Rhizoma: a runtime for self-deploying, self-managing overlays

Qin Yin[1], Adrian Schüpbach[1], Justin Cappos[2], Andrew Baumann[1], and Timothy Roscoe[1]

[1] Systems Group, Department of Computer Science, ETH Zurich
[2] Department of Computer Science and Engineering, University of Washington

Abstract. The trend towards cloud and utility computing infrastructures raises challenges not only for application development, but also for management: diverse resources, changing resource availability, and differing application requirements create a complex optimization problem. Most existing cloud applications are managed externally, and this separation can lead to increased response time to failures, and slower or less appropriate adaptation to resource availability and pricing changes.
In this paper, we explore a different approach more akin to P2P systems: we closely couple a decentralized management runtime ("Rhizoma") with the application itself. The application expresses its resource requirements to the runtime as a constrained optimization problem. Rhizoma then fuses multiple real-time sources of resource availability data, from which it decides to acquire or release resources (such as virtual machines), redeploying the system to continually maximize its utility.
Using PlanetLab as a challenging "proving ground" for cloud-based services, we present results showing Rhizoma's performance, overhead, and efficiency versus existing approaches, as well the system's ability to react to unexpected large-scale changes in resource availability.

1 Introduction

In this paper, we investigate a new technique for distributed application management over a utility computing infrastructure. Commercial "cloud computing" facilities like Amazon's EC2 [3] provide a managed, low-cost, stable, scalable infrastructure for distributed applications and are increasingly attractive for a variety of systems. However, such services do not remove the burden of application management, but instead modify it: deployment of hardware and upgrades of software are no longer an issue, but new challenges are introduced.

First, such services are not totally reliable, as recent Amazon outages show [4]. Second, changing network conditions external to the cloud provider can significantly affect service performance. Third, pricing models for cloud computing services change over time, and as competition and the pressure to differentiate in the sector intensifies, we can expect this to happen more frequently. Finally, the sheer diversity of pricing models and offerings presents an increasing challenge to application providers who wish to deploy on such infrastructures.

These challenges are typically addressed at present by a combination of a separate management machine outside the cloud (possibly replicated for availability or managed by a company like RightScale[3]) and human-in-the-loop monitoring. Such arrangements have obvious deficiencies: slow response time to critical events, and the cost of maintaining an infrastructure (albeit a much smaller one) to manage the virtual infrastructure which the application is using.

This paper evaluates an alternative approach whereby the application manages itself as a continuous process of optimization [23], and makes the following contributions. First, we present Rhizoma, a runtime for distributed applications that obviates the need for a separate management console and removes any such single point of failure by turning the application into a *self-deploying* system reminiscent of early "worm" programs [19]. Second, we show how constraint logic programming provides a natural way to express desired application behavior with regard to resources, and can be applied to simplify the task of acquiring and releasing processing resources autonomously as both external conditions and the needs of the application itself change. Finally, using PlanetLab, we show that in spite of its flexibility, our approach to resource management results in better application performance than a centralized, external management system, and can adapt application deployment in real time to service requirements.

In the next section, we provide background and motivation for our approach, and Section 3 describes how application providers deploy a service using Rhizoma. In Section 4 we detail how Rhizoma operates at runtime to manage the application and optimize its deployment. Section 5 presents our current experimental implementation on PlanetLab, and Section 6 shows results from running Rhizoma in this challenging environment. Finally, Section 7 covers related work, and we discuss future work and conclude in Section 8.

2 Background and Motivation

Deploying and maintaining applications in a utility computing environment involves important decisions about which resources to acquire and how to respond to changes in resource requirements, costs, and availability.

An operator deploying an application must first consider the offerings of assorted providers and their costs, then select a set of nodes on which to deploy the application. Amazon's EC2 service currently offers a choice of node types and locations, and the selection will become increasingly complex as multiple providers emerge with different service offerings and pricing models.

The selected compute resources must then be acquired (typically purchased), and the application deployed on the nodes. Following this, its status must be monitored, as offered load may change or nodes might fail (or an entire service provider may experience an outage [4]). These factors may require redeploying the application on a larger, smaller, or simply different set of nodes. This leads to a control and optimization problem that in many cases is left to a human operator working over timescales of hours or days.

[3] http://www.rightscale.com/

Alternatively, a separate management system is deployed (and itself maintained) on dedicated machines to keep the application running and to respond to such events. In autonomic computing this function may be referred to as an "orchestration service", and typically operates without the application itself being aware of such functionality. Such separation and (logical) centralization of management can have benefits at large scales, but introduces additional complexity and failure modes, which are particularly significant to operators deploying smaller services where the cost of an additional dedicate node is hard to justify.

To address these problems, we explore an alternative model for application management and present our experience building a runtime system for distributed applications which are *self-hosting*: the application manages itself by acquiring and releasing resources (in particular, distributed virtual machines) in response to failures, offered load, or changing policy. Our runtime, Rhizoma, runs on the same nodes as the application, performing autonomous resource management that is as flexible and robust to failure as the application itself.

In order to remove a human from direct management decisions, and to decouple Rhizoma's management decisions from the application logic, we need a way to specify the application's resource requirements and performance goals. This specification needs to be extensible in terms of resource types, must be able to express complex relationships between desired resources, and must allow for automatic optimization. These requirements led us to choose *constraint logic programming* (CLP) [21] as the most tractable way to express application demands; we introduce CLP and further motivate its use below.

2.1 Constraint logic programming

CLP programs are written as a set of constraints that the program needs to meet and an objective function to optimize within those constraints. This provides a direct mapping between the operator's expression of desired performance and cost, and the application's underlying behavior. Rather than an operator trying to find out how many program instances need to be deployed and where to deploy them, the CLP solver treats the task as an optimization problem and uses application and system characteristics to find an optimal solution.

A constraint satisfaction problem (CSP) consists of a set of variables $V = V_1, \ldots, V_n$. For each V_i, there is a finite set D_i of possible values it can take (its domain). Constraints between the variables can always be expressed as a set of admissible combinations of values. CLP combines logic programming, which is used to specify a set of possibilities explored via a simple inbuilt search method, with constraints, which are used to minimize the reasoning and search by eliminating unwanted alternatives in advance. A solution of a CSP is an assignment of values to each variable such that none of the constraints are violated.

CLP is attractive for application management for another reason: As in the resource description framework (RDF), CLP programs have powerful facilities for handling diversity in resource and information types, since CLP can use logical unification to manage the heterogeneity of resource types, measurement and monitoring data, and application policies. This allows a CLP program to

easily add new resources and data sources while continuing to utilize the existing ones. Unlike RDF query languages, however, CLP provides a natural way to express high-level optimization goals.

We are not the first to make this observation. There is an increasing consensus that constraint satisfaction has a powerful role to play in systems management (for example, in configuration management [7]), and indeed the CLP solver we use in Rhizoma was originally written to build network management applications [5]. A novel feature of Rhizoma is embedding such a CLP system within the application, allowing it to manage and deploy itself.

2.2 Network testbeds

The challenges outlined above will also be familiar to users of networking and distributed systems testbeds such as PlanetLab [15]. A number of systems for externally managing PlanetLab applications have appeared, such as Plush [2] and AppManager [9]. In this paper, we evaluate Rhizoma on PlanetLab, however our target is future cloud computing infrastructure. PlanetLab is a very different environment to cloud providers like EC2 in several important respects.

Firstly, PlanetLab is much *less* stable than services like EC2. This helps us to understand how Rhizoma can deal with server, provider or network outages. PlanetLab is an excellent source of trouble: deploying on PlanetLab is likely to exercise weaknesses in the system design and reveal problems in the approach.

Secondly, PlanetLab nodes are more diverse (in hardware, location, connectivity and monitored status) than current cloud offerings nodes, allowing us to exercise the features of Rhizoma that handle such heterogeneity without waiting for commercial offerings to diversify.

Finally, we can deploy measurement systems on PlanetLab alongside Rhizoma for instrumentation, which is hard with commercial infrastructure services.

3 Using Rhizoma

In this section, we describe how Rhizoma is used as part of a complete application deployment. The Rhizoma runtime executes alongside the application on nodes where the application has been deployed, and handles all deployment issues. Consequently, the only nodes used by Rhizoma are those running the application itself – there are no management nodes *per se* and no separate daemons to install.

We use our current, PlanetLab implementation for concrete details, and use the term "application" to refer to the whole system, and "application instance" or "instance" for that part of the application which runs on a single node.

3.1 Initial deployment

Rhizoma targets applications that are already capable of handling components which fail independently and organize into some form of overlay. To deploy such an application, a developer packages the application code together with the

Rhizoma runtime, and supplies a constraint program specifying the deployment requirements (described in Section 3.2 below) and short scripts that Rhizoma can call to start and stop the application on a node. These scripts can be extremely short (typically one or two lines) and are the only part where explicit interaction between the application and Rhizoma is required.

Other interaction is optional, albeit desirable, for applications that wish to direct resource allocation based on application metrics. For example, a Rhizoma-aware web cluster can scale up or down in response to workload changes while considering the current system configuration, so that resource consumption can be optimized without sacrificing quality of service. Application developers can also benefit from the underlying Rhizoma facilities, as described in Section 3.4.

A developer can deploy the application by simply running the package on one node (even a desktop or laptop computer). No further action and no specific software is required on any other node – Rhizoma will start up, work out how to further deploy the application on a more appropriate set of nodes, and vacate the initial machine when it has acquired them. Rhizoma can in fact be seen as a "worm", albeit a benign one, in that it moves from host to host under its own control. We discuss the relationship with early worm programs in Section 7.

3.2 The constraint program

The constraint program specifies how the application is to be deployed, and can be supplied by the developer or operator of the application. It can also be changed (with or without human intervention) while the application is running, though this capability has not been used to date. The program specifies a *constraint list*, a *utility function*, and a *cost function*.

The constraint list is a set of logical and numeric constraints on the *node set*, i.e. the set of nodes on which the application is to execute. These are conditions which must be satisfied by any solution.

The utility function $U(N)$ for a given node set gives a value in the interval $(0, 1)$ representing the value of a deployment. This function may make use of anything that Rhizoma knows about the nodes, such as their pairwise connectivity (latency, bandwidth), measured CPU load, etc.

The cost function $C(\Delta)$ specifies a cost for deploying or shutting down the application on a node. As with utility, this function may take into account any information available to Rhizoma. Its definition might range from a constant value to a complex calculation involving pricing structures and node locations.

Rhizoma will attempt to find a node set which satisfies the constraints and maximizes the value of the *objective function*, defined as the utility $U(N)$ minus the cost $C(\Delta)$ of moving to the new set from the current one.

3.3 Example: PsEPR

To provide a concrete example of deploying an application with Rhizoma, we take a publish/subscribe application inspired by the (now defunct) PsEPR service on

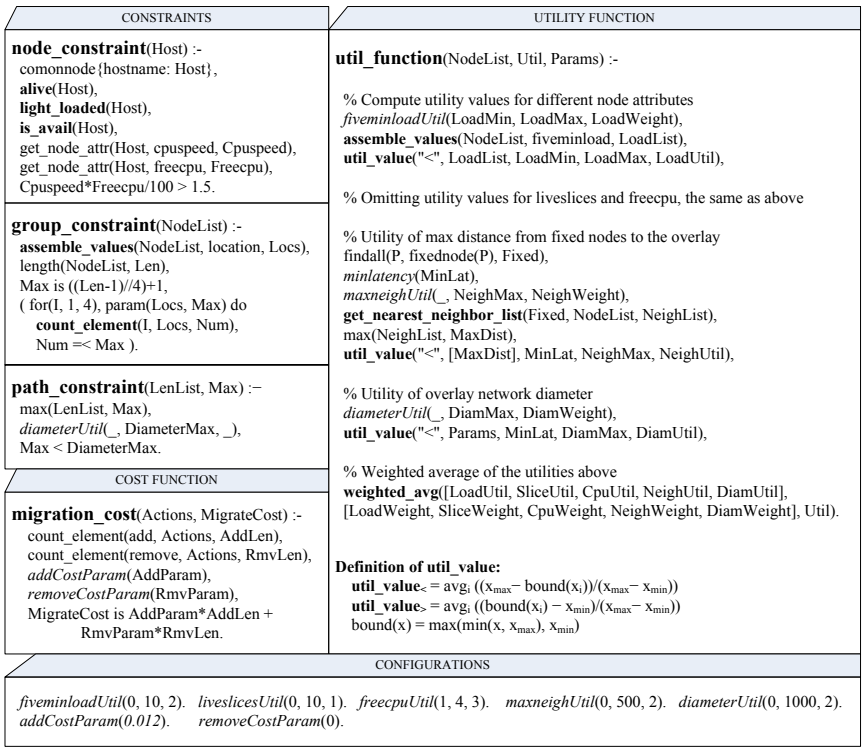


Fig. 1: PsEPR application requirements

PlanetLab [6]. Informally, PsEPR's requirements were to run on a small set of well-connected, lightly loaded, and highly available PlanetLab nodes which were sufficiently distributed to be "close" to the majority of other PlanetLab nodes.

Deployment of the original PsEPR system was performed by hand-written parallel SSH scripts. Nodes were selected based on informal knowledge of location properties, together with human examination of data from the CoMon monitoring service [14], which includes status information such as node reachability, load, and hardware specifications. Node failures were noticed by human operators, and new nodes were picked manually. The set of nodes was reviewed irregularly (about once a month) [1].

Constraints: PsEPR is an example of a distributed application with requirements that cannot be expressed simply as number of nodes or minimum per-node resources. Figure 1 shows how PsEPR's requirements can be expressed as a set of Rhizoma constraints. These constraints are applied to data acquired by Rhizoma as described in Section 5.2, and include node constraints (which a node must satisfy for it to be considered), group constraints (defined over any group of nodes), and network constraints (specifying desired network characteristics).

node_constraint uses several pre-defined Rhizoma predicates. **alive** requires that the node responds to ping requests, accepts SSH connections, has

low clock skew, and a working DNS resolver. **light_loaded** specifies maxima for the memory pressure, five-minute load, and number of active VMs on the node. Finally, **is_avail** checks that the node is not listed in Rhizoma's "blacklist" of nodes on which it has previously failed to deploy. Moreover, developers can also define new logical and numeric constraints on node properties. Here, we require the node to have a certain amount of "free" CPU cycles available, as calculated from its CPU utilization and clock speed.

group_constraint specifies that nodes are evenly distributed over four geographical regions. Here we use the fact that the data available for every node includes an integer in the range $[1, 4]$ indicating a geographical region (North America, Europe, Asia, or South America). We specify that the number of nodes in any region is no greater than the integer ceiling of the total number of nodes divided by the number of regions.

path_constraint limits a maximum diameter for each shortest network path between any two nodes of the resulting overlay network.

Utility function: The constraints define "hard" requirements the system must satisfy and limit the allowable solutions. However, a system also has "soft" requirements which are desirable but not essential. To address this, we specify a utility function that calculates a utility value for any possible deployment, for which Rhizoma attempts to optimize. Here, we construct the utility function as a weighted average of the deviation of various node parameters from an ideal. For PsEPR, we consider for every node the five-minute load, the number of running VMs (or *live slices*, in PlanetLab terminology) and available CPU, the network diameter, and the maximum latency to the overlay from each of a set of ten manually chosen, geographically dispersed anchor nodes.

In **utility_function**, **assemble_values** gathers a list of values for a given parameter for every node in the node list. **util_value** is a built-in function that computes the utility for an individual parameter given the list of values for the parameter: (x_{min}, x_{max}, weight). As an example, for the experiments reported in Section 6, the values are configured as shown at the bottom of Figure 1. **util_value** computes utility as an average of the deviations of a parameter x_i from the ideal (x_{max} or x_{min}) as defined in **utility_function** of Figure 1. **get_nearest_neighbor_list** finds for every node in the list of fixed nodes, the nearest neighbor to it in the overlay. **minlatency** is the minimum latency to any node, as determined by Rhizoma at run-time. Finally, **weighted_avg** computes the weighted average of a list of values given a list of weights.

Cost function: This function incorporates two notions: the cost of a particular deployment, and the cost of migrating to it. The former is relatively straightforward: in PlanetLab it is generally zero, and for commercial cloud computing services can be a direct translation of the pricing structure. Indeed, the ability to optimize for real-world costs is a powerful feature of Rhizoma.

However, quantifying the cost of migration is much harder, and does not correspond to something a developer is generally thinking of. In Figure 1, we

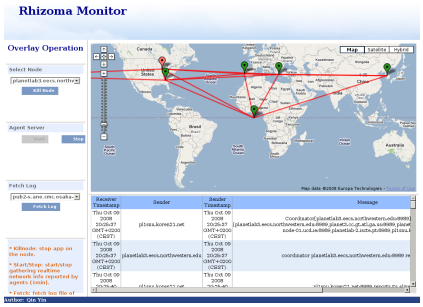


Fig. 2: Visualizing a Rhizoma application

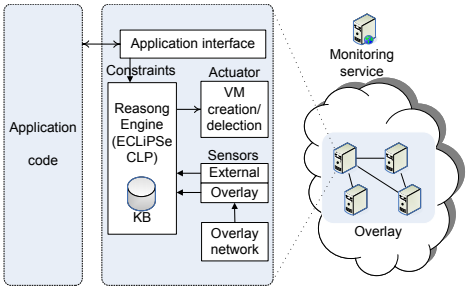


Fig. 3: Rhizoma architecture

adopt a simple linear model in which the migration cost increases with the number of nodes added and removed. The deployment effect of varying the migration cost by tuning the constant coefficients is investigated in Section 6.5. The migration cost could also consider application details and configuration changes. Ideally, it would be learned online by the system over time.

3.4 Rhizoma-aware programs

Although we have presented the minimal interface required for existing applications to be deployed with Rhizoma, the runtime's functionality is also available to applications. Rhizoma maintains an overlay network among all members of the node set, and uses this for message routing. It also maintains up-to-date status information for all nodes in the application, plus considerable external monitoring data gathered for the purpose of managing deployment, along with a reasoning engine that applications can use to execute queries.

This functionality is exposed via a service provider/consumer interface for applications written using Rhizoma's module framework. The framework maintains module dependencies through service interaction, and can be extended by developers with additional application modules.

3.5 Observing the application

Since the node set on which the application is deployed is determined by Rhizoma as an ongoing process, a human user cannot necessarily know at any moment where the application is running (though it is straightforward to specify some "preferred" nodes in the constraint program). For this and debugging reasons, Rhizoma additionally stores the IP addresses of the node set in a dynamic DNS server, and also exports a management interface, which allows arbitrary querying of its real-time status from any node in the application. It is straightforward to build system visualizations, such as the one in Figure 2, using this data.

3.6 Discussion

Rhizoma relieves service operators of much of the burden of running a service: deploying software, choosing the right locations and machines, and running a sep-

arate management service. However, this simplicity naturally comes at a price: despite their attractiveness, constraint solvers have never been a "magic bullet".

The first challenge is computational complexity. It is very easy to write constraint programs with exponential performance curves that become intractable even at low levels of complexity. In Section 4.4, we describe one approach for preventing this in Rhizoma. More generally, the art of writing good constraint programs lies in selecting which heuristics to embed into the code to provide the solver with enough hints to find the optimum (or a solution close enough to it) in reasonable time. This is a hard problem (and a topic of much ongoing research in the constraint community).

Application developers may find it difficult to write constraints in a language such as the Prolog dialect used by our CLP solver. While constraints and optimization provide a remarkably intuitive way to specify requirements at a high level, there is a gap between the apparently simple constraints one can talk about using natural language, and the syntax that must be written to specify them.

We address both of these issues by trading off expressivity for complexity (in both senses of the word). We provide a collection of useful heuristics based on our experience with PlanetLab, embedded in a library that provides high-level, simplified constraints which use the heuristics. This library can also serve as the basis for a future, simplified syntax. In this way, developers are assured of relatively tractable constraint programs if they stick to the high-level constructs we supply (and need only write a few lines of code). However, the full expressive power of CLP is still available if required.

4 Operation

The architecture of a Rhizoma node is shown in Figure 3. As well as the reasoning engine introduced in the previous section, Rhizoma consists of an overlay maintenance component, and resource interfaces to one or more distributed infrastructures (such as PlanetLab). The application interface could be as simple as configuring a constraint file or as complex as using the component services to build an application from scratch.

In this section, which describes the operation of a Rhizoma system in practice, we first introduce the sensors/actuators and knowledge base (KB in the figure), two key components of the Rhizoma architecture, and describe the maintenance of an overlay network. We then give a detailed discussion of the steady-state behavior of Rhizoma, including the components used to construct it, followed by what happens at bootstrap and when a new node is started.

4.1 Sensors and knowledge base

Sensors in Rhizoma periodically take a snapshot of resource information about node or network status from external and internal monitoring services. External monitoring services provide coarse-grained information about the whole hosting

environment, while internal monitoring services provide fine-grained measurement of more up-to-date resource information on a given overlay.

To maintain and optimize the system, Rhizoma stores data collected by the sensors in the *knowledge base* used by the CLP solver. Every member of the overlay maintains its own knowledge base, though their content and usage differ, as described in the following section. As part of the CLP solver, the knowledge base provides a query interface. High-level knowledge can be derived from low-level facts. For example, based on the overlay status data, we can compute the network diameter in terms of latency.

The knowledge base in the current implementation stores only the latest information retrieved by the sensors, however in the future we intend to time-stamp the available data and maintain historical information such as moving averages, which could be used by constraint programs.

4.2 Coordinator node

To manage the overlay, Rhizoma elects one node as a coordinator. The coordinator can be any node in the overlay, and any leadership election algorithm may be used. The currently-implemented election algorithm simply chooses the node with the lowest IP, although we intend to explore leadership-election based on node resources as future work. This node remains the coordinator as long as it is alive and in the overlay. A new coordinator is elected if the node crashes, or if the optimization process decides to move to another node.

Only the coordinator node runs the constraint program, storing in its knowledge base the complete data set, which provides it with a global view of the hosting environment. To optimize the use of communication and computation resources within the overlay, other nodes maintain only the overlay status.

4.3 Steady-state behavior

To respond to changes in resource utilization, Rhizoma performs several periodic operations in its steady state. First, the coordinator collects information from the sensors, and updates the knowledge base periodically. The periods are based on the characteristics of different sensors, such as their update frequency and data size. Real-time overlay information from the resource monitoring service is collected by the coordinator and disseminated to each member in the overlay.

Since every member in the overlay knows the current overlay status, it can take decisions to optimize the resource usage. Rhizoma currently supports shortest-path routing and minimum spanning-tree computation for broadcast communication. In the future, the performance optimization of applications subject to available overlay resources could also be investigated.

To adapt to changes in the host environment, the coordinator periodically solves the developer-provided constraints based on the current resource capacity and utilization. If the current overlay state is not suitable, it yields a list of actions to apply to it. These actions will move the current network configuration towards a new one that meets the application's constraints and has higher utility.

4.4 Optimization process

The actions derived from periodic solving include acquiring new nodes and releasing existing nodes. In principle, the solver tries to maximize the value of the objective (defined as the utility function minus the cost function) based on the current knowledge base, subject to the constraints.

In practice, this approach would lead to exponential complexity increases, particularly in a PlanetLab-like environment where the number of node options is very large (more than 600 live nodes at any time) – an optimal overlay of c nodes would require examining on the order of $\binom{600}{c}$ possible configurations. Rhizoma's solver instead derives an optimal set of at most n *add(node)* or *remove(node)* actions which will improve the utility of the current deployment subject to the cost of the actions. Here, n is a relatively small horizon (such as two or three), which makes the optimization considerably more tractable. Such incremental optimization also has a damping effect, which prevents Rhizoma from altering its configuration too much during each period.

This technique is a case of the well-known hill-climbing approach, and can lead to the familiar problem of local maxima: it is possible that Rhizoma can become stuck in a sub-optimal configuration because a better deployment is too far away to be reached. In practice, we have not observed serious problems of this sort, but it can be addressed either by increasing the horizon n, or using one of several more sophisticated optimization algorithms from the literature.

4.5 Adding or removing nodes

The actions chosen by the solver are executed by an actuator. To remove a node, the actuator calls a short cleanup script on that node, for example, copying back logs and stopping the application. To add a node, the actuator will first test its liveness, and then copy the relevant files (the application and Rhizoma) before starting Rhizoma. If Rhizoma runs successfully on the new node, it connects to other members of the overlay using a seed list passed by the actuator, replicates the overlay status into its knowledge base, and starts the application.

All nodes in Rhizoma's overlay run a failure detector to identify failed nodes. If the coordinator itself fails, a new one is elected and takes over running the constraint program. To handle temporary network partitions and the possible situation of multiple coordinators, each node maintains a "long tail" list of failed nodes that it attempts to re-contact. If a failed node is contacted, the node sets will be merged, a new coordinator elected, and the reasoning process restarted.

5 Implementation

In this section we provide an overview of the current Rhizoma runtime system, as implemented for PlanetLab. Rhizoma is implemented in Python, using the ECL$^{\text{i}}$PS$^{\text{e}}$ [5] constraint solver (which is written in C). We currently assume the presence of a Python runtime on PlanetLab nodes, although Rhizoma is capable

of deploying Python as part of the application, and a port to Windows uses this technique. Rhizoma is built in a framework loosely inspired by the OSGi module system, allowing sensors, actuators, the routing system, CLP engine, and other interfaces to be easily added or removed. The bulk of the runtime is a single-threaded, event-driven process, with the CLP engine (see below) in a separate process that communicates over a local socket.

5.1 Use of ECL^{i}PS^e

Our implementation uses the ECL^{i}PS^e constraint solver, which is based around a Prolog interpreter with extensions for constraint solving, and a plugin architecture for specialized solvers (such as linear or mixed-integer programming). At present, Rhizoma uses only the core CLP functionality of the solver.

We also use ECL^{i}PS^e to hold the knowledge base. Status information about the nodes and the connectivity between them, sensor data and overlay membership are stored in the form of Prolog facts – expressions with constant values that can easily be queried by means of the term and field names. Since ECL^{i}PS^e is based on logic programming, it is easy to unify and fuse data from different sources by specifying inference rules, the equivalent of relational database views. This provides writers of constraint programs with logical data independence from the details of the sensor information and its provenance, and also provides Rhizoma with a fallback path in case of incomplete data.

ECL^{i}PS^e runs on each node in the Rhizoma overlay, but only the coordinator executes the constraint program. This approach is suitable for an environment such as PlanetLab with well-resourced nodes and a uniform runtime environment, and it is useful to have the knowledge base available on each node. We discuss relaxing this condition for heterogeneous overlays in Section 8.

5.2 PlanetLab sensors and actuators

Sensors: Our PlanetLab implementation of Rhizoma uses three external information sources: the PlanetLab Central (PLC) database, the S3 monitoring service [22], and CoMon [14]. S3 provides Rhizoma with connectivity data (bandwidth and latency) for any two PlanetLab nodes. The CSV text format of a complete S3 snapshot is about 12MB in size, and is updated every four hours. CoMon provides status information about individual nodes and slices, such as free CPU, CPU speed, one-minute load, DNS failure rates, etc. The text format of a short CoMon node-centric and slice-centric view is about 100kB, and is updated every five minutes. PLC provides information which changes infrequently, such as the list of nodes, slices, and sites.

Rhizoma also measures a subset of the information provided by S3 and CoMon for nodes that are currently in the overlay. This data is more up-to-date, and in many cases more reliable. Inter-node connectivity and latency on Rhizoma's overlay is measured every 30 seconds, and the results are reported back to the coordinator, together with current load on the node as a whole,

obtained by querying the local CoTop daemon[4]. Rhizoma's rules privilege more frequently-updated information over older data.

Actuators: Rhizoma uses a single actuator on PlanetLab for acquiring and releasing virtual machines. Releasing a VM is straightforward, but adding a new node is a complex process involving acquiring a "ticket" from PLC, contacting the node to create the VM, and using an SSH connection to transfer files and spawn Rhizoma. Failures and timeouts can (and do) occur at any stage, and Rhizoma must deal with these by either giving up on the node and asking the solver to pick another, or retrying. Rhizoma models this process using a state machine, allowing all deployment operations to run concurrently, rather than having to wait for each action to complete or timeout before proceeding.

The actuator is naturally highly platform-specific. An experimental actuator for Windows clusters uses an entirely different mechanism involving the `psexec` remote execution tool, and we expect node deployment on Amazon EC2 to be a more straightforward matter of XML RPC calls.

5.3 Discussion

PlanetLab is, of course, not representative of commercial utility computing, though it shares many common features. PlanetLab is more dynamic (with nodes failing and performance fluctuations), and so has been useful in revealing flaws in earlier versions of Rhizoma. Current and future commercial utility-computing platforms will (one hopes) be more predictable.

Different providers also have different methods of deploying software, something that Rhizoma in the future must handle gracefully. We have started work on a cross-platform Rhizoma, using PlanetLab and clusters, which picks an appropriate node deployment mechanism at runtime, but do not present it here.

However, cloud computing is still in its infancy and PlanetLab is perhaps a more interesting case than current providers in aspects other than reliability and deployment. Note that Rhizoma does not need to deal with specific nodes and is just as capable of dealing with generic "classes" of nodes when running. Where PlanetLab offers several hundred distinct, explicitly named nodes, commercial providers typically offer a small number of node classes (for example, EC2 currently offers five node types, each available in the US or Europe).

As utility computing evolves, we expect to see many more deployment alternatives in the commercial space, and increasingly complex pricing models (as we have seen with network connectivity). An open question is whether the external measurement facilities seen in PlanetLab will be duplicated in the commercial space. If not, Rhizoma would have to rely on its own measurements.

6 Evaluation

Rhizoma is a (rare) example of a system not well served by controlled emulation environments such as FlexLab [18]. Since Rhizoma can potentially choose to

[4] CoTop is the per-node daemon responsible for collecting information for CoMon.

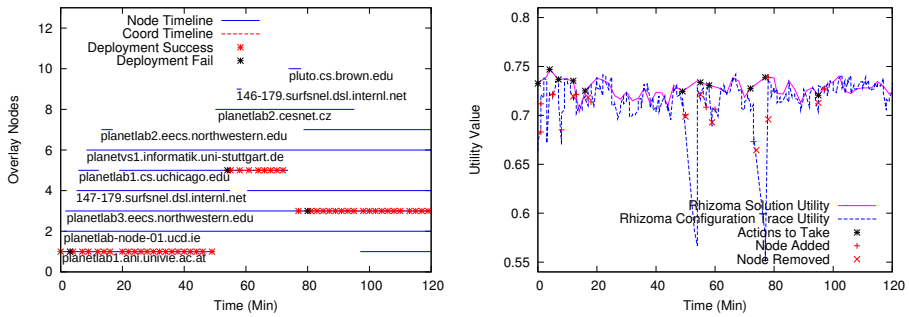


Fig. 4: Short trace deployment timeline and utility

deploy on any operational PlanetLab node (there are more than 600), realistic evaluation under FlexLab would require emulating all nodes, a costly operation and not something FlexLab is designed for.

We adopt an approach conceptually similar in some respects. We deploy Rhizoma with PsEPR application requirements (though in this experiment no "real" application) on PlanetLab for about 8 hours to observe its behavior, and log three sources of information:

1. All local measurements taken by Rhizoma, the coordinator's actions, and overlay status. This includes per-node CoTop data, per-link overlay latency, the coordinator's decisions, and successful or failed deployment attempts. This logging is performed by Rhizoma and backhauled to our lab.
2. The results of querying CoMon, S3, and PLC (as Rhizoma does) during the period of the trace. Unlike Rhizoma, we perform this centrally.
3. For this trace, we also run a measurement slice on all usable PlanetLab nodes, which performs fine-grained measurement of inter-node latency. We expect this to resemble the measurements taken by Rhizoma (1), but the extra coverage represents more of a horizon than is available to Rhizoma. This trace is stored on the nodes and transferred to our lab after the experiment.

Unless otherwise stated, our constraints, utility and cost function are as in Figure 1. For space reasons, we focus on one trace; other results are similar.

6.1 Basic performance measures

We first consider the initial two hours of the overall trace. Figure 4 shows Rhizoma's behavior during this period. From the timeline in the left graph, we see that Rhizoma ran on a total of ten different nodes, and redeployed 18 times, with the coordinator changing three times. The right graph explains Rhizoma's behavior; it shows both the utility value and actions taken by Rhizoma to change the overlay. *Configuration utility* depicts the actual performance of the Rhizoma configuration. Whenever Rhizoma detects a significant opportunity to improve the utility, it generates a new deployment plan (shown by points marked *actions to take*) which is then executed as node additions and removals. *Solution utility*

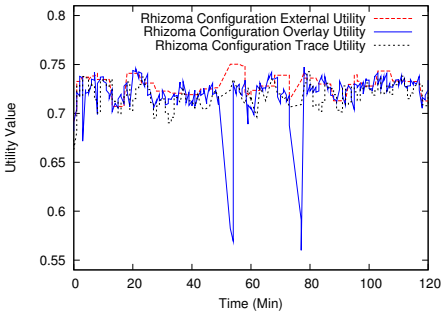


Fig. 5: Measures of utility

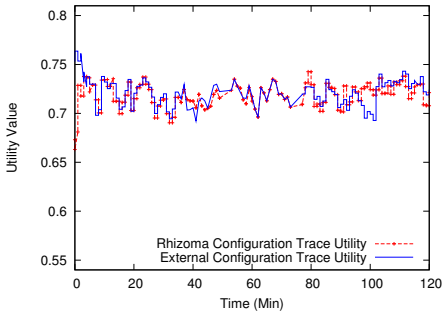


Fig. 6: Effect of overlay monitoring

shows what the solver expects the utility value to be after taking the actions. As we can see, the configuration follows this expectation but does not exactly meet it due to variance between the sensor data and actual node performance.

The two sharp drops in utility are due to short periods of very high latency (one more than three seconds) observed to the coordinator node. In both cases, Rhizoma responds by redeploying, although the first redeployment attempt fails.

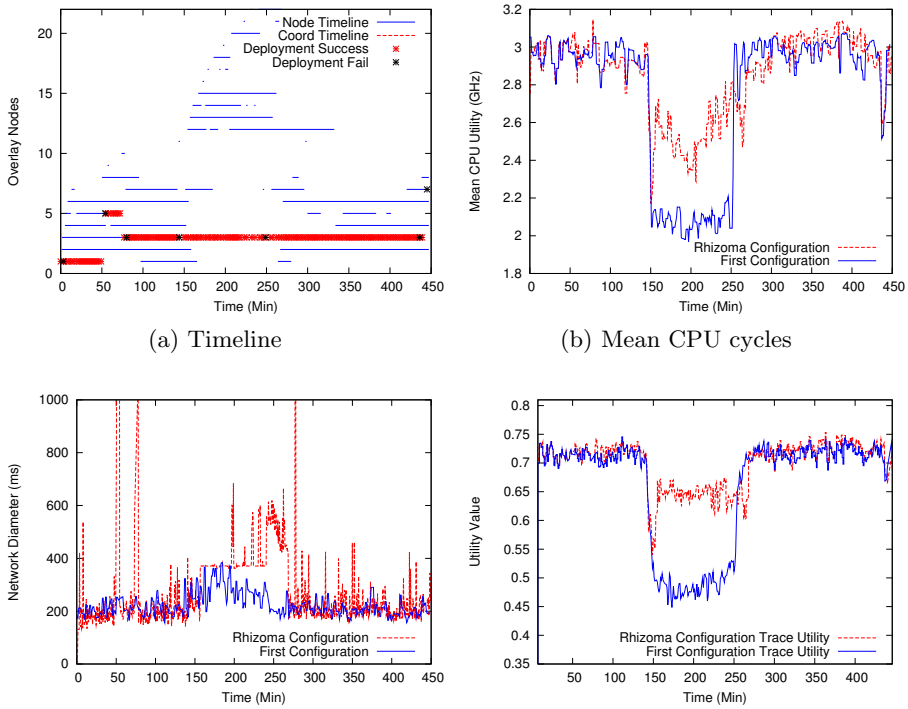
6.2 Different measures of utility

Rhizoma attempts to move its configuration to one that maximizes an objective function (utility minus cost), which expresses the cost of deploying on new nodes or vacating old ones. Utility is a measure of the value of a given configuration, but since this is itself a function of machine and network conditions, it can be calculated in different ways.

Figure 5 shows the utility of Rhizoma's actual configuration for the trace duration, as calculated using different information sources. *Overlay utility* uses the information Rhizoma uses for optimization: CoMon and S3 data, plus its own real-time overlay monitoring results – this is Rhizoma's view of itself, and matches the *configuration utility* in Figure 4. *External utility* users only CoMon and S3 data, and excludes Rhizoma's overlay measurements. This is how Rhizoma's performance appears to an observer with access to only the external monitoring information. As CoMon updates every five minutes, and S3 every four hours, the utility value changes less frequently. *Trace utility* is based on CoMon, S3, and our detailed PlanetLab-wide trace data.

As expected, the overlay and trace utilities are almost identical, since Rhizoma is in this case duplicating data collected by the monitoring slice, and both are reflected by the external utility. However, we also see that the trace utility lags behind the overlay utility, since Rhizoma's monitoring information is updated every 30 seconds, whereas the trace data is updated once per minute (due to the overhead of measuring latency between *all* pairs of PlanetLab nodes).

Furthermore, only Rhizoma observes the sharp spikes in latency to the coordinator. An observer or management system using the external data would not have noticed this problem. Under extreme conditions, this effect may lead to Rhizoma taking actions that would appear detrimental to an external observer.



(a) Timeline

(b) Mean CPU cycles

(c) Overlay network diameter

(d) Total utility

Fig. 7: Adaptivity to unpredictable event versus first configuration

6.3 Effect of overlay monitoring

Rhizoma's resource allocation decision-making is integrated with the application, rather than relegated to a separate management machine. One potential advantage is that Rhizoma can use real-time measurements of application performance in addition to externally-gathered information about PlanetLab.

We use our trace data to simulate Rhizoma without overlay data. Figure 6 shows a somewhat negative result: compared to the full system (*Rhizoma configuration*), the achieved utility of the simulation (*external*) is similar. While we believe that high-level application information can still be beneficial, it seems that in this case Rhizoma has little to gain from its own overlay measurements.

6.4 Adaptivity to failure

Figure 7 shows a larger and more dramatic section of the complete trace. At about 150 minutes, a buggy slice on PlanetLab caused a large increase in CPU usage across many nodes. Our utility function in this deployment favors available CPU over network diameter. As Figure 7a shows, this caused Rhizoma to redeploy from nodes in Europe to the US and Asia (the coordinator remains up, since although starting in Vienna by this point it was running in the US).

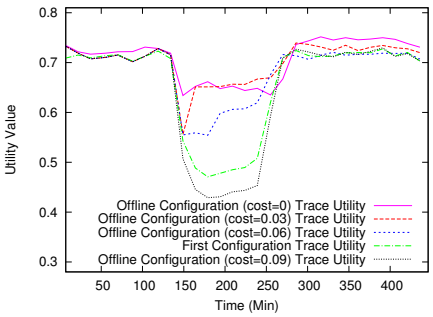


Fig. 8: Sensitivity to cost function

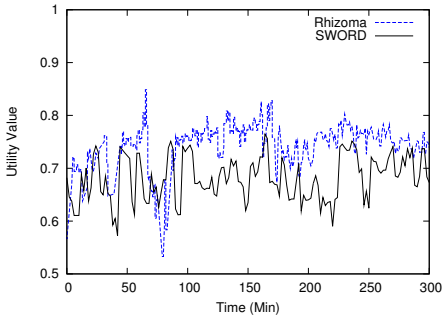


Fig. 9: Comparison with SWORD

Figure 7b shows the mean CPU availability on the node set during the event. After an initial drop, Rhizoma's redeployment recovers most CPU capacity in a few minutes, and continues to optimize and adapt as conditions change. By comparison, the mean CPU availability across the nodes in the initial stable configuration has dropped by more than 30%. The tradeoff to enable this is shown in Figure 7c: for the duration of the event, the overlay diameter increases by about 80% as Rhizoma moves out of Europe. After two hours, more CPU capacity becomes available and Rhizoma moves back, reducing the overlay diameter to its former value. The overall effect on utility is shown in Figure 7d.

This reaction to a sudden, transient change in network conditions at these timescales is infeasible with a human in the loop, moreover, it requires no dedicated management infrastructure – indeed, as Figure 7a shows, Rhizoma maintains its service even though *no* node participates in the system for the full duration of the trace. We are unaware of any other system with this property.

6.5 Cost function sensitivity

To explore the effect of the cost function on Rhizoma's behavior, we simulated different cost weights (that is, different values of *addCostParam*). Figure 8 plots the simulated utility value (calculated using the trace data), averaged over 30-minute windows. *First configuration* shows the utility of the first stable configuration achieved by the system, which is equivalent to an infinite cost.

We see that, in general, increasing cost reduces the likelihood that Rhizoma changes nodes, and thus its ability to adapt to changes in resource availability. The simulated Rhizoma with cost weight of 0.09 performs worse than the first configuration because it changed nodes to satisfy the free CPU constraint, but the nodes that it moved to were also affected, and although the first solution happens to perform slightly better during the period of 150–250 minutes, the difference (utility ≈ 0.05) is not great enough to cause it to redeploy. The period around 200–300 minutes shows a situation where the simulated Rhizoma with zero cost found a local maximum, as described in Section 4.4.

6.6 Strawman comparison with SWORD

We next present a comparison with configurations returned by SWORD [12], a centralized resource discovery tool for PlanetLab. Figure 9 shows a trace of the utility function for a Rhizoma overlay. During the trace, we also captured the results of a periodic SWORD query designed to match the Rhizoma constraint program as closely as possible, and use our PlanetLab-wide measurements to evaluate the utility function of this hypothetical SWORD-maintained network.

SWORD is not as expressive as Rhizoma, and in particular does not support network-wide constraints such as diameter, and so we omit these from Rhizoma's constraint program here. Moreover, SWORD cannot consider migration cost. Rhizoma still performs significantly better, largely due to its optimization framework. However, the differences in design and goals between the two systems make this comparison purely illustrative.

6.7 Overhead

Finally, we briefly describe the overhead of using Rhizoma to maintain an application overlay. Rhizoma currently uses link-state routing which, while suitable for the modest (tens of nodes) overlays we are targeting, would need to be replaced for very large overlays, perhaps with a DHT-like scheme.

In the current implementation, a Rhizoma node in a network of size N must send about 100 bytes each minute for failure detection, leader election, and local CoTop information, plus $128 \times N$ bytes for link-state and latency information. Rhizoma must send this information to all $N - 1$ other nodes. For a 25-node network, this therefore results in about 1500 bytes/second/node of maintenance bandwidth, which is roughly comparable with that used in DHTs [17]. Each run of the ECL$^{\text{i}}$PS$^{\text{e}}$ solver takes around five seconds of CPU time.

7 Related Work

Early examples of autonomous, mobile self-managing distributed systems were the "worm" programs at Xerox PARC [19], themselves inspired by earlier Arpanet experiments at BBN. As with Rhizoma, the PARC worms were built on a runtime platform that maintained a dynamic set of machines in the event of failures. Rhizoma adds to this basic idea the use of CLP to express deployment policy, a more sophisticated notion of resource discovery, and an overlay network for routing. We are aware of very little related work in the space of autonomous, self-managing distributed systems since then, outside the malware community. However, the use of knowledge-representation techniques (which arguably includes CLP) in distributed systems is widespread in work on intelligent agents [20], and techniques such as job migration are widely used.

Oppenheimer et al. [13] studied the problem of service placement in PlanetLab, concluding (among other things) that redeployment over timescales of tens of minutes would benefit such applications. While they target large-scale applications, their findings support our motivation for adaptive small-scale services.

In PlanetLab-like environments, management is generally performed by a separate, central machine, although the management infrastructure itself may be distributed [9–11]. The Plush infrastructure [2] is representative of the state-of-the-art in these systems. Plush manages the entire life cycle of a distributed application, provides powerful constructs such as barriers for managing execution phases, performs resource discovery and monitoring, and can react to failures by dynamically acquiring new resources. In addition to its externalized management model and emphasis on application life-cycle, the principal difference between Plush and Rhizoma is that the former's specification of resource requirements is more detailed, precise, and low-level. In contrast, Rhizoma's use of constraints and optimization encourages a higher-level declaration of resource policy.

The resource management approach closest to Rhizoma's use of CLP is Condor's central Matchmaking service [16], widely used in Grid systems. Condor matches exact expressions against specifications in disjunctive normal form, a model similar to the ANSA Trading Service [8]. Rhizoma's specification language is also schema-free, but allows more flexible expression of requirements spanning aggregates of nodes, and objective functions for optimizing configurations.

8 Conclusion and Future Work

We showed that a fully self-managing application can exist on a utility computing infrastructure, dynamically redeploying in response to changes in conditions, according to behavior specified concisely as a constraint optimization program.

A clear area for future work on Rhizoma is in autotuning the cost function based on performance measurements, and feeding application-level metrics back into the optimization process. Also in the near term, we are enhancing Rhizoma to run across multiple clusters and commercial utility computing providers, and to incorporate real pricing information into our cost functions, in addition to continuing to gain experience with using Rhizoma on PlanetLab.

Longer term, the same features of CLP that are well-suited to heterogeneous providers can be used to express additional constraints on which functional components of an application can or should run on which nodes – at present, Rhizoma assumes all nodes in the overlay run the same application software.

The Rhizoma approach is no panacea, and we see a place for both externally and internally managed applications in cloud computing. We have demonstrated the feasibility of the latter approach, and pointed out some of the challenges.

9 Acknowledgements

We would like to thank the anonymous reviewers for their comments, and Rebecca Isaacs and Simon Peter for many helpful suggestions for how to improve the paper.

References

1. R. Adams. PsEPR operational notes. http://www.psepr.org/operational.php, May 2008.
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. Amazon Elastic Compute Cloud. http://aws.amazon.com/ec2.
4. Amazon Web Services. Amazon S3 availability event. http://status.aws.amazon.com/s3-20080720.html, July 2008.
5. K. R. Apt and M. G. Wallace. *Constraint Logic Programming using ECL^{i}PS^e*. Cambridge University Press, 2007.
6. P. Brett, R. Knauerhase, M. Bowman, R. Adams, A. Nataraj, J. Sedayao, and M. Spindel. A shared global event propagation system to enable next generation distributed services. In *WORLDS'04*, Dec. 2004.
7. T. Delaet, P. Anderson, and W. Joosen. Managing real-world system configurations with constraints. In *ICN'08*, Apr. 2008.
8. J.-P. Deschrevel. The ANSA model for trading and federation. Architecture Report APM.1005.1, Architecture Projects Management Limited, July 1993.
9. R. Huebsch. PlanetLab application manager. http://appmanager.berkeley.intel-research.net/, Nov. 2005.
10. T. Isdal, T. Anderson, A. Krishnamurthy, and E. Lazowska. Planetary scale control plane. http://www.cs.washington.edu/research/networking/cplane/, Aug. 2007.
11. J. Liang, S. Y. Ko, I. Gupta, and K. Nahrstadt. MON: On-demand overlays for distributed system management. In *WORLDS'05*, pages 13–18, 2005.
12. D. Oppenheimer, J. Albrecht, D. Patterson, and A. Vahdat. Distributed resource discovery on PlanetLab with SWORD. In *WORLDS'04*, Dec. 2004.
13. D. Oppenheimer, B. Chun, D. A. Patterson, A. Snoeren, and A. Vahdat. Service placement in a shared wide-area platform. In *USENIX'06*, June 2006.
14. K. Park and V. S. Pai. CoMon: a mostly-scalable monitoring system for PlanetLab. *SIGOPS Oper. Syst. Rev.*, 40(1), 2006.
15. L. Peterson, D. Culler, T. Anderson, and T. Roscoe. A Blueprint for Introducing Disruptive Technology into the Internet. In *HotNets-I*, Oct. 2002.
16. R. Raman, M. Livny, and M. Solomon. Matchmaking: Distributed resource management for high throughput computing. In *HPDC7*, July 1998.
17. S. Rhea, D. Geels, T. Roscoe, and J. Kubiatowicz. Handling Churn in a DHT. In *USENIX'04*, June 2004.
18. R. Ricci, J. Duerig, P. Sanaga, D. Gebhardt, M. Hibler, K. Atkinson, J. Zhang, S. Kasera, and J. Lepreau. The Flexlab approach to realistic evaluation of networked systems. In *NSDI'07*, Apr. 2007.
19. J. F. Shoch and J. A. Hupp. The "worm" programs — early experience with a distributed computation. *Commun. ACM*, 25(3):172–180, 1982.
20. K. Sycara, K. Decker, A. Pannu, M. Williamson, and D. Zeng. Distributed intelligent agents. *IEEE Expert*, Dec. 1996.
21. M. Wallace. Constraint programming. In J. Liebowitz, editor, *The Handbook of Applied Expert Systems*. CRC Press, Dec. 1997.
22. 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*, 2006.
23. Q. Yin, J. Cappos, A. Baumann, and T. Roscoe. Dependable self-hosting distributed systems using constraints. In *HotDep'08*, Dec. 2008.