`

5.2.3 MostlySpread

Specialization(s)

- To `lazySpread`: `mostlySpread(s,|s|)` $\equiv$ `lazySpread(s)`

5.2.4 Quarantine

Reformulation(s)

- Using `fence`+`root`: `quarantine(s)` $\equiv$ `root(`$\uparrow$`s), fence(`$\overline{\uparrow \texttt{s}}$`,` $\overline{\texttt{s}}$`)`

5.2.5 Split

Specialization(s)

- To `lonely`: `split({vs1,`$\overline{\texttt{vs1}}$`})` $\equiv$ `lonely(vs1)`
- To `lazySpread`: `split(s/|s|)` $\equiv$ `lazySpread(s)`

5.2.6 LazySpread

Reformulation(s)

- Using `mostlySpread`: `lazySpread(s)` $\equiv$ `mostlySpread(s,|s|)`
- Using `splitAmong`: `lazySpread(s1)` $\equiv$ `splitAmong(s/|s|,`$\mathcal{N}$`/|`$\mathcal{N}$`|)`
- Using `split`: `lazySpread(s)` $\equiv$ `split(s/|s|)`

5.2.7 Ban

Reformulation(s)

- Using `fence`: `ban(vs1, ns1)` $\equiv$ `fence(vs1,`$\overline{\texttt{ns1}}$`)`
- Using `among`: `ban(vs1, ns1)` $\equiv$ `among(vs1,{`$\overline{\texttt{ns1}}$`})`

Specialization(s)

- To `fence`: `ban(vs1,`$\overline{\texttt{ns1}}$`)` $\equiv$ `fence(vs1, ns1)`

5.2.8 SingleCapacity

Reformulation(s)

- Using `cumulatedCapacity`: `singleCapacity(ns, nb, r)` $\equiv \forall$`n` $\in$`ns, cumulatedCapacity({n}, nb, r)`

5.2.9 Fence

Reformulation(s)

- Using `ban`: `fence(vs1, ns1)` $\equiv$ `ban(vs1,`$\overline{\texttt{ns1}}$`)`

Specialization(s)

- To `ban`: `fence(vs1,`$\overline{\texttt{ns1}}$`)` $\equiv$ `ban(vs1, ns1)`

5.2.10 Among

Reformulation(s)

- Using `splitAmong`: `among(vs1, ns1)` $\equiv$ `splitAmong({vs1},ns1)`

Specialization(s)

- To `gather`: `among(s,`$\mathcal{N}$`/|`$\mathcal{N}$`|)` $\equiv$ `gather(s)`
- To `ban`: `among(vs1,{`$\overline{\texttt{ns1}}$`})` $\equiv$ `ban(vs1, ns1)`

5.2.11 Gather

Reformulation(s)

- Using `among`: `gather(s)` $\equiv$ `among(s,`$\mathcal{N}$`/|`$\mathcal{N}$`|)`

5.2.12 CumulatedCapacity

Specialization(s)

- To `singleCapacity`: $\forall$`n` $\in$`ns, cumulatedCapacity({n}, nb, r)` $\equiv$ `singleCapacity(ns, nb, r)`

Constraints Index

By Concern

By Element

By User

Bibliography

[1] M. Al-Fares, A. Loukissas, and A. Vahdat. A scalable, commodity data center network architecture. In *ACM SIGCOMM Computer Communication Review*, volume 38, pages 63–74. ACM, 2008.

[2] Amazon EC2. `http://aws.amazon.com/ec2/`.

[3] Apache HTTP Server Project. `http://httpd.apache.org`.

[4] Apache Tomcat. `http://tomcat.apache.org`.

[5] Windows Azure. `http://www.windowazure.com`.

[6] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *19th SOSP*, pages 164–177, 2003.

[7] L. A. Barroso, J. Dean, and U. Hölzle. Web search for a planet: The google cluster architecture. *IEEE Micro*, 23(2):22–28, 2003.

[8] E. Bin, O. Biran, O. Boni, E. Hadad, E. Kolodner, Y. Moatti, and D. Lorenz. Guaranteeing high availability goals for virtual machine placement. In *31th ICDCS*, june 2011.

[9] N. Bobroff, A. Kochut, and K. Beaty. Dynamic placement of virtual machines for managing sla violations. In *Integrated Network Management, 2007. IM'07. 10th IFIP/IEEE International Symposium on*, pages 119–128. IEEE, 2007.

[10] F. Cappello, E. Caron, M. Dayde, F. Desprez, Y. Jegou, P. Primet, E. Jeannot, S. Lanteri, J. Leduc, N. Melab, G. Mornet, R. Namyst, B. Quetier, and O. Richard. Grid'5000: A large scale and highly reconfigurable grid experimental testbed. *IEEE/ACM Int. Workshop on Grid Computing*, 2005.

[11] Information Technology Cloud Data Management Interface (CDMI). `http://snia.org/sites/default/files/CDMI_SNIA_Architecture_v1.0.1.pdf`, sep 2011.

[12] J. S. Chase, D. E. Irwin, L. E. Grit, J. D. Moore, and S. E. Sprenkle. Dynamic virtual clusters in a grid site manager. In *HPDC '03: Proceedings of the 12th IEEE International Symposium on High Performance Distributed Computing*, page 90, Washington, DC, USA, 2003. IEEE Computer Society.

[13] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield. Live migration of virtual machines. In *2nd NSDI*, pages 273–286, 2005.

[14] B. Cully, G. Lefebvre, D. Meyer, M. Feeley, N. Hutchinson, and A. Warfield. Remus: High availability via asynchronous virtual machine replication. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, pages 161–174, 2008.

[15] Distributed Management Task Force. `http://dtmf.org`.

[16] W.-c. Feng, J. G. Hurwitz, H. Newman, S. Ravot, R. L. Cottrell, O. Martin, F. Coccetti, C. Jin, X. D. Wei, and S. Low. Optimizing 10-gigabit ethernet for networks of workstations, clusters, and grids: A case study. In *SC '03: Proceedings of the 2003 ACM/IEEE conference on Supercomputing*, page 50, Washington, DC, USA, 2003. IEEE Computer Society.

[17] Fit 4 green. `http://fit4green.eu`, 2011.

[18] H. Frazier and H. Johnson. Gigabit ethernet: From 100 to 1,000 mbps. *IEEE Internet Computing*, 3(1):24–31, 1999.

[19] L. Garshol. Bnf and ebnf: What are they and how do they work?

[20] A. Greenberg, J. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. Maltz, P. Patel, and S. Sengupta. Vl2: a scalable and flexible data center network. *ACM SIGCOMM Computer Communication Review*, 39(4):51–62, 2009.

[21] C. Guo, G. Lu, D. Li, H. Wu, X. Zhang, Y. Shi, C. Tian, Y. Zhang, and S. Lu. Bcube: a high performance, server-centric network architecture for modular data centers. In *ACM SIGCOMM Computer Communication Review*, volume 39, pages 63–74. ACM, 2009.

[22] D. Gupta, S. Lee, M. Vrable, S. Savage, A. C. Snoeren, G. Varghese, G. M. Voelker, and A. Vahdat. Difference engine: harnessing memory redundancy in virtual machines. *Commun. ACM*, 53(10):85–93, Oct. 2010.

[23] J. R. Hamilton. Architecture for modular data centers. *Innovative Data Sys. Research*, pages 306–313, 2007.

[24] R. Harper, L. Tomek, O. Biran, and E. Hadad. A virtual resource placement service. In *2011 IEEE/IFIP 41st International Conference on Dependable Systems and Networks Workshops (DSN-W)*, pages 158–163, june 2011.

[25] F. Hermenier, S. Demassey, and X. Lorca. Bin Repacking Scheduling in Virtualized Datacenters. *Principles and Practice of Constraint Programming–CP 2011*, pages 27–41, 2011.

[26] F. Hermenier, X. Lorca, J. Menaud, G. Muller, and J. Lawall. Entropy: a consolidation manager for clusters. In *Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, pages 41–50. ACM, 2009.

[27] F. Hermenier, N. Loriant, and J. Menaud. Power management in grid computing with xen. In *Frontiers of High Performance Computing and Networking–ISPA 2006 Workshops*, pages 407–416. Springer, 2006.

[28] M. Hibler, R. Ricci, L. Stoller, J. Duerig, S. Guruprasad, T. Stack, K. Webb, and J. Lepreau. Large-scale virtualization in the emulab network testbed. In *USENIX ATC*, 2008.

[29] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori. kvm: the linux virtual machine monitor. In *Proceedings of the Linux Symposium*, volume 1, pages 225–230, 2007.

[30] C. Leiserson. Fat-trees- university networks for hardware-efficient supercomputing. *IEEE Transactions on Computers*, 34:892–901, 1985.

[31] C. Liu, B. T. Loo, and Y. Mao. Declarative automated cloud resource orchestration. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*, SOCC '11, pages 26:1–26:8, New York, NY, USA, 2011. ACM.

[32] Microsoft Corporation. Windows Server 2008 - Hyper-V Product Overview - An Early Look. Technical report, 2007.

[33] MySQL. `http://mysql.com`.

[34] M. Nelson, B.-H. Lim, and G. Hutchins. Fast transparent migration for virtual machines. In *ATEC '05: Proceedings of the annual conference on USENIX Annual Technical Conference*, pages 25–25, Berkeley, CA, USA, 2005. USENIX Association.

[35] OCCI Working Group. Open Cloud Computing Interface - Infrastructure. Technical report, apr 2011.

[36] Open virtualization format specification. `http://www.dmtf.org/standards/published_documents/DSP0243_1.0.0.pdf`, feb 2009.

[37] RackSpace Hosting. `http://www.rackspace.com`.

[38] R. Raman, M. Livny, and M. Solomon. Matchmaking: Distributed resource management for high throughput computing. In *In Proceedings of the Seventh IEEE International Symposium on High Performance Distributed Computing*, pages 28–31, 1998.

[39] M. Rosenblum. Vmware's virtual platform. In *Proceedings of Hot Chips*, pages 185–196, 1999.

[40] P. Ruth, J. Rhee, D. Xu, R. Kennell, and S. Goasguen. Autonomic live adaptation of virtual computational environments in a multi-domain infrastructure. In *Autonomic Computing, 2006. ICAC'06. IEEE International Conference on*, pages 5–14. IEEE, 2006.

[41] D. J. Scales, M. Nelson, and G. Venkitachalam. The design of a practical system for fault-tolerant virtual machines. *SIGOPS Oper. Syst. Rev.*, 44(4):30–39, Dec. 2010.

[42] VMware vSphere 5. Licensing, Pricing and Packaging. Technical report, 2011.

[43] T. Sterling, D. J. Becker, D. Savarese, J. E. Dorband, U. A. Ranawake, and C. V. Packer. Beowulf: A parallel workstation for scientific computation. In *In Proceedings of the 24th International Conference on Parallel Processing*, pages 11–14. CRC Press, 1995.

[44] Top500 supercomputing. `http://top500.org`.

[45] A. Verma, P. Ahuja, and A. Neogi. pmapper: power and migration cost aware application placement in virtualized systems. In *Proceedings of the 9th ACM/IFIP/USENIX International Conference on Middleware*, pages 243–264. Springer-Verlag New York, Inc., 2008.

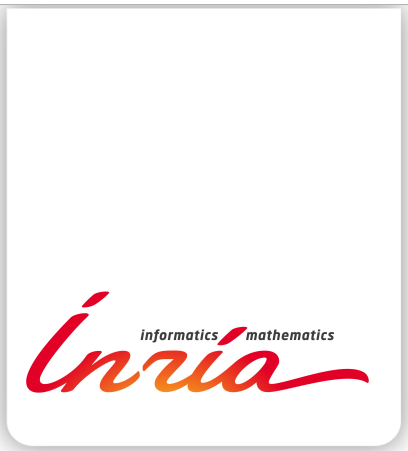
[46] Virtualization management index. Technical report, VKernel, Dec. 2010.

[47] VMWare Infrastructure: Resource Management with VMWare DRS. Technical report, 2006.

[48] VMWare Distributed Power Management Concepts and Use. Technical report, 2009.

[49] C. Waldspurger. Memory resource management in vmware esx server. *ACM SIGOPS Operating Systems Review*, 36(SI):181–194, 2002.

[50] R. Wojtczuk and J. Rutkowska. Following the white rabbit: Software attacks against intel vt-d technology. Technical report, Invisible Things Lab, 2011.



RESEARCH CENTRE
SOPHIA ANTIPOLIS – MÉDITERRANÉE

2004 route des Lucioles - BP 93
06902 Sophia Antipolis Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-0803