Doctoral Thesis

Declarative resource management for virtual network systems

Author(s):
Yin, Qin

Publication Date:
2013

Permanent Link:
https://doi.org/10.3929/ethz-a-009778948

Rights / License:
In Copyright - Non-Commercial Use Permitted
DECLARATIVE RESOURCE MANAGEMENT FOR VIRTUAL NETWORK SYSTEMS

A dissertation submitted to

ETH ZURICH

for the degree of

Doctor of Sciences

presented by

QIN YIN

Master of Science ETH in Computer Science, ETH Zurich

July 1, 1981

citizen of China

accepted on the recommendation of

Prof. Dr. Timothy Roscoe, examiner
Prof. Dr. Gustavo Alonso, co-examiner
Prof. Dr. Jeffrey S. Chase, co-examiner

2013
This dissertation addresses how to apply declarative techniques to resource management over virtual network systems on the client and the provider side.

More and more distributed applications and services run on physical or virtual compute, storage and network resources that have been rented or otherwise dynamically acquired from resource providers. Example resource providers are cloud computing platforms, grid systems, as well as research testbeds. However, applications deployed to date over these new virtual infrastructures have generally not been written specifically for such platforms. To deal with the resource management challenges, traditional approaches are applied: an external management system or human administration performs management functions. This solution has obvious deficiencies: increased complexity, poor response to resource availability and slow adaptation to application goals and pricing policies.

The first part of this thesis explores a new approach to building and managing distributed applications over virtual infrastructures: the application handles resource management issues itself as a continuous process of optimization. Two key design features of this approach are *fate-sharing between application and resource management* and *declarative resource management*. This approach has been implemented in the Rhizoma decentralized management runtime. With Rhizoma running alongside the application on the same nodes, the application can handle resource management issues itself by acquiring and releasing processing resources autonomously as both external conditions and the needs of the application itself change. This approach is validated and extended in the Anzere personal data storage and replication system. Anzere provides a declarative way to partially replicate data items at scale according to expressive policy specifications.

The second part of the thesis is motivated by the rise of virtual infrastructures which allow users to reserve or use combinations of (virtual) nodes, switches, and network links from underlying shared physical virtual infrastructures ranging from network testbeds to grid computing facilities. This trend has renewed research interest in the design and implementation of resource allocators. Virtual infrastructure resource allocation becomes more challenging as dependencies between resources become more constrained, resources become more diverse and infrastructures scale to large numbers of clients, sites, and network elements.
The basic problem addressed in the second part is how a client of a virtual infrastructure provider requests resources, how the provider allocates such resources, and how the allocation of such resources is returned to the client. Since this area is still relatively new, research work on resource allocation for virtual infrastructures still lacks benchmarks. The thesis suggests a general framework to generate realistic benchmarks for virtual infrastructure resource allocators, and presents as an example a workload generator which is based on a 5-year trace of experiments submitted to the popular Emulab testbed.

To address the virtual network mapping problem, we propose the VF2x mapping algorithm. Several novel algorithmic improvements and careful implementation enable VF2x to allocate resources to virtual networks on a large testbed in a matter of seconds using commodity hardware. VF2x is integrated as the resource allocation solver in Arosa, our resource allocator for network testbeds. Arosa explores a late-binding resource allocation strategy which is enabled by expressing both the resource requests and the resource commitments as constraints. Late-binding resources to requests enables the providers to optimize resource allocation for various metrics such as utilization, and can help to mask failures of physical resources from users as illustrated by the experiments.
Kurzfassung

Diese Dissertation beschäftigt sich mit der Anwendung von deklarativen Techniken zur Ressourcenverwaltung virtueller Netze sowohl für den Client als auch den Provider.


Der zweite Teil dieser Dissertation ist begründet durch den zunehmenden Einsatz von virtuellen Infrastrukturen, welche es dem Benutzer erlauben (virtuelle) Knoten, Switches und Netzwerkverbindungen von physisch gemeinsam genutzten


# Contents

Abstract v  

Kurzfassung vii  

1 Introduction 1  
1.1 Motivation .................................................. 1  
1.1.1 Applications over virtual infrastructures .......... 1  
1.1.2 Virtual infrastructure resource allocators ........ 2  
1.2 Contributions .................................................. 3  
1.2.1 Application resource management ...................... 3  
1.2.2 Infrastructure resource allocation ................... 5  
1.3 Overview ..................................................... 6  

I Application resource management 9  

2 Background 11  
2.1 Virtual infrastructures .................................. 12  
2.2 Application resource management ..................... 13  
2.3 Declarative programming techniques ................... 15  
2.3.1 Constraint programming .............................. 15  
2.3.2 Logic programming ................................. 15  
2.3.3 Constraint logic programming .................... 16  
2.4 Conclusion .................................................. 18  

3 Rhizoma 21  
3.1 Introduction ................................................. 22  
3.2 Using Rhizoma .............................................. 23  
3.2.1 Initial deployment .................................. 23  
3.2.2 The constraint program ............................. 24  
3.2.3 Example: PsEPR .................................... 25  
3.2.4 Rhizoma-aware programs ......................... 28  

ix
### CONTENTS

3.2.5 Observing the application ........................................ 28
3.2.6 Discussion .................................................................. 29
3.3 Operation ...................................................................... 30
  3.3.1 Sensors and knowledge base ....................................... 31
  3.3.2 Coordinator node ...................................................... 32
  3.3.3 Steady-state behavior ................................................ 32
  3.3.4 Optimization process ................................................ 34
  3.3.5 Adding or removing nodes ......................................... 35
3.4 Implementation .............................................................. 36
  3.4.1 Use of ECL\textsuperscript{PS}e ......................................... 36
  3.4.2 PlanetLab sensors and actuators .................................... 37
  3.4.3 Deployment on PlanetLab .......................................... 38
3.5 Evaluation ..................................................................... 39
  3.5.1 Basic performance measures ....................................... 39
  3.5.2 Different measures of utility ........................................ 42
  3.5.3 Effect of overlay monitoring ...................................... 43
  3.5.4 Adaptivity to failures ................................................. 46
  3.5.5 Cost function sensitivity ............................................. 46
  3.5.6 Strawman comparison with SWORD ............................ 47
  3.5.7 Overhead .................................................................. 48
3.6 Related Work ................................................................. 48
3.7 Summary ...................................................................... 49
3.8 Conclusion ...................................................................... 50

4 Anzere policy-based data replication .................................. 53
  4.1 Introduction .................................................................. 54
  4.2 Background and motivation .......................................... 55
  4.3 Target scenario and environment ..................................... 57
  4.4 Policy model ............................................................... 60
    4.4.1 Device- and content-neutral policies .......................... 60
    4.4.2 Policy stratification ................................................. 61
    4.4.3 Replication through constraints .................................. 63
    4.4.4 Acquirable resources ............................................... 65
    4.4.5 Composing Anzere policies ...................................... 67
  4.5 Implementation ............................................................ 68
    4.5.1 Anzere overlay network ............................................ 70
    4.5.2 Knowledge base ...................................................... 70
    4.5.3 CLP solver and equivalence classes ............................ 71
  4.6 Evaluation ................................................................. 73
    4.6.1 Policy sustainability ................................................ 74
    4.6.2 Reactivity ............................................................. 78
CONTENTS

4.6.3 Optimization ........................................... 81
4.7 Summary .................................................. 82
4.8 Conclusion and future work ............................. 83

II Virtual infrastructure resource allocation 85

5 Virtual infrastructure reference model 87
  5.1 Resource allocation reference model .................. 89
  5.2 Resource allocator examples ............................ 90
    5.2.1 Testbeds ........................................... 90
    5.2.2 Grid computing system ............................. 93
    5.2.3 Cloud computing systems ........................... 94
  5.3 Discussion ............................................. 95

6 Realistic benchmarking framework 97
  6.1 Related work of benchmarking ......................... 98
  6.2 General methodology ................................ 98
    6.2.1 Parameter selection ................................ 99
    6.2.2 Trace collection .................................. 100
    6.2.3 Workload characterization .......................... 100
    6.2.4 Workload generation ................................ 101
  6.3 Emulab trace-based workload generator ............... 101
  6.4 Summary ............................................... 103

7 VF2x virtual network mapping algorithm 105
  7.1 Background and related work .......................... 106
  7.2 VF2x algorithm and implementation ................... 108
    7.2.1 Implementation decisions ........................ 108
    7.2.2 VF2x with a candidate queue ..................... 111
    7.2.3 Heuristic sorting of network topologies .......... 111
    7.2.4 Timeout-and-relax for $\epsilon$ ................... 113
    7.2.5 Batching .......................................... 113
    7.2.6 Algorithm evaluation with synthetic workload .... 114
  7.3 Evaluation with realistic testbed workloads ......... 116
    7.3.1 Realistic test workload ............................ 117
    7.3.2 Additional heuristics ............................. 117
    7.3.3 Repeated requests to exhaust physical resources .. 120
    7.3.4 Long trace behavior ................................ 124
  7.4 Conclusion ............................................ 126
CONTENTS

8 Arosa 127
  8.1 Introduction ............................................. 128
  8.2 Approach ................................................. 131
    8.2.1 Resource commitments as constraints ................. 131
    8.2.2 Resource negotiation with constraints ................ 132
    8.2.3 System architecture ................................. 134
    8.2.4 Discussion ........................................... 136
  8.3 Arosa with ECL'PS^e solver ............................. 138
    8.3.1 Initial results ....................................... 138
    8.3.2 Conclusion ............................................ 140
  8.4 Arosa with VF2x ......................................... 141
    8.4.1 Resource assignment with bound tickets ............... 141
    8.4.2 Resource assignment with unbound tickets ............. 142
  8.5 Evaluation ............................................... 143
    8.5.1 Operating in the partial mode ......................... 144
    8.5.2 Operating in the complete mode ....................... 146
    8.5.3 Benefits of late-binding ............................. 149
  8.6 Conclusion and discussion ................................ 151

9 Conclusions 153
  9.1 Application resource management .......................... 153
  9.2 Virtual infrastructure resource allocation ............... 154
  9.3 Directions for future work ................................ 155
1 Introduction

1.1 Motivation

More and more distributed applications and services run on compute, storage and network resources that have been rented or otherwise dynamically acquired from resource providers. The clearest example is the rise of cloud computing; providers such as Amazon support companies who have outsourced the provisioning of (virtual) servers, storage services or messaging services for their applications. It also applies to more research-oriented environments like PlanetLab, and it is the explicit model for network-layer services in testbed platforms such as GENI. In this thesis, we use the term virtual infrastructure to refer to all cloud, utility, grid computing platforms as well as research testbeds. The development of virtual infrastructures provides distributed applications and services an opportunity to run on a set of dynamically acquirable resources. This development also brings new challenges in designing and implementing resource allocators for such infrastructures. This dissertation addresses how to apply declarative techniques to resource management over virtual infrastructures on the client (application) and the provider (resource allocator) side.

1.1.1 Applications over virtual infrastructures

The emergence of virtual infrastructures provides distributed applications and services an opportunity to run on rented or dynamically acquired compute, storage
and network resources, instead of using a conventional, fixed infrastructure which has to be deployed and maintained by the application or the service providers themselves. Applications deployed to date over these new virtual infrastructures, however, have generally not been written specifically for the characteristics of such platforms.

At the same time, while potentially reducing deployment and management cost, utility computing leaves most traditional management problems unsolved and, furthermore, introduces new ones. For example, service offerings and pricing structures for infrastructures have changed substantially. Changes in the application load still require the application to acquire more resources to handle increased load or release them to reduce cost. Partial or complete machine failures still occur. Traditionally, an external management system or human administration performs management functions to deal with the application resource management challenges.

This solution has obvious deficiencies: slow response time to critical events, slower or less appropriate adaptation to resource availability and pricing changes with human-in-the-loop, furthermore, the cost of deploying and maintaining a separate management system (albeit a much smaller one) on dedicated machines to manage the virtual infrastructure used by the application.

The first part of the dissertation tries to answer the question: how do virtual infrastructures lead to both new opportunities and new challenges in building and managing distributed applications? Specifically, we will focus on how to effectively write flexible, adaptable applications directly targeting at such deployment platforms. We explore a different approach to building and managing distributed applications over virtual infrastructures: the application handles resource management issues itself as a continuous process of optimization. The self-managing application will perform resource management (in effect, monitoring and interacting with one or more infrastructure providers) and deployment on new nodes autonomously by itself.

1.1.2 Virtual infrastructure resource allocators

The rise of virtual infrastructures has also renewed research interest in the design and implementation of resource allocators which allow users to reserve or use combinations of (virtual) nodes, switches, and network links from underlying shared physical virtual infrastructures ranging from network testbeds to grid computing facilities.

As dependencies between resources (switch ports, communication links, virtual machines) become more constrained, resources become more diverse (specialized switches, programmable middleboxes, etc.) and infrastructures scale to large numbers of clients, sites, and network elements, it becomes more difficult for clients to
express their requests to the providers of the infrastructure, and, in turn, for these
providers to allocate resources in a way that makes efficient use of the infrastruc-
ture.

This motivates the second part of the thesis. The basic problem addressed in
this part is: how a client of a virtual infrastructure provider requests resources,
how the provider allocates such resources, and how the allocation of such resources
is returned to the client. “Resources” in this sense include virtual machines, virtual
switches and routers, shares of physical network links, and the like.

This involves research work in a number of related areas. One example is the
design of efficient network mapping algorithms: how to map a virtual network
topology with resource constraints to specific nodes and links in a given shared
physical network infrastructure so as to accurately emulate the network properties
requested by the user. Another is the design and implementation of resource allo-
cators: how users can specify their resource requirements, how resource providers
can satisfy many users while optimizing utilization, revenue, or some other metric
of interest.

Though closely-related, the client and the provider have different design goals
to manage the resources over virtual infrastructures. The client focuses on sat-
isfying its own resource requirements and maximizing its utility in the face of
external condition changes as well as its own resource requirement changes; while
the provider focuses on satisfying as many clients’ resource requirements as possi-
ble and maximizing the utilization of the whole infrastructure.

1.2 Contributions

The main contributions of this dissertation are associated with two topics: how to
build and manage distributed applications over virtual infrastructures and how to
design and implement virtual infrastructure resource allocators. In the following
section, we outline the main contributions from these two aspects.

1.2.1 Application resource management

In the first part of the thesis, we explore and evaluate an alternative approach
whereby the application handles resource management issues itself as a continu-
ous process of optimization.

Fate-sharing between application and resource management

The resource management logic is bundled into the application itself, so that it
is more closely integrated with the rest of the application logic. This fate-sharing
design obviates the need for a separate management console and removes any single
point of failure by turning the application into a self-managing and self-deploying system reminiscent of early “worm” programs. It ensures that the application and its resource management are yoked together, so that they either fail together or not at all.

**Declarative resource management**

Constraint logic programming (CLP) provides a natural way to express desired application behavior with regard to resources and can be applied to simplify the task of autonomous resource management. The application expresses its resource requirements to the runtime as a constrained optimization problem in CLP. The CLP solving engine then fuses multiple real-time sources of resource availability data, from which it decides to acquire or release processing resources (such as virtual machines). The management runtime then takes actions accordingly and redeploys the application autonomously to continually maximize its utility as both external conditions and the needs of the application itself change.

**Rhizoma decentralized resource management runtime**

Given a specification of compute and network resources, with the Rhizoma runtime an application can autonomously manage the resources acquired from one or more virtual infrastructures. In Rhizoma, resource management and deployment on the new nodes are performed autonomously by the application itself. Using PlanetLab as a “proving ground”, we show that in spite of its flexibility, the Rhizoma resource management runtime results in better application performance than a centralized, external management system, and can adapt application deployment to its resource requirements in real time.

**Policy model and solver for personal data replication**

The novelty of the policy model and solver lies in three aspects: first, replication policies for personal data can be written independently of specific storage devices and can even result in the system acquiring and releasing virtual storage resources on-demand; second, policy goals can be preserved by the system, and can therefore react to changes in the environment such as the creation of new data items, resource failures or network outages, etc.; third, rich policy calculations can be scaled up to a large number of data items with only modest requirements in memory and computation, by using equivalence classes. The proposed policy model and solver are validated in the Anzere personal data storage and replication system. With the model and solver, the range of allowable personal data replication policies can be dramatically increased without sacrificing scalability.
1.2. CONTRIBUTIONS

1.2.2 Infrastructure resource allocation

The second part of the dissertation considers several questions around the design and implementation of resource allocators for virtual infrastructures.

**Virtual infrastructure resource allocation reference model**

The virtual infrastructure reference model pretty much encompasses any existing resource allocation system. The reference model defines the design space of resource allocators and helps us to survey how resource allocation is done today in three main application domains: testbeds, grid and cloud computing systems.

**Realistic benchmarking framework**

The benchmarking framework provides a general methodology for generating realistic benchmarks. According to the framework, a testbed workload generator is implemented. The generator is based on a 5-year trace of experiments submitted to the popular Emulab testbed.

**VF2x virtual network mapping algorithm**

VF2x is based on the VF2 subgraph isomorphism detection algorithm. VF2x performs network mapping more than two orders of magnitude faster than the previously-published vnmFlib (also based on VF2) but reduces solving time for near-worst-case problem instances through more careful implementation and several novel algorithmic changes: constructing a candidate queue, applying heuristic sorting, and introducing a new “timeout-and-relax” strategy.

**Late-binding resource allocation strategy**

Virtual infrastructure users provide a declarative description of their desired resources as constraints, and the providers reply with resource reservation promises expressed also as a set of constraints on resources instead of specific resources. This commitments-as-constraints strategy gives the providers more flexibility in late-binding resources to requests, and opens up a wide design space to optimize resource allocation for efficiency, cost, utilization, or other metrics.

**Arosa resource allocator for network testbeds**

This commitments-as-constraints idea is integrated in the design and implementation of Arosa, a resource allocator for network testbeds. With VF2x as the solver, Arosa performs network embedding in a shared, distributed testbed in a matter of seconds on commodity hardware. Late-binding resources to requests enables the providers to optimize resource allocation for various metrics, such as utilization, and can help to mask failures of physical resources from users as illustrated by the experiments.
1.3 Overview

This dissertation addresses declarative resource management over virtual infrastructures and has two parts: Part I (Chapter 2 to 4) addresses the problem from the application’s point of view and Part II (Chapter 5 to 8) addresses the problem from the provider’s perspective.

Chapter 2 illustrates the current status of virtual infrastructures, discusses the opportunities and challenges of building and managing distributed applications over these platforms, and analyzes several existing declarative programming techniques. This sets the background for the first part of the dissertation.

Chapter 3 explores a new approach to building and managing distributed applications over virtual infrastructures: the application manages itself as a continuous process of optimization. Fate-sharing between application and resource management and declarative resource management are the two key design features of the approach. The Rhizoma decentralized management runtime is an implementation of this approach. With Rhizoma runtime running on the same nodes, the application can manage itself by acquiring and releasing processing resources autonomously as both external conditions and the needs of the application itself change.

The functionality of Rhizoma is extended in Chapter 4 in two ways: first, the dynamic resource management mechanism applied in Rhizoma is used to manage distributed storage resources; second, the application can use the facilities of the knowledge base as well as the solver to drive its own behavior by making decisions about application logic based on application status. These two extensions are explored in the context of the Anzere personal data storage and replication system. Anzere provides a declarative way to partially replicate data items at scale according to expressive policy specifications.

Chapter 5 sets the background for the second part of the dissertation. It focuses on how to apply declarative techniques to resource management over virtual infrastructures on the provider side. It first introduces a general reference model for virtual infrastructure resource allocation, then surveys various resource allocation systems in the context of the proposed reference model, and in the end points out some research challenges in the field.

Since the area is still relatively new, research work on resource allocation for virtual infrastructures still lacks benchmarks. Chapter 6 suggests a general framework to generate realistic benchmarks for virtual infrastructure resource allocators,
and presents, as an example, our workload generator which is based on a 5-year trace of experiments submitted to the popular Emulab testbed.

Chapter 7 addresses the challenge of designing an efficient virtual network mapping algorithm and proposes VF2x virtual network mapping algorithm. Several novel algorithmic improvements and careful implementation enable VF2x to allocate resources to virtual networks on a large testbed in a matter of seconds using commodity hardware. This is evaluated by an extensive test workload generated from our Emulab trace-based workload generator.

We developed VF2x in the process of implementing our own resource allocator for network testbeds called Arosa. Arosa is introduced in Chapter 8, its design applies the idea of commitments-as-constraints, and its implementation integrates VF2x virtual network mapping algorithm as the solver. Arosa is also evaluated by a test workload generated by our Emulab trace-based workload generator.

Chapter 9 summarizes the dissertation and discusses several possible future research directions.
Part I

Application resource management
This first part of the thesis will focus on declarative resource management of distributed applications over virtual infrastructures.

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

However, applications deployed to date over these new virtual infrastructures (whether commercial services or research projects) have generally not been written specifically for the characteristics of these platforms. The applications just use the platforms as a substitute for a conventional, fixed infrastructure.

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

This chapter will provide some background about virtual infrastructures, state the problem of resource management of distributed applications running over these platforms, and review the related work in this field.

The following two chapters will present how this problem is addressed in two real systems: the Rhizoma decentralized resource management runtime and the Anzere personal data replication system.
2.1 Virtual infrastructures

More and more distributed applications and services run on compute, storage and network resources that have been rented or otherwise dynamically acquired. The clearest example is the rise of cloud computing; providers such as Amazon support companies who have outsourced the provisioning of (virtual) servers, storage services or messaging services for their applications. It also applies to more research-oriented environments like PlanetLab [Pla12], and it is the explicit model for network-layer services in testbed platforms such as GENI [Gen12]. In short, the term virtual infrastructure, which we use across the thesis, has a broad definition which includes all cloud, utility, grid computing platforms as well as research testbeds.

Commercial “cloud computing” facilities like Amazon Elastic Compute Cloud (EC2) [Ec12] provide a managed, low-cost, stable, scalable infrastructure for distributed applications and are increasingly attractive for a variety of systems. Back in 2007 Amazon EC2 only offered a choice of 4 standard node types. Later it supported the selection of different continental locations. As we expected, the selection did become increasingly complex with the emergence of different service offerings and pricing models. Nowadays, besides standard instances, EC2 offers more diverse ones: high-CPU, high-Memory, high-IO, Micro, Cluster Compute and Cluster GPU, on-demand as well as reserved instances, all with different pricing models.

Commercial “cloud storage” facilities such as Amazon Simple Storage Service (S3) [S12], allow users to store and retrieve any amount of data, at any time, from anywhere on the web. Different fee rates are set to store one gigabyte per month with different redundancy levels, to accomplish different requests (PUT, COPY, POST, LIST or GET) to the data stored in S3 and to transfer data out of an Amazon S3 region. Amazon Simple Queue Service (SQS) [Sqs12] offers a hosted queue for storing messages as they travel between computers. Different pricing is set for every 10,000 Amazon SQS requests, or to transfer data out of Amazon SQS.

These cloud and utility services offer attractive features: cost of maintenance and administration is centralized and amortized over multiple clients; the clients pay only for capacity that they actually use, and there is no minimum fee; hardware provisioning is decoupled from software deployment allowing cost savings and responsiveness to changes in demand. For example, clients are able to obtain and boot new server instances within minutes so that applications can quickly scale capacity, both up and down, as their computing requirements change.

As utility computing evolves, we expect to continue to see more deployment alternatives in the commercial space, and increasingly complex pricing models (as we have seen with network connectivity provision).
PlanetLab offers several hundred (1137 up to October 2012) distinct, explicitly named, widely distributed (546 sites) nodes. PlanetLab is a very different environment from cloud providers like EC2 in several important respects. Firstly, PlanetLab nodes are more diverse in hardware, location, connectivity (campus or corporate network from different institutions) than current cloud offerings. Secondly, PlanetLab provides detailed third-party monitoring services for all its resources (node status as well as network connectivity). Thirdly, PlanetLab is much less stable than services like EC2, as frequent node failures and network outage make it an excellent source of trouble.

Testbed platforms like GENI allow the allocation of virtual networks in addition to computing resources. They provide more diverse resources: virtual machines, programmable switches in the backbone and at the edges, Layer 2 or Layer 3 communication links, etc. Clients can program to set up the network topologies they need and to control flows in the network.

2.2 Application resource management

While potentially reducing deployment and management cost for applications - deployment of hardware and upgrades of software are no longer an issue - utility computing leaves most traditional management problems unsolved and, furthermore, introduces new ones. We focus on the following three:

Firstly, while utility computing providers such as Amazon aim to provide a stable execution environment, service offerings and pricing structures for infrastructures have changed substantially [Gar07]. As more infrastructure providers enter this commodity market, we expect to see further differentiation on service offerings and pricing models. Applications who wish to deploy on such infrastructures will need to select appropriate providers and resources to minimize their deployment cost, to adapt the deployment in the face of pricing changes, as well as to probably use more than one provider at a time for reliability reasons.

Secondly, changes in application load, performance goals, or other policies still require the application to acquire more resources to handle increased load or changed performance goals, or release them 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, partial or complete machine failures still occur. Although Amazon Web Services (AWS) aim at high availability, service level agreements are remarkably vague about the uptime guarantees provided. In fact, such services are not totally reliable as recent Amazon disruptions show [AWS11, AWS12]. Of course, platforms like PlanetLab are often much less available than in-house hosted hard-
Dealing with those application resource management challenges is traditionally a management function, performed by an external management system and/or human administration; 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). This solution has obvious deficiencies: slow response time to critical events, slower or less appropriate adaptation to resource availability and pricing changes with human-in-the-loop, furthermore, the cost of deploying and maintaining a separate management system (albeit a much smaller one) on dedicated machines to manage the virtual infrastructure used by the application.

A number of projects and products provide such centralized management services for distributed platforms including the Grid, testbeds and cloud infrastructures.

Cloud management solutions such as RightScale [Rig12] and Scalr [Sca12] offer some common core features: managing cloud infrastructure from multiple providers across public, private and hybrid clouds; creating and provisioning new objects (servers, storage, and/or applications) and destroying unnecessary ones on-the-fly; monitoring and reporting the execution status such as uptime, response time, quota use, etc.

Similar services are available for testbeds and the Grid. The AppManager [Hue08] package helps to centrally manage, install, upgrade, start, stop, and monitor applications on a PlanetLab slice. Sword [OAPV04] provides scalable resource discovery for PlanetLab, while Plush [ATSV06] doesn’t only locate the remote resources, but also contacts and prepares them, runs the job on top, handles the clean-up, and performs several other functions related to distributed application management. SmartFrog [AGP03] is used to deploy and configure Grid applications using the Globus tool-kit [FK96]. Condor [Con12] provides workload management for compute-intensive jobs running on the Grid.

In autonomic computing this centralized management function may be referred to as an “orchestration service” [IBM05], 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 application providers deploying smaller services where the cost of additional dedicate nodes is hard to justify.

This led us to investigate a different approach, to assess whether handling resource management issues within the application is both feasible and desirable. The self-managing application will perform resource management (in effect, monitoring and interacting with one or more infrastructure providers) and deployment on new nodes autonomously by itself.
2.3 Declarative programming techniques

In this thesis, we are investigating how declarative programming techniques can help with resource management in network systems. On the one hand, declarative programming techniques are applied to distributed applications to exploit the resources of modern heterogeneous networks and computing facilities as will be discussed in Chapter 3 and 4; on the other hand, declarative programming techniques are also used by resource clients and providers to negotiate for resources as will be explained in Chapter 8.

Declarative programming techniques have been playing a powerful role in system management; the declarative property makes it suitable to describe the desired system configuration without getting into the details of how to compute it. In this section, we will briefly survey three important and related declarative techniques: constraint programming, logic programming and constraint logic programming.

2.3.1 Constraint programming

In constraint programming, the program is formalized as a constraint satisfaction problem (CSP), consisting 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). Relations between variables are stated in the form of constraints, specifying the properties of a solution to be found. Constraints between variables can always be expressed as a set of admissible combinations of values. The set of constraints is then solved by assigning a value to each variable such that none of the constraints are violated.

Constraint programming is often used as a complement to other paradigms: functional, logical or even imperative programming. Constraint logic programming, which is introduced later, is actually a combination of logic programming with constraints.

The constraint language approach has been used to specify and match Grid resource requirements [LF04], to manage real-world network configuration from high-level specifications [DAJ08,Nar05], etc.

2.3.2 Logic programming

Logic programming such as Prolog consists of defining relations as a set of facts and rules, and querying about relations through questions. A relation can be specified by facts, stating the n-tuples of objects that satisfy the relation, or by stating inference rules (subgoals) of the relation. The rules are applied to derive new facts as well as to determine whether the facts are sufficient to ensure the truth of the question (goal). Questions/goals are answered (logic programs are
executed) by matching goals against facts or rules, unifying variables with terms, and backtracking when the rules cannot be applied anymore (subgoals fails).

Logic programming normally presents two types of semantics: declarative and procedural. Declarative semantics help the programmer to concentrate on the declarative meaning of the program without being distracted by the executional details. However, procedural semantics cannot be ignored since the order of the goals, facts and rules can affect the efficiency of the program.

Logic programming language is shown to be a natural choice for expressing complex queries about system behavior in a network monitoring system [RMJH02, WPR04]. Declarative logic language is also used to express overlay networks in a highly compact and reusable form [LCH+05], as well as in system management and administration which covers system, network, database and application management [IBM03].

2.3.3 Constraint logic programming

Constraint logic programming (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 restrict and guide the reasoning and search by eliminating unwanted alternatives in advance.

In CLP known elements of a problem are represented by a set of declarations; the problem itself is formulated as a set of constraints that the program needs to meet and an objective function to optimize within those constraints. Permissible or viable solutions are found using algorithmic search methods provided by the applied constraint solver.

Building applications which can acquire, manage, and release their own set of compute/storage resources in response to changing demand and conditions immediately raises the problem of complexity: how can programmers easily write and understand the code which causes the application to do “the right thing”, when it is running in a complex and dynamic environment? 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, formulating the problem as constraints and objective functions 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, or where to replicate the data and how to replicate it, CLP solver treats the task as an optimization problem and uses application and system characteristics to find an optimal solution.

Secondly, as in the resource description framework (RDF) [W3C12b], 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.

Thirdly, logic programming languages like Prolog are slightly more expressive than formats such as RDF [W3C12b], while being considerably more computationally tractable than more heavyweight subsets of first-order logic like description logics [W3C12a]. As researchers this gives us a convenient programming platform to explore the design space for more specialized solutions.

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

Fifthly, unlike RDF query languages, CLP solvers typically perform optimization and provide a natural way to express high-level optimization goals. This allows a programmer to give criteria in the form of an objective function for selecting the best solution which satisfies the constraints, maximizes the performance criteria, and minimizes the cost. This is potentially a very powerful technique as it allows an application to select the cheapest feasible deployment which meets its performance targets.

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.

We are certainly not the first to point out CLP’s applicability to system management problems – indeed, the ECLiPS constraint solver we use was originally developed as the core of a suite of network management applications [AW07]. It is also used in network routing and traffic engineering [OR04], and in operating system device configuration [SBRP11]. However, we are unaware of existing work building such functionality into the application itself, and thereby integrating application knowledge and management information.

ECLiPS constraint programming system is used in our Rhizoma and Anzere resource management systems because of its extensive library support and ease of use, however, its performance is not as good as more recent solvers. In Kotthoff’s comparative study [Kot09] with three other constraint solvers, ECLiPS is largely outperformed by Gecode [Gec12] and Minion [GJM06]. Microsoft Solver Foundation [MSF12] is promising too, supporting programming problems with more than 1000 variables and constraints, which is very important for more complicated optimization problems.
2.4 Conclusion

The above sections survey the current status of virtual infrastructures, the resource management problem of the applications deployed on top, as well as some declarative techniques for resource management.

The rest of the thesis’ first part will explore an alternative approach to the problem of application resource management over virtual infrastructures:

- Fate-sharing between application and resource management: The resource management logic is bundled into the application itself, so that it is more closely integrated with the rest of the application logic. This fate-sharing design [Cla88] ensures that the application and its resource management are yoked together, so that they either fail together or not at all.

- Declarative resource management: CLP is used as a programming interface to the application runtime to make the specification and implementation of such tightly-coupled resource policies tractable.

Several advantages from this approach will be verified in the following two chapters:

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

- 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 resources, application load, or service policies.

- 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).

To verify if handling these issues within the application in a declarative way is both feasible and desirable, two systems adopting this approach are discussed in detail in the following two chapters:

The Rhizoma decentralized management runtime investigates how, given a specification of compute and network resources, an application can autonomously manage the resources acquired from one or more virtual infrastructures. In Rhizoma, resource management (in effect, monitoring and interacting with one or more infrastructure providers) and deployment on the new nodes are performed autonomously by the application itself.
2.4. CONCLUSION

The **Anzere** personal data replication system uses the mechanism applied in Rhizoma to implement application logic. More specifically, Rhizoma manages the application’s compute resources and deploys the application on these resources accordingly based on its resource requirements. Anzere manages the replication system’s storage resources and replicates the data to these storage locations according to its storage requirements. Moreover, Anzere uses the reasoning engine to make decisions not only about resource requirements, but also about the application logic. For example, based on the application status about which data is stored on which devices, Anzere makes decisions on where and how to replicate data based on its replication policies.
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 entails a lack of communication between the application and its resource manager. As investigated in the previous chapter, the combination of a separate external management machine and/or human-in-the-loop monitoring has obvious deficiencies.

This chapter explores an alternative approach whereby the application manages itself as a continuous process of optimization. This approach presents two key design features:

- **Fate-sharing between application and resource management:** a decentralized management runtime closely coupled with the distributed application itself obviates the need for a separate management console and removes any such single point of failure by turning the application into a *self-managing system* reminiscent of early "worm" programs [SH82].

- **Declarative resource management:** the application expresses its resource requirements to the runtime as a constraint optimization problem in constraint logic programming language (CLP). The CLP solving engine then fuses multiple real-time sources of resource availability data, from which it decides to acquire or release processing resources (such as virtual machines). As external conditions or the needs of the application itself change, the management
runtime then takes actions accordingly and redeploy the application autonomously to continually maximize its utility.

The “Rhizoma” 1 decentralized management runtime is the implementation of the above ideas. Using PlanetLab as a challenging “proving ground” for cloud-based services, results are presented showing Rhizoma’s performance, overhead, and efficiency compared to existing approaches. In spite of its flexibility, our approach to resource management results in better application performance than a centralized, external management system; it can also automatically react to unexpected large-scale changes in resource availability and adapt application deployment in real time according to its resource requirements.

The contents in this chapter have been published in HotDep 2008 [YCBR08] and Middleware 2009 [YSC+09].

3.1 Introduction

Deploying and maintaining applications in a cloud or a utility computing environment involves important decisions about which resources to acquire and how to respond to changes in service load, 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 instance types and locations, and the selection will become increasingly complex as multiple providers with different service offerings and pricing models, emerge.

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 the offered load may change, nodes might fail, or an entire service provider may experience a disruption [AWS11, AWS12]. 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 has to be solved by a human operator and takes timescales of hours or even days.

Alternatively, a separate management system is deployed and maintained on dedicated machines to keep the application running and to respond to such events. However, the cost of additional dedicated nodes is hard to justify, especially for application providers deploying smaller-scale services, as argued in the previous chapter.

1Rhizoma is a Greek name for Rhizome. Rhizome’s characteristic of sending out roots and shoots from its nodes well represents our worm-like system which is able to replicate and expand itself across the hosting network.
In this chapter, we explore an alternative model for application management and share 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 failures as the application itself.

In order to eliminate the human element 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) as the most tractable way to express complex application demands.

The next section describes how application providers deploy a service using Rhizoma. In Section 3.3 we detail how Rhizoma operates at runtime to manage the application and optimizes its deployment. Section 3.4 presents our experimental implementation on PlanetLab, and Section 3.5 shows a comprehensive evaluation of the technique using a realistic application scenario in the challenging PlanetLab environment. Section 3.6 covers related work, and finally, we conclude in Section 3.7.

3.2 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 the PlanetLab implementation for concrete details, and use the term “application” to refer to the whole system, and “application instance” or “instance” for the part of the application which runs on a single node.

3.2.1 Initial deployment

To deploy Rhizoma together with 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 3.2.2 below) and short scripts that Rhizoma can call to start and stop the application on a node.
For the application that is already capable of handling components which fail independently and organize into some form of overlay, 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 interactions are 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; a Rhizoma-based partial data replication system can partially replicate data at scale while considering data replication policy specifications as well as the status of the overlay network. Application developers can also benefit from the underlying Rhizoma facilities, as briefly described in Section 3.2.4. More details of this are discussed in the next chapter where the Anzere personal data replication system, which is based on Rhizoma, is introduced.

The packaging for our PlanetLab implementation is the typical tarball approach also used by systems like Plush [ATSV06]. 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 Rhizoma’s relationship with early worm programs in Section 3.6.

### 3.2.2 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. 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 the cost for acquiring and using new resources (like virtual machines) and releasing old ones. As with utility, this function may take into account any information available to Rhizoma. Its definition might range
3.2. USING RHIZOMA

from a constant value to a complex calculation involving pricing structures and node locations.

The constraint program will attempt to find a set of \( c \) nodes which satisfy the constraints while maximizing the value of the objective function, defined as the utility function \( U(N) \) minus the cost function \( C(\Delta) \) based on the current status of the knowledge base.

A straightforward approach to an optimal solution would lead to exponential complexity increases, particularly in a computing environment where the number of node options is very large at any time. For a provider with \( N \) (several hundreds or thousands) live nodes, an optimal overlay of \( c \) nodes would require examining on the order of \( N^c \) possible configurations.

The solution that Rhizoma’s CLP solver generates is not a new application configuration which is optimal, but rather a set of “moves” that improve the current application’s status (the number of “moves” is smaller than the size of the overlay). 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 in additional constraints that limit the number of “moves”.

3.2.3 Example: PsEPR

To provide a concrete example of deploying an application with Rhizoma, we take a publish/subscribe application inspired by the (now defunct) Planetary Scale Event Propagation and Router (PsEPR) service deployed on PlanetLab [BKB+04]. 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 previously-selected 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 [PP06], which includes status information such as node reachability, load, and hardware specifications. Node failures were noticed by human operators, and new nodes had to be picked manually. The set of nodes was reviewed irregularly (about once a month) [Ada08]. The service was discontinued in May 2008 because the work involved in keeping it operational became too hard to justify.
Constraints:

PsEPR is an example of a distributed application with requirements that cannot be expressed simply as the number of nodes or minimum per-node resources. Figure 3.1 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 3.4.2, and include node constraints (which a node must satisfy in order for it to be considered), group constraints (defined over any group of nodes), and network constraints (specifying desired network characteristics).

<table>
<thead>
<tr>
<th>CONSTRAINTS</th>
<th>UTILITY FUNCTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>node_constraint(Host) :-</td>
<td>util_function(NodeList, Util, Params) :-</td>
</tr>
<tr>
<td>comonnode{hostname: Host},</td>
<td>% Compute utility values for different node attributes</td>
</tr>
<tr>
<td>alive(Host),</td>
<td>assemble_values(NodeList, fiveminload, UtilLoadMin, LoadMax, LoadUtil),</td>
</tr>
<tr>
<td>light_loaded(Host),</td>
<td>util_value(&quot;&lt;&quot;, LoadList, LoadMin, LoadMax, LoadUtil),</td>
</tr>
<tr>
<td>is_avail(Host),</td>
<td>% Omitting utility values for liveslices and freecpu, the same as above</td>
</tr>
<tr>
<td>get_node_attr(Host, cpuspeed, Cpuspeed),</td>
<td>% Utility of max distance from fixed nodes to the overlay</td>
</tr>
<tr>
<td>get_node_attr(Host, freecpu, Freecpu),</td>
<td>findall(P, fixednode(P), Fixed),</td>
</tr>
<tr>
<td>Cpuspeed*Freecpu/100 &gt; 1.5.</td>
<td>minlatency(MinLat),</td>
</tr>
</tbody>
</table>

| group_constraint(NodeList) :- | maxneighUtil(_, NeighMax, NeighWeight), |
| assemble_values(NodeList, location, Locs), | get_nearest_neighbor_list(Fixed, NodeList, NeighList), |
| length(NodeList, Len), | max(NeighList, MaxDist), |
| Max is ((Len-1)//4)+1, | util_value("<", [MaxDist], MinLat, NeighMax, NeighUtil), |
| ( for(I, 1, 4), param(Locs, Max) do | % Utility of overlay network diameter |
| count_element(I, Locs, Num), | diameterUtil(_, DiamMax, DiamUtil), |
| Num =< Max ). | util_value("<", Params, MinLat, DiamMax, DiamUtil), |

| path_constraint(LenList, Max) :- | % Weighted average of the utilities above |
| max(LenList, Max), | weighted_avg([LoadUtil, SliceUtil, CpuUtil, NeighUtil, DiamUtil], |
| diameterUtil(_, DiamMax, DiamUtil), | [LoadWeight, SliceWeight, CpuWeight, NeighWeight, DiamWeight], Util). |

<table>
<thead>
<tr>
<th>COST FUNCTION</th>
<th>CONFIGURATIONS</th>
</tr>
</thead>
<tbody>
<tr>
<td>migration_cost(Actions, MigrateCost) :-</td>
<td>fiveminloadUtil(0, 10, 2).</td>
</tr>
<tr>
<td>count_element(add, Actions, AddLen),</td>
<td>livelvesUtil(0, 10, 1).</td>
</tr>
<tr>
<td>count_element(remove, Actions, RmvLen),</td>
<td>freecpuUtil(1, 4, 3).</td>
</tr>
<tr>
<td>addCostParam(AddParam),</td>
<td>maxneighUtil(0, 500, 2).</td>
</tr>
<tr>
<td>removeCostParam(RmvParam),</td>
<td>diameterUtil(0, 1000, 2).</td>
</tr>
<tr>
<td>MigrateCost is AddParam*AddLen +</td>
<td>addCostParam(0.012).</td>
</tr>
<tr>
<td>RmvParam*RmvLen.</td>
<td>removeCostParam(0).</td>
</tr>
</tbody>
</table>

---

Figure 3.1: PsEPR application requirements

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, the five-minute load, and the number of active VMs on the node. Finally,
3.2. USING RHIZOMA

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. 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 not 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 free CPU. We also try to minimize the network diameter, and the maximum latency to the overlay from each of a set of ten manually chosen, geographically dispersed anchor nodes.

utility_function also uses a number of Rhizoma built-in predicates. 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 a specific parameter given the list of values for the parameter, a minimum $x_{min}$ and a maximum $x_{max}$. This function has two variants, $\text{util\_value}_<$ is used when the value to be considered ideal is the minimum (for example, in the case of live slices), and $\text{util\_value}_>$ when the ideal is the maximum (in the case of free CPU). $\text{util\_value}$ computes the utility as an average of the deviations of a parameter $x_i$ from the ideal ($x_{max}$ or $x_{min}$) as defined below the utility_function in Figure 3.1. get_nearest_neighbor_list finds for every node in the list of fixed nodes, the nearest neighbor to it in the overlay, and then we try to minimize the maximum latency from each of the fixed nodes to its nearest neighbor in the overlay. Finally, weighted_avg computes the weighted average of a list of values given their corresponding weights.
CHAPTER 3. RHIZOMA

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 it 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 considering. In Figure 3.1, we 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 3.5.5. The migration cost could also consider application details (such as the cost of copying data and moving data around) and configuration changes. Ideally, it would be learned online by the system over time.

3.2.4 Rhizoma-aware programs

Although we have presented the minimal interface required to deploy existing applications 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 the 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 which could invoke existing Rhizoma module services described above. For example, a listener service is provided by the overlay module to notify applications when nodes fail or join, and of other changes to the overlay. Using the channel communication interface, applications can explicitly send messages to a member in the overlay, or to a set of nodes (broadcast/multicast). Furthermore, applications can also piggyback on Rhizoma’s leadership election for their own purposes.

3.2.5 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
3.2. USING RHIZOMA

“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. Using this data, it is straightforward to build system visualizations, such as the one in Figure 3.2.

![Rhizoma Monitor](image)

Figure 3.2: Visualizing a Rhizoma application

3.2.6 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 separate management service. However, despite their attractiveness, constraint solvers have never been a “magic bullet”.

The first challenge is the computation complexity. It is very easy to write constraint programs with exponential performance curves that become intractable even at low levels of complexity. In Section 3.3.4, we describe one approach to 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 a reasonable time. This is a difficult 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). Based on our experience with PlanetLab, we provide a collection of useful heuristics, 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. If they stick to the high-level constructs we supply, developers are assured of relatively tractable constraint programs, and they need only write a few lines of code. However, the full expressive power of CLP is still available if required.

While the constraints are intuitive, this is less true of the utility and cost functions. The utility function requires developers to map their various intuitive measures which make a good application deployment to a single scalar function, but this is made easier by the relative stability of the solutions to small changes in the utility function. Quantifying the cost function, especially the cost of migration, is hard, and does not correspond to something a developer is generally thinking of as investigated in Section 3.5.5.

### 3.3 Operation

The architecture of a Rhizoma node is shown in Figure 3.3. Along with 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.

The control and data flow of a Rhizoma application form a closed loop starting from an initial deployment (possibly an unoptimized one). By interpreting and realizing the configuration, the application takes actions to acquire and release resources. After the actions become effected, Rhizoma runtime senses the status and yields a new deployment strategy reflecting the best adaptation for resource dynamics. It then feeds the new deployment back to the application and closes this loop.

The rest of this section describes the operation of a Rhizoma system in practice.
3.3. OPERATION

![Rhizoma architecture diagram]

Figure 3.3: Rhizoma architecture

We first introduce the sensors and knowledge base (KB in the figure), two key components of the Rhizoma architecture, and describe the selection and role of the coordinator node. 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.

3.3.1 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, even 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, an extension of this work could timestamp the available data and maintain historical information, such as moving averages, which could be used by constraint programs. A number of logic pro-
gramming languages extend Datalog and support formulating inductive temporal queries [Cho94]. ECLiPSe CLP also supports temporal reasoning with constraint handling rules [Fr94], a feature that could be explored further with time-stamped sensor data.

3.3.2 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. In Rhizoma, 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 (members) maintain only the overlay status. The elected node remains the coordinator as long as it is alive and in the overlay. A new coordinator is elected if the old node crashes, or if the optimization process decides to move to another node.

Because of the computation requirements of running the constraint program, and the storage requirements of storing the knowledge base, only a well-provisioned node can be elected as the coordinator. In our PlanetLab implementation of Rhizoma, this constraint is less of a problem given the fact that the PlanetLab nodes chosen for the overlay are more or less homogeneous, therefore, simply, the node with the lowest IP is elected as the coordinator.

However, in other more heterogeneous networks such as the personal device network, more device attributes (e.g., resource availability, location, owned vs. rented device, etc.) should be considered in the election mechanism. For example, in the Anzere personal data replication system, the coordinator is typically a well-provisioned node such as a home desktop PC; phones and tablets will never be elected as coordinators, as will be explained in detail in the next chapter.

3.3.3 Steady-state behavior

To respond to changes in resource utilization, Rhizoma performs several periodic operations in its steady state:

- network monitoring;
- overlay optimization;
- overlay reconfiguration.

Network monitoring

Sensors in Rhizoma periodically collect resource information about node or network status from external and internal monitoring services, and notify the coor-
3.3. OPERATION

dinator to update the knowledge base accordingly. The periods are based on the characteristics of different sensors, such as their update frequency and data size. Details of PlanetLab external sensors and Rhizoma overlay sensors are discussed in the next section. While the coordinator stores in its knowledge base all the information collected by the monitoring services, it only disseminates real-time overlay information to all the other members in the overlay. Each overlay member also stores in its knowledge base this information whose size is much smaller than the one stored in the coordinator.

Overlay Optimization

Since every member in the overlay knows the current overlay status, it can take decisions to optimize the application’s resource usage in scenarios like overlay formation, overlay communication, data transfer, etc. In the simple example shown in Figure 3.4, the bottom arc depicts the optimization of overlay formation from Configuration1 to Configuration1’. Traffic is detoured to minimize the network diameter.

The Rhizoma PlanetLab implementation supports shortest-path routing for unicast communication and minimum spanning-tree for broadcast communication. Overlay messages can be detoured and forwarded through several overlay hops.

Figure 3.4: Overlay adaptation
In the later Rhizoma implementation for the Anzere personal network, overlay routing also uses application logic to reason about diverse network and device types and provides path optimization based on latency, bandwidth, and price. For instance, using information about available interface types and node-to-node latency, the routing module can set up a minimum latency path “phone-laptop-cloud”, which uses the phone-laptop Bluetooth link and the laptop-cloud Ethernet link. The Dexferizer data transfer optimization service [UR11] extends this even further by optimizing the transfer of data objects within a user’s collection of computers and personal devices, subject to a variety of user-defined quality metrics such as cost, power consumption, and latency.

Overlay reconfiguration

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, the coordinator 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. The top arc in Figure 3.4 illustrates the overlay reconfiguration after the old coordinator node $A$ fails. Rhizoma acquires two new nodes $E$ and $F$, deploys the application and itself to these new nodes and elects $B$ as the new coordinator.

3.3.4 Optimization process

Actions derived from periodic solving include acquiring new nodes and releasing existing ones. In principle, the solver tries to maximize the value of the objective function (i.e., the utility function minus the cost function) based on the current knowledge base and subject to the constraints. In practice, this approach would lead to exponential complexity increases, particularly in a PlanetLab-like environment with 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 $n$ $\text{add}(\text{node})$ or $\text{remove}(\text{node})$ 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, preventing Rhizoma from altering its configuration too much during each period.

In each optimization step, Rhizoma’s solver chooses a combination of the two possible actions $\text{add}(\text{node})$ to add a qualified node or $\text{remove}(\text{node})$ to remove a node from the overlay. The length of each combination is at most $n$. This means that a shorter or empty action list (which leaves the current overlay unchanged)
is also a valid combination. The solver chooses nodes to add and remove in order
to maximize the objective function. This can be done by looking at different
combinations of actions together with different combinations of nodes per action.
Suppose to maintain an overlay with \( c \) nodes in a computing environment with \( N \)
active nodes, in each optimization step we generate at most \( n \) actions (\( x \) add and
\( y \) remove actions). In this scenario, to generate the action plan, the number of
configurations to be examined will be:

\[
\sum_{x=1}^{n} \binom{N}{x} \sum_{y=0}^{n-x} \binom{c}{y}
\]

This set of actions is then passed to the appropriate actuators for execution.

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
the issue can be addressed either by increasing the horizon \( n \), or by using one of
several more sophisticated optimization algorithms from the literature.

### 3.3.5 Adding or removing nodes

The actions chosen by the solver are executed by the actuators. 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 becomes
a candidate node and tries to connect to other members of the overlay using
a seed list passed by the actuator. After successfully joining the overlay as a
member, Rhizoma replicates the overlay status into its knowledge base, and starts
the application on the node.

Although Rhizoma’s solver chooses alive nodes (derived from external sensor
data) on which to deploy, it is still possible that the deployment fails. This may
occur if stale sensor data is used that doesn’t reflect a more recent problem with
the node. In this case, the node will be added to a blacklist stored in the knowledge
base for a period of time.

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.
CHAPTER 3. RHIZOMA

3.4 Implementation

In this section, we provide an overview of the Rhizoma runtime system, as implemented for PlanetLab. Rhizoma is implemented in Python, using the constraint solver ECL\textsuperscript{i}PS\textsuperscript{e} [AW07] (which is written in C). In the experiments shown in this chapter, we 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 [OSG07] 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.

3.4.1 Use of ECL\textsuperscript{i}PS\textsuperscript{e}

Our implementation uses the ECL\textsuperscript{i}PS\textsuperscript{e} constraint solver, which is based around a Prolog interpreter with extensions for constraint solving and a plug-in 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\textsuperscript{i}PS\textsuperscript{e} to hold the knowledge base. Information about the nodes and the connectivity between them collected by external and internal sensors, and the overlay membership managed by the overlay network module 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\textsuperscript{i}PS\textsuperscript{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. For example, Rhizoma inference rules privilege more frequently-updated information over older data, handle incomplete data with the union of redundant data from different sources, derive path between

![Figure 3.5: Node role state machine](image-url)
each pair of nodes from basic connectivity information, etc. This provides writers of constraint programs with logical data independence from the details of the sensor information and its provenance.

ECL/PS\textsuperscript{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, meanwhile, it is useful to have the knowledge base available on each node. We discuss relaxing this condition for heterogeneous overlays in Section 3.7.

### 3.4.2 PlanetLab sensors and actuators

**Sensors:** Our PlanetLab implementation of Rhizoma uses three external information sources: the PlanetLab Central (PLC) database, the S3 monitoring service [PP06], and CoMon [YSB+06]. 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 the current load on the node as a whole, obtained by querying the local CoTop daemon\textsuperscript{2}.

**Actuators:** Rhizoma uses a PlanetLab actuator for acquiring and releasing virtual machines ("slivers" in PlanetLab terminology). Releasing a VM is straightforward, but adding a new node is a complex process: the coordinator first contacts PLC for a “ticket”, effectively a capability for creating a VM on the target node. It then presents this ticket to the target’s node manager, hopefully creating the sliver, before using an SSH connection to copy tarballs of Rhizoma and the application to the target, and to spawn a startup script which unpacks the files and starts the application on the node. One possible optimization is to delegate the task of deployment to an existing overlay node that is closer to the target.

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,

\textsuperscript{2}CoTop is the per-node daemon responsible for collecting information for CoMon.
or retrying. Rhizoma runs all deployment operations 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); the actuator for Amazon EC2 uses XML RPC API for VM management and SSH utility for command execution - data transfer in a way very similar to our PlanetLab implementation.

3.4.3 Deployment on PlanetLab

Rhizoma tries to address the challenges raised by future cloud and utility computing infrastructures. These challenges will also be familiar to users of networking and distributed systems testbeds such as PlanetLab. PlanetLab, which is used as a “proving ground” for Rhizoma, is a very different environment to the commercial utility computing infrastructures like EC2 in several important respects, though it shares many common features.

Firstly, PlanetLab is more dynamic and much less stable than services like EC2. This helps us to understand how Rhizoma can deal with server, provider or network outages, and performance fluctuations. PlanetLab is an excellent source of trouble: deploying on PlanetLab is likely to exercise weaknesses in the system design and reveal flaws in the approach. Current and future commercial utility-computing platforms will (one hopes) be more predictable.

Secondly, PlanetLab nodes are more diverse (in hardware, location, connectivity and monitored status) than current cloud offerings, 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.

We believe PlanetLab is probably a more interesting and challenging case than current cloud computing providers for Rhizoma to exercise its ability to manage heterogeneous resources, to adapt to changing resource availability. In this chapter, Rhizoma is deployed and evaluated on PlanetLab.

The Anzere personal data replication system, which will be introduced in the next chapter, extends Rhizoma’s coverage to a more heterogeneous network environment, including personal computers, mobile phones, tablets, and virtual machines dynamically acquired from not only PlanetLab but also Amazon EC2. Different cloud providers have different methods of deploying software, something that Rhizoma needs to handle gracefully. Actually, Rhizoma does not need to deal with specific nodes from PlanetLab and is just as capable of dealing with generic “classes” of nodes offered by cloud providers. One important difference is that the external measurement facilities seen in PlanetLab are not duplicated in the
commercial space, and Rhizoma has to rely on its own measurements and some rough estimation.

3.5 Evaluation

Rhizoma is a rare example of a system not well served by controlled emulation environments such as FlexLab [RDS+07]. Since Rhizoma can potentially choose to 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 aspects. We deploy Rhizoma with PsEPR application requirements on PlanetLab for about 8 hours to observe its behavior. In this experiment, after running the ECL/PS* solver, Rhizoma waits for two minutes after a successful deployment, or for up to five minutes to declare a failed deployment, before starting to generate a new solution. The solving period therefore ranges between two and five minutes, although the calculation itself typically only takes a few seconds of CPU time.

We log three sources of information:

- 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.
- The results of querying CoMon, S3, and PLC (as Rhizoma does) during the period of the trace. Unlike Rhizoma, we perform this centrally.
- For this trace, we also run a measurement slice on all PlanetLab nodes which are alive and accessible. This slice performs fine-grained measurements of inter-node latency. We expect this to resemble the measurements taken by Rhizoma (3.5), 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 illustrated in Figure 3.1. In the rest of this section, we focus on one trace; other results are similar.

3.5.1 Basic performance measures

We first consider the initial two hours of the overall trace. Figure 3.6 and Figure 3.7 show Rhizoma’s behavior during this period. From the timeline shown in Figure 3.6, we see that during the first two hours Rhizoma ran on a total of ten

different nodes, with the coordinator changing three times. Rhizoma started on Node `planetlab1.ani.univie.ac.at` which acted as the first coordinator, then it rapidly expanded to five other nodes. When the coordinator became unavailable, Rhizoma vacated this first node, added a new node into the overlay and elected another node as coordinator. × and ∗ label the time point when an action plan is generated; × labels the action plan which succeeded and ∗ labels the one which failed due to the fact that the nodes to be added were not available.

Figure 3.7 explains Rhizoma’s behavior; it shows the utility value of the overlay configuration and the generated solution, as well as actions taken by Rhizoma to change the overlay configuration. Configuration trace 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 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 of more than three seconds) observed by the coordinator node. Further investigation into the trace revealed that the high latency measurements were
3.5. EVALUATION

due to a bug in Rhizoma's early implementation that was used to collect the trace shown in this section. In this implementation, after asynchronously fetching data from CoMon, Rhizoma copied (but should have moved) the temporary file into the local directory synchronously. Since Rhizoma is a single-threaded, event-based process, this big irregular operation caused the above outliers in latency measurements. Even though the implementation was buggy, we still choose to show this trace for the following reasons:

Firstly, even the outliers are inaccurate measurements of the overlay latency, however, in both cases, Rhizoma made the right decision and responded by redeploying. Although the redeployment was not necessary, it did show Rhizoma's ability to react to network status - the status it learns from the knowledge base.

Secondly, the trace itself is very interesting. As we show later, during the period of the trace, PlanetLab on which we deployed Rhizoma, experienced a large-scale partial outage. This turned out to be a great exercise for Rhizoma's self-managing and self-deploying abilities.

Third, this imperfect trace is a good example of how the Rhizoma trace analysis subsystem helped us to improve the system implementation. The bug was fixed simply by replacing `copy` with `move`; the bug also led us to improve the entire system implementation by removing any possible big synchronous operation, as well as applying “exponentially-weighted moving average” for the latency measurements.

Figure 3.7: Short trace deployment utility
3.5.2 Different measures of utility

Rhizoma attempts to migrate 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 3.8 shows the utility of Rhizoma’s actual configuration for the trace duration, as calculated using different information sources. Overlay utility is based on 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 3.7. External utility uses only CoMon and S3 data and excludes Rhizoma’s overlay measurements (Rhizoma is still using this extra data to make decisions). 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. Be aware that this figure, as well as several others shown below, use non-zero Y-axes in order to highlight the interesting data points and to differentiate the lines to be compared.

As expected, the overlay and trace utilities are almost identical (except for the
outliers), 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 node. An observer or a 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.

### 3.5.3 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.

![Figure 3.9: Effect of overlay monitoring](image)

We use our detailed PlanetLab-wide trace data to simulate Rhizoma without overlay data. Figure 3.9 shows a somewhat negative result: compared to the full system (Rhizoma configuration), the achieved utility of the simulation (external configuration) is similar. While we believe that high-level application information
CHAPTER 3. RHIZOMA

Figure 3.10: Timeline

Figure 3.11: Mean CPU cycles
3.5. EVALUATION

Figure 3.12: Overlay network diameter

Figure 3.13: Total utility
can still be beneficial, it seems that in this case Rhizoma has little to gain from its own overlay measurements.

### 3.5.4 Adaptivity to failures

Figure 3.10 shows a larger and more dramatic section of the complete trace. At about 150 minutes, a buggy slice on PlanetLab turned out to be a CPU hog and caused a large increase in CPU usage across many nodes. This situation is analogous to partial provider outage or degradation. Our utility function in this deployment favored available CPU over network diameter. As this figure 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).

Figure 3.11 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 3.12: 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. Figure 3.13 shows the overall effect on utility. It looks very similar to Figure 3.11 because **mean CPU** is weighted more than **network diameter** in the definition of the utility function.

This reaction to a sudden, transient change in network conditions at these timescales is infeasible with a human-in-the-loop. Moreover, no dedicated management infrastructure is required – indeed, as Figure 3.10 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.

### 3.5.5 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 3.14 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 changes nodes to satisfy the free CPU constraint, but
3.5. **EVALUATION**

![Figure 3.14: Sensitivity to cost function](image)

The nodes that it moves to are also affected; 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 finds a local maximum, as described in Section 3.3.4.

### 3.5.6 Strawman comparison with SWORD

We next present a comparison with configurations returned by SWORD [OAPV04], a centralized resource discovery tool for PlanetLab.

Figure 3.15 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; we 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 we therefore 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, differences in design and goals between the two systems
make this comparison purely illustrative.

3.5.7 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 [RGRK04]. Each run of the ECL$^{PS^c}$ solver takes around five seconds of CPU time.

3.6 Related Work

Early examples of autonomous, mobile self-managing distributed systems were the “worm” programs at Xerox PARC [SH82], 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 [SPW+96], and techniques such as job migration are also widely used.

Oppenheimer et al. [OCP*06] 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 [LKGN05, IAKL07]. The Plush infrastructure [ATSV06] 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 encourage a higher-level declaration of resource policy.

The resource management approach closest to Rhizoma’s use of CLP is Condor’s central Matchmaking service [RLS98], widely used in Grid systems. Condor matches exact expressions against specifications in disjunctive normal form, a model similar to the ANSA Trading Service [Des93]. 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.

3.7 Summary

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. Applications’ self-deployment and self-management are achieved by the Rhizoma approach which presents two key design features: closely coupling a decentralized
management runtime with the distributed application itself; and expressing the application’s resource requirements as a constraint optimization problem in CLP to the runtime who makes decisions on which processing resources to acquire or release.

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.

3.8 Conclusion

Rhizoma decentralized resource management runtime is the first prototype implementing this approach. As we have mentioned in the above text, it is desirable to extend the current implementation to support the following features:

- Heterogeneous platforms: Rhizoma runtime runs only on desktop PCs, cluster machines and PlanetLab nodes. We need to enhance Rhizoma to run on a more heterogeneous network which includes real commercial utility computing facilities. Given the general framework Rhizoma provides, it is straightforward to make such an extension. In the next chapter, the Anzere personal data replication system extends its coverage to a personal computing environment which includes personal computers, mobile phones, tablets, and virtual machines dynamically acquired from not only PlanetLab but also Amazon EC2.

- Resource reasoning based on application logic: At present, Rhizoma runtime solver makes decisions on when and where to deploy the application, and Rhizoma assumes all nodes in the overlay run the same application software. Actually, the same features of CLP that are well-suited to heterogeneous providers can be used to express additional constraints on application deployment, such as which functional components of an application can or should run, and on which nodes, or what application data can or should be stored on which storage devices. The Anzere replication system uses the same mechanism applied in Rhizoma to decide, e.g., where to store data, where to run Paxos proposer and acceptors to achieve sequential consistency.

- Application logic reasoning: In this chapter, the Rhizoma runtime and the PsEPR application do not have much interaction, and the short scripts used to start and stop the application on a node are the only part where explicit interaction between the application and Rhizoma is required. Actually, other optional interaction is more desirable for applications wishing to direct the
3.8. CONCLUSION

application logic based on the application information. The Anzere replication system takes a first step towards this direction and leverages the CLP reasoning engine to make decisions on where and how to replicate data at scale, while considering data replication policy specifications, status of the overlay network, as well as the current placement of the data collection.
Anzere policy-based data replication

We have investigated an alternative model for application management where the management logic is integrated with application instances in Chapter 3. With Rhizoma runtime running on the same nodes, the application can manage itself autonomously by acquiring and releasing distributed virtual machines in response to failures, offered load, or changing policy to continuously satisfy the desired resource requirements of the application which are expressed as constraints.

After investigating the space of declarative management for distributed computation (VMs) resources in Rhizoma, in this chapter, we will extend the functionality of Rhizoma in two ways: firstly, the dynamic resource management mechanism applied in Rhizoma is used to manage distributed storage resources; secondly, the application can use the facilities of the knowledge base as well as the solver to drive its own behavior. We will discuss these two extensions in detail by exploring a technique for partially replicating data items at scale according to expressive policy specifications in the context of the Anzere personal data storage and replication system.

Anzere replicates a user’s personal data (photos, music, etc.) across an ensemble of physical and virtual devices owned (or rented on-demand) by a single user, in the spirit of systems like Cimbiosys [RRT+09] and Perspective [SSCG09]. The personal computing and storage environment we consider include personal computers, mobile phones, tablets, and virtual machines dynamically acquired on both Amazon EC2 and PlanetLab.

Anzere is built on the Rhizoma platform, and accordingly it includes a self-managing overlay network, sensors to monitor system status (device, network,
data, etc.), ECL\textsuperscript{i}PS\textsuperscript{e} constraint solver to make deployment and replication action plans, actuators to take actions. Anzere supports a wider range of platforms than Rhizoma, and it currently runs on mobile phones (Nokia N810, N900), laptops and desktops, and VMs on PlanetLab and Amazon EC2. In addition, Anzere contains a data replication subsystem which supports file system and Amazon S3 data storage and is built on existing replication techniques, in large part on PRACTI [BDG+06], but also TACT [YV02], Bayou [TTP+95], and Paxos [Lam98].

The contents presented in this chapter are based on a joint work published in SOCC 2011 [RYJ+11]. My main contribution to this joint project focuses on the design of the policy model, the implementation of the ECL\textsuperscript{i}PS\textsuperscript{e} policy solver, and the evaluation of the solver’s performance. This chapter shows how such replication systems e.g., Anzere can scale while supporting policies much more expressive than previous schemes. The policy model and solver that I designed and implemented have the following novel features: item replication expressed as constraints, devices referred to by predicates rather than explicitly named, and replication to storage nodes acquired on-demand from the cloud. These extensions introduce considerable complexity in policy evaluation, but the proposed approach can scale well by using equivalence classes to reduce the problem space. This approach is validated in Anzere system via deployment on an ensemble of devices (phones, PCs, cloud virtual machines, etc.). The evaluation using simulation and real data shows that the approach supports rich policies and high data volumes.

4.1 Introduction

Policy-based data replication in a network of personal devices (physical and virtual) investigates how to flexibly replicate personal data in response to a rich set of policies in a way that is robust in the face of devices entering and leaving the system, and with the option to dynamically acquire new resources (virtual machines and cloud storage) in response to changes in workload and policy, if this results in a “better” configuration of the system.

Managing a user’s personal data (photos, contacts, music collection, etc.) is a long-standing problem with, as yet, no effective solution. Achieving this without relying on large, online service providers like Facebook, Google, or Yahoo is a topic of considerable research interest. A personal approach is attractive from a social point of view because it retains a greater degree of privacy and control, and is resilient in the face of a provider becoming insolvent, or the victim of a large-scale compromise of private data.

The application scenario can be summarized as follows: a user owns a small (fewer than 20) collection of different devices, which might include phones, tablets, laptops, home machines, and virtual machines and associated storage rented from
cloud providers such as Amazon. This constitutes the user’s personal cloud. The user acquires and (less frequently) creates new data items, by taking photos and videos, downloading music and documents, and editing contacts.

The goal of the Anzere system is to preserve this growing body of a user’s personal data by replication, and make it selectively available according to the user’s applications and preferences, which are specified as a set of replication policies. The problem is complicated by the limited computation and storage resources on some devices (such as mobile phones), the bandwidth required to replicate data quickly, and the fact that the set of devices involved might change without warning (e.g., due to failure, theft, or purchase of new hardware).

The main contribution of my policy model and policy solver is that the range of allowable policies (i.e., the expressivity of the policy language) can be dramatically increased over previous systems without sacrificing scalability. My work shows (i) how replication policies for personal data can be written independently of specific devices (where the data is to be stored), and can even result in the system acquiring and releasing virtual storage resources on-demand, (ii) how such a system preserves policy goals by reacting to changes in the environment such as the creation of new data items, failures or network outages, etc. and (iii) how to scale rich policy calculations up to large numbers of data items with only modest requirements in memory and computation, using equivalence classes.

The rest of this chapter is organized as follows. Next section motivates our work and reviews related work. Section 4.3 elaborates on our target scenario, and identify the key properties a personal storage system should provide. Section 4.4 presents the policy model, and the solver implementation is described in Section 4.5. The approach is evaluated in Section 4.6, and the conclusion of the first part is given in Section 4.8.

## 4.2 Background and motivation

Autonomous personal data management is the motivation for our Anzere policy-based personal data storage and replication system. Recent user studies have shown that, despite a plethora of commercial point-solutions for backup and synchronization, people still find it difficult to manage their multiple personal devices.

Oulasvirta and Sumari [OS07] have studied practical problems people face in synchronizing devices. For example, their laptops can be automatically backed up to a file server, but their smart phones cannot easily access this server. On the other hand, the option of storing all data on smart phones is not always viable due to limited storage capacity. People find ad-hoc solutions to such problems, such as carrying more devices, anticipating future needs by copying data to appropriate locations, and manually synchronizing their data when most convenient (e.g.,
before leaving for a trip).

Dearman and Pierce [DP08] report that people synchronize their devices, using portable media, emailing files to themselves, sharing directories over a network, or using third-party external servers [Dro12, Doc12, Liv12], but not without the risk of losing their data as recently reported in the news [Ama11]. All these techniques have serious limitations: they require special configuration, cannot handle all types of files, and/or raise privacy and reliability concerns. File synchronization tools are rarely adopted by regular users partly because they rarely organize their personal data through hierarchical naming, but instead use data attributes and higher-level search interfaces [Sal09, MAB+10].

Motivated by these problems, several recent systems have been proposed to address the challenges of policy-based replication of personal data (photos, music, etc.) within a network of devices, as an alternative to centralized online services. Among them, Perspective [SSCG09] and Cimbiosys [RRT+09] are closest to our work. Both systems address the challenge of personal data storage with support for content-based partial replication. Perspective, designed for home devices, provides a semantic file system interface based on the concept of view, a query which defines a set of files to be stored on a specific device. Cimbiosys allows users to selectively distribute data across their devices by associating content filters with each device. Perspective replicates filters on all devices, and assumes they change infrequently, whereas Cimbiosys allows incomplete knowledge of other replicas and frequently-changing filters using filter-based synchronization trees.

Eyo [SLLP+10, SPLL+11], a storage system for personal media collections, fully replicates policies and all content metadata across all devices – content and metadata are managed separately. The system offers a device-transparent storage API, where each device knows about all objects.

The policies supported by all these systems have been relatively simple, in order to facilitate scaling the policy calculation to large numbers of items. Furthermore, in these systems, data is specified in policies as logical predicates, but locations are specific devices. This constraint is relaxed in our approach: devices are specified by logical predicates and, indeed, acquired on-demand if necessary. Like Eyo, Anzere fully replicates content metadata and policies, but policies are evaluated by an elected coordinator. As shown later, actually this approach deals efficiently with filter changes and adapts to changing network topologies.

EnsemBlue [PF06], a distributed file system for PCs and consumer electronic devices, provides content-based partial replication through persistent queries, which specify the set of events that an application is interested in receiving. The file server then appends log records to the query when an event matching the query occurs. The application fetches the query, reads the records, and replicates the files accordingly. We share with EnsemBlue the device ensemble concept and diversity
of the storage elements. Different from EnsemBlue, Anzere enables data management through device-independent policies and is able to extend the ensemble to dynamically-acquired cloud resources.

PodBase [PKD08,PNKD11] is a system for storage management across a household’s personal devices. PodBase’s goal is to ensure that data is replicated on enough devices to tolerate failures (data durability) and that replicas are placed on devices where they may be needed (data availability). Like Anzere, PodBase is a self-managing system with automatic data management, however, it does not focus on providing a policy-based interface to support flexible replication requirements. As we show later, the data durability and availability properties PodBase targets can be expressed by our policy language.

4.3 Target scenario and environment

Anzere storage system is intended for personal clouds, personal ensembles of owned machines (mobile devices like phones and tablets, laptops as well as fixed computing devices like PCs) which can be dynamically extended to incorporate also rented cloud resources (e.g., from Amazon EC2), if advantageous for the overall system configuration.

Personal clouds are highly heterogeneous. They consist of both physical and virtual resources, storage capacities range from a few GB to terabytes on cloud resources, processors vary from embedded systems to server-class processors, link speeds range from 54 Mbps wireless to 10 Gbps Ethernet, and pricing structures vary from expensive, metered 3G connections to backbone links.

![Diagram of personal cloud network](image)

Figure 4.1: The personal cloud network of our example user. This is also the hardware setup for our experiments.
As a driving example, we use a concrete hardware configuration and a real data set, extrapolating from this wherever it is necessary to investigate scaling and tradeoffs.

The hardware configuration, which is also used for experiments in Section 4.6, is shown in Figure 4.1. It comprises an office desktop PC, a home PC server, a laptop, a Nokia N900 smart phone, and a few virtual machines on Amazon EC2 and PlanetLab (the number varies according to the system’s policy decisions). The home PC and phone have private IP addresses, and use the virtual machines for NAT traversal.

The data set, obtained from a member of our research group, is summarized in Table 4.1. Making strong claims about how representative this data set is would require a full user study and be beyond the scope of our research, but it does provide us with a starting point grounded in reality.

<table>
<thead>
<tr>
<th>Device</th>
<th>Photos</th>
<th>Music</th>
<th>Videos</th>
</tr>
</thead>
<tbody>
<tr>
<td>HomeServer</td>
<td>6958 (8.1GB)</td>
<td>4904 (23.1GB)</td>
<td>53 (4.7GB)</td>
</tr>
<tr>
<td>Laptop</td>
<td>3291 (7.2GB)</td>
<td>932 (5.7GB)</td>
<td>10 (2.2GB)</td>
</tr>
<tr>
<td>OfficePC</td>
<td>0 (0)</td>
<td>3997 (19GB)</td>
<td>0 (0)</td>
</tr>
<tr>
<td>Phone</td>
<td>89 (38.5MB)</td>
<td>868 (4.3GB)</td>
<td>0 (0)</td>
</tr>
<tr>
<td>Camera</td>
<td>25 (56.5MB)</td>
<td>0 (0)</td>
<td>0 (0)</td>
</tr>
<tr>
<td>Cloud</td>
<td>4492 (5GB)</td>
<td>0 (0)</td>
<td>28 (435MB)</td>
</tr>
<tr>
<td>Total</td>
<td>9231 (13.2GB)</td>
<td>4904 (23.1GB)</td>
<td>56 (6.8GB)</td>
</tr>
</tbody>
</table>

We see that the user’s data is not replicated fully on all devices; instead, partial replicas of the data are created on different device subsets. In this data set, the music collection has a master copy on the home server, with subsets replicated on laptop, office PC, and phone. For photos, however, the creation of partial replicas does not follow an obvious pattern – there is no master replica, photos on camera are not backed up anywhere else, a subset of the phone’s photos is replicated on the laptop. A cloud-based web host stores a (perhaps public) photo subset. The number of videos of this user is relatively small, but this might well increase in the future.

Our example user explained the goals that drive such a data distribution as follows:

- Backup (as soon as possible) photos as well as videos taken with the camera and the mobile phone on the home PC server. If the home server is
not reachable during travelling, make a copy of the files on the laptop and ultimately have the laptop synchronized with the home PC.

- Regularly check that enough free storage space is available on the phone and camera. In particular, make sure to empty the camera’s memory card after a trip.

- Avoid uploading private photos such as portraits of himself, relatives, and friends to the cloud. This is achieved by manually checking every photo before uploading.

- Store the entire music collection on the home server, and have subsets of it on the office PC, laptop, and phone. These collections are selected manually and in an ad-hoc manner, but they tend to include favorite and most recent albums. As the phone’s music collection cannot be as big as the one in the laptop and office PC, the user would like the collection on the phone to be periodically (e.g., weekly, monthly) refreshed with new content. The user currently achieves this by manually updating the phone’s collection, whenever he remembers to do so.

Although this is only one example, it shows some common usage patterns across users and suggests that applications could largely benefit from richer replication policies to control and automate the placement of a user’s data across different devices. Rather than users manually classifying their photos and moving them to the appropriate devices or cloud, an application for photo albums generation, for instance, could use a common face-recognition algorithm to automatically classify any new photo into public or private and replicate it accordingly. Likewise, rather than having users manually refreshing their phone’s music collections, a music player application could automatically use user-specific information such as album’s rating, creation date, and last played date to periodically refresh the content on the phone.

To meet the applications’ data replication requirements, the goals of our policy-based data replication are to:

- Allow intuitive specification of policies, which are not tied to specific devices, since these change over time – replacing a phone or buying an additional laptop should not require any change in the policies;

- Exploit (and decide to acquire or release) dynamic virtual resources on-demand, if necessary – that is, our approach should factor monetary cost and performance into the replication decisions;

- Efficiently replicate the user’s data according to a set of flexibly-specified policies, and scale to a large number of data items and a reasonable number of policy rules;
• React in a timely fashion to failures and changes in data, policies, devices, or external conditions such as hosting prices or network outages.

The key differences between these goals and those of previous systems are that the policy requirements are considerably broader (variable number of devices, when to rent a new VM, etc.), and devices are, like content, identified implicitly using predicates rather than identifiers. This makes policy reasoning more complex, however, it is still kept tractable in the face of such flexibility as explained later.

4.4 Policy model

This section discusses about what can be expressed as policies. The users are not expected to directly use the notation proposed here; policies are better generated by user applications (e.g., music players, photo sharing applications, etc.) or composed using graphical tools (e.g., a data distribution map across devices). The focus in this section is what semantics can be expressed by the policies.

Data is stored as a set of data items. A data item like a photo is represented as a pair of (content, metadata), where content is the binary data itself, and metadata is a list of key-value pairs. metadata can be mutable, while, for the moment, content is assumed immutable. Actually, Anzere also supports mutable content and offers high flexibility in expressing its consistency requirements.

4.4.1 Device- and content-neutral policies

A replication policy, hereafter simply a policy, represents a set of rules, filters, and constraints that applications establish to control where a user’s data is stored. A policy might address requirements such as:

• Accessibility: “Recently-acquired music available from the device the user carries”;
• Durability: “At least 3 distributed copies of my Ph.D. thesis”;
• Privacy: “No private photos in the cloud”;
• Capacity limits: “No more than 2GB music on phone devices”.

Unlike existing systems, policies are independent of the personal cloud configuration for which they are initially specified. For example, to guarantee that the photos taken with the phone are replicated for durability, current replication systems (e.g., Cimbiosys, Perspective) will express this requirement by generating a filter on the device with id=myhomedesk for objects of type=jpeg and location=myapplephone. Our approach will express the same requirement as a
4.4. POLICY MODEL

policy requiring objects of type=jpeg and location=phone to be replicated to at least one device of type=fixed and with tag=owned.

The advantage of this approach is that the replica is not bound to any specific device. The policy continues to work even when the connectivity changes (myhomedesk is not reachable by myapplephone, but myofficepc can be accessed instead), or when the device set changes (the user buys a new phone and names it as mynokiatablet). Furthermore, in principle, these general policies can also be reused by other users.

4.4.2 Policy stratification

Replication policies are a set of triplets \( \langle IP, R, DP \rangle \), here, \( IP \) is an item predicate, \( DP \) is a device predicate, \( R \) is a relationship that must hold among the items and devices identified by \( IP \) and \( DP \). The policies are expressed in a declarative way (CLP constraint logic language in our case) as in Rhizoma. Logical unification provided by logic languages is a powerful technique for fusing information from a set of heterogeneous sources, such as different device and data types, and decoupling the system from a predefined schema.

The example shown in Figure 4.2 depicts how policy stratification works. In this example, item and device metadata is automatically extracted from files and the device OS, and represented as Prolog facts. The item fact \( \text{photos/anfora.jpeg} \) describes a JPEG photo, which was created on April 28th, 2010, and is public. The device fact \( \text{nokiaN900} \) describes a phone which is a mobile, self-owned device with a rental fee of $0.

Item and device predicates at Level 2 are mostly provided by storage system developers in the embedded library and can be extended by user applications. These predicates are applied to low-level facts collected from the files and the device OS and can be used by user applications to compose policies. Specifically, at Level 2, picture_item is an item predicate which defines the set of items of type “photo”, while any_device is a device predicate which defines the set of available devices. These predicates are expressed as simple inference rules.

At Level 3, a high-level durability policy is defined requiring the system keeps at least two replicas for each photo; \( \text{rep}\geq 2 \), the relationship predicate, defines “data available from at least 2 devices.”

A more interesting policy example is “items modified within 24 hours are accessible from my NokiaN900 phone with a latency of no more than 100 ms”:

\[
\text{mod\_item}(Op, Time, Itemid) :-
\text{item(itemid:Itemid, moddate:Moddate)},
\text{mjd\_now(MjdNow)},
\text{mjd\_to\_unix(MjdNow, UnixNow)},
\]
Figure 4.2: Example of policy stratification. At Level 1, facts describing data items and devices are automatically generated from OS and application tools. At Level 2, item and device predicates are provided by the storage system developers in the library and can be extended by user applications. At Level 3, user applications use item and device predicates to compose replication policies.

\[
\text{Diff is UnixNow-Moddate, Func =.. [Op, Diff, Time], Func.}
\]

\[
\text{close_device(MyDevid, MaxLatency, Devid) :- ollink{src:MyDevid, dst:Devid, latency:Latency}, Latency \$< \text{MaxLatency}.}
\]

\[
\text{policy([[mod_item,#<,86400]],[repany],[[close_device,'NokiaN900',100]]).}
\]

The \texttt{close\_device} device predicate specifies devices which are reachable from the specified device within the given latency; it uses \texttt{ollink} facts (produced by the monitoring infrastructure described in Section 4.5.2) to get latency measurements between any two devices in the overlay.

As shown in these examples, predicates can refer to both immutable and mutable properties of an item or device: immutable properties like item category
4.4. POLICY MODEL

(picture_item) and device ownership (owned_device), as well as mutable properties such as item temporal property (mod_item, \(<\), 86400) and device locality (close_device, 'NokiaN900', 100). Multiple predicates can be specified in a single policy, as indicated by the list syntax for IP and DP at Level 3 in Figure 4.2.

Facts describing new item and device types can be extended by the user applications and added to the system on-the-fly. If a new pricing model for cloud resources is introduced, or a user’s media files are now classified using a new metadata schema, these new facts and new inference rules can be inserted into the system which will immediately be reasoned anew.

4.4.3 Replication through constraints

Consider the following two policies which affect the photos taken from the phone:

1. Mobile device files replicated to at least one fixed device, and
2. Replicate photos to a home device.

If these two policies are evaluated in isolation, in the above order, it is possible to replicate the photos first e.g., to the office PC and later to the home server. In the end, this plan gives three replicas for each photo. If they are evaluated in isolation, in a reversed order, the photos will only be replicated to the home server. The ideal solver should evaluate the existing policies, regardless of their order, and generate unambiguous efficient solution. For this example, the photos will be replicated only to the home server to satisfy the specified policies as well as to minimize the storage/bandwidth utilization.

Policies will also incorporate resource and cost constraints imposed by the environment. Mobile phones have limited storage capacity, motivating moving old photos to a better-provisioned device if they are no longer needed. If backup in the cloud incurs a rental fee, a personally owned device might be preferred.

All these requirements and constraints make the policy solving a complex constraint satisfaction/optimization problem. This led us to choose constraint logic programming (CLP) as the most tractable way to express high-level application demands and optimization goals. More background about CLP can be found in Chapter 2.

CLP programs are regular logic programs with added logical and numeric constraints, providing a natural way to express policy requirements such as number of replicas, device capacities, and cost limitations. This constraint satisfaction problem is to find a data placement in the personal cloud respecting all the constraints. Our CLP formulation can be visualized as a 2-dimensional matrix \( M \) whose columns represent the devices and whose rows represent items stored. The variables are the matrix cells, \( v(x,y) \), whose possible values are either 0 (do not store) or 1 (store). An example data distribution after two new items item3 and item4 are added is shown in Table 4.2.
Data replication policies impose constraints on the variables in a subset of the matrix cells, restricting the possible values that can be assigned to the cells. For example, policy([[IP]], [repany], [[DP]]) requires \( \forall x \exists y (IP(x) \land DP(y) \rightarrow v(x,y) = 1) \). In other words, for the data item \( x_i \): IP(\( x_i \)), the summary of \( v(x,y) \) from the columns where DP(y) is true, the following formula must be satisfied: \( \sum_{y \in \{y\}} v(x, y) \geq 1 \). The right most column of Table 4.2 shows the replica constraints and the bottom row shows the device capacity constraints that the policies impose. After trying to satisfy all the constraints, the CLP solver computes a set of solutions with the assignments to the variables in the matrix cells as shown in Table 4.3.

The CLP solver also supports optimization by maximizing a given objective function. For instance, the objective function might minimize the distance (actually bandwidth utilization) between the current data placement matrix and the new solution, or prioritize fixed devices over more resource constrained mobile ones, or minimize the overall (monetary) cost for renting cloud resources. Other optimization metrics can be added as well, and multi-objective optimization is possible using a weighted sum of the specified objective functions.

The above mentioned optimization objectives lead us to a final configuration as shown in Table 4.4. By comparing the cells of the current and the desired data
placement matrices, an execution plan is generated. Pairs of cells yielding the same value produce no action. For the case of pairs with different values, if it is 1 a \( \text{copy}(y, z, x) \) action is generated (\( z \) is a device that currently stores item \( y \)), and if the new value is 0 a \( \text{delete}(y, x) \) action is generated. These actions are then executed through responsible nodes.

Table 4.4: Current data distribution

<table>
<thead>
<tr>
<th></th>
<th>phone</th>
<th>homePC</th>
<th>laptop</th>
</tr>
</thead>
<tbody>
<tr>
<td>item1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>item2</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>item3</td>
<td>0(^a)</td>
<td>1(^a)</td>
<td>0</td>
</tr>
<tr>
<td>item4</td>
<td>0</td>
<td>1(^c)</td>
<td>1(^c)</td>
</tr>
</tbody>
</table>

\(^a\) delete (item2, phone)

\(^b\) copy(item2, phone, homePC)

\(^c\) copy (item4, homePC, laptop)

Table 4.5 shows examples illustrating the expressivity of Anzere’s policy language. However, the approach described above doesn’t scale well as the number of objects increases. Section 4.5 will address this scalability problem and show how equivalence classes which are generated dynamically from the policy specification make this approach scalable.

### 4.4.4 Acquirable resources

Once the problem of applying replication policies efficiently is cast as one of optimizing a set of possible actions, it becomes straightforward to add additional types of actions, with associated costs and benefits, to the basic framework. This feature is exploited to acquire and release cloud storage and computational resources on demand, if doing so results in an overall benefit to the personal cloud ensemble.

Anzere factors the decision of acquiring cloud resources in the policy evaluation itself. The decision is taken entirely on-the-fly by the CLP solver. If the current set of storage devices is not sufficient to satisfy the policy, Anzere searches the possible states the system can achieve by incrementally acquiring cloud resources. To support acquirable resources, new columns representing these resources will be added to the matrix model described above while the reasoning logic remains the same. For instance, one scenario evaluated in Section 4.6 is about a data synchronizing application which allows the user to specify the maximum access time tolerable by specific classes of data. This application uses data availability policy like Policy 4 in Table 4.5. While the user is travelling abroad, the system detects the increased latency from the user’s laptop to his home and office network,
## Example Anzere Policies

<table>
<thead>
<tr>
<th>Policy Type</th>
<th>Description</th>
<th>Prolog Policy</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Fault Tolerance:</strong></td>
<td>1. Music backup on a home device</td>
<td><code>policy([audio_item], [repany], [home_device]).</code></td>
</tr>
<tr>
<td></td>
<td>2. Video backup on 2 fixed, owned devices</td>
<td><code>policy([video_item], [rep, #&gt;, 2], [fixed_device, owned_device]).</code></td>
</tr>
<tr>
<td><strong>Data Availability:</strong></td>
<td>3. 1-day old music on mobile devices</td>
<td><code>policy([audio_item], [mod_item, #&lt;, 86400], [repall], [mobile_device]).</code></td>
</tr>
<tr>
<td></td>
<td>4. 1-day old photos on a fixed device at 100 ms from the laptop</td>
<td><code>policy([picture_item], [mod_item, #&lt;, 86400], [repany, 100 ms from the laptop], [fixed_device, close_device, 'laptop', 100]).</code></td>
</tr>
<tr>
<td><strong>Resource Management:</strong></td>
<td>5. 5GB free storage on phone</td>
<td><code>policy([any_item], [size, #&lt;, 5000000000], [phone_device]).</code></td>
</tr>
<tr>
<td></td>
<td>6. No private items in the cloud</td>
<td><code>policy([private_item], [repany], [cloud_device]).</code></td>
</tr>
<tr>
<td></td>
<td>7. Public photos in the cloud</td>
<td><code>policy([public_item, picture_item], [repany], [cloud_device]).</code></td>
</tr>
<tr>
<td><strong>Privacy:</strong></td>
<td>8. Rental fee for cloud storage less than 10$</td>
<td><code>policy([any_item], [cost, #&lt;, 10], [cloud_device]).</code></td>
</tr>
</tbody>
</table>

### Description
- **Policy Type:** Prolog policy
- **Cost:** 8. Rental fee for cloud storage less than 10$
- **Privacy:** 6. No private items in the cloud
- **Resources:** 5. 5GB Free storage on phone
- **Availability:** 4. 1-day old photos on a fixed device at 100 ms from the laptop
- **Reliability:** 3. 1-day old music on mobile devices
- **Tolerance:** 2. Video backup on 2 fixed, owned devices
- **Fault:** 1. Music backup on a home device
and might decide to acquire cloud machines from data centers close to the current location. On these machines, the system stores temporary copies of the data for which the user required fast access.

The release of cloud resources also occurs in an automatic manner. Cloud VMs are released when no items are stored on them any more, or the data in the cloud are migrated to cheaper/free machines as the solver tries to minimize the rental fees. For instance, cost related policy such as Policy 8 in Table 4.5 would eventually force the system to cleanup unused cloud resources. So in the example above of the user travelling abroad, when the user returns from the trip and his/her own devices re-gain faster connectivity, cloud VMs will be released.

CLP optimization can be very useful in such context. In fact, the inclusion of cloud infrastructures brings an interesting new variable to the problem: the price for renting VMs and transferring data from/to them. Price constraints are expressed through explicit cost policies or embedded in objective functions. Price models can become complex and need to closely follow the changing pricing structures of different cloud providers. Nevertheless, we feel our approach goes some way to being able to integrate such factors into the behavior of the system, much as commercial offerings like RightScale [Rig12] attempt to do today for hosted services.

The number of ordinary users today using rented cloud storage for their private files is rather limited. However, Anzere represents a solution also for those users who primarily deal with their own physical devices. More importantly, Anzere can provide an incentive for such users to use cloud storage. By automating the selection, acquisition and release of cloud resources, by offering a simple API for controlling the cost for renting such resources, and by possibly choosing the most adequate cloud infrastructures (based on price and resource requirements) for each user’s requirements, the API to cloud computing infrastructures is extremely simplified, thus making cloud computing a viable option also for regular users.

### 4.4.5 Composing Anzere policies

The users are not expected to deal with the policies in Prolog directly by themselves. Rather, future applications are expected to generate most policies on the user’s behalf using context-specific knowledge or asking users a few questions. For example, a prototype photo album application for publishing photos to the cloud is developed in the group. This application is similar to the one our example user in Section 4.3 could use for sharing public photos. Users could specify which photos to include in an album through properties such as size, format, time frame, and privacy. This information is automatically extracted from image metadata or using image processing algorithms (e.g., a face recognition algorithm classifies as private photos containing a specific person).
Another example of how existing applications could leverage our policy language is prefetching. Existing applications could integrate support for content prefetching more extensively, and allow users to specify properties such as freshness, overall size, and rating of the content to be prefetched on specific devices – very much like what web browsers do today by setting their maximum cache size). When the content is generated on a device in the personal cloud, other devices matching the prefetching policy will automatically receive such content. For example, data availability policies such as Policy 3 in Table 4.5 could, for instance, fulfill the user’s requirement to automatically refresh music content on the phone.

Regardless of how policies are specified, conflicts between them may prevent a solution from being found. Two situations may arise: in the first case, one or more newly-issued policies conflict with previously-specified ones (e.g., the system requires 2 replicas in the cloud and a new policy specifies that private data cannot be stored in the cloud). The second case occurs when the device ensemble suddenly changes, new data items are submitted or the data properties change such that the current set of policies can no longer be satisfied by the available resources (e.g., the system requires two replicas for all objects within a fixed budget which is exceeded by the generation of new items). Currently, the system reacts simply by reporting failed reasoning and detected conflicts, and waiting until they are resolved. In a single-user situation, such events are expected to happen in isolation and at a relatively low rate, allowing the feedback returned to the application (and, ultimately, the user) to be accurate enough to quickly identify the cause of the conflict. If such events occur more frequently, a possible solution that we have not yet explored is for the system to use its logs to replay events in isolation and perform a root-cause analysis.

In general, users require a tool to globally manage the policies generated by applications. Some graphical tools can provide the users an intuitive experience and the users could be able to directly observe the “effects” of their policy changes. Understanding the effects of a policy will help the users to compose the desired set of policies. The focus of our policy model, however, is to explore which policy semantics can be expressed to the system, and how they can be accomplished using a constraint-based approach.

4.5 Implementation

The policy model and solver are integrated into the Anzere personal data storage system which is built ontop the Rhizoma platform. Anzere has similar software architecture as Rhizoma as shown in Figure 4.3. Anzere includes a self-managing, self-deploying overlay network, sensors to monitor the network and device status, a knowledge base (KB) to store the system’s state, ECLiPS CLP policy solver to
make replication and deployment action plans, actuators to acquire (and release) cloud resources on-the-fly as well as to move data around to satisfy the policies.

As a data storage and replication system, Anzere also contains a data replication subsystem providing flexible consistency and partial replication. Anzere replicates user data as well as information necessary for the system operation (e.g., overlay and storage sensors’ information). Anzere data replication subsystem supports the three PR-AC-TI [BDG+06] properties: Partial Replication addresses the different resource requirements of an ensemble of heterogeneous devices; Arbitrary Consistency permits tuning consistency when dealing with different types of mutable content; Topology Independence ensures device failures and mobility do not compromise system operation.

This section will introduce first the overlay network, the sensors, the actuators, and the data replication subsystem. The components related to reasoning the application’s requirements (data replication policies in Anzere’s case) to get a better configuration (data placement and data replication), more specifically, the CLP policy solver and the knowledge base, will be discussed in detail afterwards.

![Figure 4.3: The Anzere system architecture. Main components are the overlay network (sensors, actuators and routing functionality), the knowledge base, the CLP solver, and the data replication subsystem. This software stack runs on each node in the personal cloud network with the exception of the CLP solver which runs only on well-provisioned nodes.](image-url)
4.5.1 Anzere overlay network

Anzere inherits the self-managing and self-deploying overlay network from Rhizoma, and reuses its overlay protocols for membership management, failure detection and coordinator election. An overlay node is elected as coordinator, while the others act as members. Rhizoma’s election mechanism is extended in Anzere, so that Anzere is able to elect the coordinator based on any device attributes (e.g., resource availability, location, owned vs. rented device, etc.). In case of coordinator failure or overlay partition, a new coordinator is re-elected automatically and the liveness is guaranteed by replicating the knowledge base across a few overlay nodes.

Although a user’s personal cloud is generally small, its complexity arises from the heterogeneity of its resources. Anzere reuses the resource management functionality in Rhizoma, and extend its PlanetLab-specific sensor and actuator modules to support more heterogeneous platforms. Currently, Anzere supports sensors for mobile phones, tablets, virtual machines dynamically acquired on both Amazon EC2 and PlanetLab, as well as actuators for Amazon EC2 and PlanetLab.

Anzere sensors monitor the status of the personal cloud as well as the hosting infrastructure. With this information, Anzere is able to detect and react to events such as node failures, variations of link quality, introduction of new network links as well as cloud outages. Anzere actuators take actions according to the action plan generated by the CLP policy solver. The action plan includes copy and delete actions to move around data items as well as acquire and remove actions to dynamically acquire and release specified cloud resources.

Like Rhizoma, Anzere is also engineered through a modular framework (inspired by the OSGi [OSG07] module management system), which not only enables easy system maintenance, but also allows us to customize the functionality running on each device based on its hardware/OS, as well on the role the node plays in the system (overlay coordinator, member). The CLP policy solver and knowledge base, for instance, do not run on resource-constrained devices, but instead on well-provisioned coordinator nodes. Specialized sensor modules for phones, PlanetLab, Amazon EC2 are included only in the corresponding distributions.

4.5.2 Knowledge base

The knowledge base stores the distributed state of the system in the form of Prolog facts. This information is collected through overlay and storage sensors, running on every node in the ensemble.

Overlay sensors monitor the status of the network, detect failures, and collect information such as device type and status, number and type of network interfaces available, latency and bandwidth between any two devices. This information is
stored in the form as the following facts:

device(devid,location,type,cost,processor,mem,disk).
olnode(hostname,cpuspeed,freecpu,fiveminload,mem,freemem,gbfree).
ollink(src,dst,link-type,latency,bandwidth).

Storage sensors inform the knowledge base about policies, item metadata (extracted from files using ExifTool [Exi12]), and so-called item2dev information, i.e., a summary of which items are stored at which nodes. This information is essential for the CLP solver to build a map of the current data distribution and react to changes in policies and data items. In the knowledge base, the facts embedding this information are as follows:

policy([[item_pred]], [relation], [[device_pred]]).
item(itemid,type,size,createdate,moddate,tag).
item2dev(itemid,devid).

Data collected through the overlay sensors is replicated across a few resource-qualified nodes (i.e., nodes that can afford to take over the role of coordinator and run the CLP solver in case of coordinator failures), while storage sensor’s information is replicated across all overlay nodes (the overhead is relatively small, e.g., metadata describing 20,000 items accounts for about 6-8 MB). Bodies of data items (content of photo, audio, video files) are replicated to nodes selected by the CLP policy solver based on the active policies and the knowledge base.

4.5.3 CLP solver and equivalence classes

Policy evaluation in Anzere is centralized at one overlay coordinator node running the CLP solver. This node is typically a well-provisioned node such as an office or home desktop PC or a cloud VM. Phones and tablets will never be elected as coordinators. The coordinator does not represent a single point of failure because the overlay is capable of dynamically electing a new coordinator if the old one fails or disconnects. However, a legitimate concern is whether this represents a scaling bottleneck. As we will show in Section 4.6, this centralized approach has not caused any performance degradation or scalability issues in system operation. Liveness is guaranteed by replicating the knowledge base across all or a few other overlay nodes, to speed up recovering from a lost coordinator.

The ECL/PS* constraint programming system, which is written in C and Prolog, supports the development and deployment of our CLP policy implementation. Due to online reasoning, scalability is a primary concern in our policy solver implementation. The number of devices is unlikely to be a scaling problem, since a personal cloud is expected to be small (fewer than 20). Yet, the number of data
items is potentially an issue for the scalability of the policy engine as it is large and grows over time. Our CLP implementation needs to optimize placement of hundreds of thousands of objects.

To address this problem, equivalence classes is then introduced to group data items. Our CLP implementation directly derives the smallest set of equivalence relations from the set of item predicates appearing in the active policies. These equivalence relations partition the item set into disjoint subsets, namely equivalence classes. Assume a system managing a user’s picture, video, and audio items has the following two policies:

\[
\text{policy}([[\text{picture item}],[\text{private item}]], [\text{repany}], [[\text{owned device}]]).
\]
\[
\text{policy}([[\text{audio item}]], [\text{rep},#>=,1],[[\text{fixed device}]]).
\]

The CLP program computes four equivalence relations:

\[
[[\text{picture item}],[\text{private item}]]
[[\text{picture item}],[\text{public item}]]
[[\text{audio item}]]
[[\text{video item}]]
\]

The number and type of equivalence classes as well as the granularity of how data is aggregated into such classes is derived directly from the user’s policies, specifically the item predicates present in the user’s active policies. By doing so, the number of classes is much smaller than the number we would obtain by considering all possible combinations of metadata properties, which would be exponential. Moreover, unlike metadata, item predicates are boolean predicates, thus ensuring a finite number of combinations.

With the introduction of equivalence classes, the rows in the matrix model described in the previous section do not refer anymore to item identifiers but instead to equivalence classes. This change enormously reduces the number of variables the CLP solver has to manage, and hence its solving time. As shown in Section 4.6, equivalence classes bring a substantial performance improvement to the system.

The solution generated by the CLP solver consists of an action plan (copy or delete actions) for different Anzere nodes to execute. These actions are executed through the replication subsystem’s API. If the destination node of an action is a cloud resource which is currently not present in the network overlay, the action plan will also include acquire (or remove) actions. The corresponding cloud actuator is then invoked and the node is dynamically acquired or released to complete the action.

The solver periodically re-calculates the execution plan to incorporate new items and policy changes, as well as to react to changes in the device status and
4.6. EVALUATION

topology (e.g., “device within 100 ms access latency from the phone”). Policies with temporal properties (e.g., recently modified items) also require the CLP solver to reason periodically to approximate continuous time requirements. An event-driven invocation of the solver is not supported yet in the system but would be a nice optimization. For example, the solver can be invoked only when substantial changes have occurred in the environment.

Generating and applying the action plan in an uncertain environment with unreliable network connectivity, where devices can fail or be turned on and off dynamically, requires some care.

First, the system may become unstable if actions are generated and executed whenever some condition in the environment changes. For instance, route flapping can cause the network latency to change rather frequently and mobile devices can disconnect from time to time. As in Rhizoma, Anzere overlay network also smooths these variations so that only stable network changes are processed by the solver.

Another concern is about the execution of the actions: the action plan might be left not fully executed when some responsible devices are turned off, and the actions happen asynchronously. To address this, the coordinator node running the solver maintains a coherent view of the system state. Every time an action is completed in a device, an acknowledgment is sent back to the coordinator (in the form of item2dev fact). This helps to implement the move action which is a copy followed by a delete: only after the copy has completed successfully, the coordinator issues a delete command. Since the coordinator has an up-to-date view of which items are present on which devices, if some devices are turned off before completing their execution plan, the coordinator will know their latest status and re-issue requests for missing actions when these devices come back before the next policy evaluation. In this way, policies are enforced in a timely manner.

We ran into cases where thousands of actions were requested at the same time (e.g., when simulating the permanent crash of a home PC) and could not complete before the next solver invocation, or complete without updating the knowledge base due to delayed acknowledgments. To avoid duplicated action requests in such circumstances, Anzere keeps a queue of pending actions, which will be removed from the queue upon completion or timer expiry.

4.6 Evaluation

Anzere policy model and the solver is evaluated from the following perspectives:

- Policy sustainability: converging to a suitable data placement solution in compliance with the specified policies
- with acceptable reasoning time;
- with low memory consumption;
- scalable to large number of data items.

• Reactivity

- to policy changes;
- to device changes (e.g., home server crash);
- to device mobility.

Our target scenario is the one depicted in Section 4.3: a single user scenario, with a dataset of 15,000–25,000 data items and a policy set of up to 43 policies. The policies used in the following experiments are listed in Table 4.6, and some of them are overlapping with the ones in Table 4.5. The experiments use a real personal cloud (shown in Figure 4.1) emulating the configuration of an example user. The testbed consists of an office desktop PC, a home PC, a laptop, a Nokia N900 smart phone, and cloud resources (two Amazon EC2 VMs, one located in Europe and one in the US, and one PlanetLab VM). The usage of cloud resources varies according to policy decisions. The home server and the phone have private IP addresses, and the Anzere routing module takes care of automatically establishing tunnels for these devices using the office PC or cloud machines as traffic forwarders. The phone and laptop use WiFi for communication.

At startup, the office PC acts as the overlay coordinator and runs the CLP solver. All nodes run the overlay sensors to collect information on its device status and link performance. The experiments use a workload consisting of real-life samples of photo, music, and video files, with average sizes of 1.1 MB, 3.3 MB, and 4.2 MB, respectively. Metadata files, generated using ExifTool [Exi12], are about 300–400 bytes. We start with evaluating the performance of the CLP solver and then analyze Anzere’s reactivity with the full system in operation.

### 4.6.1 Policy sustainability

The first experiments evaluate the scalability aspects of the CLP solver and resource overhead of policy evaluation. The CLP solver runs on a single well-provisioned cluster node, which has a 2.3GHz AMD Opteron processor and 16GB of RAM. For the experiments, a number of data sets are generated directly from the item and item2dev facts describing the media files of our example user’s data collection as shown in Table 4.1. For smaller data sets, we use random subsets of the original fact list, while for larger ones, we randomly duplicate items in the original list, until the desired data set size is reached. In terms of solving time all
Table 4.6: The full policy set used occurred in Section Evaluation 4.6.

<table>
<thead>
<tr>
<th></th>
<th>Description</th>
<th>Policy Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>recent music on phones</td>
<td><code>policy([[audio_item], [mod_item,#&lt;,86400]], [repall], [[phone_device]]).</code></td>
</tr>
<tr>
<td>2.</td>
<td>photos on any fixed device</td>
<td><code>policy([[picture_item]], [repany], [[fixed_device]]).</code></td>
</tr>
<tr>
<td>3.</td>
<td>music on more than two personal devices</td>
<td><code>policy([[audio_item]], [rep,#&gt;=2], [[pc_device]]).</code></td>
</tr>
<tr>
<td>4.</td>
<td>videos on any home device</td>
<td><code>policy([[video_item]], [repany], [[home_device]]).</code></td>
</tr>
<tr>
<td>5.</td>
<td>public items on at least 2 devices</td>
<td><code>policy([[public_item]], [rep,#&gt;=2], [[any_device]]).</code></td>
</tr>
<tr>
<td>6.</td>
<td>private items on at least 2 fixed devices</td>
<td><code>policy([[private_item]], [rep,#&gt;=2], [[fixed_device]]).</code></td>
</tr>
<tr>
<td>7.</td>
<td>video items recently modified on at least one mobile device</td>
<td><code>policy([[video_item], [mod_item,#&lt;,86400]], [rep,#&gt;=1], [mobile_device]]).</code></td>
</tr>
<tr>
<td>8.</td>
<td>photos in the cloud</td>
<td><code>policy([[picture_item]], [repany], [[cloud_device]]).</code></td>
</tr>
<tr>
<td>8.a.</td>
<td>no private items in the cloud</td>
<td><code>policy([[private_item]], [repnone], [[cloud_device]]).</code></td>
</tr>
<tr>
<td>8.b.</td>
<td>public photos in the cloud</td>
<td><code>policy([[public_item], [picture_item]], [repany], [[cloud_device]]).</code></td>
</tr>
<tr>
<td>9.</td>
<td>photos recently modified on a fixed device at 1-minute latency from the laptop</td>
<td><code>policy([[picture_item], [mod_item,#&lt;,86400]], [rep,#&gt;=1], [[fixed_device], [close_device,'laptop1',100]]).</code></td>
</tr>
</tbody>
</table>
policies are basically equivalent, however, the number of equivalence classes derived from the policies makes a big difference. Graphs shown below depict median values and standard deviation based on 20 repetitions with different data sets.

![Graph showing solving time vs. data set size.](image)

**Figure 4.4: Solving time vs. data set size.**

Two algorithms are compared: the brute force algorithm, which constrains each item’s placement by reasoning item by item, and the equivalence class (EC) algorithm, which generates equivalence classes from the policy set, groups items accordingly, and then solves the placement problem accordingly. Figure 4.4 plots the solving time of the two algorithms for an increasing number of items, where 10 policies are specified. The 10 policies are the ones in Table 4.6 except Policy 8. These 10 policies correspond to 12 equivalence classes. For both algorithms, the solving time increases linearly with the number of items. A dataset of 100,000 items can be reasoned in a reasonably short time. To process the entire data collection of our example user, as shown in Table 4.1, which contains about 15,000 items, it takes less than 5 seconds. The Anzere policy solver can scale to handle a very large number of data items and equivalence classes allow the system to roughly double the speed of the solving process. A linear increase of the solving time for the EC algorithm is due to the overhead of expanding the solution matrix (expressed in equivalence classes) into an execution plan consisting of per-item actions. This suggests that the performance could be improved further by producing actions based on equivalence classes and delegating the conversion into item identifiers to
the devices responsible for their execution.

After proving that the size of the data set does not cause a scalability bottleneck, we will consider the other two variables: the number of devices and the number of policies. An increase in the number of devices brings a linear increase to the solving time as this corresponds to an increase in the number of columns in the solving matrix. As the number of devices in a personal cloud is likely to be small, in the experiments, we keep the overlay size constant at 7 nodes. However, an increase in the number of policies, hence equivalence classes, can cause an exponential growth in solving time. Nevertheless, Anzere still executes reasonably fast with stable behavior when varying the number of equivalence classes from 4 to 78 (corresponding to 4 and 43 policies) and measuring the solving time for data sets of tens of thousands of items. Results are reported in Figure 4.5. Note that the observation above about delegating to overlay nodes the task of converting the solution from equivalence classes to per-item actions also applies here, and could reduce the performance gap between the data sets.

Figure 4.5: Solving time vs. number of equivalence classes.

These results show how the current implementation is capable of fully supporting our target scenario of a single user with a collection of roughly 20,000 data items and 30 policies. The only constraint that remains to consider is the resource overhead of the ECL/PS solver.

Figure 4.6 depicts the solver’s upper-bound memory consumption for an in-
creasing number of items, when the same 10 policies (full policy set except Policy 8) are specified. The upper bound memory consumption is the total heap space and sum of four different stack peaks (storing Prolog variables, backtracking information, checkpoints, etc.). The peak value gives the maximum stack space allocated during the session. For a data set of 10,000 items, the upper-bound memory usage is within 64 MB, and even with 100,000 items, the upper bound is still within 300 MB. These are more than acceptable requirements for current mainstream personal computers. Recall that the overlay coordinator and the CLP solver always run on well-provisioned nodes, not on phones or tablets.

![Figure 4.6: CLP memory consumption.](image)

4.6.2 Reactivity

Next, we will observe the full Anzere system in operation and analyze how the policies enable Anzere to autonomously react and recover from node failures, user mobility, and policy changes.

In the first experiment, we pre-populate the system with about 6,500 media files distributed across all personal devices. Then we permanently crash the home server which stores roughly 1,200 files, thus making the system vulnerable. The policy set used in this experiment is Policy 1-7 from Table 4.6. After the loss of the home server, some policy becomes ineffective such as Policy 4 which specifies to
store videos on a home device. At the same time, the remaining personal devices are insufficient to create enough replicas to satisfy the given durability policy. Anzere reasons about how to recover and reestablish a stable state. As Figure 4.7 shows, it autonomously acquires a VM from EC2 and creates the missing replicas (roughly 1,200 files, for a total of 1.5 GB). The files are copied to the EC2 VM partly from the office PC and partly from the laptop. This process successfully stabilizes the system and decreases the risk of data loss to an acceptable level in a fully automated manner.

In the second experiment, a case of user mobility is emulated. Assume the laptop initially resides in the office network in Europe, and later travels with the user and reappears in the US. In the US, the laptop initiates a read workload – downloading about 70 recently-created photos from the office PC in Europe, or roughly 100 MB. This could be the case of a user living in Europe and traveling to the US for some time. We evaluate the expected duration of such a read workload, when Policy 1-7 and Policy 9 in Table 4.5 are enabled at runtime.

Policy 9 requires recent photos to be copied close to the user’s laptop for fast access. When the laptop appears in the US, the solver reacts by acquiring a local cloud VM and copying the photos to it. Figure 4.8 shows the access time (per photo) achieved by the user before and after traveling to the US; for comparison we also show what would have been achieved by remotely accessing the office PC
CHAPTER 4. ANZERE POLICY-BASED DATA REPLICATION

Figure 4.8: User mobility

(in Europe) from the US. This simple experiment demonstrates the power of our policy architecture. CLP allows for almost any conceivable policy to be specified. The policies can be applied not only for long-term data placement, but also for short-term events, such as downloading bulk data or leaving the home network. The achieved performance improvements are obvious. In our experiment, we used a laptop with WiFi connection for simplicity, but for a mobile phone with limited 3G connectivity, the performance gain would be even more remarkable.

Anzere is also designed to deal with dynamic policy changes. To show this, we use a collection of 1,000 photos (of which 25% are private) and store them on a cloud VM and the office PC. In the initial setting of the experiment, Policy 1-8 from the policy set are active. At runtime we change the policy set by disabling Policy 8, and enabling the following two: Policy 8.a and Policy 8.b. This change causes the system to delete all private photos from the cloud, but also replicate the content previously stored on the cloud to the home server, as the policy set requires at least two replicas of private items on fixed devices. The solver generates roughly 500 actions. As Figure 4.9 shows, photos are deleted from the cloud and at the same time, the same set of photos is copied to the home server (from the office PC). The system automatically reacts to the policy change and reestablishes a stable state.
4.6.3 Optimization

As mentioned in Section 4.4.3, the CLP solver can optionally include a user-defined objective function and find the data placement that optimizes this function. This section evaluates the overhead of objective function optimization in policy solving.

In this experiment, the policy solver tries to minimize the bandwidth usage while satisfying all the requirements of the policy set. As shown in Figure 4.10, the system scales well up to 30 equivalence classes, but after that the solving time can be over 100 seconds for large data sets. This result suggests that using optimization is feasible only with small policy sets, and its overhead is justified only if the cost metric (bandwidth usage in this case) varies considerably across the possible solutions.

In the current implementation, the optimization bottleneck is the ECL\(^1\)PS\(^e\) branch_and_bound search library, which has shown to become very slow with more than 100 variables (corresponding to roughly 32 equivalent classes). Two possible directions could be investigated to address this performance issue: modern solvers are capable of handling a much higher number of variables, in the order of 1000 variables [MSF12], and could therefore improve the search time; alternatively, the search algorithm could be replaced by some heuristic customized implementation.

Figure 4.9: Change in privacy policy


4.7 Summary

We present a declarative way to partially replicate data items at scale according to expressive policy specifications, and show how such replication systems can scale while supporting much more expressive policies than previous schemes: data replication expressed as constraints, devices referred to by predicates rather than explicitly named, and replication to storage nodes acquired on-demand from the cloud. These extensions introduce considerable complexity in policy evaluation, but the solver can still scale well by using equivalence classes to reduce the problem space and gain a better performance. The resource overhead for running the policy solver is also reasonably low.

The proposed policy model and solver are validated in the Anzere personal data storage and replication system via deployment in a real personal cloud consisting of phones, PCs, cloud virtual machines, etc. using simulations and realistic user data. With the specified policies, Anzere cannot only replicate data items in a stable system status, but can also autonomously react to device failures, policy changes, and user mobility.
4.8 Conclusion and future work

The Anzere personal data storage and replication system is a proving application for the Rhizoma dynamic resource management framework. Meanwhile, it also serves as an illustrative example of how applications can use the facilities of the Rhizoma solver to drive their own behavior as well. As mentioned at the end of the previous chapter and discussed in detail in this chapter, Anzere extends the functionality of Rhizoma to support more heterogeneous platforms, resource reasoning as well as application logic reasoning.

With the framework in place, extending Anzere to deal with other content type is mostly straightforward. Dexferizer \([\text{UR11}]\) data transfer optimization service is one such example. Dexferizer optimizes the transfer of data objects within a user’s collection of computers and personal devices, subject to a variety of user-defined quality metrics such as cost, power consumption, and latency. By abstracting object transfer as a high-level service, and employing declarative networking techniques to cast object transfer as a constrained optimization problem, Dexferizer transparently exploits techniques as diverse as swarming and multi-hop transfer through virtual machines.

Anzere only considers how a single-user would store and replicate his/her personal data. One interesting question is then how to efficiently share data among users in a multi-user scenario. The concerns about data ownership, level of trust, visibility to other users’ data collection in personal cloud architecture make the data sharing problem a rather challenging one. Some colleagues from the group make some initial attempts at optimizing object transfers among personal clouds.

As for the future possible directions of the Rhizoma approach (decentralized and declarative resource management), here are some thoughts:

One possible direction is to build the execution model for an application and find its optimal deployment and execution across different cloud computing providers. In addition to the computation and storage resources which Rhizoma and Anzere have taken into account, data communication is also an important factor to consider given the different pricing for requests and data transfers into, out of and within the cloud.

Another interesting direction to pursue is to build a more practical utility function for the application. In Rhizoma, the utility function only describes how much one specific deployment configuration satisfies the application’s resource requirements. It would be desirable for the utility function to be able to describe the performance of the application given a specific resource offering configuration. This requires an accurate performance model for the application which depends very much on the characteristics of the application.

An even cooler design would turn Rhizoma from a self-managing system to a self-optimizing one. For the moment, the utility function as well as the cost func-
tion are given and fixed during Rhizoma’s life cycle. This can be improved with a similar idea to profile-based optimization in compiler design, which means Rhizoma auto-tunes its utility and cost function based on performance measurements and feeds application-level metrics back into the optimization process.
Part II

Virtual infrastructure resource allocation
In the first part of the thesis, we discussed how to apply declarative techniques to resource management over virtual infrastructures on the client side. The client application is organized as a decentralized overlay and expresses its resource requirements as a constraint optimization problem. The management runtime which is closely coupled with the application itself decides to acquire/release processing or storage resources on-the-fly. It then takes action accordingly and redeploy the application autonomously to continually meet the application’s resource requirements, and to maximize its utility as both external conditions and resource needs of the application itself change.

In the second part of the thesis, we will address a different but closely related problem: how to apply declarative techniques to resource management/allocation over virtual infrastructures on the provider side.

The rise of virtual infrastructures has renewed research interest in the design and implementation of resource allocators which allow users to reserve or use combinations of (virtual) nodes, switches, and network links on a variety of virtual infrastructures ranging from network testbeds to grid computing facilities.

As dependencies between resources (switch ports, communication links, virtual machines) become more constrained, resources become more diverse (specialized switches, programmable middleboxes, etc.) and infrastructures scale to large numbers of clients, sites, and network elements, it becomes more difficult for clients to express their requests to the providers of the infrastructure, and, in turn, for these providers to allocate resources in a way that makes efficient use of the infrastructure.
This motivates the second part of the thesis. The basic problem addressed in this part is how a client of an infrastructure provider (e.g. a networking testbed, cloud hosting service, or grid installation) requests resources, how the provider allocates such resources, and how the allocation of such resources is returned to the client. “Resources” in this sense include virtual machines, virtual switches and routers, shares of physical network links, and the like.

Though closely related, the client and the provider have different design goals to manage the resources over virtual infrastructures. The client focuses on satisfying its own resource requirements and maximizing its utility in the face of external condition changes as well as its own resource requirement changes. The provider, however, focuses on satisfying as many clients’ resource requirements as possible and maximizing the utilization of the whole infrastructure. This separation of concerns is new, resource management in many-core OS has a similar separation. For example, for interacting software components (Cells), two level scheduling separates global decisions about the allocation of resources to Cells from application specific scheduling of resources within Cells [CBC+10].

The second part of the thesis, therefore, will focus on how to apply declarative techniques to resource management over virtual infrastructures on the provider side. It starts with the virtual infrastructure reference model which is used as a reference to survey how resource allocation is done today in three main application areas: testbeds, grid- and cloud-computing systems. These related work serves as the background for the second part of the thesis. In this chapter, we also point out the main research challenges in the field, some of which are addressed in detail in the following chapters.

Since the area is still relatively new, research work on resource allocation for virtual infrastructures lacks realistic benchmarks. Therefore, in the next chapter, Chapter 6, we suggest how to generate realistic benchmarks for virtual infrastructure resource allocators. The general benchmark methodology is presented with our workload generator as an example, which is based on a 5-year trace of experiments submitted to the popular Emulab testbed 1. We demonstrate the importance as well as potential applications of such benchmarks by pointing out various evaluation scenarios where the generated test workload could be used.

The generated workloads are actually used in different evaluation scenarios. They are used in Chapter 7 to compare different versions of the proposed virtual network mapping algorithm (VF2x), and in Chapter 8 to investigate the dynamic behavior of the Arosa resource allocator and to evaluate the “late-binding” resource reservation strategy.

1We would like to thank Robert Ricci from the Flux research group, University of Utah, for his generous sharing of the Emulab trace and many helpful advices, suggestions to our research work.
5.1 Resource allocation reference model

Resource allocation systems typically involve at least two stakeholders: *infrastructure providers* and *resource clients/consumers*. Some systems also include *brokering intermediaries*, which coordinate and federate infrastructure providers and offer their resources to a set of clients.

As many contemporary testbeds, our reference model also distinguishes between *tickets* and *leases*: leases are issued by the testbed and convey an immediate authorization to access specific, named resource components allocated to the user, whereas tickets are reservations for a set of resources which must be redeemed (and changed for a lease) at the appointed time.

Figure 5.1 depicts a general resource allocation reference model from the perspective of a resource client. It includes two stages: *resource reservation* and *resource allocation*. This reference model pretty much encompasses any existing resource allocation system, and it is very general that almost all resource allocation systems are a (non-strict) subset of this model – for example, some choose to conflate leases and tickets, some avoid the resource reservation stage. This reference model has been published in APSys 2012 [YR12a].

![Timeline of a resource request](image)

Figure 5.1: Timeline of a resource request

During the resource reservation stage, the user submits the resource *request* specifying the desired virtual network which includes the elements in the virtual network, constraints on and dependencies between these elements, and the period of time for which the requested resources are desired. If satisfiable, the provider will return a *ticket* as a reservation of a set of resources. The ticket, signed by the provider, is actually a promise by the provider to bind the resources to the client and grant access to them at a specific point in the future. The request and the ticket might use the same specification language. There are some more interesting properties about *tickets*:

- The ticket (and also the request) can be either “bound” or “unbound”. The “bound” ticket, which is used in most existing systems, serves as a reservation
of a set of concrete resources. The “unbound” ticket describes elements in the virtual network not yet mapped to physical components, and concrete resources will be bound to the ticket at a later time.

- More general operations such as splitting and merging of tickets are also possible, in line with existing testbed proposals.
- Tickets can be returned later to the testbed with a request, and atomically exchanged for a new ticket.

Given a ticket, a user may redeem it for a *lease* before its expiry time (some time before the planned start time of the resource usage). This redemption confirms the allocation of concrete resource components, activates these resources and allows access to them at the specified start time. The user may surrender the ticket in case the reserved resources are not needed any more and the resource allocation will not take place. Leases, however, must always refer to specific resources (and, indeed, grant access to them). The user may also surrender the lease in case the activated resources are not needed any more, but it will involve some cost.

The design and implementation of resource allocators vary in how to represent the resource requests, what to return in tickets and leases, which virtual network mapping algorithm to use, how to satisfy multiple requests while optimizing utilization, revenue, etc.

### 5.2 Resource allocator examples

In this section, various resource allocation systems are discussed in the context of the proposed reference model which helps to define the design space of resource allocators. The allocation systems discussed here include networking testbeds, cloud hosting services and grid systems.

#### 5.2.1 Testbeds

Network testbeds are used by network researchers to experiment with new protocols, applications, and systems, and are generally shared between institutions to reduce the considerable capital cost involved. A testbed typically consists of a set of physical resources – compute nodes (usually PCs), switches, links, etc. – together with a control plane which allocates resources. Testbed users submit “requests” in the form of specifications of particular network configurations (nodes, topology, etc.) they would like to experiment with, when the resources are needed, and for how long. In response, the control plane allocates, if possible, a set of resources to the user at the requested time.
5.2. RESOURCE ALLOCATOR EXAMPLES

The resources can be physical machines or links, but are often virtual “slices” of real resources. Such multiplexing greatly increases the number of users the testbed can support at a time. Unlike commercial facilities, network testbeds are typically run as community affairs funded by grants and participating institutions, so the main goal is to handle as many experiments as possible.

We start with systems that purely allocate **virtual machines**. Early distributed testbeds like PlanetLab [Pla12] allow clients to specify precisely the physical machines on which to create single, independent virtual machines (sometimes called *slivers*), and all individual physical resources (close to 1000 servers in a globally dispersed collection of sites) are advertised and visible to the clients. PLC (PlanetLab Center) supports two methods by which *slices* (each service runs in a slice of PlanetLab’s global resources) are actually instantiated on a set of nodes: *direct* and *delegated*.

The *direct* model does not support the concept of *tickets*. The user specifies the set of machines (among the advertised full set of machines) that they would like to run the experiments/services. The corresponding VMs/slivers are created and resources are assigned to them (the lease is activated) as soon as the slice is added to those nodes through the `AddSliceToNodes` API provided by PLC. The lease is released if the slice is removed from the node or the slice is expired.

Using *delegation*, the user contacts PLC for a ticket that encapsulates the right to create VMs or redistribute a pool of resources on the nodes where the controller slice is added to. The user then contacts the node manager to redeem the ticket. The node manager creates a new VM (to activate the lease) without regard for available resources and grants access to the resulting “slivers” in the reply.

As a community-run, purely best-effort service, PlanetLab does not need to provide any resource guarantees, make provisioning decisions, or allocate VMs so as to balance load or guarantee performance. Thus, PlanetLab expects slices to engage one or more resource allocation brokerage services to acquire resources. Despite this, the diversity of resources offered by PlanetLab (albeit all resembling Linux virtual machines) has led to third-party resource managers [OAPV04, ATSV06, YSC+09] which allow clients to specify requests for resources in the form of declarative queries over the available nodes. In addition, the design of PlanetLab is general enough to support resource brokerage services on top: Sirius [sir12] performs admission control on a finite resource pool set aside and allows users to reserve VMs system-wide for an hour at a time. Alternative market-based services, such as SHARP [FCC+03], Tycoon [LRA+05], Gacks [CH05] provide mechanisms for trading of resources.

VICCI [Vic12] is a programmable cloud-computing research testbed whose software stack is based on the PlanetLab software distribution. PlanetLab currently supports a best-effort model of resource allocation, and VICCI extends the im-
implementation to support resource guarantees, for example, allowing VMs to be allocated entire processor cores. VICCI’s resource allocation model is the same as PlanetLab’s direct model.

Some testbeds allow the allocation of virtual networks in addition to computational resources. Topological requirements of virtual networks constrain resources further and resource dependencies become more acute: two virtual switches might need to be connected by a share of a physical link, for example.

Emulab [Lep12] network emulation testbed allows users to use ns2 configurations to request a set of nodes for experiments, and to dictate how the nodes are configured to emulate different network topologies. Emulab does not support advance reservation but performs guaranteed-share scheduling. Its resource allocation model is very much like the direct model of PlanetLab. If the resource request is accepted, a listing of nodes (virtual or physical) and IP addresses will be allocated to the virtual network topology for the experiment. The lease will be released if the experiment is terminated or swapped out. Emulab embeds the network requests into its physical topology using network mapping algorithm called assign [RAL03]. assign builds on simulated annealing to find a near-optimal solution in bounded solving time (a few seconds).

Recent proposals for distributed testbeds such as GENI presume a federated, distributed physical infrastructure over which virtual networks (“slices”) are instantiated, and are designed to support “at scale” research in networking and innovation in services. GENI [Gen12] aims at both high best-effort utilization and admission control for guarantees to some clients. Thus, some subset of global GENI resources will be subject to admission control at a fine granularity, while most applications will run in the best-effort way. GENI includes several control frameworks: PlanetLab VINI, ProtoGENI, ORCA/BEN, etc.

VINI [Vin12] is a PlanetLab-like testbed where users can configure virtual topologies within their slices. VINI’s resource allocation model is the same as PlanetLab’s direct model. VINI users have to get a description of the available resources which tells the physical nodes and links on which it is possible to create VMs and virtual links. The user then describes the desired resources to the slice by tagging the nodes as sliver and links as vlink. If the submission of the description is successful, the tagged resources will be allocated accordingly.

ProtoGENI [Pro12] is largely based on Emulab software, and its resource allocation model is a subset of the reference model. ProtoGENI uses tickets to achieve an optimistic reservation mechanism: the users make the resource availability query right before running the slice embedder, then attempt to get tickets for the components that were chosen. In the current implementation, a “use it or lose it” policy is applied – after getting the ticket, the client must redeem it for a lease within tens of minutes, or it may be given to another user. In the future,
5.2. RESOURCE ALLOCATOR EXAMPLES

ProtoGENI will use tickets for reservations.

ORCA/BEN [ORC12] is a control plane (ORCA) for a metro-scale optical testbed (BEN). ORCA control plane [Cha09] also differentiates tickets from leases. Resource reservation is not supported yet – but will be in the future. If the application’s request for \( u \) slivers of resource type \( r \) is approved, a ticket will be issued for \( u \) units from a specific pool, and the client later redeems the ticket to obtain a lease for the resources. ORCA uses an ontology language called NDL-OWL to express rich resource requests as queries, including topology embedding.

The concepts of network virtualization are not totally new. The early research work explored how to virtualize networks efficiently to support multiple, custom control planes over a single physical infrastructure. Representative work includes open signaling [CDMK+99], Tempest [MRLC97], and the Genesis Kernel [KCC+01]. More recently, OpenFlow [MAB+08] has applied similar ideas to IP-and Ethernet-based networks. Based on this, systems like FlowVisor [SGY+10] act as transparent proxies between OpenFlow switches and controllers to provide slices of a physical network. In the Internet architecture research, so-called “pluralists” even view support for co-existing virtual networks as the key feature of the Internet architecture [APST05].

5.2.2 Grid computing system

Grid computing federates a bunch of loosely coupled, heterogeneous, and geographically dispersed computer resources from multiple administrative domains. These computer resources are accessed by non-interactive jobs that involve a large number of files. For example, the Globus grid system [Glo12] distributes large scientific applications across a set of computer resources.

Several grid resource allocation models are discussed in [FBA+03]. In the simplest form, the user may select a machine suitable for running the job and then send the job to the selected machine. More advanced grid systems would include a job “scheduler” which applies a batch scheduling job queue: when a job reaches the front of the queue, it gets dedicated access to the resources for its run. As a further step, a block of machines can be reserved in advance for a designated set of jobs with deadlines or quality of service requirements.

The scheduling and reservation process is fairly straightforward when only one resource type, usually CPU, is involved. As an example, Condor [Con12] allocates machines to parallel jobs based on a match-making mechanism called ClassAds. Machine characteristics and job requirements are represented in a common framework making it possible to decide, for a particular request-resource pair, whether the requirements are satisfied.

However, additional grid optimizations can be achieved by considering more resources in the scheduling and reservation process, and the dependencies be-
between resources motivate a richer specification for resource requests. Systems like RedLine [LF04] use constraints to describe resource offerings and requests, treat resource matching problem as a constraint satisfaction problem, and explore constraint-solving technologies to implement matching operations. RedLine can model and solve bilateral, gang, set and congruence matching problems.

More recent grid computing research work shows more and more interest in network virtualization. One example is VNET which builds layer 2 virtual networks for virtual machine grid computing [SD04]. Another example is the user-controlled lightpath provisioning (UCLP) system, which integrates lightpath resources with data and computation Grids. UCLP allows end users provision and dynamically reconfigure optical networks, as well as create customized logical IP Articulated Private Network (APNs).

5.2.3 Cloud computing systems

Acquiring resources on public clouds is conceptually similar to the PlanetLab case. Public clouds like Amazon EC2 [Ec12] have many more machines than testbeds or grid systems, but advertise a small number of VM classes (based on location and approximate computing power), each of which consists of a large homogeneous pool. While clients can request VMs in a specific location, EC2’s scale means that these VMs can be allocated entirely independently of each other. Resource allocation for public clouds differs from testbeds in three ways as shown below. These factors allow a simple and intuitive API to EC2 and simplify Amazon’s task of provisioning physical plant.

- Firstly, machines are allocated piecemeal and access to the VMs is granted at the time when a request is made – there is no notion of (or need for) future reservations of resources.
- Secondly, until recently allocating shares of the interconnect between nodes was not a feature offered by providers.
- Finally, in cloud backends, the use of admission control and capacity planning ensures that jobs have enough resources to run with little wait time, something hard to achieve in network testbeds.

Private/hybrid cloud systems like Eucalyptus [Euc12] and OpenNebular [C1212] take the middle ground between public clouds and testbeds. They encourage third-party development of the scheduler module. For example, the Haizea [SMLF09] lease manager for OpenNebula leases VM resources under a variety of terms, including rank-based match-making scheduling, reservations and best-effort request queuing.
5.3 DISCUSSION

Topology awareness has recently attracted attention from cloud systems. Topology aware resource allocation [LTRK11] in IaaS-based cloud systems aims to better support data-intensive workloads by making providers more aware of hosted application requirements and giving users fine-grained visibility into, or control over, the infrastructure.

5.3 Discussion

The above discussed resource allocation systems share many similar research challenges to allocate resources from an underlying shared physical infrastructure to requests which are composed of virtual machines, virtual links and virtual routers. These challenges have inspired research in a number of related areas.

One example is the design of efficient network mapping algorithms: how to map a virtual network (VN) topology with resource constraints to specific nodes and links in a given shared physical network (PN) infrastructure so as to accurately emulate the network properties requested by the user. This may involve virtual network embedding, resource constraint satisfaction, and even concepts from operations research such as yield maximization, in mapping multiple requests for resources to a given physical infrastructure.

Another is the design and implementation of resource allocators: how users can specify their resource requirements, how resource providers can satisfy many users while optimizing utilization, revenue, power consumption, or some other metric of interest. There is also research work investigating different resource allocation strategies: guaranteed reservation or “best-effort” allocation or both, “bound” or “unbound” resource reservation promises, allowing resource overcommitments or not, etc.

However, since the area is still relatively new, research work on resource allocation lacks realistic benchmarks to evaluate system performance, to compare design ideas, etc. Therefore, a general methodology to generate realistic benchmarks for virtual infrastructures is presented in Chapter 6 with our workload generator as an example. This workload generator is based on a 5-year trace of experiments submitted to the Emulab testbed.

A novel virtual network mapping algorithm, VF2x, is discussed in Chapter 7. Based on the VF2 subgraph isomorphism detection algorithm designed for matching large graphs, VF2x incorporates several novel algorithmic improvements. These and careful implementation make VF2x perform more than two orders of magnitude faster than the fastest previously-published algorithm vnm-Flib (also based on VF2). The performance of VF2x is evaluated by an extensive test workloads generated from the above workload generator. It is shown that VF2x can allocate resources to virtual networks on a large testbed in a matter of
seconds using commodity hardware.

A new approach to negotiating virtual resources between clients and virtual infrastructure providers is introduced in Chapter 8. Clients specify their requests as constraints, and the providers reply with resource allocations expressed also as declarative set of constraints on resources. This gives providers more flexibility in late-binding resources to requests, and opens up a wide design space to optimize resource allocation for various metrics, such as utilization, and can help to mask failures of physical nodes from clients. This allocation strategy is also evaluated with an extensive test workload generated from our workload generator. The experiments suggest that late-binding resources to requests enabled by representing resource reservation as constraints achieves better network resource utilization compared to the fixed assignment solution, and better masks network failures from clients with resource reservation.
The research work emerging with the rise of virtual infrastructures calls for benchmarks to compare different virtual network mapping algorithms, to investigate dynamic resource allocation behavior, to evaluate various resource allocation strategies, etc. The evaluation criteria also vary according to the evaluation scenario. For example, evaluation criteria to compare various virtual network mapping algorithms could be solving time, number of successful mappings, revenue-to-cost ratio, etc. However, to date there is no clear consensus on what constitutes a realistic workload for such systems, let alone on a common benchmark for comparing ideas.

Existing work mostly uses simulators to generate virtual network requests and substrate networks for evaluation. However, randomly generated graphs cannot faithfully represent real requests and physical infrastructures. Such synthetic workloads may even miss some key aspects in reality that significantly impact system design, while benchmarks based on real trace are less susceptible to this.

We argue that the accuracy of system performance evaluation results depends on the accuracy and realism of the benchmark workloads used, and realistic benchmarks should be constructed from a realistic workload characterization model which can only be derived from real-world workload traces. Another desirable feature for the benchmark workloads is generality. In other words, generated benchmark workloads are preferably general enough to be applicable to a range of evaluation scenarios.

Of course, this is not a new position and a similar idea has been applied in benchmarking database management systems, operating systems, web services,
This chapter is going to be a short one and will cover:

- A general methodology to generate realistic benchmarks for virtual infrastructure allocators;
- An example workload generator based on a 5-year trace of experiments submitted to the Emulab testbed [Emu12].

The workload generator plays an important role in the following two chapters, and the generated test workloads are applied in different evaluation scenarios which will be discussed in detail in the corresponding chapters.

Even though we focus on realistic benchmarks for testbeds, the general methodology presented in this section is also applicable to grid and cloud systems. The contents presented in this chapter have been published in APSys 2012 [YR12a].

### 6.1 Related work of benchmarking

Benchmarking consists of running a system to evaluate its performance with respect to some pre-defined benchmark metrics, typically by executing a number of standard well-designed benchmark workloads against it. Benchmarking is essential for evaluating system performance, debugging, comparing design ideas, etc. It is important and well-established in many areas such as database management systems [TPC12], high-performance computing [HPC12], HTTP servers [Apa12, BC98], I/O systems [CP94], operating systems [KASR11], parallel job schedulers [CCF+99], etc.

Benchmark workloads are used to test both the current available system and new emerging system. The generation of realistic workloads requires an accurate workload characterization model which captures the key features of the real-world workload traces and preferably is able to predict the future workload. Martin characterized Internet web server workloads and analyzed their performance implications [AW97]. Others have proposed comprehensive models for supercomputer [CB01] and parallel computer [SEY04] workloads. Recently, Ganapathi et.al. have proposed statistics-driven workload modeling for the cloud [GCF+09], and several groups have analyzed Google compute cluster traces [MHCD10, CGGK10, ZBH11]. The workload characterization results are used to predict workload patterns [KYTA12] and to synthesize realistic workloads [WBMG11]. Calzarossa et.al. present a thorough survey of workload characterization [CS93] and analyze the issues and methodologies involved [CMT00].
6.2 General methodology

In terms of the benchmarks for virtual infrastructure resource allocators, it is not yet clear what a representative workload for a resource allocator is. The reason is that virtual infrastructures are still relatively new, particularly those which are distributed in nature and permit complex topological requests.

In this section, we present a general methodology to generate realistic benchmarks and use our workload generator as an example. Our workload generator is based on a trace of Emulab experiments for the 5 years prior to June 2007. While we make no authoritative claims that this is representative (indeed, the information in the trace is limited and Emulab is unlikely to be representative of other virtual infrastructures), we argue that it is at least based on plausible data and assumptions.

Based on a thorough survey of related work in other fields, we summarize the steps to realistic workload generation:

- Parameter selection: choosing the parameter set which can thoroughly describe the system behavior;
- Trace collection: using monitoring tools to collect system workload trace;
- Workload characterization: analyzing the collected trace, characterizing it using statistic methods to construct workload distribution models;
- Workload generation: using the constructed model to generate workloads.

6.2.1 Parameter selection

The choice of the parameter set depends critically on the system for which the benchmarks are generated. According to the virtual infrastructure resource allocation reference model, we think the following parameters are of particular importance to characterize VN request traces:

- VN topology requests: the requested virtual networks with the constraints on the requested hosts, switches and links as well as how these network components are connect;
- VN request arrivals: the time when the requests are submitted;
- Lead time: the time between request submission and lease activation;
- Duration: for how long the resources are used;
• $\Pr(\text{ticket surrender})$: the probability that a ticket is surrendered before it expires;

• $\Pr(\text{ticket expiry})$: the probability that a ticket expires;

• $\Pr(\text{lease surrender})$: the probability that a lease is surrendered before it expires after the specified duration.

### 6.2.2 Trace collection

The collection of the trace used to generate realistic benchmarks should ideally use non-intrusive measurement or monitoring tools, possibly merge information from different sources, ensure a long enough time span, cover details of all the selected parameters, and retain anonymity to ensure users’ privacy.

If the resource allocation reference model is followed strictly, the collected trace should include at least the following information for each request: requested constrained network topology, desired duration, timestamps for resource reservation, lease activation and release, as well as all the possible events of ticket surrender, ticket expiry and lease surrender.

The Emulab resource allocator follows only a non-strict subset of the reference model, therefore, its trace doesn’t provide all the information listed above. Furthermore, even though Emulab logs cover many details of its system execution, the published Emulab trace is limited due to anonymity reasons. Nevertheless, it is a valuable source of information: the trace contains the complete sequence of Emulab topology requests for 5 years up to June 2007, essentially covering every experiment submitted to Emulab before that date.

### 6.2.3 Workload characterization

Since the behavior of a real workload is complex and difficult to reproduce, a compact, repeatable and accurate model is needed. Such a model has to capture the statistical characteristics of the real load. More specifically, for the resource allocation reference model, given a complete and detailed workload trace, a generative workload model should be able to describe the frequency distribution of VN requests, the distribution of the request inter-arrival time, the distributions of lead time and duration, probabilities of ticket expiration and early ticket/lease surrenders. A more realistic workload model should also capture the relationship among these parameters. The statistical models which describe the request stream, the temporal characteristic models as well as the probability models of ticket/lease events can later be used to generate realistic workloads.

The published Emulab trace only contains information about the virtual network requests and the physical network topology snapshots without the original
6.3. EMULAB TRACE-BASED WORKLOAD GENERATOR

request timestamps. The limited information provided by the trace led us to a mixed approach: sampling the complete trace to generate request stream as well as making assumptions about the missing parameters.

6.2.4 Workload generation

The workload generator should be able to synthesize a rich set of new customized realistic workloads which automatically scale across current and future virtual infrastructures. Statistics-driven workload generator generates workloads by sampling the distributions of the selected parameter set.

The distribution models used in our Emulab trace-based workload generator are either characterized from the trace or assumed if the parameters are missing. Based on these distribution models, we annotate the Emulab request stream with different time parameters – when to request resources, when to allocate resources and when to release them – and generate workloads from this.

Actually, our workload generator can synthesize various workloads applicable for different evaluation scenarios under different variants of the resource allocation reference model (e.g. with or without resource reservation phase).

6.3 Emulab trace-based workload generator

The published Emulab trace contains 23818 anonymized experiments which are in the format of 23GB of .top and .ptop files. A .top file describes a virtual topology requested by an Emulab user, and a .ptop file describes the physical network topology that was available at the time the system tried to swap in the experiment. In the trace, there are 127586 .top files (virtual topology requests) among which 52080 are unique - since identical .top files belonging to the same experiment are stored between different swap-ins.

The request stream for our workload generator is extracted from the available trace by sampling the complete 5-year request set. Virtual network requests include information about several dimensions: requested VN size, topology, node and link constraints, etc. Figure 6.1(a) shows more information about the distribution of the requested VN network sizes. About 43% of the requests are small, relatively simple topologies with fewer than 5 nodes. Half of the requests are medium-sized networks with 6-100 nodes, and about 7% of them are large networks with more than 100 nodes.
Figure 6.1: Requested virtual networks
The number of components of the requested virtual networks is shown in Figure 6.1(b). As for the number of components of the requested virtual networks, only 12% have multiple components and a majority of the requests are connected graphs. For these 88% connected graphs, 13.3% are single nodes, 63.7% have tree topologies and 10% have other topologies.

We made assumptions about the distributions of the parameters missing from the trace, such as temporal and probability parameters.

Request arrivals are modeled as a Poisson process. Poisson distribution expresses the probability of a given number of events occurring in a fixed interval of time and therefore is well suited for modeling request arrival rates. As a reference, we roughly calculated the request arrival rate for the Emulab request trace. Based on the fact that Emulab received 127586 top requests over 4 years, we conclude that Emulab received 4 requests per hour on average. Using 4 as a base value, we vary arrival rate parameter $\lambda$ to increase or decrease offered load. This distribution model naturally does not take into account load spikes on the system (such as before conference deadlines).

Lead time and duration are modeled as a Gamma distribution. Gamma distributions have long been used for modeling demand distribution in queuing systems [SGA91, Jai91]. It has two parameters: shape($k$) and scale($\theta$). The conclusion from Jelena et.al.’s comparative study of network testbed usage characteristics [MHS12] confirms the validity of our assumption about request duration. The paper shows that distributions of features such as VN request durations (VN sizes as well) are heavy-tailed: they span a wide range of values with most points clustered at small values, and with few points residing in the long tail.

For our workload generator, we make some further assumptions about the probability parameters to simplify the problem: first, ticket expiry probability is 0, meaning that all the tickets are redeemed before they are expired; second, early ticket surrender probability is 0, meaning that all the tickets are redeemed for leases; third, early lease surrender probability is 0, meaning that resource leases are always released after duration time period.

The generated request workloads are actually used in different evaluation scenarios, as explained in detail in the following chapters, to demonstrate the importance as well as potential applications of such benchmarks.

6.4 Summary

We present the general methodology to generate realistic workloads for virtual infrastructure resource allocators, and these are the first steps towards a standardized benchmark suite for virtual infrastructure resource allocators. We argue that such a goal is an important one for the research community to make progress
in this field.

While the Emulab trace we use has only limited information, it does provide a first cut at creating workloads to test a variety of properties of allocators in a range of different scenarios. The workloads are used in Chapter 7 to compare different versions of the proposed virtual network mapping algorithm (VF2x), and in Chapter 8 to investigate the dynamic behavior of the Arosa resource allocator and evaluate the “late-binding” resource reservation strategy.

In Chapter 7, the generated test workload is used to compare different variants of the proposed virtual network mapping algorithm VF2x with respect to the solving time, as well as scalability. The experimental evaluation is conducted by mapping a set of generated VN topology requests to the same physical network separately or by exhausting the physical network. The test workload for the evaluation, which has no temporal attributes, is sampled from the set of all Emulab virtual network requests.

In Chapter 8, the generated workload is first used in a simple resource allocation model (the resources are allocated at the moment the request is submitted, if the resource requirements can be satisfied) to investigate Arosa resource allocator’s dynamic behavior under continuous requests and varying degrees of request load. The generated workload is also used in a more general model (resource reservation is supported) to evaluate the strategy of expressing resource offers as constraints and late-binding resources to requests. The results show that late-binding achieves superior infrastructure utilization and partially masks the effects of physical node failures from users.
VF2x virtual network mapping algorithm

Virtual infrastructures consist of a set of physical resources – compute nodes (usually PCs), switches, links, etc. – together with a control plane which allocates resources. Users submit resource requests in the form of specifications of particular network configurations (nodes, topology, etc.) they would like to experiment with, when they require the resources, and for how long (they need them). In response, the control plane allocates, if possible, a set of resources to the users at the requested time. Resources can be physical machines or links, but are often virtual “slices” of real resources.

The allocation of local resources, such as CPU and memory, is well-studied in networking testbeds, cloud hosting services and grid systems as surveyed in Chapter 5. In networking testbeds like PlanetLab, resource allocation brokerage services [OAPV04,ATSV06,YSC+09] allow clients to specify requests for resources in the form of declarative queries over the available nodes. The Condor grid system allocates machines to parallel jobs based on a match-making mechanism called ClassAds [Sol04]. Amazon EC2 allocates VMs with requirements on location and approximate computing power from a set of large homogeneous pools.

Virtual infrastructure multiplexing enables a plurality of diverse virtual networks to coexist on a shared physical network infrastructure [APST05,TT05,CB10]. Widely distributed testbeds like GENI [Gen12] aim to support a potentially large number of experiments simultaneously on a complex, widely distributed physical network by mapping each requested network onto a share or “slice” of physical hosts, switches and links. Topological requirements of virtual networks constrain resources further and resource dependencies become more acute. These
make the problem of resource allocation for virtual networks more challenging.

A significant challenge in the context is virtual network mapping: how to map a virtual network (VN) topology with resource constraints to specific nodes and links in a given physical network (PN) infrastructure so as to accurately emulate the network configurations requested by users.

This mapping problem is difficult in theory and in practice due to the four properties summarized in [YYRC08]: diverse topologies, resource constraints, online requests and admission control. Even simplified variants of the mapping problem with relaxed properties turn out to be difficult: assigning nodes in a switched Ethernet-connected testbed without violating bandwidth constraints can be reduced to the NP-hard multiway separator problem [And02], and even when the nodes are preselected, the link mapping problem for VN requests with link constraints is still NP-hard [YYRC08].

In this chapter, we present VF2x, a new algorithm based on the VF2 subgraph isomorphism algorithm for matching large graphs [PCFSV04]. VF2x performs network mapping more than two orders of magnitude faster than the previously-published vnmFlib (also based on VF2) but reduces the solving time for near-worst-case problem instances through more careful implementation and several novel algorithmic changes: constructing a candidate queue, applying heuristic sorting, and introducing a new “timeout-and-relax” strategy. The contents presented in this chapter have been published in TridentCom 2012 [YR12b].

In the following section, we first provide some background and related work of the problem, and in Section 7.2 we describe the VF2x algorithm, discuss its implementation issues and algorithmic improvements, and evaluate its effectiveness through synthetic workloads. Using the test workload generated by the Emulab trace-based workload generator, we evaluate the performance of VF2x in Section 7.3. In Section 7.4 we make a conclusion.

### 7.1 Background and related work

In a federated, distributed physical infrastructure over which virtual networks are instantiated, low cross-sectional bandwidth is expected. However, recent virtual network mapping algorithms make different assumptions in order to apply efficient heuristics to make the mapping problem tractable.

In networking testbeds, for example, early centralized testbeds supporting specification of network properties, such as Emulab [Lep12] and DETER [det04], emulate a variety of network topologies using a small number of high-port-count, high-capacity switches to approximate a physical crossbar between machines. This testbed mapping problem with bandwidth constraints has been proven to be NP-complete [MAS10]. Simulated annealing has been successfully applied to this
situation [RAL03], but does not work well for large-scale virtualized network testbeds [HRS+08].

Recent proposals for distributed testbeds such as GENI presume a federated, distributed physical infrastructure. At present, the various proposed GENI frameworks address the mapping problem in different ways: ProtoGENI [Pro12] inherits the existing assign mapping algorithm from Emulab; ORCA-BEN [Cha09] uses NDL-OWL ontology language to express substrate and a sequence of request and release operations, and relies on Jena RDF and OWL reasoning engines to perform topology mapping [BXE+09]. The cost of the mapping operation for ORCA-BEN is not prohibitive since the BEN network is small.

Zhu and Ammar [ZA06] assume unlimited physical network resources and then try to achieve low and balanced load on both physical nodes and links. Their algorithm, VNA-I, subdivides the general topology in the VN request into multiple small star topologies, and then exploits the flexibility of these small topologies.

Lu and Turner [LT06] consider the offline problem of mapping a virtual network with a backbone-star topology to the physical network with the aim of getting sufficient capacity to accommodate any traffic pattern specified by traffic constraints.

Yu et al. [YYRC08] propose a two stage solution to the problem. They first map the virtual nodes, and then map the virtual links using the shortest path and multi-commodity flow (MCF) algorithms. The authors focus on improving the link mapping through substrate support for path splitting and migration. Razzaq and Rathore [RR10] also propose a two stage solution, but without the assumption of substrate support: virtual nodes are first mapped as closely as possible to the physical network, then virtual links are mapped to the shortest paths which satisfy the demands.

Chowdhury et al. [CRB09] propose algorithms which provide better coordination between the two stages. The virtual nodes are mapped to the physical nodes in a way that facilitates the mapping of the virtual links to the physical paths in the subsequent phase. The mapping problem is solved using a Mixed Integer Program (MIP) formulation. The authors also assume substrate support.

Several algorithms consider network migration and reconfiguration. Butt et al. [BCB10] note the importance of differentiating physical network resources and argue that topology-aware mapping, together with reoptimizing bottleneck mappings, can improve the acceptance ratio. Schaffrath et al. [SSF10] formalize the mapping problem as a linear mixed integer program and allow dynamic reconfiguration of existing mappings.

Of particular interest is the vnmFlib network mapping library implemented by Lischka and Karl [LK09]. Noting the relationship between the network mapping problem and subgraph isomorphism detection, they develop a backtracking algorithm based on the VF2 algorithm used in the pattern recognition community for
finding subgraph isomorphisms in large graphs. VnmFlib maps virtual nodes and links in a single stage and achieves better and faster mappings than the two-stage approach used by Yu et al. [YYRC08], and works especially well for large virtual networks with strong resource constraints.

While many algorithms, including VF2x, can produce optimal results (subject to some utility function), this frequently involves an exhaustive search and is therefore expensive. Moreover, it is rarely required; in practice a near-optimal solution in reasonable time is preferable. Emulab’s simulated annealing algorithm [RAL03] is a good example of exploiting this property.

However, the Emulab approach suffers from two limitations: firstly, it does not always return the same result due to its use of randomness, which makes the debugging of the algorithm challenging. Furthermore, it sometimes fails to find a solution that satisfies all the constraints even if such a solution exists. This is illustrated by the “ugly” example included in the Emulab source code. In contrast, VF2x is deterministic in all its heuristics, and for this “ugly” mapping problem, can find a solution with no violation of constraints in considerably less time.

7.2 VF2x algorithm and implementation

We found vnmFlib [LK09] to be especially suitable for the virtual network mapping problem, as explained at the end of last section. However, in the process of trying to apply vnmFlib directly to our testbed resource allocator, we find several limitations of both the algorithm and its implementation which we will explain in detail below. This motivated us to develop VF2x, our own VF2-derived virtual network mapping algorithm.

7.2.1 Implementation decisions

VF2 is the best known subgraph isomorphism detection algorithm which determines whether $G$ contains a subgraph isomorphic to $G'$. Algorithm 7.1 shows the basic logic of the VF2 algorithm. It is also the main skeleton of VF2-based virtual network mapping algorithms, such as vnmFlib and VF2x. Variable depth maintains account of how many nodes have been successfully mapped so far. At each step, a new pair of nodes $(n, m)$ is generated to try to match against each other. The feasible $(n, m)$ function checks the syntactic (structure of the graph) feasibility of the mapping. If it succeeds, the mapping is remembered; otherwise, we try the next pair until we reach a dead end and backtrack().

The simple example in Figure 7.1 depicts how the algorithm works. Here, we try to match nodes $[a, b, c]$ in $G'$ to nodes $[1, 2, 3, 4, 5]$ in $G$. As shown in the search
Algorithm 7.1: VF2x basic algorithm

| Input: | Attributed graph $G_{vn}$ and $G_{pn}$ |
| Output: | Mapping from $M(G_{vn})$ to $G_{pn}$ |

1. while $|M| \neq |G_{vn}|$ and depth $> 0$ do
2.   $(n, m) \leftarrow$ genpair();
3.   if $n < 0$ or $m < 0$ then
4.     backtrack();
5.     depth $\leftarrow$ depth $- 1$;
6.   else if feasible$(n, m)$ then
7.     $M(m) \leftarrow n$;
8.     depth $\leftarrow$ depth $+ 1$;
9.   end
10. end

The matching process backtracks several times to try new candidate pairs until one solution is found.

$vnmFlib$ and $VF2x$ are two $VF2$-based virtual network mapping algorithms and implementations which try to find a set of nodes and links in the physical network (PN) to emulate the requested virtual network (VN). As with $vnmFlib$, $VF2x$ extends $VF2$ with semantic constraints on virtual node and link attributes (a key requirement for testbed applications) and a predefined distance value $\epsilon$ which, unlike $VF2$, allows virtual links to be mapped to multi-hop paths in the physical network of at most $\epsilon$ length. Furthermore, the $\text{feasible} (n, m)$ function checks not only the syntactic feasibility but also the semantic (node and link attributes) feasibility of the mapping.
VF2x improves dramatically over vnmFlib through a combination of a careful implementation and several algorithmic improvements. The implementation techniques that improve the performance of the algorithm can be summarized as follows: VF2x is implemented entirely in C (about 1,500 lines) based on the existing igraph library [igr12] implementation of VF2; VF2x supports modeling of network topologies as both directed and undirected graphs, while vnmFlib is restricted to topologies modeled only as directed graphs; unlike vnmFlib, VF2x avoids non-tail recursion; and finally, VF2x computes neighbors in a lazy manner and populates the adjacent neighbor table on demand, while vnmFlib recomputes neighbors every time.

By themselves, these implementation decisions make a dramatic difference in performance. Table 7.1 compares our VF2x implementation to vnmFlib using two simple mapping problems. To fairly compare the two implementations, we turn off the “splittable” option (used to map the flow in a virtual link to multiple paths in the physical network), solve the mapping problems for directed graphs only, and set $\epsilon$ to 2 for both implementations. The hardware used is the same as that used in Section 7.3. As seen from Table 7.1, even these improved implementation techniques alone, without employing vnmFlib’s sorting heuristics, already allow VF2x_basic to solve the problems in roughly two orders of magnitude less time than vnmFlib.

We are not the first to observe the performance issues with vnmFlib [GLW+10], but our goal here is to demonstrate that VF2-based algorithms in general can be implemented with good performance for reasonable problem sizes.

<table>
<thead>
<tr>
<th>Table 7.1: VF2x vs. vnmFlib</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>vnmFlib with sorting heuristics</td>
</tr>
<tr>
<td>VF2x w/o algorithm improvements</td>
</tr>
<tr>
<td>Speedup per step</td>
</tr>
</tbody>
</table>

<sup>a</sup> VN: 3 nodes and 3 links; PN: 6 nodes and 9 links.
<sup>b</sup> VN: 11 nodes and 10 links; PN: 14 nodes and 15 links.

Besides careful implementation, VF2x_basic can be further optimized through algorithmic changes which prune or reorganize the search space further and thus lead to a better solution in less time. With these changes, the number of matching steps is reduced. Details of these and other algorithmic changes, and their perfor-
mance impact, are given in the rest of this section. More evaluation of how VF2x handles much larger graphs is discussed in the following sections.

### 7.2.2 VF2x with a candidate queue

To map a virtual node $n$ to the physical network, rather than matching $n$ against every node available from the physical network, VF2 matches only against those nodes within $\epsilon$ hops of the physical nodes already mapped.

VF2x makes a further change to the algorithm to reduce the theoretical complexity. This change is called candidate queue. The intuition of this is to help VF2x to prune the search space still further. We first find all the nodes that $n$’s neighbors have been mapped to. In the example shown in Figure 7.2, $n$’s neighbors $n_1, n_2$ are mapped to $m_1, m_2$. We then calculate the intersection of $E(m_1)$ and $E(m_2)$. Here, $E(m)$ is the set of nodes at most $\epsilon$ hops away from $m$ in the physical network. In fact, the only possible candidates for $n$ are elements of this intersection and we call it $n$’s candidate queue. In Figure 7.2, the candidates are the light gray circles in the shadowed area. The candidate queue improves the solving time by reducing the number of candidates to match: genpair() only generates candidate pairs $(n, m)$ where $m$ belongs to the candidate queue.

![Figure 7.2: The generation of n’s candidate queue](image)

Figure 7.3 shows how candidate queues help to prune the search space with the very same example from Figure 7.1. After nodes $a$ and $b$ are mapped to nodes 2 and 3, $c$ is only going to be tested with node 5 which is the common neighbor of nodes 2 and 3 (here, $\epsilon = 1$). In this way, the branches that are crossed out will not be visited.
7.2.3 Heuristic sorting of network topologies

Like VF2, VF2x is based on a depth-first search strategy. The sequence of the generated candidate pairs depend on how the nodes are ordered in the virtual and physical networks. This suggests that sorting the nodes according to heuristics can reorganize the search tree and find a better solution in less time, without changing the algorithm’s correctness.

The intuition of heuristic sorting is to reorganize the search space to map more-constrained resources first. Different heuristics can be applied. vnmFlib sorts the generated candidate pairs based on the resource consumption of the virtual nodes; VF2x sorts graph nodes such that switches are ordered before hosts (whether a graph node is a switch or a host is indicated by the additional “type” attribute which is introduced for a testbed environment), furthermore high-degree nodes are sorted before low-degree ones.

An illustrative example of heuristic sorting is shown in Figure 7.4. In this example, the nodes are sorted by their degrees, and nodes \([a, b, c]\) in \(G'\) are mapped
to nodes [2, 5, 3, 1, 4] in G. As we can see, this heuristic manages to prune the branches that are crossed out. This simple example illustrates the effectiveness of this heuristic of mapping more-constrained resources first.

7.2.4 Timeout-and-relax for $\epsilon$

The vnmFlib authors propose two algorithms to pick an appropriate $\epsilon$ (recall that virtual links are only mapped to paths shorter than $\epsilon$): VnmFlib-simple uses a fixed $\epsilon$, while vnmFlib-advanced starts with $\epsilon = 1$, exhaustively searches for a mapping, and then successively increments $\epsilon$ until one is found or until $\epsilon = 10$.

We make further observation about the parameter $\epsilon$. For smaller $\epsilon$, fewer candidate nodes need to be considered. Thus, if an easy solution exists, it is likely that it will be found early in the search; however, the tighter constraints can increase the required exploration of the search space to find a solution to a difficult problem. At the same time, for larger $\epsilon$, more candidate nodes need to be considered, thus it needs more time to find a solution for an easy problem, but the relaxed constraints can increase the success rate for difficult ones.

This observation leads us to adopt an additional heuristic to mitigate occasional worst-case performance of VF2x (the general problem remains NP-hard), which we term “timeout-and-relax”. Instead of always running VF2x to completion, we start with small $\epsilon$ and impose a small time limit on the solving time, aborting the algorithm if it exceeds the limit. We then increase $\epsilon$ and the value of timeout, and retry.

The intuition of “timeout-and-relax” is to dynamically tradeoff solving time for solution optimality. This strategy provides a good compromise between shorter solving time of a smaller $\epsilon$, and higher success rate of a larger $\epsilon$. This optimization prevents the algorithm to produce optimal results, however, it is nevertheless worthwhile. As shown in Section 7.3.2, this strategy brings down the average solving time considerably compared to fixed $\epsilon$ and timeout values.

7.2.5 Batching

Another extension VF2x makes for a testbed environment is to support batch solving: mapping several virtual network requests in one solving process even though they share the same physical network resources. The original VF2 algorithm does not support this and can only detect isomorphism for the disconnected subgraphs (VN requests) by mapping the virtual nodes from these requests to different physical nodes. With this extension, different nodes in the same request are mapped to different physical ones, while the same physical nodes can be shared by virtual nodes from different requests. In Figure 7.15, the physical switches are shared by virtual switches from different requests in the same batch.
7.2.6 Algorithm evaluation with synthetic workload

Using the GT-ITM tool, we generated 10 separate physical networks with 100, 150, 200, \ldots, 550 nodes. Each node pair is randomly connected with probability 0.1. In this way, the physical network with 100 nodes is connected by around 500 links. We also generated 200 virtual networks. The size of the virtual networks is uniformly distributed between 2 to 20 and the nodes are connected with probability 0.5. CPU capacity and the link bandwidth follow a uniform distribution ranging from 1 to 100 for the physical networks and ranging from 1 to 50 for the virtual networks. While synthetic, these randomly generated virtual and physical networks are of the same characteristics as those used by other researchers [ZA06, YYRC08, LK09].

To evaluate each algorithmic change introduced above, we run a set of experiments to map 200 virtual networks to each of the 10 physical networks using variants of VF2x. In these experiments, $\epsilon$ is fixed and set to 2 and timeout is set to 5s. The VF2x algorithm variants compared are: VF2x\_basic, VF2x\_queue, VF2x\_queue\_sort and VF2x\_queue\_sort\_dynamic. VF2x\_basic is an extended VF2 which supports semantics constraints and maps virtual links to multi-hop paths. VF2x\_queue extends VF2x\_basic with a candidate queue which further prunes the search space. VF2x\_queue\_sort applies heuristic sorting to VF2x\_queue and sorts the nodes by the degree. Heuristic sorting reorganizes the search space in a way to find a better mapping in less time. VF2x\_queue\_sort\_dynamic extends VF2x\_queue\_sort further with “timeout-and-relax” strategy in which $\epsilon$ is dynamic with different timeouts: ($\epsilon = 2, 0.5s$), ($\epsilon = 3, 1s$), ($\epsilon = 4, 3.5s$).

We ran all the experiments on a machine with an Intel Core2 Q6700 (quad-core 2.66 GHz) CPU and 4GB memory. The machine was running Ubuntu Linux 10.04 LTS Lucid Lynx, and we used version 0.5.4 of the igraph library.

Figure 7.5 compares the four algorithm variants by mapping 200 virtual networks to the same 100-node physical network. As we can see, algorithmic improvements dramatically reduce the solving time and improve the mapping success rate: among 200 requests, within 5s, VF2x\_basic fails to map 64 requests, VF2x\_queue fails to map 56 requests, VF2x\_queue\_sort fails to map 22 requests, while VF2x\_queue\_sort\_dynamic manages to bring the number down to 6. As we can see, within the same time limit, the timeout-and-relax strategy can improve the success rate significantly.

Figure 7.6 plots the solving time of VF2x\_queue\_sort\_dynamic to map 200 randomly generated virtual networks to three different physical networks: 100 nodes with 483 links, 300 nodes with 4513 links, and 500 nodes with 12458 links. As shown in the figure, for more than 65% of the requests, VF2x\_queue\_sort\_dynamic uses less time to map them to the 100-node physical network. However, it takes substantially more time for about 20% of the requests, and even times out for 6 requests out of 200. Physical network size does play an important role in the
Figure 7.5: Solving time CDF for VF2x variants

Figure 7.6: Solving time CDF for various PN sizes
algorithm and we investigate this subject further in Section 7.3.3.

Figure 7.7 plots the solving time of VF2x_queue_sort_dynamic to map 4 virtual networks of 4 nodes/4 links, 8 nodes/12 links, 12 nodes/34 links, and 16 nodes/54 links separately to the 10 different physical networks. As the figure shows, the solving time increases roughly exponentially with the physical network size as well as the virtual network size. However, this is still acceptable for these problem sizes since VF2x_queue_sort_dynamic is able to map the virtual network of 16 nodes/54 links to the physical network of 500 nodes/12458 links within 100ms.

![Figure 7.7: Solving time CDF for various VN sizes](image)

**7.3 Evaluation with realistic testbed workloads**

Since virtual infrastructures are still relatively new, particular those which are distributed in nature and permit complex topological requests, it is not clear what kind of workload is representative for a testbed resource allocator. Much related work uses GT-ITM to simulate requests and testbed network topologies, as we have done to analyze different VF2x algorithm variants in Section 7.2.6.

However, these randomly generated undirected graphs cannot faithfully represent real testbed requests and infrastructures, which was the main motivation for
our realistic benchmarking framework explained in Chapter 6. In this section, we describe the generation of a more plausible test workload from our Emulab trace-based workload generator, and use the generated workload to evaluate VF2x. The hardware used is the same as that used in Section 7.2.6.

7.3.1 Realistic test workload

As for physical topology, we extracted information from ProtoGENI\(^1\) and build our physical topology with 627 nodes (including switches and hosts) and 1163 links based on the resource information from the Utah ProtoGENI site. We simplify the topology by summing up the bandwidth of all links from the same host to the same switch instead of including all “duplicated” links. This physical topology is used for all the experiments shown in this section.

We assume that virtual network requests arrive dynamically. Users submit requests stating desired network resources, when they are needed, and for how long. In response, the testbed resource allocator will decide whether to accept the requests and if so, which specific physical resources will be allocated.

In order to construct a realistic request workload, two important temporal variables must be modeled: the request arrival rate and duration which defines for how long the user actively uses the allocated resources.

Based on the distribution models, we annotate the Emulab request stream with two time parameters – when to request resources and when to release them – and generate our workload from this. The workload is used in Section 7.3.4 to evaluate our testbed resource allocation with VF2x mapping algorithm.

7.3.2 Additional heuristics

Analysis of the test workload

Unlike the random virtual and physical networks generated by GM-ITM, nodes both in the physical topology described above and in our new request stream have an additional “type” attribute indicating whether they are switches or nodes. By analyzing the number of components of the requested virtual network graph, we found that the majority of the requests are connected graphs. Among them, most have tree topology and 20% of the trees are single nodes, as discussed in Chapter 6. Among these tree-topology requests, the structure in Figure 7.8 is common, where the hosts between switches are running DummyNet and acting as delays.

To explore the best heuristics we can apply to the test workload, we use the physical network described in Section 7.3.1, and generate a series of trees of various sizes following the structure pattern shown in Figure 7.8. We ran the VF2x

\(^1\)http://www.protogeni.net/trac/protogeni
mapping algorithm with different $\epsilon$ values (the maximum length of the physical path that a virtual link can be mapped to) and different sorting heuristics.

$\epsilon$ values are 1, 2, 3 and 4. Our baseline for comparison is “doing nothing”: ordering virtual and physical nodes as they are declared in the specification; the second heuristic is the one we describe in Section 7.2: sorting virtual and physical switches and hosts in descending order of their degrees: first switches from high to low degree, then hosts from high to low degree (so that switches are mapped first). By comparing Figures 7.9(b) and 7.9(a), we can see that the heuristic to order/map switches and hosts based on their degrees can reduce the solving time and increase the success rate within a solving time limit of 10s.

These two experiments both show some interesting properties of $\epsilon$. First, the smaller the $\epsilon$ value is, the fewer candidate nodes are to be considered, which results in less solving time. Second, the smaller the $\epsilon$ value is, the more constrained it is to find a satisfiable solution. This can result in longer solving time for the cases where a solution with smaller $\epsilon$ exists but requires more exploration of the search space, or when failing to find a mapping solution due to a timeout. Third, the smaller the $\epsilon$ value is, the more sensitive the solving time is to the order of the nodes. In Figure 7.9(b), a tree of size 5 takes more time to solve than a tree of size 9. The 5-tree orders the nodes as $S0, S1, S2, H0, H1$ while the 9-tree orders them as $S1, S2, S0, H0, H1, H2, ... H5$ according to the sorting heuristic. These observations led us to adopt the “timeout-and-relax” technique: we start with small $\epsilon$ and small timeout, and increase $\epsilon$ and the value of timeout after each timeout. This approach aims at a good compromise between faster average solving time of smaller $\epsilon$, and higher success rate of larger $\epsilon$.
7.3. EVALUATION WITH REALISTIC TESTBED WORKLOADS

(a) “Doing nothing” heuristic sorting

(b) Degree-based heuristic sorting

Figure 7.9: Solving time for trees of various sizes
Applying heuristics to the test workload

Having shown the effectiveness of the sorting heuristic, in the next experiment we use our request stream to investigate the effectiveness of the “timeout-and-relax” strategy. We randomly choose 2000 requests out of 52089 Emulab requests and individually map each of them to the same physical topology introduced in Section 7.3.1. We compare two different strategies: the simple strategy in which $\epsilon$ is fixed and set to 4 with $\text{timeout} = 10s$; the timeout-and-relax strategy in which $\epsilon$ is dynamic with different timeouts: $(\epsilon = 2, 1s), (\epsilon = 3, 2s), (\epsilon = 4, 7s)$. In both experiments, the sorting heuristic described above is applied.

Figure 7.10 and Figure 7.11 plot the solving time against the virtual network size. Here, the virtual network size is defined as the total number of nodes and links. Figure 7.12 depicts the solving time CDF for fixed $\epsilon = 4$ and dynamic $\epsilon = [2, 3, 4]$. From these figures, we can see that: with timeout-and-relax, VF2x solves most (more than 85%) mapping problems in less time, while it also achieves slightly higher success rate (from 94% to 96.3%) and the number of timeout cases decreases from 119 to 74. Figure 7.13 shows this comparison in more detail by plotting the solving time with fixed $\epsilon$ against the solving time with dynamic $\epsilon$. As we can see, most of the mapping problems are solved in less time with dynamic $\epsilon$ and a majority of them are twice as fast. The dots plotted in the center right of the figure are the 45 requests which fail with fixed $\epsilon$ but succeed with dynamic $\epsilon$. The dots plotted in the top right corner are the 74 requests (out of 2000) which fail to be mapped within 10s. The algorithm used in all the next set of experiments is \texttt{VF2x\_queue\_sort\_dynamic} with the candidate queue, the above sorting heuristic and timeout-and-relax strategy applied.

7.3.3 Repeated requests to exhaust physical resources

In this section, we use a simple test workload to investigate the sequential and global allocation strategies. The test workload is a round-robin sequence of four requests from Emulab request history which contain one node, 4 nodes, 7 nodes and 13 nodes separately. For the two allocation strategies, we investigate the solving time to map physical resources to the successive requests, as well as the total proportion of physical resources that are allocated. In the experiment, we deliberately repeat the request in order to investigate, for one specific request, the influence of the shrinking physical resource pool.

With the sequential solving strategy, we allocate resources to each request, mark the resources allocated as unavailable and never reallocate them, and continue until we exhaust all physical resources. During this process, physical resource utilization is increasing and the free physical resources are decreasing. Figure 7.14 plots the relationship between the solving time for one specific request and the
7.3. EVALUATION WITH REALISTIC TESTBED WORKLOADS

Figure 7.10: Solving 2000 Emulab requests with fixed $\epsilon=4$

Figure 7.11: Solving 2000 Emulab requests with dynamic $\epsilon=4$
CHAPTER 7. VF2X VIRTUAL NETWORK MAPPING ALGORITHM

Figure 7.12: Solving time CDF

Figure 7.13: Solving time comparison: fixed vs. dynamic ε
shrinking physical network. It shows how, initially, the solving time decreases since the available resources are abundant and the search space is shrinking. This trend stops in the middle when the platform has much fewer resources available and the solver has to nearly exhaust the (smaller) search space to find a solution.

With the global solving strategy, whenever we process a new request, we re-assign resources to all the requests seen so far in one mapping process, without respecting any previous resource assignment. As we have observed in [YR11], with global solving, the execution time increases exponentially as more constraint programs are solved simultaneously by the ECLiPSe solver we used [AW07]. To our surprise, with the batch solving described in Section 7.2.3, the solving time in VF2x does not increase exponentially.

As shown in Figure 7.15, after a sharp increase at the beginning, the execution time increases linearly. This is due to the fact the requests are repeating themselves. At the beginning, VF2x takes some time (still within 10ms) to find mappings for the first 4 requests. Then, since successive requests are identical, they reuse the switch mappings generated beforehand as long as the switches still have enough capacity left. This significantly prunes the search space. As a conclusion, batch solving is a surprisingly promising technique for solving a set of structurally recurring requests efficiently.
CHAPTER 7. VF2X VIRTUAL NETWORK MAPPING ALGORITHM

7.3.4 Long trace behavior

In this section, we use the full test workload generated in Section 6.3 and the physical network topology described in Section 7.3.1 to evaluate our testbed resource allocator with the VF2x mapping algorithm.

We run the simulations for 200 time windows with different request arrival rates: $\lambda = 4, 8$ which corresponds to 800, 1600 requests in each simulation instance. We use a client simulator to generate and emit a stream of resource requests and release requests (sampled from Emulab request trace) to the allocator. Upon receiving a resource request, the allocator runs VF2x to decide whether to accept the request, if yes, it decides which specific physical resources to allocate and removes them from the available resource pool of the physical network; upon receiving a release request, the allocator will revoke the resources and return them to the physical network. In this process, the physical network is always changing.

Figure 7.16 shows the solving time (including failures and timeouts) of VF2x in mapping the requests of different sizes to the changing physical network in the continuous resource allocation process. As we can see, with a bigger $\lambda$ value, the provider receives more requests in a given time slot, and this results in a higher failure rate because of its limited capacity. Nevertheless, as shown in Figure 7.16,
7.3. EVALUATION WITH REALISTIC TESTBED WORKLOADS

Figure 7.16: Executing the generated trace of arrival rate $\lambda = 4, 8$
VF2x algorithm is fast, efficient and able to allocate resources on a large testbed in seconds.

7.4 Conclusion

We have presented VF2x, a new general virtual network mapping algorithm based on the VF2 subgraph isomorphism detection algorithm. VF2x supports semantic constraints and is able to map virtual links to multi-hop paths. Several algorithmic improvements are introduced: using a candidate queue, applying heuristic sorting and adopting timeout-and-relax strategy.

The design of suitable network mapping algorithms for network testbeds (and, indeed, similar scenarios such as datacenter networks and cloud facilities) is still in its infancy, and the idealized problem is, in theory, computationally hard.

However, we have shown that a combination of careful attention to implementation details, good choice of algorithm, and pragmatism with regard to timeouts can provide a fast and efficient way of embedding a stream of virtual network requests in a physical substrate – when combining implementation, algorithmic optimizations, heuristics like sorting, and strategies like timeout-and-relax, VF2x is over two orders of magnitude faster than previous systems like VnmFlib.

While VF2x is motivated by testbeds research, it is also applicable to larger network infrastructures where the request load is heavier, the requested VNs are bigger and more complicated, and the mapping time is an important factor in system design.

As briefly mentioned in this chapter, VF2x was invented in the process of implementing our own virtual infrastructure resource allocator. We intended to apply VnmFlib directly to our testbed resource allocator, however, we found several limitations of both the algorithm and its implementation. This motivated us to design and implement the VF2x algorithm. The integration of VF2x in our Arosa resource allocator will be discussed in detail in the next chapter.
Resource allocation for virtual infrastructures becomes more challenging as both compute and network resources are involved. The development of such distributed infrastructures raises two closely-related challenges: firstly, how users should specify resources (nodes, links, switches) they wish to reserve in a flexible manner, and secondly, how the infrastructure can best commit or offer resources to users to achieve the highest possible utilization of the infrastructure.

In this chapter, we describe a new approach to negotiate network resources between users and infrastructure providers: users provide a declarative description of their desired resources as constraints, and providers reply with resource reservation promises expressed also as a set of constraints instead of specific resources. This commitments-as-constraints idea gives providers more flexibility in late-binding resources to requests, and opens up a wide design space to optimize resource allocation for efficiency, cost, utilization, or other metrics. The idea has been published in Apsys 2011 [YR11].

Commitments-as-constraints is integrated into the design and implementation of Arosa, a resource allocator for network testbeds. Arosa allows users to reserve resources on nodes, switches, and (virtual) links. The latest version of Arosa integrates as the solver VF2x virtual network mapping implementation, which has been discussed in detail in Chapter 7. The performance of Arosa is evaluated on a workload generated by our Emulab trace-based workload generator introduced in Chapter 6.

With VF2x, Arosa performs network embedding in a shared, distributed testbed in a matter of seconds on commodity hardware. At the same time, Arosa pro-
vides flexibility in allocation policies by describing both resource requests and resource commitments ("tickets") as constraint programs. This allows resources to be bound to requests on a "just-in-time" basis. Late-binding resources to requests enables providers to optimize resource allocation for various metrics, such as utilization, and can help to mask failures of physical resources from users as suggested by the experiments shown in Section 8.5.

8.1 Introduction

Virtual infrastructures (e.g., networking testbeds, cloud hosting services, or grid installations) face an increasing challenge in resource allocation. As inter-dependencies between resources (virtual machines, virtual switches and routers, shares of physical network links) become more constrained, resources become more diverse (specialized switches, programmable middle-boxes, etc.) and infrastructures scale to a larger number of users, network elements, it becomes more difficult to efficiently allocate resources to users from a shared virtual infrastructure.

The basic problem we address in this chapter is how a client of a virtual infrastructure provider specifies resources he/she wishes to use in a flexible manner, how the provider best commits or offers resources to the user to achieve the highest possible utilization of the infrastructure, and how the allocation of such resources is returned to the user.

Designers of virtual infrastructures address the first by moving to a query-based model: users supply a declarative description of resources they desire, and the provider allocates (if possible) specific resources to satisfy the request. The request is a set of constraints on the resources to be allocated, and (optionally) some objective function to allow the resource allocator to select the "best" (for the user) allocation from available options. The resource allocators normally take a relatively flexible, first-come-first-served approach which commits specific resources at request time. However, committing specific resources reduces the scope for resource renegotiation and overbooking, and, as shown later, reduces the effective capacity of the testbed.

The resource allocation process is quite straightforward at a high level, and existing infrastructures employ simple mechanisms. Here, we will take networking testbeds as an example to survey how it is done today, and in doing so make the argument that existing solutions to the problem will not suffice for large-scale, distributed facilities where networking resources are explicitly allocated.

Testbeds like Emulab address the resource allocation problem by being a centralized facility: Emulab exploits high-capacity network switches which approximate a physical crossbar between machines, thereby rendering the problem of embedding users’ virtual network requests in Emulab’s physical infrastructure
tractable (even so, the problem is not trivial). However, recent proposals for distributed network testbeds such as GENI will not be amenable to such solutions. Systems like GENI are based on a complex, widely distributed physical network topology, and aim to support a potentially large number of experiments simultaneously by mapping each requested network topology onto a share or “slice” of physical switches and links. The expected low cross-sectional bandwidth in GENI-like testbeds and the committing specific resources idea lead to inefficient allocation with simple greedy approaches.

A simple example (Figure 8.1) should make this clear. Given a physical network of 2 switches and 8 hosts, a simple strategy might allocate for REQUEST1 4 hosts under one switch, and return the user explicit commitments of these four hosts in TICKET1 as a reservation. However, this fixed reservation leaves REQUEST2 unsatisfiable which leads to poor utilization of the expensive infrastructure. Furthermore, if one of the hosts fails and becomes inaccessible after the reservation, the user will be exposed to such resource failures which happen between reservation and use.

![Diagram](https://via.placeholder.com/150)

(a) Problem

![Diagram](https://via.placeholder.com/150)

(b) Solution

Figure 8.1: Virtual network allocation example

In this chapter, we explore a different approach: not only do users specify their requests as constraints, but providers reply with resource promises which are also expressed as sets of declarative constraints on resources. We conjecture that this
allows providers much greater flexibility in *late-binding* resources to requests, and opens up a wide space of techniques for optimizing physical resource utilization, including statistical overbooking.

In the example above, instead of returning reservations of concrete resources, our approach will express the commitments also as a set of constraints and late-bind resources to requests. As shown in Figure 8.1(b), both requests can be satisfied with the illustrated allocation solution. Moreover, with late-binding, resource failures will be transparent to the affected users.

Actually, as discussed in Chapter 5, similar challenges are faced by Infrastructure-as-a-Service (IaaS) providers and scientific Grid platforms, to a lesser, though increasing, degree. In this chapter, we focus on network research platforms, but we expect that as network topology becomes increasingly important for performance and acts as a market differentiator [LTRK11] in these areas, our ideas will have wider applicability.

The idea of expressing resource requests, as well as resource *reservations*, as a declarative set of constraints, will be introduced in Section 8.2. This changes the resource negotiation process between users and providers, and in turn, opens up a rich design space for providers to optimize resource allocation for efficiency, cost, utilization, revenue, or other metrics.

This commitments-as-constraints idea is integrated into the design and implementation of Arosa as explained in Section 8.3 and Section 8.4. Arosa is a resource allocator for network testbeds; it allows users to reserve resources on nodes, switches, and (virtual) links. Two versions of Arosa are investigated. The first version implements the virtual networking mapping function with ECLPS constraint solver, however, this version has some performance issues. This motivated us to survey existing virtual network mapping algorithms and in the end led to the invention of VF2x as discussed in Chapter 7. VF2x has been integrated into the second version of Arosa.

Section 8.5 evaluates the performance of Arosa with VF2x virtual network embedding algorithm on a test workload generated by our Emulab trace-based workload generator. Arosa performs network embedding in a shared, distributed testbed in a matter of seconds on commodity hardware. Moreover, the commitments-as-constraints idea gives providers more flexibility in late-binding resources to requests. Our experiments suggest that the late-binding of resources enabled by representing resource reservations as constraints achieves higher infrastructure utilization and better masks network failures from users.
8.2 Approach

This section starts with how to express resource requests and commitments as constraints. Then we give a high-level overview of how Arosa, our resource allocator for network testbeds, operates by depicting the resource negotiation process between the user and the provider. Much of this is similar to allocators used in systems like ProtoGENI and ORCA. The main differences are: Arosa supports specifying resource requests as well as reservation commitments as constraints; meanwhile, Arosa also supports the reference model introduced in Chapter 5 in a flexible way. We will point out more differences as they appear.

The design and system architecture of Arosa will be explained in this section, while other key features of the system, such as late-binding and ticket reassignment, will be described in the following sections.

Even though Arosa is a resource allocator designed and implemented for network testbeds, however, the general idea of commitments-as-constraints and the strategy of late-binding enabled by the idea is nevertheless applicable to IaaS cloud providers and scientific Grid platforms.

8.2.1 Resource commitments as constraints

We build the Arosa resource allocator for network testbeds to investigate a different approach to negotiate testbed resources between users and testbed providers: users specify their requests as constraints, and providers reply with resource promises expressed also as a declarative set of constraints on resources.

A resource request or commitment is a list of constraints on:

- Start time and duration: when the requested resources are desired and for how long;
- Resource types: how many compute and network resources are required, what are the desired characteristics of these resources (CPU, memory of the compute units, table entries of the switches, latency or bandwidth of the links, etc.);
- Topology: users can either specify the concrete network topology of how the virtual hosts and switches are connected, or specify the topological properties of the virtual network, such as the maximum fanout, the network diameter, possibly together with some aggregated properties of the resource set, such as the total number of cores.

A very simple example request with topological properties may look like this:

\[
\begin{align*}
\text{Time} & : 8..12, \quad \% \text{request from 8am for 4 hours} \\
n\!(\text{Hosts}) & = 3 \quad \% \text{three hosts}
\end{align*}
\]
sum(Hosts, cpu) > 8 % total CPU units larger than 8
num(Switches) > 1 % more than one switch
di(Topology) = 3 % network diameter is 3

In the following, we will explain in more detail how the user and the provider negotiate testbed resources with constraints.

### 8.2.2 Resource negotiation with constraints

Arosa, our resource allocator for network testbeds, supports specifying resource requests as well as reservation commitments as constraints; meanwhile, it also supports the reference model introduced in Chapter 5 in either a partial or a complete mode. As most existing testbeds, the partial mode does not support advance resource reservation. Whenever a resource request is received and if it is satisfiable, the allocator operating in the partial mode will immediately return a lease for a set of concrete resources. Working in this mode, the allocator cannot benefit from the commitments-as-constraints idea or accordingly late-binding.

In most of the following text, we will focus on Arosa operating in the complete mode. Arosa working in the complete mode supports the two stages described in the reference model: resource reservation and resource allocation. In this mode, the life cycle of an Arosa request follows the timeline depicted in Figure 5.1, and the allocator makes a differentiation between tickets and leases.

In existing systems, a ticket, signed by the provider, serves as a reservation of a set of concrete resources – a promise by the provider to bind the resources to the user and grant access to them at a specific point in the future. Arosa differs in the sense that the ticket is rarely bound to a concrete set of resources, but is essentially another set of constraints, like a user request. In this case, tickets are “unbound”: they describe elements in the virtual network not yet mapped to physical components. Only leases must always refer to specific resources (and, indeed, grant access to them).

Figure 8.2 depicts how users (researchers) and the provider negotiate for compute and network resources in the complete mode. This is similar to the protocols used in other systems, but the significant differences are, firstly, that tickets generally do not specify definite resources (or even definite numbers in some cases), and secondly, that resource negotiation is ongoing, a user can at any time return a ticket (as constraint programs) or a lease (concrete assignment) and exchange it automatically for some other ticket or lease with a new request (also as a constraint program).

In this simple example, the user sends Request0 for 3 hosts. The provider accepts this request and issues Ticket0 for 3 (non-specific) hosts. Later, the user refines its request, returns Ticket0 and requests in exchange 3 hosts with at least 8
CPU units. This request is again satisfied and Ticket1 is returned. In this resource reservation phase, tickets are “unbound”, and the user is not subject to failures of specific nodes and does not have to frequently check the availability of the reserved “unbound” nodes.

Eventually, the user presents the ticket, redeems it and requests a lease for the resources. Only then, the provider will allocate specific resources, bind these resources to the ticket and grant the user a lease. Only after that will the user know the resources he can use. At this point, even though the mapping is bound,
the user must still wait until the lease is activated (at the start time specified in the request and its corresponding ticket) to get the control over the resources.

As shown from the full lifetime of a request in Figure 5.1, even after resource allocation (the ticket is redeemed), the user is still able to negotiate with the provider to exchange leases for different resources by returning the granted lease and presenting a new/refined request. Clearly in this case the testbed has much less freedom in allocating resources from the specific infrastructure. In the example in Figure 8.2, the user adds an additional constraint on network diameter. The testbed can interpret this in many ways, but one might expect it to minimize the overhead of lease reallocation by keeping as much of the current allocation as possible unchanged (two hosts under one switch), releasing the last host under this switch, and allocating another switch with one host connected to it. In practice, though, lease reallocation depends on the underlying infrastructure and has to make the tradeoff between resource migration cost, quality of service and platform utilization [ABF +11]. Resource reallocation and lease exchange are not fully explored in the current implementation of Arosa.

Finally, the user may surrender the ticket or lease at any time and terminate the negotiation process. The revocation which happens before the lease is activated has no influence in terms of the usage of the resources.

### 8.2.3 System architecture

The software architecture of Arosa is shown in Figure 8.3. The principal components are as follows:

- The **Ticket Manager** responds to resource requests and decides whether to issue a ticket and, if so, what it will contain. It is also responsible for removing tickets which have expired or been surrendered.
- The **Lease Manager** responds to lease requests and decides what to grant in the lease. It is also responsible for removing the lease if it expires or a user surrenders it.
- The **Resource Knowledge Base** keeps a list of all the valid tickets and leases it has issued, and maintains an up-to-date list of all physical resources available, and their current condition. Typically it also keeps some kind of objective function the provider seeks to maximize – most likely utilization in the case of a network testbed, but in commercial settings, this will probably be some function of yield or revenue.
- The **Component Manager** interacts with the testbed infrastructure itself, activates the leases at the specified start time and prepares the resources for users’ usage (for example, to start requested VMs or create a virtual switch).
The novelty of Arosa lies principally in two aspects: firstly, expressing the resource requests and reservation commitments as constraints and accordingly being able to late-bind resources to requests; secondly, using VF2x for resource assignment/reassignment to map virtual network requests to the shared physical infrastructure. These two aspects are mainly implemented in the Ticket Manager and the Lease Manager.

When the provider receives a request for a ticket, it should only return a ticket which describes resources it knows it can allocate in the future as constraints and does not have to be bound to any specific resources. More precisely, the provider should be able to prove the existence of an assignment of physical resources to users which optimizes its object function subject to the following constraints:

- No concrete assignments already made in an issued lease may change until the lease expires.
- The assignments corresponding both to the ticket about to be issued, and to the complete set of tickets already issued, must satisfy the constraints expressed in those tickets.
- The assignment must respect the physical topology and resource availability in the testbed during the time interval covering all the tickets and leases.

Only if the request is to redeem a ticket for a lease does the provider need to return a concrete resource assignment to the user. This operation may, of course,
fail: changes in the testbed (such as failures) after the ticket was issued may result in the provider being unable to satisfy the request.

This late-binding leaves the provider considerable freedom in deciding whether a feasible allocation exists. One option is to try to produce, from scratch, a complete resource assignment across the entire relevant timescale for all the unbound tickets. This straightforward procedure can give optimal use of the infrastructure (without resorting to revoking leases), and therefore extracts maximum benefit from the fact that tickets do not imply definite assignments. However, while delivering optimal utilization of the testbed, this assignment includes embedding many virtual networks into a physical network which is known to be NP-hard, and is thus unlikely to be computable in a reasonable time as shown in Section 8.3.1.

A second option is to perform a purely incremental allocation for each new request. This mirrors the strategy used by systems like ProtoGENI and ORCA today: the provider retains the previous concrete assignments and assigns resources for a new request only from previous unassigned ones. This is cheap to compute, but can lead to sub-optimal allocation of resources.

Between these extremes there is a tradeoff: the more existing reservations our system can reconsider, the greater the complexity of the allocation but the higher the potential efficiency of the result. Arosa adopts a middle ground to solve this NP-hard problem, as we will describe below, it maintains a “mostly concrete” resource assignment internally, but issues tickets which are generally non-specific about which resources may have been allocated – in fact, they are essentially the users’ requests and convey no information other than “yes”. Key to this approach is the ability to rapidly re-calculate an assignment of resources, a problem dominated by the time required to embed all the virtual networks being considered simultaneously into the (unallocated) physical resources available.

This approach allows Arosa to transparently do a limited reassignment of resources if a request for a ticket cannot be trivially satisfied, or some other event causes a change in the system state.

8.2.4 Discussion

The idea of commitments-as-constraints opens up a wide design space to optimize resource allocation for efficiency, cost, utilization, or other metrics. In this chapter, we focus mostly on the late-binding of resources enabled by representing resource reservation as constraints.

As explained in the example in Section 8.1 and illustrated by experiments in Figure 8.12(a), the late-binding of resources achieves higher resource utilization. Furthermore, in the case of network elements with varying capabilities, late-binding avoids committing a machine early to a task which does not fully exercise it.
The late-binding of resources accommodates changes in physical resource availability. Failures can be masked if resources can be reallocated before they are required. For example, Arosa can often mask failures of physical nodes and still honor issued tickets by performing ticket reassignment, as shown in Figure 8.13.

Moreover, the late-binding of resources afforded by unbound tickets enables a fourth option to decide whether a feasible resource allocation exists – other than the two extremes and the middle-ground Arosa takes. This fourth option, which we do not investigate further, is to employ sophisticated yield-management techniques, including the use of overbooking and variable price models, to gamble: issue a ticket if there is a high probability it will be satisfiable when the time comes. The probability can be estimated in a variety of ways, for instance, taking into account the chance of no-shows, estimating the chance that resources will be released early or that tickets will be surrendered, or reducing the guarantees delivered by a ticket in response to anticipated overload. Yield management is the process of understanding, anticipating, and reacting to consumer behavior in order to maximize revenue. It was first explored in the airline industry, particularly by American Airlines [SLD92]. To improve profit, airlines overbook flights or offer discounts when it appears as if seats will otherwise be vacant. Resource overbooking was first considered in the context of shared hosting platforms (i.e., clustered environments) in [USR02]. This research work showed that revenue benefits from controlled overbooking of resources can be dramatic.

Late-binding opens up a wide space of allocation strategies. Arosa, which makes a tradeoff between the two extremes, is just one of them. More other solutions are also possible: some may be appropriate for academic testbeds which follow PlanetLab’s community model, while others may employ sophisticated yield-management techniques, including the use of overbooking and variable price models, to maximize commercial revenue.

Issuing tickets as constraints offers other potential advantages for virtual infrastructure resource allocators, which we haven’t explored yet. One example is that resources can be specified at different levels of generality. Very general requests, such as “4 machines for 2 hours in the morning” are both easy to express and fit naturally into the provider’s framework. As another example, multi-stage renegotiation is possible with constraint-based tickets. For the moment, it is up to the user to refine the resource request by adding or modifying constraints and the provider just reply with a “yes” or “no”. With tickets as constraints, it is also possible for the provider to start the negotiation with the user by replying “4 machines for 2 hours in the morning are not available, though 4 machines for 2 hours in the afternoon are satisfiable with some discount.”
8.3 Arosa with ECL\(^e\)PS\(^e\) solver

Arosa resource allocator is implemented in Python. Our first implementation uses the ECL\(^e\)PS\(^e\) constraint solver (which is written in C) for network embedding. In this implementation, status information about the available resources from the physical infrastructure, such as the hosts, the switches, the connectivity between them, are stored in the form of Prolog facts – expressions with constant values that can easily be queried by means of the term and filed name. The resource requests shown above are simply a set of constraints on the variables to be determined, namely, Hosts and Switches.

This implementation is briefly evaluated with synthetic workloads to establish the feasibility of our new approach as well as to investigate the performance of our quick-and-dirty prototype.

In the experiments, Arosa is running on a machine with an Intel Core2 Q6700 (quad-core 2.66 GHz) CPU and 4GB memory. The machine was running Ubuntu Linux 10.04 LTS Lucid Lynx.

8.3.1 Initial results

We used Mininet [LHM10] to generate a physical tree network (depth 3, fanout 6) with 216 hosts and 43 switches, and randomly annotated the nodes with different capabilities. Our test workload is a round-robin sequence of 4 pre-defined requests, Req\(_1\) and Req\(_2\) are simple requests for networks of 2 and 5 nodes respectively, while Req\(_3\) and Req\(_4\) are more complex requests for larger networks (7 and 11 nodes) with specific topology requirements. For different allocation strategies, we are interested both in the time to perform successive allocations, and the total proportion of physical resources that can be allocated.

In our first experiment, we allocate resources to each request when it arrives, and never reallocate resources. As Figure 8.4 shows, after the first 34 requests are satisfied (70% utilization), only small requests can be met. Note also that solving time (whether successful or not) decreases as available resources are reduced.

In the second experiment, we try full ticket reallocation: as each request arrives, we globally allocate all resources to the complete set of requests so far with no a priori allocations. This problem is NP-complete, and therefore as Figure 8.5 shows, execution time increases exponentially and after several hours, our solver fails to allocate even half of the available network.

Finally, we pick a design point between these two: allocate sequentially as in the first experiment case, but when allocation fails retry by remapping, in one go, the current request and all existing requests. These requests are ordered by their complexity, and solved independently as we did in the first experiment. Intuitively, this represents a compromise between the exponential solving time of
8.3. AROSA WITH ECL$^4$PS$^E$ SOLVER

![Sequential solving graph](image)

Figure 8.4: Sequential solving

![Global solving graph](image)

Figure 8.5: Global solving
reconsidering all a prior requests altogether, and the severely constrained approach of fixing previous allocations. Remapping the more complex requests first might be expected to result in more freedom to find space for new requests.

Figure 8.6: Constraint remapping

Figure 8.6 shows the results. Versus the simple sequential approach, runtime is much higher (though not prohibitive, even with our unoptimized solver). We satisfy more requests (we now fail first at request 40), resulting in greater utilization (16.7% remaining free). However, to remap resources, our first implementation needs to run for more than 10 minutes.

8.3.2 Conclusion

As a quick-and-dirty prototype, this first implementation with ECL\textsuperscript{1}PS\textsuperscript{e} solver helped us to get some initial results aimed at establishing the feasibility of our approach. In this preliminary evaluation, we also explored a bit of the space for optimization that is enabled by late-binding resource requests. We investigated a very simple remapping strategy: allocate resources to the requests sequentially, but when allocation fails retry by remapping, in one go, the current request and all existing tickets.

Our simple first experiments suggest that the late-binding of resources enabled by representing resource tickets as constraints opens up a significant space for
optimization of resource usage by platform providers. It is also very obvious from
the results that there is much scope for improving the run time of our solver for
embedding resources as well as for remapping resources.

The half-satisfying results from the first implementation lead us to explore an
enhanced algorithm for network embedding. This effort actually results in VF2x
virtual network embedding algorithm and implementation which has been dis-
cussed in Chapter 7. Arosa’s implementation is upgraded accordingly to integrate
the VF2x virtual network embedding implementation as the solver.

8.4 Arosa with VF2x

In this section, we will only focus on how the VF2x algorithm is used in Arosa to
assign resources to requests.

In Arosa, when the provider receives a request for a ticket, the provider will
privately try to compute an assignment of physical resources which meets all the
constraints and requirements listed in the request. If the provider successfully
solves this mapping problem, it will return the user a ticket – for the moment,
ticket contains the same set of constraints as the request.

For the provider, the key problem is how to generate such an assignment given
the topology of the physical resources, their availability, the set of active leases,
the set of already-issued tickets, and their privately stored assignments. In the
following text, we will first only consider “bound” ticket and later generalize the
problem from bound tickets to “unbound” ones which can be remapped.

8.4.1 Resource assignment with bound tickets

To simplify the problem, in this section, we only consider “bound” tickets. Sup-
pose, the provider receives a request $R_5$ at time $T_7$, shown in Figure 8.7. At this
time, in the system, there are two active leases $R_1$ and $R_3$, and three issued tickets
$R_0$, $R_2$ and $R_4$. To compute an assignment for $R_5$, we need to consider resource
consumption in the duration between $T_10$ and $T_13$. From the figure, we can see
that $R_5$ has one conflicting lease $R_3$, two conflicting tickets $R_0$ and $R_2$.

Since in this section, the tickets are “bound”, it means all the resources al-
dicated to the conflicting leases and all the resources reserved for the conflicting
tickets are not available for the current request to use. One simple solution is to
get a copy of the physical network topology, minus the resources occupied by the
leases and tickets, and map the virtual network request to the remaining physical
topology using VF2x.

This solution works, however, it is not very efficient. To see why, suppose the
physical network has a host with four VMs, $R_0$ gets 2 reserved as well as $R_2$, and
the solution mentioned above will consider this host as unavailable for \( R_5 \) since no VMs are left. However, if we take a closer look at \( R_0 \) and \( R_2 \), these two tickets have non-overlapping durations, which means these two tickets will use at most 2 VMs from this host and \( R_5 \) can still use up to 2 VMs.

Our solution is as follows: first, we take a copy of physical network topology, deduct the resources allocated/reserved to the leases/tickets, here \( R_2 \) and \( R_3 \), whose redeem time is before \( T_{10} \); then start from \( T_{10} \) and simulate the system running and responding to all the redeem and release events between \( T_{10} \) and \( T_{13} \). This includes releasing the resources reserved for \( R_2 \) at \( T_{11} \), allocating resources for \( R_0 \) at \( T_{12} \), and stopping at \( T_{13} \). During this process, we record the minimal resources left at each time point, and finally use the minimal remaining physical topology to run VF2x to find a resource mapping for \( R_5 \).

When the provider finds a resource assignment for \( R_5 \), this assignment is then privately stored in the provider’s resource knowledge base before the ticket is returned.

### 8.4.2 Resource assignment with unbound tickets

We can now generalize this problem from “bound” tickets to the more complex issue of computing a resource assignment for a request given that all the tickets...
are “unbound” and can potentially be remapped.

To illustrate this, consider a new scenario where $R_5$ requests a specialized host (for example, one with an FPGA-based accelerator), and the only such node not already allocated to the leases has been (privately to the provider) reserved by $R_0$, even though $R_0$ does not require the extra functionality. At this point, we can take advantage of “unbound” tickets and find a reassignment for $R_0$ to satisfy $R_5$.

The key challenge is to decide which tickets to be considered for remapping, and how this remapping is to be done. The tickets to be considered for remapping can be one conflicting ticket ($R_0$), several conflicting tickets ($R_0$ and $R_2$), or even include tickets whose duration is not overlapping with the request ($R_0$ and $R_4$). The reason for the last case is that, $R_0$ and $R_4$ are overlapping.

Remapping can be conducted in different ways. One option is to take a copy of the physical topology, deduct the resources allocated by the leases and the resources reserved for the not-to-be-remapped tickets, and run VF2x to solve $R_0$ and $R_5$ collectively in one round. A second approach is to use heuristics to reorder the request and the to-be-remapped tickets, and fix their assignment one after another sequentially.

In Section 8.5, we choose the conflicting ticket who reserves the most resources, and run group solving for this ticket and the received request.

8.5 Evaluation

The initial results lead us to search for or invent a more efficient solver for embedding as well as remapping resources. VF2x virtual network mapping algorithm, which has been discussed thoroughly in Chapter 7, is the outcome of this effort. In the following text, we will focus on the Arosa resource allocator which applies VF2x virtual network mapping algorithm as the solver.

The detailed evaluation of Arosa’s latest implementation with VF2x is done with a more plausible workload generated from the Emulab trace-based workload generator introduced in Chapter 6.

The generated workload is first used in a simple resource allocation model: the resources are allocated at the moment the request is submitted if the resource requirements can be satisfied. This simple allocation model is applied to investigate Arosa resource allocator’s dynamic behavior under continuous requests and varying degrees of request load. The generated workload is also used in a more general model in which resource reservation is supported. We use this general model to evaluate the performance of Arosa with VF2x as the solver, and to investigate the idea of commitments-as-constraints and late-binding resources to requests. We will show how late-binding can result in superior testbed utilization and how Arosa can partially mask the effects of physical node failures from users.
In the following experiments, the hardware used is the same as that described in Section 8.3. We used version 0.5.4 of the igraph library to evaluate arosa with the VF2x network embedding solver.

### 8.5.1 Operating in the partial mode

In this section, we investigate the dynamic behavior of Arosa resource allocator under continuous requests and varying degrees of request load. In the experiment, Arosa is operated under a simple resource allocation model (the partial mode): the allocator does not support reservation and assigns resources to the user for duration time immediately after the resource request is received.

We use the Emulab trace-based workload generator to generate a stream of resource requests and release requests, and a client simulator to emit the generated requests to the resource allocator who assumes the partial mode and only returns “bound” leases: upon receiving a resource request, the allocator runs VF2x to decide whether to accept the request, if yes, it decides which specific physical resources to allocate and removes them from the available resource pool of the physical network; upon receiving a release request, the allocator will revoke the resources and return them to the physical network. In this process, the physical network as described in Section 7.3.1 is always changing.

To evaluate Arosa resource allocator under such an allocation model, we set duration distribution (Gamma) parameters as follows: \(shape = 0.3\) and \(scale = 20\), and vary \(\lambda\) from 4 to 8 and 16 to simulate increasing offered load. These distributions are used to annotate the Emulab request stream with two parameters: when to request resources and when to release them. We run simulations for 200 time windows with different request arrival rates to investigate arosa’s dynamic behavior under continuous requests with respect to resource utilization.

In Figure 8.8, the dotted line depicts the utilization of the physical network (dividing the number of allocated hosts by the total number of physical hosts) under a request arrival rate of \(\lambda = 4\). The solid line shows the utilization in an “ideal” scenario where all the requests are accepted and satisfied. Of 800 requests, the allocator refuses 69 due to resource insufficiency and fails to map 7 requests to the physical network within 10s. For these timeout cases, we can relax their resource constraints and retry the mapping with VF2x.

Figure 8.9 shows the physical network utilization under different request loads. After some warm-up time, the system enters a more steady state as leases are requested and released “evenly”. In the end, when no more requests are received, the allocator gradually releases all the resources. By comparing lines for different \(\lambda\), we can see that the higher the load, the more likely it is that the allocator can fit small requests into the network and achieve higher utilization.
Figure 8.8: Executing the generated trace of arrival rate $\lambda = 4$  

Figure 8.9: Executing the generated trace of arrival rate $\lambda = 4, 8, 16$
8.5.2 Operating in the complete mode

In this section, we will investigate mainly the performance of Arosa with VF2x as the solver. In the experiment, Arosa resource allocator operates under the full resource allocation reference model (in the complete mode). The time-line of the resource allocation process shows several important time points related to the requests: request submission time, ticket redeem time, lease activation time (the same as the start time specified in the ticket) and lease release time (specified as duration in the ticket, however, early release is also possible).

We equipped Emulab requests with the above parameters in our workload generator: request arrival rate, lead time and duration. We vary the request arrival rate from 1 to 4, 16 and 32 to increase the offered load. The parameters for lead time Gamma distribution are set to shape = 0.8 and scale = 20, which results in 1000 generated lead times being distributed as follows: 105 fall in less than a hour, 149 in 1 to 4 hours, 168 in 4 to 8 hours, 362 in 8 to 24 hours, 216 in 1 to 7 days. The duration Gamma distribution is modelled with shape = 0.3 and scale = 20. With this model, out of 1000 generated durations, 266 fall in less than 10 minutes, 117 in 10 to 30 minutes, 90 in 0.5 to 1 hour, 459 in 1 to 24 hours, 68 in 1 to 7 days.

We use client simulator to send requests to the provider to request for tickets, to redeem tickets for leases and to release leases. Upon receiving the requests, the provider who holds a physical network infrastructure as described in Section 7.3.1 takes actions accordingly to solve the resource requests, allocate resources for the leases and revoke lease resources (tickets are bound in this experiment). Figure 8.10 and Figure 8.11 shows the solving time of VF2x in mapping the requests of different network sizes to the changing physical network, as well as the failures and timeouts in this continuous resource allocation process. As we can see, with a bigger λ value, the provider receives more requests in a given time slot, which leads to a higher reservation failure rate because of its limited capacity, as well explained in Figure 8.10(b) and Figure 8.11(b). Nevertheless, as shown in the figures VF2x algorithm is fast, efficient and able to allocate resources on a large testbed in seconds.

Other than the solving time for the requests, utilization of the platform is another very important aspect we would like to investigate. Figure 8.12(a) depicts the utilization of the system under the trace using λ = 16. After about 20 minutes warm-up period of allocating resources to redeem requests, the provider keeps its utilization above 60%. The platform’s utilization keeps stable when leases are redeemed and released “evenly”. In the end, when no more requests are received, the provider gradually releases all the resources.
8.5. EVALUATION

Figure 8.10: Executing the trace with $\lambda = 1.4$
Figure 8.11: Executing the trace with $\lambda = 16, 32$
8.5. EVALUATION

8.5.3 Benefits of late-binding

In this section, we investigate the benefits of late-binding “unbound” tickets from two different perspectives:

Provider: we investigate the difference in provider’s resource utilization with bound and unbound ticket under the same trace of requests.

User: we investigate how much the user can be masked from network failures by introducing random node failures into the system at the point when redeem requests are received.

Higher resource utilization

The base line of this experiment is shown in Figure 8.10 and Figure 8.11 with $\lambda = 16$. In this experiment, the client simulator uses the exact same trace, while the provider treats the tickets as “unbound”. When a ticket request can not be satisfied respecting all the privately stored ticket assignments, the provider takes a simple remapping strategy: among all the unredeemed tickets whose duration is overlapping with the request, picks the one consumes the most resources and solves this ticket together with the current request in one pass using VF2x batch matching.

Figure 8.12(a) shows the comparison of the resource utilization between the baseline and this simple remapping strategy, and in total 12 failed requests can be remapped. Figure 8.12(b) zooms into the period between 70 and 85 minutes. As shown in the figures, even this simple remapping strategy achieves higher resource utilization most of the time with an improvement up to 10%, and achieves over 80% system utilization.

Masking network failures

Large-scale distributed testbeds are prone to frequent failures. In [PBFM06], the most common reasons for nodes being down include hardware failures and network reconfiguration. Figure 8.13 demonstrates the ability of masking those kinds of network failures using remapping. In this experiment, the client simulator uses the trace of $\lambda = 16$ model to send requests to the provider. On the provider side, at the time when a redeem request is received, a random set of the hosts (5%, 20% and 40%) fails. The provider then tries to assign non-failure resources to all the active leases and the to-be-redeemed ticket (who have been affected by the failures) through remapping. If it succeeds we call it a redeem success. At the next redeem point, forget the failures introduced before and repeat the above procedure.

Figure 8.13 shows a trace with around 1360 redeem requests. When the failure rate is 5%, only 6 requests can be redeemed using the privately stored assignments,
Figure 8.12: Utility comparison
8.6 Conclusion and discussion

We have proposed a new approach to negotiate network resources between users and infrastructure providers: users provide a declarative description of their desired resources as constraints, and providers reply with resource reservation promises expressed also as a set of constraints instead of specific resources.

We also presented Arosa, a fast resource allocator for network testbeds. Arosa allows users to express the resource requests and reservation commitments as constraints and accordingly is able to late-bind resources to requests. Furthermore,
Arosa uses VF2x for resource assignment to map virtual network requests to the shared physical infrastructure.

The design of suitable resource allocators for network testbeds (and, indeed, similar scenarios such as datacenter networks and cloud facilities) is still in its infancy, and the idealized problem is, in theory, computationally intractable.

However, we have shown that VF2x is a fast, efficient algorithm and is able to embed virtual network requests in a large testbed in seconds. The efficiency of VF2x virtual network mapping algorithm in turn enables flexibility. Being able to quickly solve resource requests allows Arosa to late-bind resource allocation. We have shown that this increases testbed utilization, and can mask physical failures.

We have evaluated only a simple remapping strategy, and we conjecture that there are also better heuristics for resource remapping than the simple one we evaluate here: the field of Operations Research has a wealth of results on this kind of problem.

At present in Arosa, as opposed to tickets, we do not consider remapping leases: once a lease is granted, we assume the resources are taken until the lease expires or it is surrendered. Actually, lease remapping is also very important, for example, for achieving a better placement of the service [OCP+06], or for re-optimizing the utilization of the substrate network [YYRC08]. In practice, though, lease remapping has to make the tradeoff between the resource migration cost and the quality of service/platform utilization [ABF+11]. Without any information or estimates on the actual load to be introduced by requests, online adjustments (migration, for example) may be needed.

Finally, the late-binding of resources combined with fast solving of network mapping serve as a foundation for more sophisticated techniques like overbooking and online resource renegotiation, which is worth further investigation.
This dissertation has contributed towards applying declarative techniques to resource management over virtual infrastructures on the client and the provider side. Though closely related, the client and the provider have different design goals to manage the resources over virtual infrastructures. The client focuses on satisfying its own resource requirements and maximizing its utility in the face of external condition changes as well as its own resource requirement changes; while the provider focuses on satisfying as many clients’ resource requirements as possible and maximizing the utilization or revenue of the whole infrastructure.

9.1 Application resource management

The first part of the dissertation explores an alternative approach to the problem of application resource management over virtual infrastructures. The novelty of the proposed approach lies in two key features:

- Fate-sharing between application and resource management: the resource management logic is bundled into the application itself, so that it is more closely integrated with the rest of the application logic.
- Declarative resource management: constraint logic programming is used as a programming interface to the application runtime to make the specification and implementation of such tightly-coupled resource and application policies tractable.
Two systems are designed and implemented following this approach: the Rhizoma decentralized resource management runtime and the Anzere personal data storage and replication system. Both systems are deployed over real virtual infrastructures: Rhizoma uses PlanetLab as a challenging “proving ground” for cloud-based services while Anzere is deployed in a heterogeneous personal cloud which consists of an ensemble of personal devices: personal computers, mobile phones, tablets, cloud virtual machines, etc. The evaluations of the two systems both verify the following advantages of the above approach:

- **Expressiveness:** constraints can naturally express an application’s resource requirements – where it should and should not run, the global and local properties of the resource set the application needs, as well as application-specific requirements such as fault tolerance, data availability, privacy policies in Anzere.
- **Adaptation responsiveness:** by coupling the management functionality with the core application logic, the system can react more quickly to changes in available resources, pricing policies, or application load and status by determining where and how an application should be deployed, where and how personal data should be stored and replicated, etc.
- **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 from different providers or considering the overhead of system reconfiguration/migration).

### 9.2 Virtual infrastructure resource allocation

The basic problem addressed in the second part of the dissertation is how a client of an infrastructure provider (e.g., a networking testbed, cloud hosting service, or grid installation) requests resources, how the provider allocates such resources, and how the allocation of such resources is returned to the client.

- **Realistic benchmarking framework:** a general methodology is proposed for generating realistic benchmarks to evaluate system performance and to compare different design ideas. Following this methodology, an Emulab trace-based workload generator is implemented and the generated test workloads are used later to compare different versions of the VF2x virtual network mapping algorithm, and to evaluate the late-binding resource allocation strategy.
- **The VF2x virtual network mapping algorithm:** VF2x is based on the VF2 subgraph isomorphism detection algorithm. Several novel algorithmic improvements and careful implementation make VF2x perform more than two
orders of magnitude faster than the previously-published vnmFlib (also based on VF2). The evaluation of VF2x by an extensive test workload shows that VF2x can allocate resources to virtual networks on a large testbed in a matter of seconds using commodity hardware.

- Commitments-as-constraints: this idea opens up a wide design space to optimize resource allocation for efficiency, cost, utilization, or other metrics. Among all the advantages, we investigated mostly providers’ flexibility in late-binding resources to requests.

- Arosa resource allocator for network testbeds: Arosa allows users to express resource requests and reservation commitments as constraints and accordingly is able to late-bind resources to the requests. Furthermore, Arosa uses VF2x to map virtual network requests to the shared physical infrastructure. The evaluation shows that Arosa performs network embedding in a shared, distributed testbed in a matter of seconds. The experiments also suggest that the late-binding of resources enabled by representing resource reservation as constraints achieves better network resource utilization compared to the fixed assignment solution, and better masks network failures from clients with resource reservation.

### 9.3 Directions for future work

We discuss several interesting directions for future work.

**Self-optimizing resource management runtime**

For the moment, the utility function as well as the cost function are given and fixed during the self-managing application’s life cycle. Inspired by the idea of profile-based optimization in the compiler design, we would turn Rhizoma from a self-managing system to a self-optimizing one by auto-tuning the utility and cost function based on performance measurements and feeding application-level metrics back into the optimization process. The utility function now only describes how much one specific deployment configuration satisfies the application’s resource requirements. A more accurate utility function would preferably be able to describe the performance of the application given a specific resource offering configuration. To support the deployment across different cloud computing providers, the cost function needs to be evaluated across different pricing models.

**Resource renegotiation and overbooking with constraints**

Our work of virtual infrastructure resource allocation focuses mostly on the late-binding of resources enabled by representing resource reservation as constraints and investigates its advantages of optimizing resource utilization and
CHAPTER 9. CONCLUSIONS

masking physical resource failures. Actually, the late-binding of resources afforded by Arosa’s unbound tickets also enables gambling (the fourth option) to decide whether a feasible allocation exists. This is the domain of yield management and operations research, but we conjecture that the employment of sophisticated yield-management techniques including the use of overbooking and variable price models, will be beneficial to improve the utilization or the revenue of the underlying virtual infrastructure. Overbooking improves revenue in airline industry [SLD92], and brings dramatic revenue benefit to shared hosting platforms [USR02].

Virtual infrastructure federation

In the second part of the thesis, we assumed two stakeholders: infrastructure providers and resource clients, interacting with each other. According to the allocation reference model with these two roles, the task of federating resources from different virtual infrastructures is left to the resource client. The client needs to explicitly request resources from a specific infrastructure provider, and takes care of resource management and application deployment across different infrastructures itself. Actually, Rhizoma provides the perfect framework for such self-managing and self-deploying application requirements. As demand for virtual infrastructure federation grows, however, we expect the emergence of brokering intermediaries, which coordinate and federate infrastructure providers and offer their resources to a set of clients. It is worth investigating the potential benefits of expressing both resource requests and commitments as constraints in such a system setting.

Data center resource allocation

The Arosa resource allocator targets at systems like GENI which are based on a complex, widely distributed physical network topology, and aim to support a potentially large number of experiments simultaneously by mapping each requested network topology onto a share or “slice” of physical switches and links. The VF2x virtual network mapping algorithm also takes account of the expected low cross-sectional bandwidth in GENI-like testbeds. It is an interesting topic to investigate how Arosa resource allocator can be applied in a data center environment to handle both constrained virtual machine placement and real-time task scheduling problems at the same time.

Constraint solver optimization

ECLiPS constraint programming system is used in our Rhizoma and Anzere resource management system because of its extensive library support and ease of use. However, ECLiPS is an interpreted, high-level language with higher execution time overhead compared to imperative languages as shown from the comparison between Arosa with ECLiPS solver and Arosa with VF2x in Chapter 8.
An ECL\textsuperscript{i}PS\textsuperscript{e} CLP program works by propagating constraints and then probing values rather than assigning values in a straight-forward iterative way, which may lead to longer execution time. More recent constraint solvers have better performance [Kot09]. A more modern Satisfiability Modulo Theories (SMT) solver like Z3 [DMB08] could express most of the same constructs we use in ECL\textsuperscript{i}PS\textsuperscript{e} but would almost certainly significantly improve execution time. Another promising direction is to leverage declarative optimization platforms such as Cologne [LRL+12] to declaratively specify and incrementally execute constraint optimization problems in distributed systems.
CHAPTER 9. CONCLUSIONS
Bibliography


ProtoGENI. \url{http://www.protogeni.net/trac/protogeni}, October 2012.


[YCBR08] Qin Yin, Justin Cappos, Andrew Baumann, and Timothy Roscoe. Dependable self-hosting distributed systems using constraints. In
Proceedings of the Fourth conference on Hot topics in system dependability, HotDep’08, San Diego, California, 2008.


