Planning Configuration Relocation on the BonFIRE Infrastructure

Citation for published version:
Herry, H & Anderson, P 2013, Planning Configuration Relocation on the BonFIRE Infrastructure. in *Cloud Computing Technology and Science (CloudCom), 2013 IEEE 5th International Conference on* . vol. 2, Institute of Electrical and Electronics Engineers (IEEE), pp. 164-169.
https://doi.org/10.1109/CloudCom.2013.120

Digital Object Identifier (DOI):
10.1109/CloudCom.2013.120

Link:
Link to publication record in Edinburgh Research Explorer

Document Version:
Peer reviewed version

Published In:
Cloud Computing Technology and Science (CloudCom), 2013 IEEE 5th International Conference on

Planning Configuration Relocation on the BonFIRE Infrastructure

Herry Herry* and Paul Anderson†
School of Informatics, University of Edinburgh
Edinburgh, UK
*h.herry@sms.ed.ac.uk, †dcspaul@ed.ac.uk

***Abstract*—The Nuri configuration tool uses automated planning techniques to reconfigure computing infrastructures from one declarative state to another, while maintaining any required constraints at all stages of the generated workflow. This paper presents the results of our experiments using Nuri to relocate complete 3-tier web applications between two BonFIRE sites. We describe the architecture of the tool and present the results of the experiment which demonstrate that Nuri is capable of planning, and executing this relocation in a reasonable time, with guaranteed constraints on the system throughout the reconfiguration.**

I. Introduction

Cloud computing is now an accepted option for deploying application services – services can be easily created, deleted and relocated to cater for changes in requirements, demand and economics. Within a single installation, virtual machine migration makes this a straightforward process – machine images can be transferred between physical machines without changes to the configuration, and without even interrupting a running process. However, to fully exploit the potential of the cloud, we need to be able to move services between different infrastructures, reliably and without disrupting the operation of the overall service during the transfer process. This is much more complex; live VM migration is not normally possible between different sites, so new copies of the services must be instantiated on the new infrastructure, and any clients of the service must be reconfigured to reference the new service, before we can tear down the old service. This is likely to involve some reconfiguration of the services themselves (as a minimum, the IP addresses will change). Any significant distributed application will require a more complex reconfiguration, and a carefully crafted workflow to preserve a running service during all stages of the transfer. We will use the term *relocation* to refer to this process, since *migration* is commonly used for the simpler case where virtual images are moved without reconfiguration.

Armstrong [1] offers a solution to this problem using a *recontextualization* approach with dynamic virtual devices. However this only works where an identical VM image can be used on both sites. If the virtualisation technology is different (for example), or if other aspects of the relocation require reconfiguration above the VM level, then this is not appropriate. System configuration tools such as Puppet [2] allow us to configure a system according to a broadly declarative specification. If the initial system is configured using such as tool, then the same specification can be used to configure the destination system in exactly the same way. These tools are designed to support the portability of application configurations, even in the face of differences at other layers (for example, the operating system). However, they make no guarantees about the order of the changes that will be applied to move the system from one configuration to another; to maintain a consistent application during the transfer process requires a manually created workflow. The prototype Nuri configuration tool [3], [4] is a declarative configuration tool which uses automated planning techniques to generate workflows between any two arbitrary states, while maintaining any required (declarative) constraints at all stages of the workflow.

The BonFIRE infrastructure provides an ideal testbed on which to explore the relocation process; it is a federated cloud infrastructure with heterogeneous platforms – for example, HPLabs site, INRIA and EPCC, are all maintained independently by different teams, using different technologies (Cells and OpenNebula). There is no capability for transferring virtual machines between sites, but there is a common broker that can be used to manage virtual machines on multiple the sites.

This paper describes an experiment which show that the Nuri configuration tool is capable of automatically planning and orchestrating the relocation of multi-tier applications in this environment. Moreover, the planning and execution happen within a reasonable amount of time, and the global constraints are maintained automatically at every stage of the relocation.

We start by describing the SFP language which is used to model the configuration specification of the system. We then illustrate the relocation process with a simple example to transfer a cloud application service from one BonFIRE site to another. This is followed by the details of the planning and implementation stages, and a brief description of the Nuri architecture. Finally, we present the experimental results using an extended example.

II. Modelling

To model the system, we use the object-oriented configuration language SFP [4]. Inspired by SmartFrog [5], SFP adopts a prototype-based language approach which allows an object to be used as a template for another object. Each resource

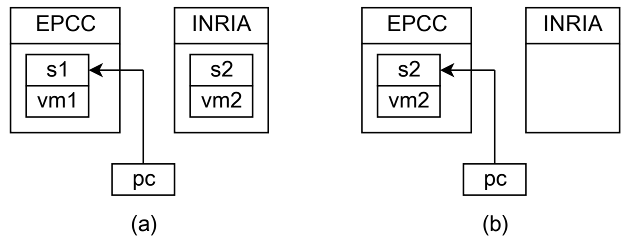


Fig. 1: System configuration of the example system in the current (a) and desired (b) state.

is modelled as an object whose state is represented by a collection of attribute-value pairs, and the union of all these object states represents the state of the system.

A unique feature (among declarative configuration languages) of SFP is that it allows us to define a procedure which represents the capability of an object to change its (or another object's) state by modifying particular attribute values, whenever the current state satisfies a particular condition. Each procedure declaration consists of a set of parameters, a set of conditions, a set of effects, and a numeric cost value. This information is used by the automated planner to determine the actions available to move from one configuration state to another.

SFP also allows us to define the desired state as a "loose" constraint. This allows the administrator to work at a higher level by defining a whole set of possible desired states. It also affords the planner more flexibility in searching for the best solution, and it allows us to define "global constraints" which must be satisfied at every stage during a sequence of reconfiguration steps.

III. Example

Assume we have a production service (s_1) which is running on the EPCC BonFIRE site, and is currently being used by a client (*pc*). Assume that we also have a new version of the service (s_2) which is being tested on the INRIA site. After ensuring that the new version has passed the tests, we would like to replace the production service with the new one, while preserving the global constraint that the client is always using a running service. Figure 1a and 1b illustrate the current and desired state of the system respectively.

We use SFP configuration language to model the configuration state of the system. Figure 2 shows a declarative specification of the current state. In the desired state, three attributes are different:

1) The value of attribute `created` in line 7 is set to `false` since the vm_1 and s_1 are no longer required;
2) The value of attribute `in_cloud` in line 15 is set to `proxy.epcc` since we require vm_2 and service s_2 at INRIA instead of EPCC;
3) The value of attribute `refer` in line 20 is set to `vm2.s2` since we require pc to use s_2 (the new version).

The final specification after the changes is shown in figure 3. Note that the global constraint remains the same.

```
 1  include "schemata.sfp"
 2  proxy isa Machine {
 3    epcc isa Cloud { location = "uk-epcc"; }
 4    inria isa Cloud { location = "fr-inria"; }
 5  }
 6  vm1 isa VM {
 7    created = true
 8    in_cloud = proxy.epcc
 9    s1 isa Service { installed = true
10                     running = true
11                     version = 1; }
12  }
13  vm2 isa VM {
14    created = true
15    in_cloud = proxy.inria
16    s2 isa Service { installed = true
17                     running = true
18                     version = 2; }
19  }
20  pc isa Client { refer = vm1.s1; }
21  global { pc.refer.running = true; }
```

Fig. 2: The (declarative) specification of the current state of the example system in SFP, which uses the schema defined in figure 4.

```
 1  include "schemata.sfp"
 2  proxy isa Machine {
 3    epcc isa Cloud { location = "uk-epcc"; }
 4    inria isa Cloud { location = "fr-inria"; }
 5  }
 6  vm1 isa VM { created = false
 7    s1 isa Service; }
 8  vm2 isa VM {
 9    created = true
10    in_cloud = proxy.epcc
11    s2 isa Service { installed = true
12                     running = true
13                     version = 2; }
14  }
15  pc isa Client { refer = vm2.s2; }
16  global { pc.refer.running = true; }
```

Fig. 3: The (declarative) specification of the desired state of the example system in SFP, which uses the schema defined in figure 4.

Before deploying the changes, our prototype will submit these specifications[1] to the planner which automatically generates the following sequence of actions:

1) `proxy.inria.delete(vm=vm2)`
2) `proxy.epcc.create(vm=vm2)`
3) `vm2.s2.install`
4) `vm2.s2.start`
5) `pc.redirect(s=vm2.s2)`
6) `proxy.epcc.delete(vm=vm1)`

Clearly, the execution of the above plan will achieve the desired state as defined in figure 3: service s_2 will be relocated from INRIA to EPCC site, all resources in vm_1 will be deleted (including service s_1), and pc will refer to service s_2. Note

[1]In practice, the user will only need to specify the desired state (figure 3). The current state (figure 2) will be automatically generated on the master node by aggregating the state of all client nodes.

```
schema Machine
schema Cloud {
  location = ""
  sub create_vm (vm : VM) {
    condition { vm.created = false; }
    effect { vm.created = true
             vm.in_cloud = this; }
  }
  sub delete_vm (vm : VM) { ... }
}
schema VM extends Machine {
  created = false; in_cloud isref Cloud
}
schema Service {
  installed = false
  running = false
  sub install { ... }
  sub uninstall { ... }
  sub start { ... }
  sub stop { ... }
}
schema Client { refer isref Service
  sub redirect (s : Service) { ... }
}
```

Fig. 4: The schema (schemata.sfp). For brevity, details of some procedures are omitted. A complete version of the schema is available at:
http://homepages.inf.ed.ac.uk/s0978621/bonfire/schemata.sfp.

that the intermediate states during configuration do not violate any global constraint – i.e. *pc* will always refer to a running service.

IV. Planning

Deployment of a specification involves a sequence of configuration steps which execute procedures to change the attribute value towards the desired state, preserving the global constraints at every step. To achieve this, we employ a technique described in [4] which compiles the current and desired state specifications, together with the global constraints, into a classical planning problem (using a Finite-Domain Representation (FDR) [6]). We then use an automated planner to generate the workflow.

The complexity of finding a solution plan is generally classified as a PSPACE-complete problem [7], which can be very hard to solve. Fortunately, there are several heuristic search techniques which can be used to significantly reduce the computation time – for example a combination of an admissible heuristic, with an A^* search algorithm to generate an optimal plan, and a combination of an inadmissible heuristic with a greedy algorithm to generate a satisfying plan[2]. Although the first technique has the advantage of producing an optimal plan, in most cases it requires much longer to find a solution. The second technique will find a solution in a shorter time, but this is not guaranteed to be optimal.

In the system configuration domain, the planning time is affected by several factors; the first is the total number of variables in the configuration specification that need to be considered – this is related to the number of components since the number of variables is the sum of components' attributes; the second is the complexity of the goal state as well as the preconditions and postconditions of the operators – this is related to the coupling degree of the planning problem which determines the difficulty of the solution.

Our prototype combines these two techniques into a 2-stage search:

1 We focus on solving the planning problem as quickly as possible using the first combination – i.e. an inadmissible heuristic such as h^{FF} [9] or h^{cea} [10] with a greedy search algorithm.
2 We optimise the solution plan found on the first stage by solving the same planning task, but only use the actions selected by the plan. We then use the first combination – i.e. an admissible heuristic, such as $h^{\text{LM-cut}}$ [11], with A^* search algorithm.

With this 2-stage search, we can achieve a suboptimal plan within a reasonable time. In practice, we have found that this usually produces a globally optimal plan for most test-cases.

V. Implementation

Figure 5 shows the architecture of the Nuri configuration tool. A master node controls a set of client nodes. The master node is responsible for generating and orchestrating the execution of the plan to achieve the desired state. Each client has an agent which is responsible for instantiating software components, generating the current state of the node, and invoking the actions. The communication between the nodes, either master/client or between clients, uses an HTTP/JSON protocol[3].

Currently, the whole of the configuration tool is implemented in Ruby, apart from the planner which is implemented in C++. We use a modified version of FastDownward planner [12]. Every computing resource is managed by an instance of a Nuri module which consists of:

1) An SFP schema, which is a declarative description of an abstract resource;
2) A Ruby class, which has implementation code for querying the current state of the component and implementing the SFP procedures.

There is a clear separation between the SFP description and the Ruby implementation. The mapping from SFP to Ruby and vice versa is done by the agent using the SFP-Ruby library. This clear separation allows us to have different implementations, for example one in Ruby and another in Java, communicating transparently via the SFP configuration language.

The Nuri tool[4] consists of several parts whose interactions can be summarised as follows[5]:

[2]A satisfying plan is a non-optimal solution. More detail on admissible and inadmissible heuristics can be found in [8].

[3]Or HTTPS.

[4]Source code: https://github.com/herry13/nuri

[5]Numbers correspond to the numbers in figure 5.

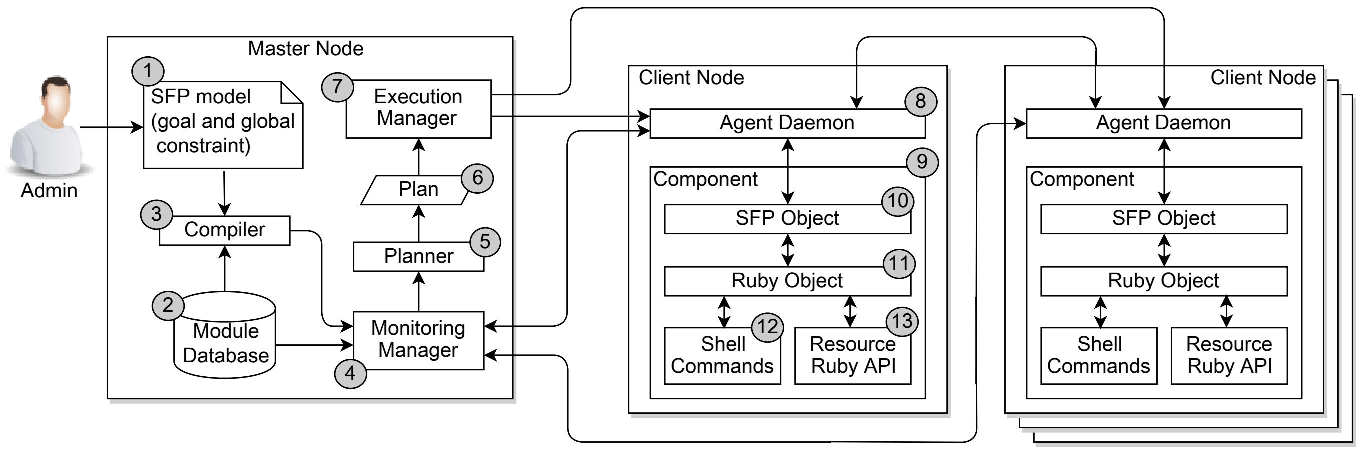


Fig. 5: The architecture of Nuri configuration tool.

1) An administrator defines and submits a declarative specification (1) of the desired state and the global constraints to the master (in SFP).
2) The SFP compiler (3) compiles this specification, together with any necessary files (for example schema files from a module database (2)), to resolve any references to a schema or object template. The compilation result is then sent to the monitoring manager (4).
3) The monitoring manager (4) will send the relevant part of the compiled specification to each target agent (8). Based on this specification, the agent creates a set of components (9), each of which is an instance of a Nuri module. Each component manages a particular resource using the Ruby implementation code (11). Since it is possible that an agent may not have an implementation of particular module, the monitoring manager is capable of sending implementations from the module database (2) to the agent.
4) After all the components have been created, the agent generates the local current state by invoking a Ruby method *update_state* on each component, and sends the state of all components to the monitoring agent. By combining the current state of all agents with the desired state and the global constraints, the monitoring agent then generates a configuration task which is sent to the planner (5) for solving.
5) The planner compiles the configuration task into a classical planning problem, and automatically searches for a plan (6). If a solution is found, the plan is submitted to the execution manager (7) for provisioning.
6) The execution manager (7) orchestrates the execution of the plan by scheduling execution on the actions based on the ordering constraints defined in the plan. Each request contains an action description (i.e. the reference of the target component and its method that should be executed) and the value of each parameter. Whenever an agent receives an execution request, it searches the target component and invokes its method by passing the parameters' value. The execution result will be verified to detect any failure, in which case the execution manager will automatically stop the plan execution, and restart the process (i.e. back to step #4), in order to generate an alternative plan.

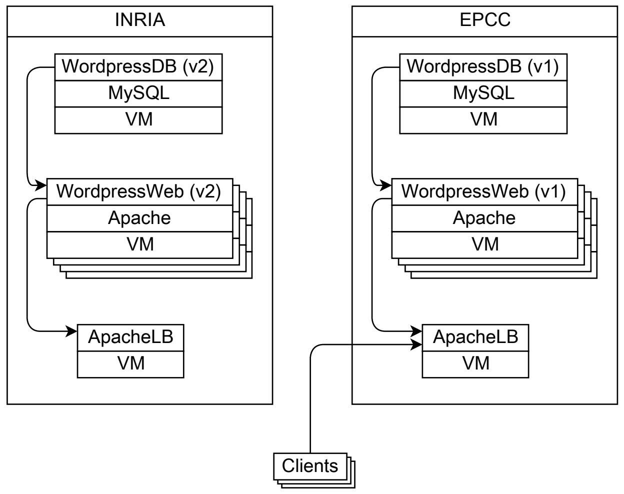


Fig. 6: The current state of the two systems as described in section VI.

VI. Extended Example

In order to evaluate Nuri on the BonFIRE infrastructure, we used an example involving a typical 3-tier web application consisting of a load balancer, a set of Wordpress application servers, and a database service.

We evaluated a scenario where there are two systems; the first system has version number 1, running on the EPCC BonFIRE site, and being used by a number of clients. The second system has version number 2, is running on the INRIA BonFIRE site, and being tested by the engineers. This current state is illustrated in figure 6. Since the second system has passed all tests, we would like to relocate this version to the

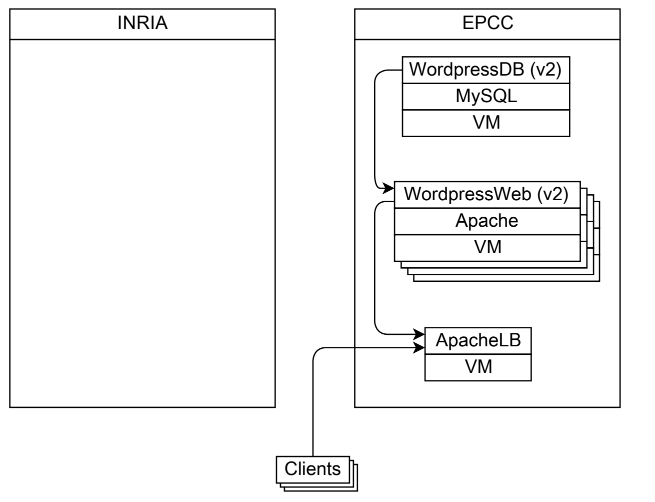
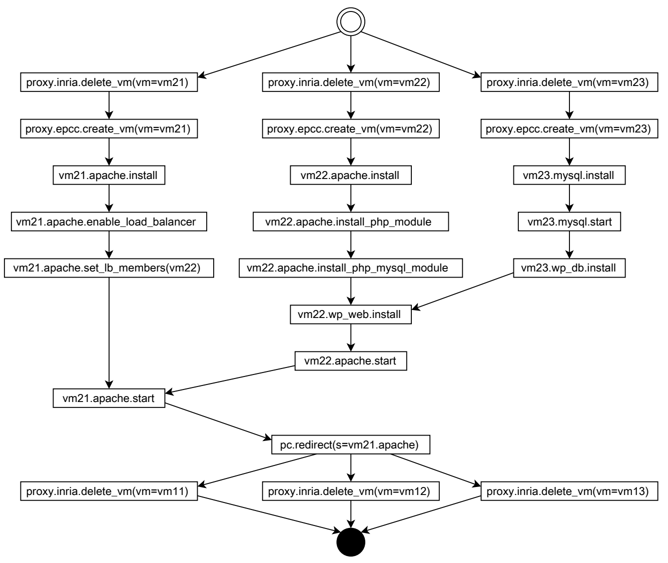


Fig. 7: The desired state of the two systems as described in section VI.



proxy.inria.delete_vm(vm=vm21)
proxy.inria.delete_vm(vm=vm22)
proxy.inria.delete_vm(vm=vm23)
proxy.epcc.create_vm(vm=vm21)
proxy.epcc.create_vm(vm=vm22)
proxy.epcc.create_vm(vm=vm23)
vm21.apache.install
vm22.apache.install
vm23.mysql.install
vm21.apache.enable_load_balancer
vm22.apache.install_php_module
vm23.mysql.start
vm21.apache.set_lb_members(vm22)
vm22.apache.install_php_mysql_module
vm23.wp_db.install
vm22.wp_web.install
vm22.apache.start
vm21.apache.start
pc.redirect(s=vm21.apache)
proxy.inria.delete_vm(vm=vm11)
proxy.inria.delete_vm(vm=vm12)
proxy.inria.delete_vm(vm=vm13)

Fig. 8: The generated plan with one application layer.

EPCC site to replace the first system. Figure 7 illustrates this desired state.

Although the software is deployed independently on different VMs, the relocation process must satisfy some global constraints due to service dependencies. In addition, we do not want to disturb any usage of the service by the clients during the process. Thus, the following global constraints should be maintained:

- The web services depend on the database service: whenever the web services are running then the database service must be running as well;
- The load balancer depends on the web services: whenever the load balancer is running then the web service must be running as well;
- All clients should always refer to a running server.

To implement this relocation, we defined the desired state and the global constraints in SFP. Figure 10 shows an example of the configuration specification of the desired state with one application layer. By submitting this specification to the master node, we generated the plan shown in figure 8. The execution of this plan relocated the first system from INRIA to EPCC, configured the client to use this system, and finally deleted the first system from EPCC site. The plan execution did not violate any global constraints – the services were started in the correct order, and the client was redirected to use another running service before the old one was deleted.

In the experiments, we used a virtual machine with 2 CPUs and 2 GB RAM in EPCC as the Nuri master. Each BonFIRE site is managed by an instance of a proxy module that runs on a Nuri agent. This proxy module uses a *Restfully* Ruby library to connect to the BonFIRE broker to create or delete VMs on a particular site. For the managed system, we used *small* (1 CPU, 1 GB RAM) instance VMs. For the software stack, we used unmodified Debian Squeeze 10G v5, Apache Web Server, MySQL Database Server, and the Wordpress Content Management application, each managed by a Nuri module[6].

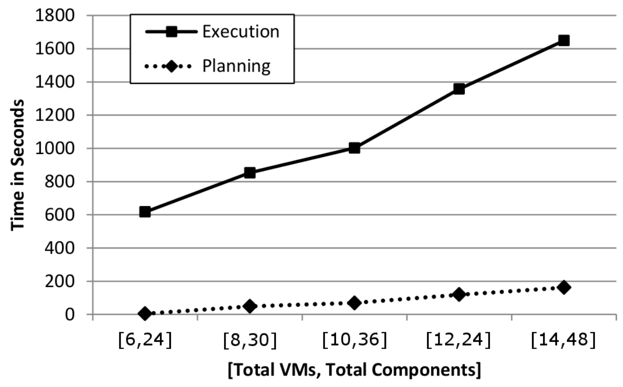


Fig. 9: Planning and execution time for ”configuration reallocation“ between two BonFIRE sites with different number of application layer. The axis is the total number of VMs and components of the two systems.

All VMs were connected to BonFIRE WAN network.

We ran several experiments using the same scenario but with different number of application layers to measure the effect of the size of the system on the planning and execution times. We ran each experiment five times and then took the average. Figure 9 illustrates the comparison of planning and execution times between various numbers of application layers. These results show that the Nuri planner could generate the plan for every case in a reasonable time – i.e. between 4-162 seconds. However, we believe that it is possible to improve these times by using better heuristic and search techniques and this is currently under investigation. On the other hand, the execution time is near optimal since the Nuri execution engine implements a partial-order execution algorithm – i.e. operations are executed in parallel whenever possible. But

[6]There is a Nuri module which manages the client configuration and is capable of redirecting service references in a similar way to a DNS lookup.

```
...
proxy isa Machine {
  sfpAddress = "172.18.240.38"
  // proxy component for EPCC site
  epcc isa Bonfire { location = "uk-epcc"
    experiment = "autocloud"; }
  // proxy component for INRIA site
  inria isa Bonfire { location = "fr-inria"
    experiment = "autocloud"; }
}
pc isa Client {
  sfpAddress = "172.18.240.39"
  // change reference to the latest system
  refer = vm21.apache
}
// "virtually" move machines of the latest
// system to EPCC site by setting "in_cloud"
// with value "proxy.epcc"
vm21 isa VM { created = true
  in_cloud = proxy.epcc
  apache isa Apache {
    running = true
    is_load_balancer = true
    lb_members = (vm22.apache)
  }
}
vm22 isa VM { created = true
  in_cloud = proxy.epcc
  apache isa Apache { running = true; }
  wp_web isa WordpressWeb {
    version = 2
    installed = true
    http = vm12.apache
    database = vm23.wp_db
  }
}
vm23 isa VM { created = true
  in_cloud = proxy.epcc
  mysql isa Mysql { running = true }
  wp_db isa WordpressDB {
    version = 2
    installed = true
    mysql = vm23.mysql
  }
}
// delete machines of old system
vm11 isa VM { created = false
  apache isa Apache; }
vm12 isa VM { created = false
  apache isa Apache
  wp_web isa WordpressWeb; }
vm13 isa VM { created = false
  mysql isa Mysql
  wp_db isa WordpressDB; }
// global constraints
global {
  // pc always refers to a running system
  pc.refer.running = true
  // dependencies between services
  if vm11.apache.running = true
    then vm12.apache.running = true
  if vm12.apache.running = true
    then vm13.mysql.running = true
  if vm21.apache.running = true
    then vm22.apache.running = true
  if vm22.apache.running = true
    then vm23.mysql.running = true
}
```

Fig. 10: Configuration specification for the desired state.

figure 9 shows that the execution time is linear with the number of VMs. We suspect that although VM creation requests were submitted in same time by Nuri, but the broker or the site agent processed them in a queue.

VII. Conclusions

This paper has presented experimental results which demonstrate that declarative specifications, combined with automated planning are a viable approach to practical "configuration relocation" for cloud applications. By defining only the specification of the desired state and the global constraints, the Nuri configuration tool can automatically generate a workflow to implement the relocation within a reasonable time. The execution of the generated workflow is guaranteed to achieve the desired state as well as preserving the necessary constraints during the relocation.

Acknowledgment

This research is fully supported by a grant from HP Labs Innovation Research Program Award.

References

[1] D. Armstrong, D. Espling, J. Tordsson, K. Djemame, and E. Elmroth, "Runtime virtual machine recontextualization for clouds," in *Euro-Par 2012: Parallel Processing Workshops*. Springer, 2013, pp. 567–576.

[2] Puppet Labs, "Puppet," 2013. [Online]. Available: http://www.puppetlabs.com/puppet

[3] H. Herry, P. Anderson, and G. Wickler, "Automated planning for configuration changes," in *Proceedings of the 25th Large Installation System Administration Conference (LISA '11)*. Usenix Association, 2011.

[4] H. Herry and P. Anderson, "Planning with global constraints for computing infrastructure reconfiguration," in *AAAI-12 Workshop on Problem Solving using Classical Planners (CP4PS'12)*. AAAI Press, 2012.

[5] P. Goldsack, J. Guijarro, S. Loughran, A. Coles, A. Farrell, A. Lain, P. Murray, and P. Toft, "The smartfrog configuration management framework," *ACM SIGOPS Operating Systems Review*, vol. 43, no. 1, pp. 16–25, 2009.

[6] M. Helmert, "Concise finite-domain representations for PDDL planning tasks," *Artificial Intelligence*, vol. 173, no. 5-6, pp. 503–535, 2009.

[7] T. Bylander, "The computational complexity of propositional strips planning," *Artificial Intelligence*, vol. 69, no. 1, pp. 165–204, 1994.

[8] M. Ghallab, D. Nau, and P. Traverso, *Automated Planning: theory and practice*. Morgan Kaufmann, 2004.

[9] J. Hoffmann and B. Nebel, "The FF planning system: Fast plan generation through heuristic search," *Journal of Artificial Intelligence Research*, vol. 14, no. 1, pp. 253–302, 2001.

[10] M. Helmert and H. Geffner, "Unifying the causal graph and additive heuristics." in *ICAPS*, 2008, pp. 140–147.

[11] M. Helmert and C. Domshlak, "Landmarks, critical paths and abstractions: What's the difference anyway?" in *Nine-Teenth International Conference on Automated Planning and Scheduling*, 2009.

[12] M. Helmert, "The fast downward planning system," *Journal of Artificial Intelligence Research*, vol. 26, no. 1, pp. 191–246, 2006.