Sequencer: smart control of hardware and software components in clusters (and beyond).

Dr. Pierre Vignéras
pierre.vigneras@bull.net
Extreme Computing R&D
Bull, Architect of an Open World
www.bull.com

September 15, 2011

Abstract

Starting/stopping a whole cluster or a part of it is a real challenge considering the different commands related to various device types and manufacturers, and the order that should be respected. This article presents a solution called the *sequencer* that allows the automatic shutting down and starting up of clusters, subset of clusters or even data-centers. It provides two operation modes designed for ease of use and emergency conditions. Our product has been designed to be efficient and it is currently used to power on and power off one of the largest cluster in the world: the Tera-100, made of more than 4000 nodes.

Keywords: emergency power off, start/stop procedure, actions sequencing, workflow, planning, cluster management, automation, case study.

1 Introduction

Emergency Power Off (EPO) is often not considered from a system administration point of view in traditionnal clusters[1]. Even for Tier-IV infrastructures [1], EPO may happen for various reasons. In such cases, stopping (or starting, both cases are addressed) macro components such as a whole rack or a rack set requires an appropriate sequence of actions. Considering the vast number of different components a cluster is composed of:

nodes: compute nodes[2], login nodes, management nodes, io (nfs, lustre, ...) nodes, ...

hardware: power switches (also called Power Distribution Units or PDUs), ethernet switches, infiniband [2] switches, cold doors[3], disk arrays, ...

powering on/off a whole set of racks can be a real challenge.

First, since it is made of a set of heterogeneous devices, starting/stopping each component of a cluster is not straightforward: usually each device type comes with its own poweron/off command. For example, shutting down a node can be as simple as an

[1]In this article, we consider clusters because they are the first target of our solution. However, this solution also applies to general data-centers.

[2]From a hardware perspective, a node in a cluster is just a computer. A distinction is made however between nodes depending on their roles in the cluster. For example, user might connect to a login node for development, and job submission. The batch scheduler runs on the management node and dispatch jobs to compute nodes. Compute nodes access storage through io nodes and so on.

[3]A cold door is a water-based cooling system produced by Bull that allows high density server in the order of 40 kW per rack.

'`ssh host /sbin/halt -p`'. However, it might be preferable to use an out of band command through the IPMI BMC[4] if the node is unresponsive for example. Starting a node can be done using a wake on lan [3] command or an IPMI [4] command. Some devices cannot be powered on/off remotely (infiniband or ethernet switches for example). Those devices might be connected to manageable Power Distribution Units (PDUs) that can remotely switch their outlets on/off using SNMP [5] commands. On the extreme case, manual intervention might be required to switch on/off the electrical power.

For software components, there is also a need to manage the shutdown of multiple components on different nodes. Considering high availability framework, virtualization, localization, and clients, using standard calls to `/etc/init.d/service [start|stop]` is often inappropriate.

Finally, the set of instructions for the powering on/off of each cluster's components should be ordered. Trivial examples include:

- powering off an ethernet switch too soon may prevent other components, including nodes, from being powered off;
- powering off a cold door should be done at the very end to prevent cooled components from being burnt out.

By the way, this ordering problem is not only relevant to hardware devices. A software component can also require that a sequence of instructions is executed before being stopped. As a trivial example, when an NFS daemon is stopped, one may prefer that all NFS clients unmount their related directories first in order to prevent either the fill of syslog with NFS mount error (when NFS mount option is 'soft') or the load average brutal increase due to the freezing of softwares accessing the NFS directories (when NFS mount option is 'hard').

Therefore, in this article, the generic term 'component' may define a hardware component such as a node, a switch, or a cold door, or a software component such as a lustre server or an NFS server.

[4]Baseboard Management Controller.

Our proposition — called the *sequencer* — addresses the problem of starting/stopping a cluster (or a data-center). Its design takes into account emergency conditions. Those conditions impose various constraints addressed by our solution:

- Predictive: an EPO should have been validated before being used. It should not perform unknown actions.
- Easy: an EPO should be easy to launch. The emergency cause may happen at any time, especially when skilled staff is not present. Therefore, the EPO procedure should be launchable by "unskilled" humans.
- Fast: an EPO should be as fast as possible.
- Smart: an EPO should power off each component of a cluster in the correct order so most resources will be preserved.
- Robust: an EPO should be tolerant to failure. For example, if a shutdown on a node cooled by a cold door returned an error, the corresponding cold door should not be switched off to prevent the burnout of the node. On the other side, the rest of the cluster can continue the EPO process.

This article is organized as follow: section 2 exposes the design of our solution while some implementation details are dealt with in section 3 following by scalability issues in section 4. Some results of our initial implementation are given section 5. Section 6 compares our solution to related works. Finally, section 7 presents future works.

2 Design

Three major difficulties arise when considering the starting/stopping of a cluster or of a subset of it:

1. the computing of the dependency graph between components (power off a cold door after all components of the related rack have been powered off);

2. the defining of an efficient (scalable) instructions sequence where the order defined by the dependency graph is respected (powering off nodes might be done in parallel);
3. the execution of the instructions set itself, taking failure into account properly (do not power off the cold door, if related rack's nodes have failed to power off).

Therefore, the sequencer is made of three distinct functional layers:

Dependency Graph Maker (DGM): this layer computes the dependency graph according to dependency rules defined by the system administrator in a database.

Instructions Sequence Maker (ISM): this layer computes the instructions sequence that should be executed for starting/stopping the given list of components and that satisfies dependency constraints defined in the dependency graph computed by the previous layer.

Instructions Sequence Executor (ISE): this layer executes the instructions sequence computed by the previous layer and manages the handling of failures.

Finally, a chaining of those layers is supported through an "all-in-one" command.

This design provides therefore two distinct processes for the starting/stopping of components:

Incremental Mode: in this mode, each stage is run separately. The output of each stage can be verified and modified before being passed to the next stage as shown on figure 2.1. The incremental mode generates a script from constraints expressed in a database table and from a components list. This script is optimized in terms of scalability and is intepretable by the instructions sequence executor that deals with parallelism and failures. This mode is the one designed for emergency cases. The instructions set computed should be validated before being used in production.

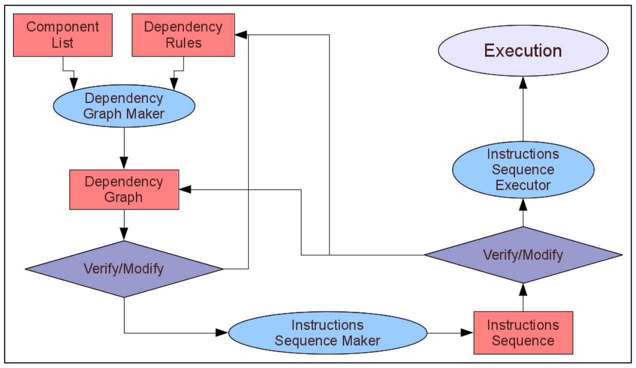


Figure 2.1: Incremental Mode: each stage output can be verified and modified before being passed to the next one.

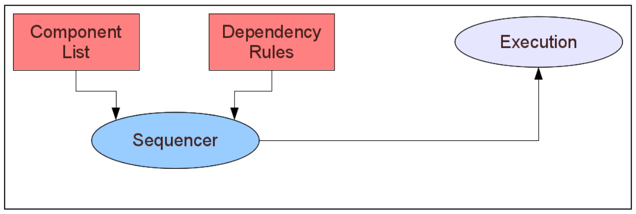


Figure 2.2: Black Box Mode: using the sequencer for simple non-critical usage.

Black Box Mode: in this mode, illustrated in figure 2.2, chaining feature is used to start/stop components as shown by the following syntax:

```
# clmsequencer \                  # command name
stop \                            # ruleset name
colddoor3 node[100-200]           # components list
```

This command is somewhat equivalent to the following:

```
# clmsequencer \                  # command name
depmake \                         # dgm stage
stop \                            # ruleset name
colddoor3 node[100-200] \         # components list
|clmsequencer seqmake \           # ism stage
|clmsequencer seqexec             # ise stage
```

and can therefore be seen as a syntactic sugar.

The computation of the dependency graph and of the instruction set can take a significant amount of time, especially on very large clusters such as the

Tera-100[5]. This is another good reason for choosing the incremental mode in emergency conditions where each minute is important.

3 Implementation

3.1 Dependency Graph Maker (DGM)

The Dependency Graph Maker (DGM) is the first stage of the sequencer. It takes a components list in parameter, and produces a dependency graph in output. It uses a set of dependency rules described in a database table. The CLI has the following usage:

```
  # clmsequencer depgraph [--out file]
ruleset cl_1...cl_N
```

The output is a human readable description of the computed dependency graph in XML format that the Instructions Sequence Maker can parse. By default, the computed dependency graph is produced on the standard output.

The `--output file` option allows the computed dependency graph to get written in the specified `file`.

The `ruleset` parameter defines which ruleset should be used to compute the dependency graph. Ruleset will be explained in section 3.1.2 on the sequencer table.

Finally, other parameters `cl_1...cl_N` define on which components the dependency graph should be computed. Each parameter describes a list of component in a specific format describes in next section 3.1.1.

3.1.1 Components list specification

The first stage of the sequencer takes as an input a list of components. This list is of the form:

```
prefix[a-b,c-d,...][#type][@category]
```

where:

prefix[a-b,c-d,...]: is the standard contracted notation for designing a set of names prefixed by `'prefix'` and suffixed by a number taken in the range given by intervals $[a-b]$, $[c-d]$, and so on. For example, `compute[1-3,5,7-8]` defines names: `compute1`, `compute2`, `compute3`, `compute5`, `compute7`, `compute8`.

category: is optionnal and defines the table[6] where given names should be looked for their `type` (if not given). The type of a component is used in the definition of the dependency table as described in section 3.1.2. Category examples (with some related types) are: `node` (`io`, `nfs`, `login`,`compute`), `hwmanager` (`bmc`, `cmc`, `coldoor`[7]) and `soft` (`nfsd`, `nagios`, `sshd`).

Some examples of full component list names are given below:

`R-[1-3]#io@rack`: the io racks R-1, R-2 and R-3;

`bullx[10-11]#mds@node`: the lustre mds node bullx10 and bullx11;

`colddoor1#coldoor@hwmanager`: the cold door numbered 1;

`esw-1#eth@switch`: the ethernet switch esw-1;

`server[1-2]#nfsd@soft`: NFS daemons running on server1 and server2.

3.1.2 Sequencer Dependency Rules: the sequencer table

The Dependency Graph Maker requires dependency rules to be specified in a database table. This table describes multiple sets of dependency rules. A *ruleset* is defined as a set of dependency rules. For example, there is one ruleset called `smartstart` containing all

[5]Tera-100 is ranked #6 in the Top500 november 2010 list of fastest supercomputers in the world and #1 in Europe. It is composed of several thousands of Bull bullx series S servers. See `http://www.top500.org/` for details.

[6]It is considered a good practice to have a database where the cluster is described. In a bullx cluster, each component is known and various informations are linked to it such as its model, its status, its location and so on. There should be a way to find a type from a component name. In this article, we use a database for that purpose, any other means can be used though.

[7]Cold doors are spelled 'coldoor' in bullx cluster database.

the dependency rules required for the starting of components. Another ruleset containing all dependency rules required for the stopping of components would be called `smartstop`.

The format of this table is presented below. One line in the table represents exactly one dependency rule. Table columns are:

`ruleset:` the name of the ruleset this dependency rule is a member of.

`name:` the rule name, this name is used as a reference in the `dependson` column, it should be unique in the ruleset;

`types:` the component types the rule should be applied to. A type is specified using the full name (that is, `'type@category'`). Multiple types should be separated by the "pipe" symbol as in `compute@node|io@node`. The special string `'ALL'` acts like a joker: `'ALL@node'` means any component from table node matches, while `'ALL@ALL'` means any component matches, and is equivalent to `'ALL'` alone.

`filter:` an expression of the following two forms:

- `%var =~ regexp`
- `%var !~ regexp`

where `'%var'` is a variable that will be replaced by its value on execution (see table 1 for the list of available variables). The operator `'=~'` means that component will be filtered in only if a match occurs while `'!~'` means the component will be filtered in only if a match does not occur (said otherwise, it a match occurs, it will be filtered out).

If the expression does not start with a known `'%var'` then, the expression is interpreted as a (shell) command that when called specifies if the given component should be filtered in (returned code is 0) or out (returned code is different than 0). Variables will also be replaced before command execution, if specified. As an example, to filter out any component which name starts with the string `'bullx104'`, one would use: `'%name =~ ^bullx104'`. On the other side, to let a script decide on the component id, one would use: `'/usr/bin/my_filter %id'`.

Finally, two special values are reserved for special meanings here:

- String `'ALL'`: any component is filtered in (i.e. accepted);
- The `'NULL'` special DB value: any component is filtered out (i.e. refused).

`action:` the (shell) command that should be executed for each component that matches the rule type (and that have been filtered in). Variables will be replaced, if specified (see table 1 for the list of available variables). If the action is prefixed with the `'@'` symbol, the given action will be executed on the component using an `'ssh'` internal connexion. Depending on the action exit code, the Instruction Sequence Executor may continue its execution, or abort. This will be discussed in section 3.3.

`depsfinder:` the (shell) command that specifies which components the current component depends on. The command should return the components set on its standard output, one component per line. A component should be of the following format: `'name#type@category'`. Variables will be replaced, if specified (see table 1 for the list of available variables). When set to the `'NULL'` special DB value, rule names specified in the `dependson` column are simply ignored.

`dependson:` a comma-separated list of rule names, this rule depends on. For each dependency returned by the depsfinder, the sequencer looks if the dependency type matches one of the rule type specified by this field (rule names specified should be in the same ruleset). If such a match occurs, the rule is applied on the dependency. When set to the `'NULL'` special DB value, the script specified in the `'depsfinder'` column is simply ignored.

`comments:` a free form comment.

Name	Value	Example
%id	The full name of the component	bullx12#compute@node
%name	The name of the component	bullx12
%type	The type of the component	compute
%category	The category of the component	node
%ruleset	The current ruleset being processed	smartstop
%rulename	The current rule being processed	compute_off

Table 1: List of available variables.

The framework does not allow the specification of a timeout for a given action for two main reasons:

1. Granularity: if such a specification was provided at that level (in the sequencer table), the timeout would be specified for all components type specified by the `'type'` column whereas it seems preferable to have a lower granularity, per component. This is easily achievable by the action script itself for which the component can be given as a parameter.

2. The action script, for a given component, knows what to do when a timeout occurs much better that the sequencer itself. Therefore, if a specific process is required after a command timeout (such as a retry), the action script should implement itself the required behavior when the timeout occurs and returns the appropriate return code.

As an example we will consider the sequencer table presented in table 2.

3.1.3 Algorithm

The objective of the Dependency Graph Maker is to output the dependency graph based on the dependency rules defined in the related table and on the components list given as a parameter. The computing of the dependency graph involves the following steps:

1. **Components List Expansion**: from the given components list, the expansion should be done. It returns a list of names of the form: `'name#type@category'`. Such name is called *id* in the following.

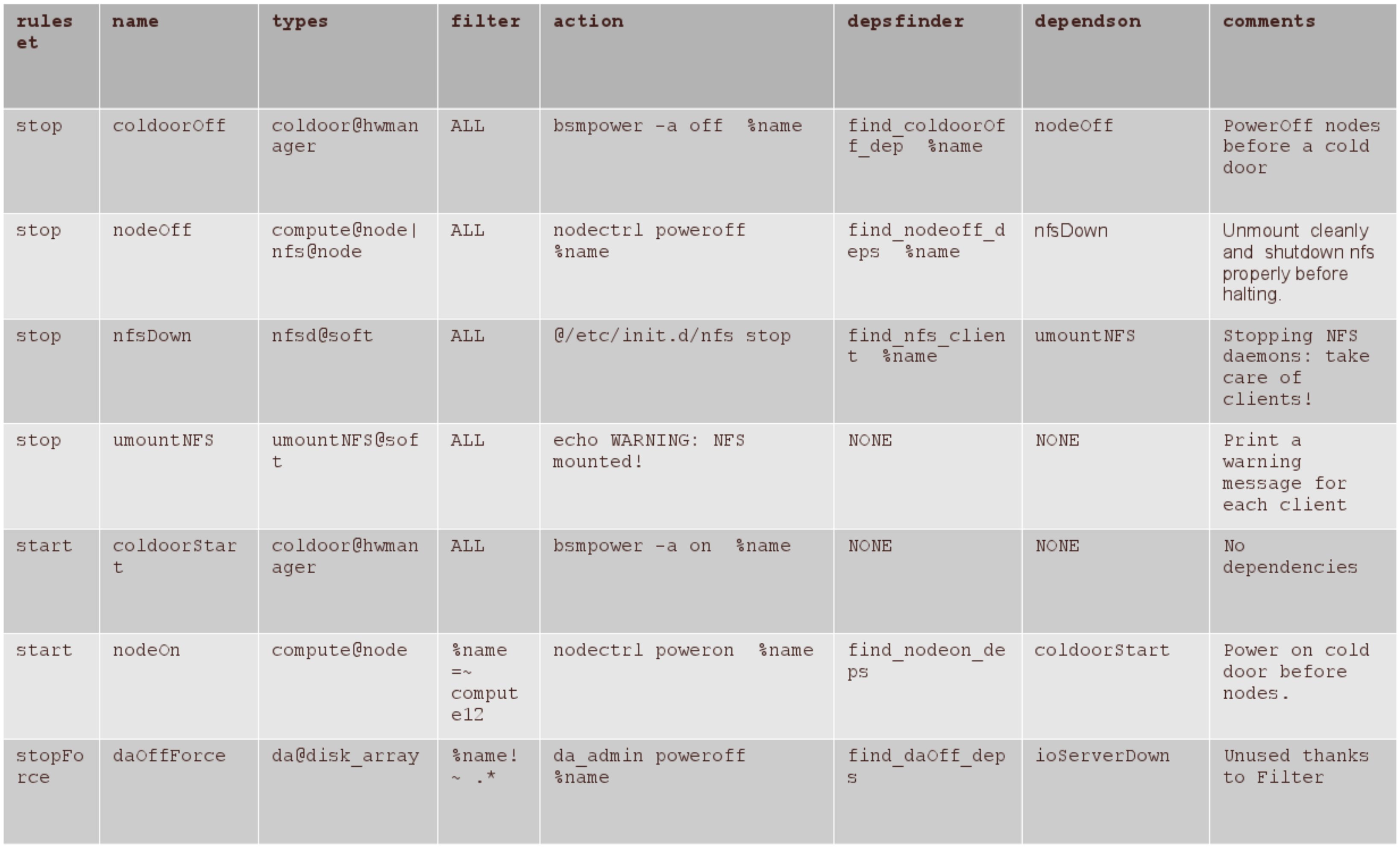
2. **Dependency Graph Creation**: the dependency graph is created as a set of disconnected nodes where each node is taken from the list of ids. A node in the dependency graph has the following form: '`id [actionsList]'` where `'actionsList'` is the list of actions that should be executed for the component with the corresponding '`id`'. This graph is updated during the process by:

   (a) Node additions: when processing a given component `'c'`, through a dependency rule (one row in the related ruleset table), the command line specified by column `'depsfinder'` is executed. This execution may return a list of components that should be processed before the current one. Each of those components will therefore be added to the dependency graph if it is not already present.

   (b) Arc additions: for each components returned by the `'depsfinder'` script, an arc is added between `'c'` and that returned component;

   (c) Node modification: when processing a given component, the content of the column `'action'` of the ruleset table of the related dependency rule is added to the node actions list.

3. **Rules Graph Making**: from the dependency rules table, and for a given ruleset, the corresponding graph – called the *rules graph* – is created. A node in that graph is a pair (s, t) where



ruleset	name	types	filter	action	depsfinder	dependson	comments
stop	coldoorOff	coldoor@hwmanager	ALL	bsmpower -a off %name	find_coldoorOff_dep %name	nodeOff	PowerOff nodes before a cold door
stop	nodeOff	compute@node|nfs@node	ALL	nodectrl poweroff %name	find_nodeoff_deps %name	nfsDown	Unmount cleanly and shutdown nfs properly before halting.
stop	nfsDown	nfsd@soft	ALL	@/etc/init.d/nfs stop	find_nfs_client %name	umountNFS	Stopping NFS daemons: take care of clients!
stop	umountNFS	umountNFS@soft	ALL	echo WARNING: NFS mounted!	NONE	NONE	Print a warning message for each client
start	coldoorStart	coldoor@hwmanager	ALL	bsmpower -a on %name	NONE	NONE	No dependencies
start	nodeOn	compute@node	%name =~ compute12	nodectrl poweron %name	find_nodeon_deps	coldoorStart	Power on cold door before nodes.
stopForce	daOffForce	da@disk_array	%name!~ .*	da_admin poweroff %name	find_daOff_deps	ioServerDown	Unused thanks to Filter

Table 2: An example of a sequencer table.

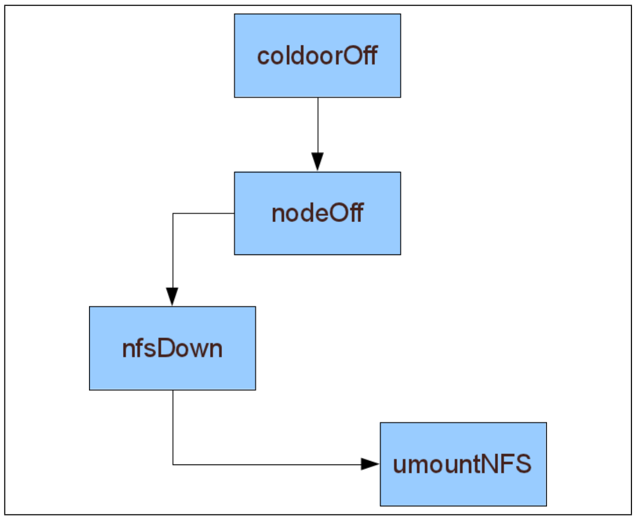


Figure 3.1: Rules Graph of the '`stop`' ruleset defined in the sequencer table 2.

s is the rule symbolic name, and t is the component types defined by the '`types`' column in the dependency rule table. This graph is used for the selection of a component in the components list to start the dependency graph update process. From table 2, the rules graph of the `stop` ruleset is shown figure 3.1. Note that cycles are possible in this graph. As an example, a PDU (related to a `switch` type in the sequencer table) that connects (power) an ethernet switch which itself connects (network) a PDU.

4. **Updating the Dependency Graph**: from each ids (resulting from the expansion of each initial components list), the corresponding rule in the given ruleset of the sequencer table should be found. For that purpose, a *potential root* is looked for in the ids set. A potential root is an id that *matches* one root[8] in the rules graph. A match between an `id` of the form '`name#type@category`' and a rule '`R`' occurs when `type` is in '`R.types`' and when id has been *filtered in*. If such a match cannot be found, then, a new rules graph is derived from the preceding one by removing all roots and their related edges. Then, the root finding is done on that new graph, and so on recursively until either:

   - the rule graph is empty: in this case, the given components list cannot be started/stopped (entirely);
   - the rule graph is only made of cycles: any id can be used as the starting point;
   - a match occurs between '`id`' and '`R`': in this case, the dependency graph is updated from the application of rule '`R`' to '`id`'. Each time such an application is made, '`id`' is removed from the initial id set.

[8]A node in the graph with no parent.

As an example, consider a rack cooled by a bullx cold door '`cd0`' containing an NFS server '`nfs1`' and a compute node '`c1`'. Consider also another NFS server '`nfs2`' that is not in the same rack. We also suppose that:

- '`c1`' is client of both '`nfs1`' and '`nfs2`';
- '`nfs1`' is client of '`nfs2`';
- '`nfs2`' is client of '`nfs1`'[9].

Using table 2, and the component list: '`nfs1#nfsd@soft, cd0, nfs2`', objectives are:

- power off '`c1`' and '`nfs1`' before '`cd0`', because powering off a cold door requires that each equipement cooled are powered off first;
- stop NFS daemons on '`nfs1`' because it is requested (this should be done before powering off '`nfs1`');
- power off '`nfs2`' because it is requested (but the NFS daemon will have to be stopped before);
- for each NFS client[10] a warning should be written before the actual stopping of used NFS server.

[9]Yes, it might seem strange here. This serves the purpose of our example.

[10]One might use the content of `/var/lib/nfs/rmtab` for an (inaccurate) list of NFS clients, the '`showmounts`' command or any other means.

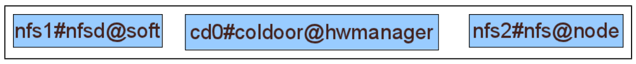


Figure 3.2: The initial dependency graph.

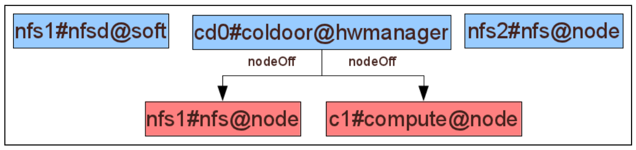


Figure 3.3: The dependendy graph after the call to the 'cd0#coldoor@hwmanager' depsfinder.

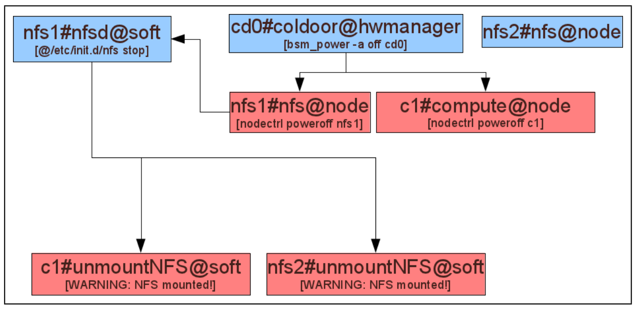


Figure 3.4: The dependency graph after the application of rule 'coldoorOff' on 'cd0#coldoor@hwmanager'.

With the table, the ruleset and the component list, the sequencer starts to expand the component list into an id set: 'nfs1#nfsd@soft, cd0#coldoor@hwmanager, nfs2#nfs@node'. Then the dependency graph is initialized: each id in the set has a related node in the graph as shown in figure 3.2.

Then the sequencer looks for a *potential root* using the rules graph (shown on figure 3.1). A match exists between rule 'coldoorOff' and 'cd0#coldooor@hwmanager'. Therefore, the sequencer starts applying rule 'coldoorOff' to 'cd0#coldooor@hwmanager'. The depsfinder of the rule is called. For each id returned by the depsfinder[11], the graph is updated: a node with an empty action list is made and an edge from the current id to the dependency is created. Each returned id is added to the id set.

Then, for each dependency, the sequencer checks if a match exists with one of the rules defined in the 'dependson' column, and for each match, the matching rule is applied recursively.

In our case, the cold door depsfinder returns every cooled component: 'nfs1#nfs@node' and 'c1#compute@node'. Therefore, the graph is updated as shown in figure 3.3. The 'coldoorOff' rule defines a single dependency in its 'dependson' column: 'nodeOff'. Both components match, the rule is applied. The application of the rule 'nodeOff' on 'c1#compute@node' leads to the execution of the depsfinder which does not return anything. Therefore, the application of the rule ends by adding the action 'nodectrl poweroff %name' to the related node in the dependency graph and by the removal of the related id from the id set.

This implies that a given rule is applied at most once on a given id.

The sequencer continues with the next dependency which is 'nfs1#nfs@node'. The application of the rule 'nodeOff' leads to the execution of the depsfinder which returns 'nfs1#nfsd@soft'. This node is already in the graph (it is in the initial id set). Therefore, the dependency graph is just updated with a new edge. This id matches the dependency rule specified 'nfsDown' and this last rule is applied on that id. The depsfinder on 'nfs1#nfsd@soft' returns all known clients which are 'c1#unmountNFS@soft' and 'nfs2#unmountNFS@soft'.

Finally both dependencies match the rule 'umountNFS' but its application does not lead to any new node in the dependency graph. However, the graph is updated so each node is mapped to its related action, recursively, up to the node the sequencer started with: 'cd0#coldoor@hwmanager' as show on figure 3.4.

At that stage, the id set contains only the last element from the originial component list:

[11]Note that it is not required that depsfinder returns ids with a predefined category as soon as a match occurs in the sequencer table. Predefined categories are used to ease the mapping between a given component and a type. In a large cluster (or data-center), it may not be easy to determine what is the real type of a given component name.

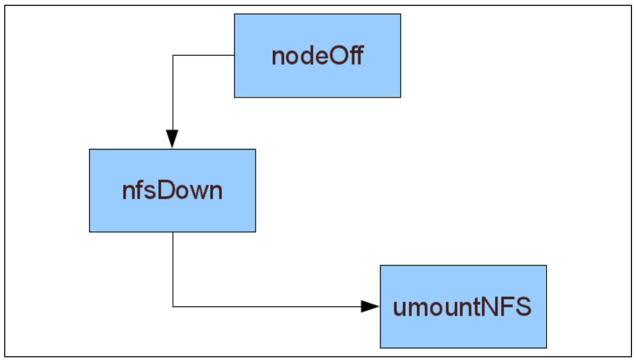


Figure 3.5: The rules graph with first level roots removed.

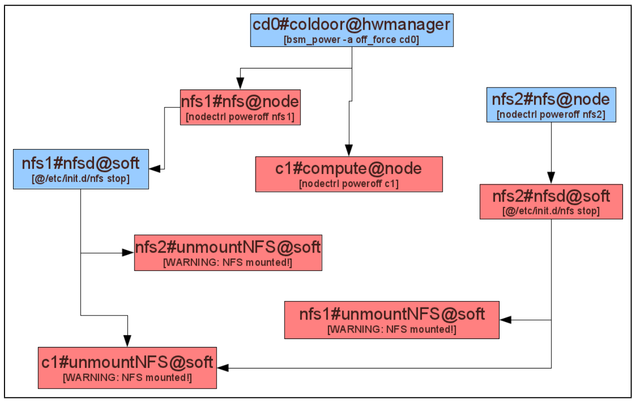


Figure 3.6: The final dependency graph.

`'nfs2#nfs@node'` (others were removed by preceding rule applications). Unfortunately that id does not match any root rule in the rules graph. Thus, the rules graph is (virtually) modified so roots are removed. This leaves us with the rules graph shown on figure 3.5.

From that graph, id `'nfs2#nfs@node'` matches root rule `'nodeOff'` which is therefore applied. The depsfinder returns `'nfs2#nfsd@soft'` which is new and therefore added in the dependency graph. The rule `'nfsDown'` is applied on that id (since a match occurs) giving us two dependencies `'c1#unmountNFS@soft'` and `'nfs1#unmountNFS@soft'`.

The algorithm ends after the mapping of those new ids with their related actions as shown in the final graph shown on figure 3.6.

Remember that a rule is never applied twice on a given id. Therefore, the action from rule `'unmountNFS'` which is `'echo WARNING: NFS mounted!'` on id `'c1#unmountNFS@soft'` is not added twice.

The sequencer displays this dependency graph in an XML format (using the open-source python-graph library available at `http://code.google.com/p/python-graph/`) on its standard output. This output can be given directly to the second stage of the sequencer. Note that contrary to the rules graph, the dependency graph should not contain a cycle (this will be detected by the next stage and refused as an input).

3.2 Instructions Sequence Maker (ISM)

The Instructions Sequence Maker (ISM) is the second stage of the sequencer. Its role is to transform a dependency graph into a set of instructions that can be given as an input to the third stage, the Instructions Sequence Executor (ISE).

A set of instructions is specified as an XML document, within an `<instructions>` XML tag. Three kind of instructions can be specified:

Action: defined by the `<action>` tag. It specifies the actual command that should be executed. Attributes are:

- `'id'`: Each action should be identified by a unique string. This attribute is mandatory. It is usually[12] of the form `'name#type@category!rule'`
- `'deps'`: a list of ids this action depends on (*explicit dependencies*). This attribute is optionnal. Default is the empty string.
- `'remote'`: the command should be executed using the current shell unless this attribute is set to `'true'`. In this case, an

[12]It is not required for the instructions sequence XML document to be created by the Instructions Sequence Maker. It may be created/modified by hand or by any other programs.

internal ssh connexion is made to execute the given command on each components described by the `'component_set'` attribute (see below). This attribute is optionnal. Default is `'false'`.

- '`component_set`': the set of components this action should be executed on in the following format: `'name[range]#type@category'`. This attribute is ignored by the ISE unless the remote attribute is set to `'true'`. This attribute is optionnal. Default is `'localhost#type@cat'`.

Sequence: defined by the `<seq>` tag. It specifies a set of instructions (hence, one of Action, Sequence or Parallel) that must be executed in the given order. This defines *implicit dependencies* between instructions as opposed to *explicit dependencies* defined by the `'deps'` attribute of an Action.

Parallel: defined by the `<par>` tag. It specifies a set of instructions (hence one of Action, Sequence or Parallel) that can be executed in any order. This explicitly defines that there is no dependency between each instruction. The ISE is free to execute them in parallel. Note that the ISE may or may not execute those instructions in parallel. This is not a requirement for the successful completion of a parallel instruction.

Transforming a dependency graph into an instructions sequence is straightforward if performance is not the main goal. A simple topological sort [6] on the input dependency graph returns a sequence of actions where constraints are respected.

For example, on our example where the final dependency graph computed by the DGM is given on figure 3.6, a topological sort[13] gives the sequence shown on sample 1.

This sequence is valid, but not efficient: it requires 9 sequential steps. This transformation algorithm is called `'seq'` in the sequencer and it can be selected. Three other algorithms are provided within the sequencer:

- `'par'`: this algorithm inserts each node in the dependency graph using a single parallel (`<par>` XML tag) instruction and explicit dependencies (`'deps'` attribute of the `<action>` XML tag). Such an algorithm is optimal in terms of performance, but it produces an instructions sequence file that is difficult to read by a human because of all those explicit dependencies.

- `'mixed'`: this algorithm inserts each leaf nodes in the dependency graph using a parallel instruction, then remove those leaf nodes from the dependency graph and starts again. Every such parallel instructions are included in a global sequence one (`<seq>` XML tag). This algorithm tends to execute set of actions by steps: all leaf nodes are executed in parallel. Once they have terminated, they are removed from the graph, and another batch of leaf nodes are executed in parallel up to the end.

- `'optimal'`: this algorithm produces an instructions sequence that is as efficient as the `'par'` algorithm but much more readable. It uses implicit dependencies as much as possible[14] using sequence instructions. This algorithm is selected by default.

Describing in details those algorithms with their advantages and constraints is beyond the scope of this paper.

3.3 Instructions Sequence Executor (ISE)

The Instructions Sequence Executor (ISE) is the last stage of the sequencer. It takes in input an instructions sequence as computed by the ISM or created/edited by hand or by any other means. It then runs the instructions specified taking into account:

[13]For a given directed acyclic graph, several valid topological sort outputs can be found.

[14]Our XML instructions sequence format can only express trees if implicit dependencies are used exclusively.

Sample 1 Result of the topological sort on the dependency graph given figure 3.6.

```
<instructions>
    <seq>
        <action id="c1#unmountNFS@soft!unmountNFS">echo WARNING: NFS mounted!</action>
        <action id="nfs1#unmountNFS@soft!unmountNFS">echo WARNING: NFS mounted!</action>
        <action id="nfs2#unmountNFS@soft!unmountNFS">echo WARNING: NFS mounted!</action>
        <action id="nfs1#nfsd@node!nfsDown" remote="true" component_set="nfs1#nfsd@node">
         /etc/init.d/nfsd stop
        </action>
        <action id="nfs1#nfs@node!nodeOff">nodectrl poweroff nfs1</action>
        <action id="nfs2#nfsd@soft!nfsDown" remote="true" component_set="nfs2#nfs@node">
         /etc/init.d/nfsd stop
        </action>
        <action id="c1#compute@node">nodectrl poweroff c1</action>
        <action id="nfs2#nfs@node!nodeOff">nodectrl poweroff nfs2</action>
        <action id="cd0#coldoor@hwmanager!coldoorOff">bsmpower -a off cd0</action>
    </seq>
</instructions>
```

- parallelism: actions that do not have dependencies between them might be executed in parallel. There is a customizable maximum limit on the number of actions that can be executed in parallel by the ISE. This helps limiting the load increase of the system due to a vast number of forks in a large cluster.
- dependencies: an action is not executed unless all its dependencies (explicit and implicit) have completed successfully. An executed action is considered successful in two cases:
  - its returned code is 0 (alias OK);
  - its returned code is 75 (alias WARNING also known as EX_TEMPFAIL in `sysexits.h`) and the option `'--Force'` has been given to the ISE.

The implementation of the ISE uses the ClusterShell [7] python library as the backend execution engine. Describing the implementation of the ISE is beyond the scope of this article.

4 Scalability Issues

Using the sequencer on a large cluster such as the Tera-100 can lead to several issues related to scalability.

4.1 Complexity

Several complexity in space and time can be identified for:

1. the production of the dependency graph produced by the DGM;
2. the production of the actions graph produced by the ISM;
3. the execution of the actions graph by the ISE.

This last complexity was our first concern due to our customer requirements. If theorical complexity has not (yet) been formally determined, the execution time of the sequencer for the production of the dependency graph of the Tera-100 is:

- 13 minutes 40 seconds for the start ruleset with 9216 nodes and 8941 edges in the dependency graph;
- 2 minutes 1 second for the stop ruleset with 9222 nodes and 13304 edges in the dependency graph.

The time taken by the ISM for the production of the actions graph from the previously computed dependency graph using the 'optimal' algorithm is:

- 4.998 seconds for the start with 4604 nodes and 8742 edges in the actions graph;
- 6.343 seconds for the stop with 4606 nodes and 9054 edges in the actions graph.

Finally, the time taken by the ISE to execute these actions graph is:

- 4 minutes 27 seconds for the start with:
  - 99.7% of actions executed (successfully or not);
  - 6.6% of actions that ends on error for various reasons;
  - 0.3% of actions not executed because some of their dependencies ends on error or was not executed.
- 9 minutes 23 seconds for the stop ruleset with:
  - 96.7% of actions executed (successfully or not);
  - 15.3% of actions that ends on error for various reasons;
  - 3.3% of actions not executed because some of their dependencies ends on error or was not executed

Explaining differences between those metrics is beyond the scope of this paper. However, from such results, the sequencer can be considered has quite scalable.

4.2 Mantainability

The maintenance of the various graph used by the sequencer:

- rules graph;
- the DGM produced dependency graph;
- the ISM produced actions graph XML file;

is also an issue on large systems. Identifying wrong dependencies in a flat file can be hard, especially with large graph represented with several thousands of lines.

The sequencer can exports those graph in the DOT format. It therefore delegates to specific graph tools, the identification of non trivial problems for maintenance purposes. For instance, rules graph are usually small and the standard `'dot'` command that comes within the graphviz [8] open-source standard product can be used for their vizualisation. This is fast and easy. For other much larger graph, however, specialized tools such as Tulip [9] might be used instead.

4.3 Usability

In the context of large systems, giving correct inputs to a tool, and getting back a usable output can be a big challenge in itself. In the case of the sequencer, inputs are the sequencer table and the components list.

For the maintenance of the table, the sequencer provides a management tool that helps adding, removing, updating, copying and even checksuming rules. For the components list, the sequencer uses what is called a `guesser` that given a simple component name fetches its type and category. This allows the end user to specify only the name of a component.

Apart from the output produced by the first two stages that have already been discussed in the previous section on maintanability, the last output of great interest for the end-user, is the ISE output. To increase further the usability of the sequencer, several features are provided:

Prefix Notation: each executed action output is prefixed by its id. When the ISE executes an action graph produced by previous stages, those ids contain various informations such as the type, the category, and the rulename this action comes from. This helps identifying which action produced which output (the bare mininum). Moreover, such an output can be passed to various filters such as grep or even gathering commands such as `'clubak'` from ClusterShell [7] or `'dshbak'` from pdsh [10]. In the case of 'clubak' command, the separator can be given as an option. As a side effect, this allows the end-user to group similar output by node, type or category.

Reporting: The ISE can produce various reports:

- 'model': each action with their dependencies (implicit and explicit) are shown; this is used to determine what the ISE will do before the actual execution (using a '--noexec' option);
- 'exec': each executed action is displayed along with various timing informations and the returned code;
- 'error': each executed action that exited with an error code is displayed along with their reversed dependencies (their parent in the dependency graph); this is used to know which action has not been executed because of a given error;
- 'unexec': each non executed action is displayed along with its missing dependencies — a dependency that exited with an error code and that prevented the action from being executed; this is used to know why a given action has not been executed.

5 Results

The sequencer has been designed for two main purposes:

1. Emergency Power Off: this is the reason of the three different independent stages;
2. Common management of resources in clusters (and data-centers): this is the main reason for the chaining mechanism.

Our first experiment with our tool shows that it is quite efficient. Our main target was the powering on/off of the whole Tera-100 system which leads to the execution of more than 4500 actions in less than 5 minutes for the start and in less than 10 minutes for the stop.

6 Related Works

Dependency graph makers exist in various products:

- Make [11], SCons [12], Ant [13] for example are used for the building of softwares; they focus on files rather that cluster components and are therefore not easily adaptable to our problem.
- Old System V init [14], BSD init [15], Gentoo init[16] and their successors Solaris SMF [17], Apple launchd [18], Ubuntu upstart [19] and Fedora systemd [20] are used during the boot of a system and for managing daemons. To our knowledge none of those products can be given a components list as an input so actions are executed based on it.

Solutions for starting/stopping a whole cluster are most of the time manual, described in a step-by-step chapter of the product documentation and hard wired. This is the case for example with the Sun/Oracle solution [21] (command 'cluster shutdown'). It is not clear whether all components in the cluster are taken into account (switches, cooling components, softwares, ...) and whether new components can be added to the shutdown process. IBM uses the open-source xcat [22] project and its 'rpower' command which does not provide dependencies between cluster components.

From a high level perspective, the sequencer, can be seen as a command dispatching framework similar to Fabric [23], Func [24] and Capistrano [25] for example. But the ability to deal with dependencies lacks in these products making them unsuitable for our initial problem.

The sequencer can also be seen as a workflow management system where the pair DGM/ISM acts as a workflow generator, and the ISE acts as a workflow executor. However, the sequencer has not been designed for human interactive actions. It does not deal for example with user access rights or tasks list for example. It is therefore much lighter than common user oriented workflow management systems such as YAWL [26], Bonita [27], Intalio|BPMS [28], jBPM [29] or Activiti [30] among others.

We finally found a single real product for which a comparison has some meaning: ControlTier [31].

ControlTier shares with the sequencer various features such as possible parallel execution of independent actions and failure handling. However, the main difference is in the way workflows are produced: they are dynamically computed in the sequencer case through dependency rules (depsfinder scripts) and the component input list whereas they are hard wired through configuration files in the case of ControlTier.

To our knowledge, our solution is the first one to address directly the problem of starting/stopping a whole cluster or a part of it, taking dependencies into considerations, still remaining integrated, efficient, customizable and robust in the case of failure. We suppose the main reason is that clusters are not designed to get started/stopped entirely. Long uptime is an objective! However automated tools ease the management of clusters, making start/stop procedure faster, reproducible, and reducing human errors to a minimum.

7 Conclusion and Future Works

The sequencer solution presented in this article is the first of its kind to our knowledge. It has been designed with EPO in mind. This is the reason for its 3 independent stages and for the incremental mode of execution. Still the sequencer provides the chaining feature making its use pertinent for small clusters or small part of a big one.

The sequencer fulfills our initial objectives:

- Predictive: the incremental mode allows a computed instructions sequence to be verified, modified and recorded before being run.
- Easy: executing a recorded instructions sequence requires a single command: `'clmsequencer < instructions.sequence'`
- Fast: the sequencer can execute independent instructions in parallel with a customizable upper limit.
- Smart: the order in which instructions are executed comes from a dependency graph computed from customizable dependency rules and a given cluster components list.
- Robust: failures are taken into account by the sequencer component by component.

Our solution is highly flexible in that most of its inner working is configurable such as:

- the dependency rules,
- the dependency fetching scripts,
- the action to be taken on each component,
- the dependency graph,
- the final instruction sets.

The sequencer has been validated on the whole Tera-100 system. A shutdown can be done in less than 10 minutes, and a power on takes less than 5 minutes (more than 4500 actions for both rulesets). The sequencer framework will be released under an open-source license soon. Several enhancements are planned for the end of this year including:

- smarter failure handling;
- live reporting/monitoring;
- performance improvement of dependency graph generation through caching;
- post-mortem reporting;
- replaying.

8 Aknowledgment

The author would like to thank Matthieu Pérotin for his helpful review, Marc Girard for his support during the development of the solution presented in this paper and the reviewers for their very clear comments and remarks.

References

[1] W. Turner, J. Seader, and K. Brill, "Industry standard tier classifications define site infrastructure performance," tech. rep., Uptime Institute, 2005. 1

[2] P. Grun, "Introduction to infiniband for end users," tech. rep., InfiniBand Trade Association, April 2010.
`http://members.infinibandta.org/kwspub/Intro_to_IB_for_End_Users.pdf`. 1

[3] AMD, "Magic packet technology." White Paper, November 1995. Publication# 20213, Rev: A Amendment/0
`http://support.amd.com/us/Embedded_TechDocs/20213.pdf`. 2

[4] N. D. Intel, Hewlett-Packard, "Ipmi v2.0 rev. 1.0 specification markup for ipmi v2.0/v1.5 errata revision 4," 2009.
`http://download.intel.com/design/servers/ipmi/IPMI2_0E4_Markup_061209.pdf`. 2

[5] S. R. International, "Snmp rfcs."
`http://www.snmp.com/protocol/snmp_rfcs.shtml`. 2

[6] A. B. Kahn, "Topological sorting of large networks," *Commun. ACM*, vol. 5, pp. 558–562, November 1962. 11

[7] "Clustershell opensource project," 2011.
`http://sourceforge.net/apps/trac/clustershell`. 12, 13

[8] AT&T Labs Research and Contributors, "Graphviz." Web page, June 2011.
`http://graphviz.org/`. 13

[9] D. Auber, "Tulip : A huge graph visualisation framework," in *Graph Drawing Softwares* (P. Mutzel and M. Jünger, eds.), Mathematics and Visualization, pp. 105–126, Springer-Verlag, 2003. 13

[10] J. Garlick, "pdsh: Parallel distributed shell."
`http://sourceforge.net/projects/pdsh/`. 13

[11] Free Software Foundation, "GNU Make." Web page, July 2004.
`http://www.gnu.org/software/make/`. 14

[12] The SCons Foundation, "SCons." Web page, June 2011.
`http://www.scons.org/`. 14

[13] The Apache Software Foundation, "The Apache Ant Project." Web page, July 2004.
`http://ant.apache.org/`. 14

[14] Novell, Inc (now SCO), *System V Interface Definition, Fourth Edition, Volume 2*, June 1995.
`http://www.sco.com/developers/devspecs/vol2.pdf`. 14

[15] FreeBSD, *FreeBSD System Manager's Manual, init(8)*, Sept. 2005.
`http://www.freebsd.org/cgi/man.cgi?query=init&sektion=8`. 14

[16] "Gentoo Initscripts," Mar. 2011.
`http://www.gentoo.org/doc/en/handbook/handbook-x86.xml?part=2&chap=4`. 14

[17] R. Romack, "Service managemen facility (smf) in the solaris 10 operating system." Sun BluePrints OnLine, Feb. 2006.
`http://www.linux.com/archive/feature/125977`. 14

[18] J. Wisenbaker, "launchd in depth." AFP548, July 2005.
`http://www.afp548.com/article.php?story=20050620071558293`. 14

[19] M. Sobell, "Ubuntu's upstart event-based init daemon," Feb. 2008.
`http://www.linux.com/archive/feature/125977`. 14

[20] L. Poettering, "Rethinking pid 1," Apr. 2010.
`http://0pointer.de/blog/projects/systemd.html`. 14

[21] Oracle, *Oracle Solaris Cluster System Administration Guide.* http://download.oracle.com/docs/cd/E18728_01/html/821-1257/ghfwr.html. 14

[22] L. Octavian, A. Brindeyev, D. E. Quintero, V. Sermakkani, R. Simon, and T. Struble, "xCAT 2 Guide for the CSM System Administrator." IBM Red Paper, 2008. http://www.redbooks.ibm.com/redpapers/pdfs/redp4437.pdf. 14

[23] Jeff Forcier, "Fabric." Web page, June 2011. http://fabfile.org/. 14

[24] MichaelDeHaan, AdrianLikins, and SethVidal, "Func: Fedora Unified Network Controller." Web page, June 2011. https://fedorahosted.org/func/. 14

[25] Jamis Buck, "Capistrano." Web page, June 2011. http://github.com/capistrano/capistrano/wiki/. 14

[26] M. Adams, S. Clemens, M. L. Rosa, and A. H. M. ter Hofstede, "Yawl: Power through patterns," in *BPM (Demos)*, 2009. 14

[27] M. V. Faura, "Bonita." http://www.bonitasoft.com/. 14

[28] I. Ghalimi, "Intalio|bpms." http://www.intalio.com/bpms. 14

[29] T. Baeyens, "Jbpm." http://www.jboss.org/jbpm. 14

[30] T. Baeyens, "Activiti." http://www.activity.org/. 14

[31] Alex Honor, "Control Tier." Web page, June 2011. http://controltier.org/. 14