Dependable self-hosting distributed systems using constraints

Qin Yin[*], Justin Cappos[†], Andrew Baumann[*], Timothy Roscoe[*]

[*] *Systems Group, Department of Computer Science, ETH Zurich*

[†] *Department of Computer Science and Engineering, University of Washington*

Abstract

We describe a technique for writing distributed applications which manage themselves over one or more utility computing infrastructures: by dynamically acquiring new computational resources, deploying themselves on these resources, and releasing others when no longer required. Unlike prior work, such management functionality is closely integrated with the application, allowing greater freedom in application-specific policies and faster response to failures and other changes in the environment without requiring any external management system. We address the programming complexity of this approach by applying constraint logic programming, and describe *Rhizoma*, an experimental runtime to explore these ideas. We present early experience of deploying self-hosting applications on PlanetLab using Rhizoma.

1 Introduction: the case for self-hosting

This paper makes a case for "self-hosting" applications over utility-computing infrastructures, and reports on early work on a runtime for such applications using constraint specifications as a programming tool.

We are increasingly seeing distributed applications and services which use hardware and software resources managed by infrastructure providers rather than managed by the service or application provider themselves. In these scenarios, services run on computation resources that have been rented or otherwise dynamically acquired.

The clearest example is the rise of utility computing, where providers such as Amazon's EC2 [1] and competing services already support companies who have outsourced the provisioning of servers for their applications. However, it is also the case in more research-oriented environments like PlanetLab and the Grid, and furthermore this is the explicit model for network-layer services in testbed platforms such as GENI.

Such platforms offer attractive features: cost of maintenance and administration is centralized and amortized over multiple customers, and hardware provisioning is decoupled from software deployment allowing cost savings and responsiveness to changes in demand. However, applications deployed to date over these new infrastructure models (whether commercial services or research projects) have generally not been written specifically for the characteristics of these platforms, instead using them as a substitute for a conventional, fixed infrastructure.

It seems reasonable, therefore, to ask how such utility computing platforms lead to both new opportunities and new challenges in building distributed applications. We are interested in how to effectively write dependable applications directly targetting such deployment platforms.

We argue that while potentially reducing deployment and management cost for applications, utility computing leaves most traditional dependability problems unsolved and furthermore introduces new ones. We focus on three:

Firstly, partial or complete *machine failures* still occur. Although services such as EC2 aim at high availability of virtual hardware, service level agreements are remarkably vague about the uptime guarantees provided. Of course, platforms like PlanetLab are often much *less* available than in-house hosted hardware.

Secondly, *changes in load, performance goals, or other policies* still require that a service acquire more resources or release those it has to reduce cost. At present such provisioning decisions generally involve a human in the loop, though in an enterprise setting a centralized orchestration service can perform this function as part of a global resource allocation policy.

Finally, while utility computing providers such as EC2 aim to provide a stable execution environment, *pricing structures* for infrastructure have changed substantially [8]. As more infrastructure providers enter what is mostly a commodity market, we can expect to see further differentiation on pricing models. Customers seeking to minimize costs will need to adapt in the face of such changes, and some may well wish to use more than one provider at a time.

Dealing with these challenges is traditionally a management function, performed by an external management system and/or human administrator, while the application itself is not involved in the process beyond conventional fault-tolerance and exploiting new resources as and when they become available (as in P2P systems).

We are investigating a different approach, to see if handling these issues *within* the application is both feasible and desirable. In our experimental runtime system, *Rhizoma*, resource management (in effect, monitoring and interaction with one or more infrastructure providers) and deployment on new nodes is performed autonomously by the application itself.

Our system has two salient features: first, it bundles resource management policy into the application, where it is more closely integrated with the rest of the application logic, and second, it uses a constraint system as a programming interface to the runtime, to make the specification and implementation of such tightly-coupled policies tractable. We hope to see several advantages from this approach:

Expressiveness: Constraints can naturally express an application's resource requirements – where it should and should not run, and the global and local properties of the resource set the application needs.

Optimization: Optimizing over the set of constraint solutions allows a powerful declaration of how an application should be deployed given alternatives, in terms of performance (along a variety of dimensions) and cost (capturing complex pricing models of different providers).

Adaptation responsiveness: By coupling the management functionality that determines where and how an application is deployed with the core application logic, the system can react more quickly to changes in available resources, load, or policies.

In the next section we describe constraint logic programming (CLP) and its use in resource management and optimization. Section 3 presents the architecture of Rhizoma, our self-hosting runtime supporting application adaptability. Some early experimental results quantifying this advantage are shown in Section 4. We discuss related work in Section 5, and conclude in Section 6.

2 Why CLP?

Building applications which can acquire, manage, and release their own set of computational resources in response to changing demand and conditions immediately raises the problem of complexity: how can programmers easily write and understand code which causes the application to do "the right thing", when running in a complex and dynamic environment?

We are investigating the suitability of constraint logic programming [17] for this task. Several features of CLP make it an attractive option at first glance for specifying the resource requirements and desired adaptive behavior of a distributed application.

Firstly, *logic programming*'s powerful facilities for expressing concepts such as unification help us to manage the heterogeneity of resource types, measurement and monitoring data, and application policies. Logic programming languages like Prolog are slightly more expressive than formats such as RDF [16], while being considerably more computationally tractable than more heavyweight subsets of first-order logic like description logics [15]. As researchers this gives us a convenient programming platform to explore the design space for more specialized solutions.

Secondly, CLP extends logic programming with the addition of logical and numeric *constraints*, which are a natural way to express both local and global resource requirements such as physical location, computational power, communication quality, replication conditions, and cost limitations, provided that a feasible solution to a constraint set can be found reasonably efficiently.

An example of a small overlay-based service on the PlanetLab platform can illustrate the use of constraints. Given information about node properties and current status, the application developer can specify requirements using the following:

```
node_constraint(Host) :-
  node{hostname: Host, resptime: Resptime,
       fiveminload: Fiveminload,
       cpuspeed: Cpuspeed, freecpu: Freecpu},
  alive(Host), Resptime < 0.5, Fiveminload < 5,
  Cpuspeed*Freecpu/100 > 1.
```

`node_constraint` defines the constraints on each node. The pre-defined `alive` predicate requires that a node respond to ping requests, accept SSH connections, have acceptable clock skew and a working DNS resolver. `Resptime` and `Fiveminload` ensure light load, while the final clause guarantees a minimum CPU capacity.

More interesting are global constraints:

```
group_constraint(HostList) :-
  length(HostList, N), N = 3,
  % Geographical constraints
  node_loc(HostList, LocList),
  geographical_constraint(LocList),
  % Aggregated memory constraints
  node_mem(HostList, FreememList),
  node_mem_constraint(FreememList),
  % Inter-node constraint
  edge_constraint(HostList).

geographical_constraint(LocList) :-
  alldifferent(LocList).
```

```
node_mem_constraint(FreememList) :-
  avg(FreememList, Avg), Avg > 80. % built-in

edge_constraint(HostList) :-
  make_graph_pl(HostList, Graph),  % built-in
  diameter(Graph, Diameter),       % built-in
  Diameter < 300.
```

These constraints include the number of nodes, spatial distribution, aggregate properties on specific attributes (such as minima, maxima, averages or sums) and edge constraints on latency, bandwidth, or network diameter. In this example, we choose three nodes on different continents with minimum average free memory, and a network diameter lower than 300ms.

The example also illustrates the need to reduce the complexity of constraint specifications, making them accessible to application developers. By providing a library of built-in constraints, the most common requirements can be succinctly expressed, with the power of the full constraint language remaining available for the few cases where it is necessary.

Thirdly, CLP solvers typically perform *optimization*, allowing a programmer to give criteria in the form of an *objective function* for selecting the best solution which satisfies the constraints. This is potentially a very powerful technique as it allows an application to select the cheapest feasible deployment which meets its performance targets. For our simple example, we define the objective function to be a weighted average of the deviation of various node parameters from an ideal:

$$obj_i = \frac{1}{\sum_i weight_i} \left(\frac{x_i - a_i}{b_i - a_i} weight_i \right)$$

Here, a_i and b_i give a range for parameter x_i (b_i is an ideal value, and may be greater or less than a_i). Example values are shown in Table 1.

In reality, this is not sufficient. An objective function like the one above may measure the "goodness" of a particular state, but fails to take into account the cost of reconfiguring the application from one state to another.

In practice, in our runtime system Rhizoma, we optimize the objective minus a *cost function* which captures how hard it is to acquire and use new resources like virtual machines and release old ones. Furthermore, the solution that Rhizoma's CLP solver generates is not a new application configuration, but rather a set of "moves" that improve the service's state. This is a powerful technique for reducing the computational complexity of evaluating the constraint program for two reasons: firstly, it *a priori* limits the search space to configurations which are not wildly different from the application's current deployment, and secondly, it allows a programmer to cleanly express this tradeoff both in the cost function, and additional constraints that limit the number of "moves".

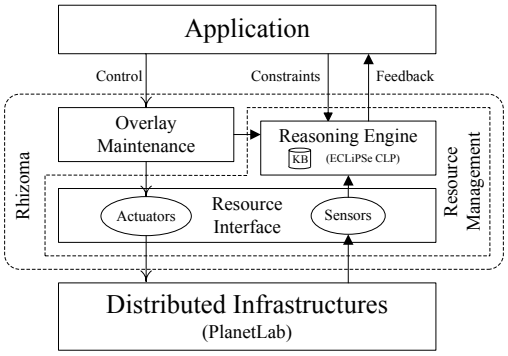


Figure 1: Rhizoma per-node architecture

Finally, CLP is a relatively mature technology, and easily embeddable CLP solvers are readily available – it has been used extensively for scheduling, planning, and routing problems where a diverse set of resources must be reconciled with a complex set of requirements. Our aim is not to push the envelope of constraint techniques, but to investigate their usefulness for our system.

We are certainly not the first to point out CLP's applicability to problems of system management – indeed, the ECL^{i}PS^e constraint solver we use for Rhizoma was originally developed as the core of a suite of network management applications [4]. However, we are unaware of existing work building such functionality into the application itself, and thereby integrating application knowledge and management information.

3 Rhizoma architecture

Our runtime system, Rhizoma, bundles management logic with every instance of the distributed application. As shown in Figure 1, application developers specify system requirements in the form of constraints. Then, based on real-time network statistics, Rhizoma automatically checks the system state for conformance with the requirements, generating feedback to the application layer whenever the constraints are not satisfied. Applications translate such feedback into overlay operations (such as allocating or releasing nodes) to effect changes to the distributed infrastructure on which they execute.

Resource management, a cross-layer component critical to the architecture of Rhizoma, is shared between the application and Rhizoma itself. It consists of three parts: a *reasoning engine* embedding the ECL^{i}PS^e constraint programming system [4], along with *sensors* and *actuators* wrapped by the resource interface. Sensors collect status information about the underlying distributed infrastructure and update a *knowledge base* (KB) maintained by Rhizoma. The reasoning engine evaluates the constraints provided by the application against the infor-

	(a, b)	weight
freecpu	(1, 3)	3
freemem	(50, 100)	2
fiveminload	(10, 0)	2
liveslice	(10, 0)	1

Table 1: Objective function parameters

mation in the KB, either periodically or in response to a change in resource availability. It then reports the results to the application, which may take actions on the overlay.

Although the reasoning engine is capable of running on any node in the system, a leadership election is used to select one node to perform reasoning and disseminate the results. If this node fails, a new leader will be elected to perform the reasoning. Management is therefore decentralized and thus as robust to failure as the application itself – there is fate-sharing between the management logic and the application.

The overlay maintenance component facilitates control and communication within the overlay network. It provides a simple management API for the application. Control commands from the application are interpreted as operations on the distributed infrastructure, which in turn are executed by the actuators. The overlay maintenance component also serves as a data source for the KB, providing real-time overlay status information. In this way, a control-flow cycle is formed from one decision to the next, making the system self-adaptable.

4 Early experiments

In order to evaluate how Rhizoma helps applications adapt to resource changes, we tested our early prototype using a simple PlanetLab application.

PlanetLab might seem an unusual choice for implementing mechanisms for dependability: it is considerably more heterogeneous, less predictable, and less available than other utility computing platforms, and not remotely representative of platforms like EC2. Our methodology is to use PlanetLab as an early "proving ground", since its variability is likely to expose issues with our design that would take longer to discover through simulation or deployment on more stable platforms. Once these issues are shaken out, we will investigate Rhizoma in environments that more closely match our motivating scenarios.

The constraints for our test application are the same as those in Section 2. The parameters chosen for the objective function are shown in Table 1, and the migration cost is set as $mcost = 0.1n$ (where n is the number of changed nodes) to penalize overlay churn. The best solution is then selected by Rhizoma, minimizing the difference of the objective value and migration cost.

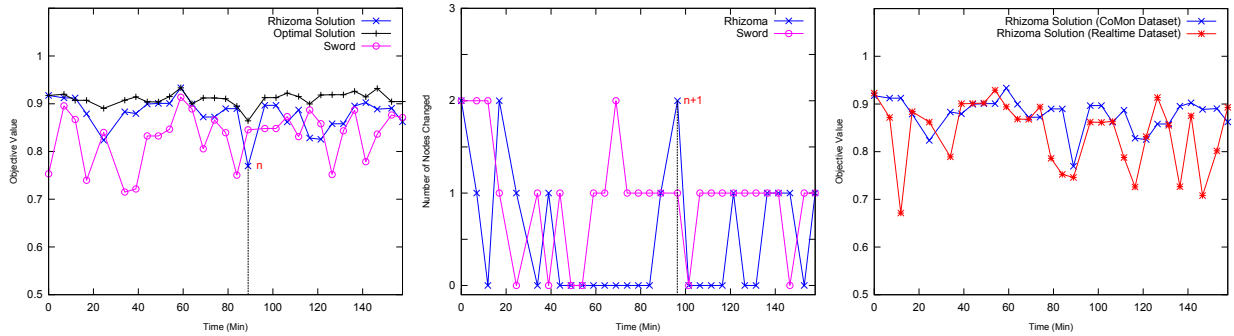
We started our application manually at first on an arbitrary PlanetLab node. Rhizoma periodically generates a deployment solution according to network status reported by PlanetLab monitoring services [14, 19] and the provided constraints. Meanwhile, a comparable query is sent to the SWORD [13] service. SWORD is a service commonly used for deploying PlanetLab applications. It accepts XML-encoded user queries expressed as utility functions, and returns a set of nodes that maximizes the utility. The query we send does not include any of the group constraints, because they cannot be expressed as a SWORD utility function. The reasoning period is set to five minutes, matching that of the most frequently updated monitoring service.

Figure 2 shows the experimental results. Figure 2(a) plots the value of the objective function on different solutions.[1] In this figure, *Rhizoma solution* is our final deployed solution, *optimal solution* is a hypothetical solution maximizing the objective value with $mcost = 0$, and *SWORD solution* is our comparison. Figure 2(b) shows the number of nodes changed by the Rhizoma and SWORD solutions. Rhizoma performs 15 migrations in total, whereas SWORD changes the node set 28 times, as it does not record the previous solution, nor consider migration costs.

As shown in Figures 2(a) and (b), Rhizoma usually performs better than SWORD, although SWORD can outperform Rhizoma, as it derives a solution irrespective of cost. Despite being conservative, Rhizoma still exhibits adaptability and responsiveness. Because it considers migration cost, Rhizoma keeps the system stable even as the solution's objective value degrades in the nth round. However, it reacts to this loss, and redeploys to two new nodes in the next round to increase the objective value.

The experiment shows Rhizoma's ability to optimize and adapt to changes in resource availability. However, to see the real performance of the Rhizoma solution, we plotted the objective value of this solution as calculated using two different data sets: one based on the same sources used by Rhizoma, and a second using a real-time data set gathered every 30 seconds by our own monitoring service, which collects the same information locally on every node. These results appear in Figure 2(c), which shows that the data sets that Rhizoma uses for its reasoning tend to lag behind, while the network status varies more frequently. They also explain the outlier in the third round, during which the *Rhizoma solution* appears to perform better than the *optimal solution*, because Rhizoma saw a stale data set, leading to two successive rounds

[1] The objective values are calculated based on the CoMon [14] (updated every five minutes) and S3 [19] (updated every four hours) data sets.



(a) Objective value comparison (b) Number of nodes changed (c) Real objective value

Figure 2: Rhizoma experimental results

with the same solution and objective value. The system's responsiveness could therefore be improved if real-time overlay status information was fed back into the knowledge base.

5 Related work

Constraints have been used to manage real-world network configuration from high-level specifications [6,12], and logic programming more generally has been widely applied in projects to manage networks [18], express network protocols [11] and in system administration [10]. A number of projects provide centralized management services for distributed platforms including PlanetLab [2,3,9,13] and the Grid [5,7].

6 Conclusions and future work

CLP provides a clean programming interface to describe heterogeneous network resources as well as tightly-coupled application resource policies. However, skills and experience are needed to write constraints that can be efficiently evaluated. Adding more powerful constraints can help improve application performance, but such over-specification may result in no solutions being found. In order to reduce the complexity of writing constraint logic, we intend to provide more built-in constraint solvers. Some concerns of future work include which solvers to provide and what interfaces to use.

The general efficiency of the constraint solver can also be improved in two ways: by using the branch-and-bound algorithm to find sub-optimal but "good-enough" solutions with a second *acceptability* bound, and by using an incremental reasoning strategy instead of starting from scratch each time.

Since the resource management and application logic are tightly integrated, the complexity of resource allocation as well as overlay maintenance increases with the size of the overlay. We are primarily targeting applications of limited size – no more than a hundred nodes – in a heterogeneous network environment. Even though the overlay won't scale to thousands of nodes, the efficiency of resource allocation and the consistency of overlay maintenance remain a challenge.

Rhizoma tries to adapt the application overlay quickly to node failures, available resources and modified policy. However, transient constraint violation is unavoidable when changing from one stable network configuration to another. To take a simple example, if the application decides to move from some heavily-loaded to more lightly-loaded nodes, the migration process will involve first replicating to the new nodes before shutting down the heavily-loaded nodes, and thus will lead to a temporary violation of a constraint that calls for a fixed number of nodes. Future work involves identifying which application constraints may be violated, and giving the application developer more control over such situations.

Finally, rather than the simple application used currently, we intend to explore applications with diverse resource and topology constraints to quantify the relationship between solution performance and migration cost, and to develop general re-usable constraint routines.

In summary, we are exploring how to build dependable distributed applications over utility-computing infrastructures. Two novel ideas, using constraint specifications as a programming tool and building resource management policy into application logic, are proposed, and an initial evaluation of our runtime, Rhizoma, is presented. We have argued by example that constraints enable greater expressiveness of resource requirements and continuous optimization of resource allocation. Our early experiments have shown that Rhizoma's constraint-based runtime can perform sophisticated resource management in the face of changing conditions and machine failures over a dynamic infrastructure.

References

[1] Amazon Elastic Compute Cloud. http://aws.amazon.com/ec2.

[2] J. Albrecht, R. Braud, D. Dao, N. Topilski, C. Tuttle, A. C. Snoeren, and A. Vahdat. Remote control: distributed application configuration, management, and visualization with Plush. In *LISA'07*, pages 1–19, 2007.

[3] P. Anderson, P. Goldsack, and J. Paterson. SmartFrog meets LCFG: Autonomous reconfiguration with central policy control. In *LISA'03*, pages 213–222, 2003.

[4] K. R. Apt and M. G. Wallace. *Constraint Logic Programming using ECL^{i}PS^e*. Cambridge University Press, 2007.

[5] A. Bricker, M. Litzkow, and M. Livny. Condor technical summary. Tech. Rep. 1069, University of Wisconsin-Madison, 1991.

[6] T. Delaet, P. Anderson, and W. Joosen. Managing real-world system configurations with constraints. In *ICN'08*, Apr. 2008.

[7] I. Foster and C. Kesselman. Globus: A metacomputing infrastructure toolkit. *Intl. J. Supercomp. Apps. & High Perf. Comp.*, 11(2):115–128, 1997.

[8] S. L. Garfinkel. An evaluation of Amazon's grid computing services: EC2, S3 and SQS. Technical report TR-08-07, School for Engineering and Applied Sciences, Harvard University, 2007.

[9] R. Huebsch. PlanetLab application manager. http://appmanager.berkeley.intel-research.net/.

[10] IBM. *Tivoli Enterprise Console Rule Developer's Guide*, 1st edition, Aug. 2003. SC32-1234-00.

[11] B. T. Loo, T. Condie, J. M. Hellerstein, P. Maniatis, T. Roscoe, and I. Stoica. Implementing declarative overlays. *SIGOPS Oper. Syst. Rev.*, 39(5):75–90, 2005.

[12] S. Narain. Network configuration management via model finding. In *LISA'05*, pages 15–15, 2005.

[13] D. Oppenheimer, J. Albrecht, D. Patterson, and A. Vahdat. Distributed resource discovery on PlanetLab with SWORD. In *WORLDS'04*, Dec. 2004.

[14] K. Park and V. S. Pai. CoMon: a mostly-scalable monitoring system for PlanetLab. *SIGOPS Oper. Syst. Rev.*, 40(1), 2006.

[15] W3C. OWL web ontology language. http://www.w3.org/TR/owl-ref.

[16] W3C. Resource description framework. http://www.w3.org/RDF.

[17] M. Wallace. Constraint programming. In J. Liebowitz, editor, *The Handbook of Applied Expert Systems*. CRC Prs., Dec. 1997.

[18] M. Wawrzoniak, L. Peterson, and T. Roscoe. Sophia: An information plane for networked systems. In *HotNets-II*, Nov. 2003.

[19] P. Yalagandula, P. Sharma, S. Banerjee, S. Basu, and S.-J. Lee. S3: a scalable sensing service for monitoring large networked systems. In *INM'06*, pages 71–76, 2006.