Technical Report

UCAM-CL-TR-615
ISSN 1476-2986

Number 615

UNIVERSITY OF CAMBRIDGE
Computer Laboratory

Global public computing

Evangelos Kotsovinos

January 2005

15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom
phone +44 1223 763500

http://www.cl.cam.ac.uk/

This technical report is based on a dissertation submitted November 2004 by the author for the degree of Doctor of Philosophy to the University of Cambridge, Trinity Hall.

Technical reports published by the University of Cambridge Computer Laboratory are freely available via the Internet:

http://www.cl.cam.ac.uk/TechReports/

ISSN 1476-2986

Summary

High-bandwidth networking and cheap computing hardware are leading to a world in which the resources of one machine are available to groups of users beyond their immediate owner. This trend is visible in many different settings. *Distributed computing*, where applications are divided into parts that run on different machines for load distribution, geographical dispersion, or robustness, has recently found new fertile ground. *Grid computing* promises to provide a common framework for scheduling scientific computation and managing the associated large data sets. Proposals for *utility computing* envision a world in which businesses rent computing bandwidth in server farms on-demand instead of purchasing and maintaining servers themselves.

All such architectures target particular user and application groups or deployment scenarios, where simplifying assumptions can be made. They expect centralised ownership of resources, cooperative users, and applications that are well-behaved and compliant to a specific API or middleware. Members of the public who are not involved in Grid communities or wish to deploy *out-of-the-box* distributed services, such as game servers, have no means to acquire resources on large numbers of machines around the world to launch their tasks.

This dissertation proposes a new distributed computing paradigm, termed *global public computing*, which allows any user to run any code anywhere. Such platforms *price* computing resources, and ultimately *charge* users for resources consumed. This dissertation presents the design and implementation of the *XenoServer Open Platform*, putting this vision into practice. The *efficiency* and *scalability* of the developed mechanisms are demonstrated by experimental evaluation; the prototype platform allows the global-scale deployment of complex services in less than *45 seconds*, and could scale to *millions* of concurrent sessions without presenting performance bottlenecks.

To facilitate global public computing, this work addresses several research challenges. It introduces *reusable mechanisms* for representing, advertising, and supporting the discovery of resources. To allow flexible and federated control of resource allocation by all stakeholders involved, it proposes a novel *role-based resource management* framework for expressing and combining distributed management policies. Furthermore, it implements effective *service deployment* models for launching distributed services on large numbers of machines around the world easily, quickly, and efficiently. To keep track of resource consumption and pass charges on to consumers, it devises an *accounting* and *charging* infrastructure.

Acknowledgements

Although a doctoral dissertation is always the result of largely solitary work, I have been truly fortunate to have cooperated with many outstanding people, to whom I express my gratitude here.

My supervisor, Tim Harris, fully deserves the first spot in this list; I am indebted to him for being a reliable source of ideas, advice, and direction throughout my work towards this dissertation. I am also grateful to Steven Hand and Ian Pratt, who both encouraged my good ideas and questioned my not so good ones. Their support has been invaluable, and their deep technical knowledge, creativity, and enthusiasm about systems research has been a constant source of inspiration.

I would like to thank Jean Bacon, Jon Crowcroft, Steven Hand, Maja Vukovic, and Ian Wakeman for providing plenty of very useful feedback that helped me plan and shape this dissertation. For supporting me, surviving long conversations on my ideas, and proofreading parts of this dissertation, thanks are due to my friends and colleagues Alberto Fernandes, Alex Ho, Boris Dragovic, Christian Kreibich, Christina Tsouparopoulou, Dimitrios Selemetas, Eva Kalyvianaki, Giorgos Portokalidis, Katerina Biliouri, Maja Vukovic, Rajiv Chakravorty, and Tim Moreton.

I am grateful to Keir Fraser and the Xen project team, Russ Ross, and David Spence for making the Xen Virtual Machine Monitor, the CoW NFS server, and XenoSearch available respectively. I would like to thank Andy Chung for staying late — in fact, very, very late — in the Lab when required, and Priyanka Sinha and Tom Wilkie for their practical contributions. I also take this opportunity to thank Evangelos Markatos and Katerina Gialama, who stimulated my interest in distributed systems research during my time at ICS-FORTH, setting me on course to start a PhD.

I wish to thank the Marconi Corporation, the Cambridge European Trust, and the Neil Wiseman Memorial Fund of the Computer Laboratory, University of Cambridge, for providing financial support for my studies.

Last but not least, my gratitude for the motivation and support I have received from my parents, Nikos and Catherine, and my sister, Panagiota, is beyond words.

Contents

List of Figures

List of Tables

Glossary

AFS	Andrew File System
ALAN	Application-Level Active Networking
ALC	Advertisement Locations Catalogue
ANTS	Active Node Transfer System
API	Application Programming Interface
ARPANET	Advanced Research Projects Agency Network
CDN	Content Distribution Network
CORBA	Common Object Request Broker Architecture
CPU	Central Processing Unit
CoW	Copy-on-Write
DDoS	Distributed Denial of Service
DHT	Distributed Hash Table
DNS	Domain Name System
DRMAA	Distributed Resource Management Application API
DRMS	Distributed Resource Management System
F/S	File System
FTP	File Transfer Protocol
GARA	General-purpose Architecture for Reservations and Allocation
GESA	Grid Economic Services Architecture
GGF	Global Grid Forum
GRAAP	Grid Resource Allocation Agreement Protocol
GRAM	Globus Resource Allocation Manager
GRIP	Grid Resource Information Protocol
GUI	Graphical User Interface
HTTP	HyperText Transfer Protocol
I/O	Input/Output
ID	Identifier
IP	Internet Protocol
IPSEC	Internet Protocol Security

ISP	Internet Service Provider
IT	Information Technology
JDBC	Java Database Connectivity
JVM	Java Virtual Machine
JXTA	Project Juxtapose
LDAP	Lightweight Directory Access Protocol
MD5	Message Digest 5
MDS	Metacomputing Directory Service
MMOG	Massively Multiplayer Online Game
MVM	Management Virtual Machine
NFS	Network File System
NIC	Network Interface Controller
NIST	National Institute of Standards and Technology
OGSA	Open Grid Services Architecture
OGSI	Open Grid Services Infrastructure
ORB	Object Request Broker
OS	Operating System
P4	Pentium 4
PC	Personal Computer
PCC	Proof-Carrying Code
PDA	Personal Digital Assistant
PDP	Programmed Data Processor
PLAN	Programming Language for Active Networks
PLC	PlanetLab Central
PVM	Parallel Virtual Machine
QoS	Quality of Service
R/O	Read-Only
RAM	Random Access Memory
RBAC	Role-Based Access Control
RBRM	Role-Based Resource Management
RMI	Remote Method Invocation
RPC	Remote Procedure Call
RPM	Revolutions Per Minute
RSL	Resource Specification Language
RUS	Resource Usage Service
SCSI	Small Computer System Interface
SETI	Search for ExtraTerrestrial Intelligence
SFS	Self-Certifying File System
SOAP	Simple Object Access Protocol

SQL	Structured Query Language
SSH	Secure Shell
SSL	Secure Socket Layer
SWORD	Scalable Wide-Area Overlay-based Resource Discovery
TCP	Transmission Control Protocol
TLS	Transport Layer Security
UDP	User Datagram Protocol
UR	Usage Record
URL	Uniform Resource Locator
VCI	VM Control Interface
VM	Virtual Machine
VMM	Virtual Machine Monitor
VPN	Virtual Private Network
WS	Web Service
WSDL	Web Services Description Language
XIS	XenoServer Information Service
XML	Extensible Markup Language

Terminology

Control plane: the part of the system that deals with operations required to deploy and manage services.

Coordination: the process leading to achieving harmonious functioning of parts of a distributed system for effective results.

Execution environment: an environment that encompasses a set of resources and can accommodate the execution of tasks – for example, a Unix process, a Virtual Machine, or a JVM environment. Multiple execution environments may coexist on a server.

Federated systems: systems that comprise parts owned and administered by different organisations.

Global public computing: the model of distributed computing where globally dispersed, mutually untrusted, competing members of the public purchase computing resources on servers around the world for the execution of untrusted services.

GuestOS: an instance of an operating system ported to run over a Virtual Machine Monitor.

Purchase order: an item that represents the commitment of a sponsor to funding sessions he or she deploys on servers.

Resource advertisement: the process of publicising information about available resources on a server.

Resource description: the process of representing computing resources in a well-defined, structured manner.

Resource description coordination: the process of making sure representation of common resources is consistent between different servers.

Resource discovery: the process of locating resources suitable for running a distributed service.

Resource management: the process of controlling how resources are allocated to different users or user groups.

(Distributed) Service: a software system that runs on one or more servers to carry out an operation on behalf of users, and consists of a number of distributed components, called tasks.

Server: a machine that undertakes the execution of tasks.

Service deployment: obtaining adequate computing resources on one or more servers to run a service, and launching the tasks it comprises on the servers.

Session: an agreement between the resource provider and the user; the provider agrees to provide the requested resources and the user promises to pay for resource consumption.

Sponsor: an entity that funds the execution of a distributed service or task.

Stakeholder: anyone with an interest in what an entity does.

Task: a software component, which may be independent or part of a distributed service.

Virtual Machine: a type of execution environment. In the prototype implementation, a currently running guestOS – the difference between a guestOS and a Virtual Machine being analogous to that between a program and a process.

XenoServer: a server that undertakes the safe execution of untrusted tasks in exchange for monetary rewards.

Chapter 1

Introduction

Over the last few years, distributed computing has evolved from a promising area of research to a valuable solution addressing real and challenging problems. The user community has widely embraced large-scale distributed systems such as *peer-to-peer* file sharing networks, which have become extremely popular by allowing millions of globally dispersed users to exchange files simply and easily. *Scientific applications* split large computational problems into smaller sub-problems, and distribute these over numerous machines. On-line *multi-player games* rely on networks of servers around the world, which serve their nearby clients.

However, the development of distributed services has been disproportional to that of *infrastructural support* for their deployment. How can researchers, or even other members of the public acquire resources on large numbers of globally dispersed machines over short timescales to deploy their large-scale distributed services?

This dissertation proposes a new distributed computing paradigm, termed *global public computing*, where members of the public can acquire resources and deploy distributed services on networks of machines scattered around the world in exchange for money. It describes the design, prototype implementation, and evaluation of a global public computing infrastructure that addresses the needs of this user community, and investigates reusable *resource management mechanisms* for coordinating and managing global public computing infrastructures. Furthermore, it presents a *global-scale service deployment* facility, provided to allow the launching of complex distributed services easily and efficiently.

1.1 Motivating examples

Several applications that make use of computing resources on large numbers of machines have been developed for commercial, scientific, and entertainment purposes, usually solving *embarrassingly parallel*[1] problems.

In 1995 Disney and Pixar used a server farm of 117 uniprocessor and multiprocessor SPARCstation workstations, comprising a total of 294 processors, to render Toy Story [Rob95], the world's first ever full-length, entirely computer-generated animated movie. Rendering the 114,000 frames of the 77-minute movie required unprecedented amounts of raw computing power; one single-processor computer of that type alone would have needed 43 years of non-stop operation to render the movie.

SETI@home [WCL+01] analyses data from the world's largest radio telescope, located in the Arebico Observatory in Puerto Rico, in the hope of detecting signals generated by alien civilisations. Folding@home [LSP03] studies protein folding in an effort to understand the cause of many serious diseases, such as Alzheimer's, Bovine Spongiform Encephalopathy (BSE), and Parkinson's. Both initiatives require enormous amounts of computing resources that cannot be provided by any mainframe or server farm alone. @home systems pioneered the transition to *global-scale distributed computing* by exploiting idle CPU cycles on hundreds of thousands of mainly ordinary desktop machines. SETI@home is harnessing a total of up to 60 Tflops/sec; it would have taken a single computer more than two million years to analyse the data that SETI@home has analysed since 1999 to the time of writing[2]. Even the $350-million Earth Simulator Center [Sat04], home to the world's most powerful supercomputer at the time this dissertation was written, achieves less than two thirds of SETI@home's computing power.

Ultima Online [Ele97] is one of the most popular massively multi-player computer games[3]. Played by thousands of paying users simultaneously, it runs a

[1] An embarrassingly parallel problem is a computing problem that can easily, or very obviously, be split up into parts that can be computed in parallel. In these problems each step can often be computed independently from every other step, thus each step could be made to run on a separate processor to achieve quicker results (definition from `http://www.wikipedia.org`).

[2] SETI@home current total statistics, from `http://setiathome.ssl.berkeley.edu/`.

[3] A massively multiplayer online game (MMOG) is a type of computer game that enables hundreds or thousands of players to simultaneously interact in a game world they are connected to via the Internet. Typically this kind of game is played in an online, multiplayer-only persistent world. Non-MMOGs usually have less than 50 players online and are usually played on private servers (definition from `http://www.wikipedia.org`).

network of game servers placed at key network locations around the world to distribute load and maintain low latencies between those and the players. Unlike @home, Ultima cannot rely on spare CPU cycles; guaranteed resource availability is necessary, and users are much less likely to provide such guarantees for free, especially towards a goal that is not closely related to the "common good". Therefore, Ultima controls and maintains a network of proprietary, dedicated game servers.

Existing technology falls short of facilitating general mechanisms for setting up and maintaining such networks of servers easily, flexibly, and at an affordable cost, for on-line gaming or any other type of global-scale distributed services. Leasing dedicated servers is tedious and expensive, requires manual configuration, and is based on monthly contracts. Offering a new massively multi-player on-line game is estimated to cost the developer eight to eleven million US$[4], a significant part of which is for funding the server infrastructure [Cox00]. Moreover, game servers cannot be dynamically relocated to reflect fluctuation in player numbers in the different geographical regions, for instance as a result of the different time zones.

How can a new gaming company dynamically obtain computing resources on a number of machines at particular network locations around the world? How can resource acquisition be flexible enough to allow services to be moved around the world to adapt to changes in demand? How can new, untrusted, and experimental distributed services — such as novel network protocols or a next-generation Internet — be tried out in a realistic setting?

This dissertation gives answers to the aforementioned questions, by introducing a new model of distributed computing and demonstrating how this can be put into practice.

1.2 A new computing paradigm

Looking back at the history of computing, a number of different computing trends can be identified, with respect to which user classes had access to computing resources, where those resources were located, and what applications could be deployed on the machines — as shown in Figure 1.1. In the early days, only a

[4]Estimate by John Smedley, operations director at Sony Online Entertainment, according to [Bla01].

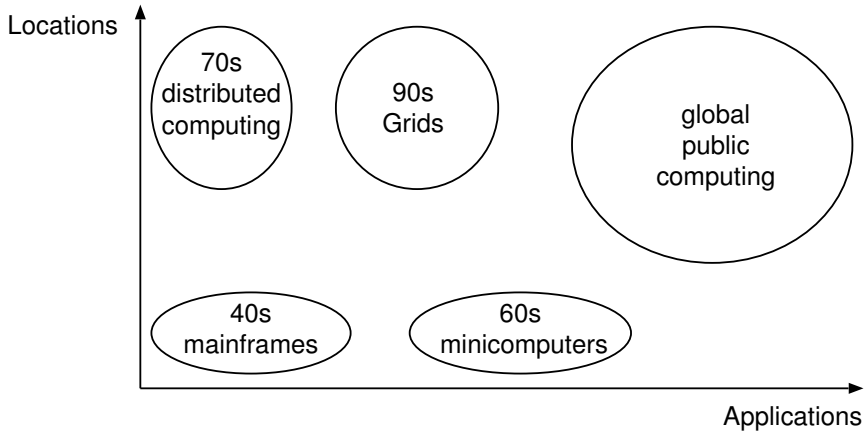

Figure 1.1: Comparison of computing models with respect to the number of locations at which a service can be running and the range of applications that can be deployed

restricted group of privileged users could access mainframes, which could only run machine-specific applications. Later, smaller machines, able to run a wider range of applications, became a commodity. Distributed computing allowed specific programs to run at more than one location, and subsequently Grid computing extended that to provide support for distributed execution of any trusted Grid-enabled application.

A few users, machine-specific code, on the mainframe. The first *mainframe* computing systems, built from the 1940s to the 1960s, were enormous room-sized machines. Mainframes were owned by large institutions, such as universities or companies, and were running proprietary operating systems. Specially written programs were developed offline, examined carefully — as errors would lead to waste of expensive resources — and given to trained staff that would run these on the machines on behalf of the users and return the results.

Time-sharing [CDD62, CV65, Bul80] allowed several users to run jobs concurrently on one processor, or in parallel on many processors, usually providing each user with his or her own terminal for input and output. Grosch's law[5] encouraged the purchase of large supercomputers to be time-shared among a group of users [Den64], since that was believed to provide more performance per dollar than buying smaller computers for each one of them [Fan65, DS68].

[5]Observation made by Herb Grosch in 1965, stating that computing performance increases as the square of its cost.

Any user, any code, in the lab. The drastic drop in the cost of electronics, as a result of the invention of integrated circuits, made Grosch's law seem irrelevant at the time, and led to the development of *minicomputers*[6] in the 1960s and *microcomputers*[7] in the 1970s. For moderately demanding applications, the dominant paradigm shifted from paying subscription fees for central time-shared computing services to buying a smaller time-sharing computer that provided in-house computing. Machines and users were usually located within the same group in an organisation.

The development of operating systems, compilers, and high-level programming languages, such as Unix and C, which run on several different kinds of machines, allowed the execution of code that was not specifically written for a particular machine. Furthermore, the development of Virtual Machine [ABCC66, PPTH72] technology allowed the safe partitioning of a single physical machine to several virtual parts, and the secure concurrent execution of potentially insecure code.

Any user, specific applications, anywhere. The introduction of computer networks at around the same time as the minicomputers led to the foundation of *distributed computing* [Fly66, Ens78, NH82, Lam86]. As minicomputers were far cheaper than mainframes while less computationally powerful, scientists observed that they did not necessarily need mainframes even for large computational jobs. Ironically, the consensus at the time became the opposite of what Grosch's law had previously suggested; linking several minicomputers together could provide the same computational power as that of a large mainframe and yet cost significantly less. Distributed computing projects split large problems into smaller parts and solved each part on a different computer, then combined the results.

[6]Minicomputers were multi-user computers which made up the middle range of the computing spectrum, in between the largest multi-user systems (mainframe computers) and the smallest single-user systems (microcomputers or personal computers). They usually took up one or a few cabinets, compared with mainframes that would usually fill a room. One of the most successful minicomputers was Digital Equipment Corporation's 12-bit PDP-8, launched in 1964 (from `http://www.wikipedia.org`).

[7]Microcomputers are — usually single-user — computers with a microprocessor as their CPU. They occupy physically small amounts of space, not more than can be put onto most tables or desks. The first generation of microcomputers was launched in the mid-1970s, the MITS Altair being one of its most well-known examples, and was followed by "home computers", such as the BBC Micro, the Commodore 64, and the IBM Personal Computer (IBM PC) (from `http://www.wikipedia.org`).

Two of the first wide-area distributed computing applications were *Creeper* and *Reaper*. Creeper made its way through the nodes of the ARPANET [MW77] in the 1970s, using their idle CPU cycles to copy itself onto the next node. Reaper came next and travelled through the same network, deleting all remaining copies of Creeper. Creeper and Reaper were essentially worms, as they replicated themselves to networked machines without their owners' permission; at the same time though, these were applications that considered the possibility of using distributed computational power. Similar worms were created in the next few years, moving from machine to machine and using idle cycles for utile purposes, such as rendering graphics [SH82].

The concept of distributed computing evolved over the years from sharing resources between a group of machines owned by a single institution — as in the Toy Story case — to a more *federated* collaborative model, where resources may be owned by different organisations and reside in different physical locations — as in the @home projects and peer-to-peer file sharing applications.

The emergence of object-based distribution middleware, such as CORBA and Java RMI, made the development of distributed applications faster and easier. Abstractions offered by XML [BPS98], SOAP [Rym01], and Web Services [GGKS02] enhanced the interoperability of distributed components, allowing autonomous services to be loosely coupled in order to achieve the performance of complex operations.

Cooperative users, specific applications, anywhere. All aforementioned systems support the development and execution of single, specific, *ad hoc* distributed applications. The evolution of distributed deployment platforms and *Grid computing* took this model one step further by providing infrastructures for deploying user-defined applications on globally dispersed machines, and allowing for more generic resource sharing. Resource providers and users are cooperative, as the former give out their resources for free and the latter generally behave well and do not misuse resources.

Programs to be deployed on Grids need to be *Grid-aware*; their source code needs to be modified to comply with the API required by the Grid on which they are to be deployed. As a result, they often need to be written in a specific programming language supported by that API. Moreover, servers participating in Grid infrastructures only support applications that are executable in specific versions of particular operating systems; thus, programs to be deployed on Grids

often need to be compiled to be executable on one of those specific operating system versions. Additionally, applications to be deployed on Grids generally need to be *trusted*; malicious, potentially harmful, or experimental code can harm other applications, or even the infrastructure itself.

Any user, any code, anywhere. The next generation of distributed computing paradigms is *global public computing*. In one phrase, "anyone can run any code anywhere"; servers scattered across the globe make resources available to all members of the public, not just cooperative scientists, and to all applications, not only well-behaved scientific experiments. They do so in exchange for money; users are ultimately charged for the resources their applications use on the servers.

At the same time, global public computing allows users to choose not only the resources that their services need, but also the *location* of those resources. Allowing users to deploy software at key points in the network can help reduce delays, remove network bottlenecks, and minimise long-haul traffic charges.

Apart from offering a flexible solution for deploying currently existing distributed services, enabling global public computing will give birth to *next-generation distributed services*. When the infrastructure is in place to allow global-scale code deployment at a low cost of entry, new services and business opportunities will emerge, similar to the way the evolution of computer networks and the Internet generated an extensive range of on-line opportunities.

1.3 The XenoServer vision

This dissertation proposes the *XenoServer Open Platform* for global public computing, to address the needs of general-purpose distributed service deployment. The name derives from the Greek word "$\xi\varepsilon\nu o\varsigma$" (xenos), which means foreign or unknown, much like the tasks that XenoServers accept and safely execute. The aim of the platform is to provide a substrate for the deployment of global-scale services, by allowing the dynamic and flexible acquisition of globally dispersed computing resources.

To build a general-purpose public computing infrastructure that allows any user to run any code anywhere, a number of important research challenges need to be addressed at the same time:

Any user. The users that provide and use resources of the platform cannot be assumed to be well-behaved, cooperative scientists. As resource owners can no longer be expected to provide resources for free for the sake of science, explicit monetary *payments* are necessary; resource consumers must be billed for the resources their tasks consume.

It is anticipated that in most cases resource owners and consumers will not know or trust each other before a transaction takes place. There is a need for coordination mechanisms that will enable *authentication* and *secure charging and billing* in such an inherently untrusted environment.

As participation is open to everyone, resource owners may wish to control the amount of resources to be allocated to different user groups or users with particular properties. At the same time, other stakeholders — such as infrastructural authorities or network administrators — need ways to influence how resources are apportioned on servers under their jurisdiction, either in their own interest, or on behalf of users under their control. Mechanisms for *resource management* based on *federated policies* are required.

In an open platform, users may use resources to perform illegal activities. It is crucial that *logging* and *auditing* information about user activities be kept. Determining the point of balance between *anonymity* and *security* is a complicated matter, not to be statically fixed at the design stage. To the greatest extent possible, it should be left open and adjustable, ideally on a per user or per server rather than platform-wide basis.

Finally, global-scale service deployment using previously available techniques involves a significant cost in terms of effort; configuring the machines, replicating the necessary code elements, installing, starting up, and managing the service throughout its lifetime needs to be done individually and highly manually. An infrastructure to provide service deployment at a *low cost of entry*, in terms of money and effort, is needed.

Any code. No assumptions can be made on the set of applications that may be executed on a global public computing platform. Any application, written in any language and running *out-of-the-box*, without requiring compliance to a platform-specific API or middleware, has to be supported.

As servers are charging for resource provision, they need to make sure that resources reserved for an application are always available to it, no matter what

else is running on the machine. The code to be executed on the servers can be potentially untrusted, unsafe, buggy, experimental, or even malicious. Providing strong *resource isolation* and *protection* between mutually untrusted services running on the same machine is crucial.

Anywhere. One of the visions which initially led scientists to the idea of distributed computing was that of constructing "distributed supercomputers" [SB78, Fre89, FC90, SC92]. Building or buying extremely powerful central supercomputers was — and still is — often prohibitively expensive, while organising large numbers of machines to unite their processing capabilities to form distributed machines provides tremendous processing power at a more reasonable cost.

However, as the price of raw computing resources has been constantly falling, the need for such "processing monsters" has been on the decline [RTBS01]. At the same time, the increasing availability of ubiquitous network connectivity allows for the decentralisation of intelligence — services migrate close to where they are needed [PS01].

This is not to dispute the usefulness of distributed supercomputers; there are still cases where extreme computational resources are needed, such as the @home projects described earlier. It becomes increasingly apparent though that a significant part of the potential and appeal of general-purpose, global public computing lies in satisfying different, more complicated user needs than just that for raw CPU cycles.

The *location* of computing resources is becoming increasingly significant. Users can experience *communication latencies* of more than two seconds when connecting to servers on other continents, while having to pay substantial amounts for long-haul network traffic [Rog98]. Service providers, such as popular web sites or game servers, spend significant amounts to maintain dedicated mirror sites in several parts of the world, in order to balance load, minimise latencies, and restrict international or transcontinental traffic [Lin03]. Services provided by transient mobile devices need access to a flexible infrastructure of reliably connected servers, in order to be replicated there for maintaining permanent network presence [RRPK01].

Enabling users to *discover servers* that have adequate available resources and are at convenient network locations is important. Mechanisms are needed to allow servers to *describe* and *publicise* their resource availability, and users to locate servers using both *location-based* and *resource-based* criteria.

It is necessary that the proposed platform supports the convenient and efficient *deployment* of *large-scale* or *global-scale services*; research groups around the world have been developing experimental systems such as new network protocols [Dun99], high-performance peer-to-peer applications [TXKN03], distributed file systems [KS91], distributed operating systems [PPD+95], and mobile agent systems [HCK95], which need to be tried out in a realistic large-scale setting. Commercial global-scale services, such as massively distributed multi-player games, are already thriving [Son99, Ele97].

As services can be deployed on large numbers of servers, *efficient global-scale deployment* mechanisms are required that do not incur prohibitive volumes of network traffic, or waiting times.

1.4 Dissertation outline

The respective contributions of the chapters included in this dissertation are the following. This chapter identifies motivating examples and introduces a new paradigm for distributed computing, termed *global public computing*, where any member of the public is allowed to run any code anywhere on the platform, and gets charged for the resources consumed. It introduces the vision for implementing a distributed platform that provides this functionality to users, and presents a framework of general research challenges to be tackled.

Chapter 2 sets the scene of distributed computing and deployment platforms. The aim is twofold; first, to analyse the *research context* in which global public computing is introduced, and point out the necessity and importance of the new computing model proposed by identifying shortcomings of conventional deployment platforms. Second, to set specific *requirements* for global public computing, and base the rest of the dissertation on devising a system that meets them.

Chapter 3 describes the design and implementation of the *XenoServer Open Platform*, which substantiates a practical global public computing infrastructure; it focuses on the high-level design of the platform, describing the components it consists of, their functionality, and the interactions between them.

Chapter 4 introduces a new approach to resource management for public computing systems. In contrast to most distributed deployment platforms, which assume or enforce centralised control of resources, the presented framework explicitly supports federated control. It proposes a *role-based resource management*

scheme to provide convenient mechanisms for defining federated policies; this allows resource owners and other stakeholders to define how resources are to be apportioned between different users and user groups, and supports combining potentially overlapping policies flexibly and easily. Moreover, it demonstrates that the proposed resource management solution is reusable in other global public computing settings and distributed deployment environments.

Chapter 5 analyses the *implementation* of the XenoServer Open Platform, providing details of the internal architecture of each of the platform components and showing how they implement the desired functionality. It discusses *service deployment configurations*, and proposes mechanisms for launching complex services on large numbers of XenoServers around the world conveniently and efficiently. Furthermore, it describes the prototype tools developed for the interaction of users with the system.

Chapter 6 *evaluates* the mechanisms employed in the XenoServer Open Platform. It demonstrates that the implemented solutions perform and scale more than adequately well, and analyses in detail how each of the research challenges identified in Chapter 2 is addressed.

The final chapter presents the *conclusions* reached and suggests areas with potential for *future work*. It outlines research challenges that may be generated by a large-scale deployment of the XenoServer platform, discusses new types of distributed services whose emergence the platform may enable, outlines trust management and security issues, and proposes investigating dynamic pricing as a means of maximising profit and regulating resource congestion.

1.5 Publication record

Subsets of my work towards this dissertation or related to it have been published in refereed international conferences and workshops as follows:

- *Global-Scale Service Deployment in the XenoServer Platform* (with Tim Moreton, Ian Pratt, Russ Ross, Keir Fraser, Steven Hand, and Tim Harris). In Proceedings of the First Workshop on Real, Large Distributed Systems (WORLDS '04), December 2004, San Francisco, CA.
- *Pinocchio: Incentives for Honest Participation in Distributed Trust Management* (with Alberto Fernandes, Sven Östring and Boris Dragovic). In

Proceedings of the 2nd International Conference on Trust Management (iTrust 2004), March 2004, Oxford, UK. Also published in Springer-Verlag Lecture Notes in Computer Science (LNCS), Volume 2995, pp. 63-77, ISBN: 3-540-21312-0.

- *Role-Based Resource Management* (with Tim Harris). In Proceedings of the 8th CaberNet Radicals Workshop, Ajaccio, Corsica, France, October 2003.

- *The XenoServer Open Platform: Deploying Global-Scale Services for Fun and Profit* (with David Spence). Poster, in Proceedings of ACM SIGCOMM 2003, August 2003, Karlsruhe, Germany.

- *XenoTrust: Event-Based Distributed Trust Management* (with Boris Dragovic, Steven Hand and Peter Pietzuch). In Proceedings of the 2nd IEEE International Workshop on Trust and Privacy in Digital Business (DEXA-TrustBus 2003), September 2003, Prague, Czech Republic.

- *Managing Trust and Reputation in the XenoServer Open Platform* (with Boris Dragovic, Steven Hand, Tim Harris, and Andrew Twigg). In Proceedings of the 1st International Conference on Trust Management (iTrust 2003), May 2003, Heraklion, Crete, Greece. Also published in Springer-Verlag Lecture Notes in Computer Science (LNCS), Volume 2692, pp. 59-74, ISSN: 0302-9743.

- *Controlling the XenoServer Open Platform* (with Steven Hand, Tim Harris, and Ian Pratt). In Proceedings of the 6th International Conference on Open Architectures and Network Programming (IEEE OPENARCH 2003), April 2003, San Francisco, California, US.

- *Distributed Resource Discovery and Management in the XenoServers Platform* (with Tim Harris). In Proceedings of the 7th CaberNet Radicals Workshop, Bertinoro, Italy, October 2002.

I have also contributed to the following Computer Laboratory Technical Report:

- *Xen 2002* (with Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Anil Madhavapeddy, Rolf Neugebauer, Ian Pratt, Andrew Warfield). University of Cambridge Computer Laboratory Technical Report 553, January 2003.

The following documents are under review at the time of writing:

- *Resource Management in Global Public Computing* (with Tim Harris). Submitted to an international journal.
- *Replic8: Location-Aware Data Replication for Ubiquitous Environments* (with Douglas McIlwraith). Submitted to a refereed international conference.

The Xen Virtual Machine Monitor, XenoSearch, XenoTrust, and Pinocchio, are results of collaborative work and are not parts of research carried out in the scope of this dissertation.

Chapter 2

Research context

The concept of distributed computing, where a program is divided into smaller parts that run on different machines, is not new, and has been attracting increasing research focus in the last few years. This chapter describes related work under four main categories. *Distribution middleware* approaches provide convenience substrates for object-based distributed computing. *Large-scale distributed applications*, such as peer-to-peer systems, are examples of successful computing at the global scale. *Active networks* propose an infrastructure that allows the incremental deployment of new protocols on network elements. *Deployment platforms* allow users to run code on remote machines owned by users that can potentially belong to different administrative domains.

As the number of existing systems in each category is prohibitively large, producing an exhaustive list of those and describing the operation of each one would be out of the scope of this dissertation. I choose to focus on a selection of systems that are representative of the different kinds of systems commonly used, and offer useful indicative examples of typical functionality provided by such systems.

2.1 Distribution middleware

Several middleware architectures, aiming to simplify the development of interoperable object-based distributed applications, have been developed. Here I discuss the relevance to global public computing of three of the most popular of them; CORBA, Java RMI, and Web Services.

CORBA [Obj91] provides middleware to allow architecture-independent development of object-based distributed applications transparent to the programmer. It allows communication between nodes in heterogeneous environments at the object level with the help of Object Request Brokers (ORBs). ORBs discover and instantiate objects on remote machines, marshal and unmarshal object parameters, and handle security, object retrieval, and method invocations.

CORBA also provides a significant number of support services. The *collection* service allows the manipulation of several objects as a group. The *concurrency* service mediates simultaneous accesses to an object such that consistency is not compromised. The *event* and *notification* services provide a substrate for easier asynchronous interaction between objects. The *naming* service handles associations between names and objects, including name binding and resolution. The *object trading* service facilitates the offering and discovery of instances of services of particular types.

RMI [WRW96, Sun99] enables the creation of distributed object-based applications in Java. Similarly to CORBA, it uses serialisation techniques to marshal and unmarshal object parameters. Unlike CORBA, which provides APIs for most programming languages, RMI requires that code be written in the Java programming language.

Using RMI, entire objects can be passed and returned as parameters in remote method invocations — unlike CORBA, where parameters need to be primitive data types, references, or structures composed of the two. This is an important property; *any* new Java code can be sent across the network and *dynamically executed* at run-time by foreign JVMs. This provides significant flexibility benefits for designing distributed services, as developers do not need to define a fixed codebase at development time — although they need to ensure that the necessary class definitions are available. This feature makes RMI more relevant to general-purpose distributed computing than CORBA.

Web Service technology [GGKS02] is a subsequent development in methodologies for constructing distributed, component-based applications. The Web Service Description Language (WSDL) [CCMW01] supports the syntactical description of interfaces in terms of messages, operations, and protocols supported. Just like CORBA and RMI, it is built on the idea of separation of a component's interface from its internal mechanism, thus allowing for transparency in interoperation.

Middleware-based approaches provide interoperability benefits in cases where the distribution of computation takes place locally. It is reasonable to expect that software components used inside a single enterprise may be written in the same programming language or comply with a particular API. The same assumption does not hold in the context of global public computing; using a middleware-based approach as a general-purpose public computing platform would mandate that applications be rewritten before being deployed on the platform. This would raise the *cost of entry* to prohibitive heights; it is necessary that the platform is able to accommodate any existing code without requiring modifications or recompilation.

Distribution middleware targets a different problem from the one that global public computing addresses. It provides mechanisms for building distributed applications transparently by allowing communication between different machines to take place at the object level. Global public computing provides support for the dynamic acquisition of computing resources on globally dispersed machines, on which CORBA, RMI, or Web Service-based applications may themselves be deployed — as discussed in Section 2.4.6.

2.2 Large-scale distributed applications

Ad-hoc distributed applications that run on large numbers of machines around the world, either to divide the computational load of demanding applications or to allow sharing files and other resources, have become increasingly popular in the last few years.

Section 2.2.1 discusses *peer-to-peer* applications, and Section 2.2.2 focuses on *scientific computing* systems. While the goals and environment of these systems are significantly different from the ones of global public computing, similarities or shared goals and challenges are examined wherever possible.

2.2.1 Peer-to-peer systems

The area of peer-to-peer systems has received significant research focus in the last few years; at the same time, several such applications have become popular within the user community [ITI04], as they are *free*, can be *anonymous* and present a *low cost of entry* to the users.

Peer-to-peer systems consist of a number of nodes that share *resources*, most commonly *files*. Their main characteristic is that peers obtain resources from other peers by *direct communication*, without the involvement of a central server, allowing them to scale to large numbers of users.

The application that has linked its name with the skyrocketing of peer-to-peer file sharing applications' popularity was *Napster*[1], which was launched in 1999. Technically, Napster was not a pure peer-to-peer system, as a central server was used for file discovery. Peers sent information about their file availability to the server, which exported search interfaces to peers. File download operations were then performed directly between peers. This centralised architecture, while simplifying design, generated both technical and legal problems; the Napster file discovery server became a bottleneck and single point of failure, and at the same time a point of vulnerability for legal action, as it stored global information about available files [Cau00].

Gnutella[2] was designed for full decentralisation. All peers in the Gnutella network perform exactly the same operations. Searching for files is completely decentralised and carried out by limited flooding — broadcasting search queries inside a section of the network. While Gnutella does not suffer from single points of failure and is less prone to litigation, discovering files is relatively inefficient and scalability is compromised [Sri01, Rit01].

A hybrid between the centralised discovery mechanism used by Napster and Gnutella's flooding is the solution employed by *KaZaA*[3]. Its model is similar to Gnutella's in that there is no fixed file discovery server, but search functionality is not carried out by all peers; a subset of peers, called the *super-nodes*, are responsible for that. Ordinary peers submit information about their file availability to the closest super-node and contact super-nodes to search for files.

JXTA [JXT01, Gon02] comprises a set of open source, peer-to-peer protocols that allow heterogeneous devices on the network to communicate and collaborate in a peer-to-peer manner. It provides the building blocks required for the rapid development of peer-to-peer applications, such as functionality for peer addressing and resource discovery and sharing. Its ultimate aim is to substantiate a common platform for the development of peer-to-peer applications or the transformation of conventional applications into peer-aware ones.

[1] `http://www.napster.com`
[2] `http://www.gnutella.com`
[3] `http://www.kazaa.com`

While there may be some common problems found in both peer-to-peer and public computing systems, their goals differ significantly. All peer-to-peer systems are *data-oriented*, while in global public computing the *location* of computing resources is important. In typical peer-to-peer systems, users are *transient* and *anonymous*. These two features make the peer-to-peer model powerful for sharing files, but inappropriate for global public computing; long-term presence and relative stability of hosting machines are desirable properties for service deployment, as guaranteed resource availability is required — especially when users are *paying* for the resources they wish to reserve and use. At the same time, anonymity allows peers to provide a low quality of service to others at no cost; even if a ratings scheme or reputation system is used, they can escape their negative score by registering a new identity [Dou02].

2.2.2 Scientific computing

Several scientific projects are harnessing idle computational resources on users' desktop computers to perform large-scale distributed computations and reduce the time required to obtain results. Usually one or more central servers distribute the computation and submit different sub-problems to different machines. The results are then sent back to the servers, where they are assembled.

One of the most successful projects of this type is SETI@home [WCL+01], which searches for extra-terrestrial intelligence in signals received by SETI's radio-telescopes. Launched in May 1999 to search through signals collected by the Arecibo Radio Telescope in Puerto Rico — the world's largest radio telescope — the project originally received far more terabytes of data every day than its assigned computers could process. Volunteers were invited to download the SETI@home software to donate their computers' idle processing time to the project. Currently, about 40 GB of data is received daily from the telescope and sent to computers all over the world to be analysed. Over two million people, the largest number of volunteers for any Internet distributed computing project to date, have installed the SETI@home software.

Folding@home [LSP03] performs processing dedicated to finding cures for diseases by studying protein folding, while PatriotGrid [Gri03] focuses on diseases that are known to be potential weapons of bioterrorism. The Parabon project[4] uses distributed computational resources for accelerating cancer research.

[4] `http://www.parabon.com`

Climate change is the subject of climateprediction.net[5], which uses idle CPU power of machines around the world in an effort to produce a forecast of the climate in the twenty-first century. It does so by quantifying the uncertainties of complex climate models, projections, and scenarios.

MD5CRK[6] was a distributed computing project that hoped to cast doubt on the security of the MD5 message digest algorithm by finding two inputs which produce the same digest [OW99]. MD5CRK was suspended after other researchers devised a technique that allows detecting such collisions without requiring vast amounts of raw computational resources [WFLY04].

The Parallel Virtual Machine (PVM) [Sun90, GS92] is a software package that permits a collection of networked Unix and Windows computers to be used as a single large parallel computer. PVM aggregates the processing power and memory of the networked machines, and is often used for solving large scientific, industrial, and medical computational problems.

Such systems are often wrongly classified as peer-to-peer. While all participating nodes in scientific computing systems do perform the same functionality — similarly to peers in peer-to-peer systems — there is an important difference; in scientific computing, very little interaction takes place between the nodes themselves. Most information flows between the nodes and a number of central servers, which divide the problem, deliver it to the nodes, coordinate node activity, and combine the solutions.

Scientific computing systems underline the potential of distributed computing and its applicability in the real world. However, the technical challenges they face are limited compared to the ones in global public computing; only specific and trusted *ad hoc* applications are deployed, users are cooperative, and resource management is often trivial — if the CPU is idle, then the application runs.

2.3 Active networks

Upgrading or replacing existing network protocols, or even deploying new ones in the wide area, is a very difficult and costly venture. There lies the motivation behind *active networking* [TW96], which aims to address the issue of protocol

[5] `http://www.climateprediction.net/`
[6] `http://www.md5crk.com`

deployment by devising an infrastructure that enables users to inject programs into the network for execution.

Active networking proposes that packets are augmented with code segments to be executed on network elements, such as switches or routers. In the extreme case, each packet transmitted carries a code segment, which is run when it arrives on every node it traverses on the way to its destination.

ANTS [WGT98, WGT99] substantiates an initial implementation of an active network, but exhibits important technical limitations; it allows any user to run code on network elements without any authentication, which is deemed too heavyweight for the per packet processing required at each node. Issues related to the incentive compatibility and sustainability of active networks — namely who would pay for resources consumed by arbitrary users on network elements and why — are not addressed. Furthermore, ANTS permits code deployment without effective resource isolation between processes contained in different packets; Java bytecode verification is mentioned as a possible protection technique, regardless of the fact that a significant proportion of network elements may not be able to run the user-space Java Virtual Machine.

Related to active networks is the research area of *active services* [AMK98], which focuses on application-level deployment of protocols on proxies connected to the network rather than on the network elements themselves. Application-Level Active Networking (ALAN) [Gho02] provides an infrastructure for the circulation of Java code to proxies near the end clients, to allow for customised application-specific protocols to be deployed. Again, the issues of funding for resource consumption and resource protection are not clearly addressed.

Some active networking or services approaches focus on ensuring that code supplied in packets for execution on nodes in the network is not harmful. This is done by mandating that it is accompanied by a digital signature from a trusted compiler — as in SPIN [BSP+95], a safety proof — as with PCC [Nec97], or written in a particular, safe language — as in SafetyNet [WJOP01] and PLAN [HKM+98]. A survey of semantic techniques for active networks is provided in [RT04].

Global public computing does share some motivation with active networking and active services in that they both envisage the deployment of code on machines under different ownership and administration. However, they exhibit distinctive ultimate goals; global public computing aims to allow anyone to run any code anywhere, while active networks limit the scope to trusted administra-

tors upgrading protocols on network elements. This brings a number of important differences to the surface, which are discussed in detail in Section 2.4.6.

2.4 Distributed deployment platforms

This section examines infrastructures that have been developed to allow the deployment of distributed services. While none of these support the deployment of generic computation by members of the public, they do exhibit higher technical relevance to global public computing. Here, the operation of several such systems is outlined, and their shortcomings with respect to their use for general-purpose global public computing are highlighted.

Grid computing allows cooperative scientists to run well-behaved and trusted distributed applications on large numbers of machines around the world, generally without direct control of their location. Section 2.4.1 examines *Globus*, one of the most popular Grid computing initiatives.

The *Condor* deployment platform, analysed in Section 2.4.2, provides an infrastructure for allowing computationally intensive applications to be deployed on clusters of machines. It facilitates sharing distributed resources by grouping them in resource pools.

Descriptions of Globus and Condor are followed by a discussion about work done in the *Global Grid Forum* in Section 2.4.3, whose purpose is to devise proposals for interoperable standards for the way Grid platforms are designed and built.

Section 2.4.4 describes *PlanetLab*, an overlay testbed for distributed computing experiments. Similar to Grids, cooperative user communities and well-intentioned services are assumed, no explicit rewards are provided for resource suppliers, and no charging or payments infrastructure is supported. In contrast to Grids, PlanetLab targets applications for which the location of resources is usually more important than the resources themselves.

This is followed by a discussion on commercial *utility computing*, envisaging computing resources as a generic service and making an analogy to other services such as electrical power, in Section 2.4.5. These systems aim to maximise efficiency of resource utilisation by providing businesses with on-demand computing power, and charging them for specific usage rather than at a flat rate.

Section 2.4.6 sets the *requirements* for supporting global public computing, and analyses which of these are met by each of the deployment platforms analysed. Subsequently, Section 2.5 proposes a generic global public computing platform as a common resource acquisition substrate.

Distributed deployment platforms — particularly PlanetLab and Globus — are active research projects, and as such they are constantly evolving. My description of their architecture is based on the existing literature explaining the overall operation plans. The intention is to avoid discussing implementation details that may change frequently or the temporary solutions that may be employed; however, it is possible that part of what is presented here may already have been changed or that it may not have been introduced yet in the implementation of those systems at the time of writing.

2.4.1 Globus

The Globus project [FK97] aims to provide an infrastructure to form *networked virtual supercomputers* for deploying distributed services — an approach earlier referred to as *metacomputing*[7]. A low-level toolkit provides the enabling technology for the Grid, allowing participants to share distributed computing resources. The toolkit supports basic mechanisms such as authentication, resource discovery, monitoring, management, and data access.

The functionality of this toolkit is exposed to applications as a *metacomputing abstract machine*, on which a range of services and applications can be built. The Globus resource management infrastructure [CFK+98] allows the representation, discovery, and management of resources in Grids. Simple *policies* for resource management may also be defined. Entities that participate in the resource discovery and management process in Globus include the *information service*, *resource brokers*, *coallocation agents*, and *local resource managers*, as shown in Figure 2.1.

Layered implementation. The coordination protocol proposed by Globus is structured in layers, each one implementing different parts of the Grid's functionality [FKT01]. The bottom layer, called the *fabric* layer, consists of the resources

[7] A metacomputing system, or metasystem, is defined as a collection of computer systems, accessible via a networked environment, which users perceive as a single computer [SC92].

to which access is mediated by the higher-level protocols. These resources are typically computational, storage, network, and code repositories.

Above the fabric layer is the *connectivity* layer, which includes the communication and authentication protocols needed for Grid interactions. In an Internet-based deployment of such Grids, communication is handled by the TCP/IP suite, while authentication is typically based on a public-key infrastructure combined with an extended version of Transport Layer Security (TLS) [DA99].

The *resource* layer sits above the connectivity layer. It uses facilities found in the connectivity layer to provide negotiation, initiation, monitoring, control and accounting of sharing operations on individual resources. Since this layer's protocol is only concerned about individual resources, global issues — such as resource naming — are not addressed here.

The resource layer includes two sub-protocols; the *information* protocol is used to obtain information about the structure and state of a physical resource, as well as its current load and usage policy. The *management* protocol is used to negotiate and manage access to the resource. Example protocols of this layer are the Grid Resource Information Protocol (GRIP) and the HTTP-based Grid Resource Access and Management [Fit01], which are both used by Globus.

The *collective* layer builds on top of the resource layer and contains protocols that are associated with collections of distributed resources. Such protocols may include directory services, and services for scheduling, brokering, monitoring, diagnostics, data replication, and accounting. A typical example of a service running in the collective layer of Globus is the Metacomputing Directory Service (MDS), which provides resource discovery mechanisms.

Resource description and discovery. Resources can be represented as instances of *resource object* classes [FFK+97]. A resource object defines the attributes associated with an entity as well as the types of values these attributes may contain. The two typical resource objects used are termed `GlobusHost` and `GlobusResource`, which may be extended for adding custom attributes. The core attributes of `GlobusHost` include the host name, type of operating system, and the total main and cache memory of the host. Resources in Globus can be represented in a language similar to the one proposed in [MJK94].

The Globus Resource Allocation Manager (GRAM) [CFK+98] is a set of service components being used for providing a common standard interface to appli-

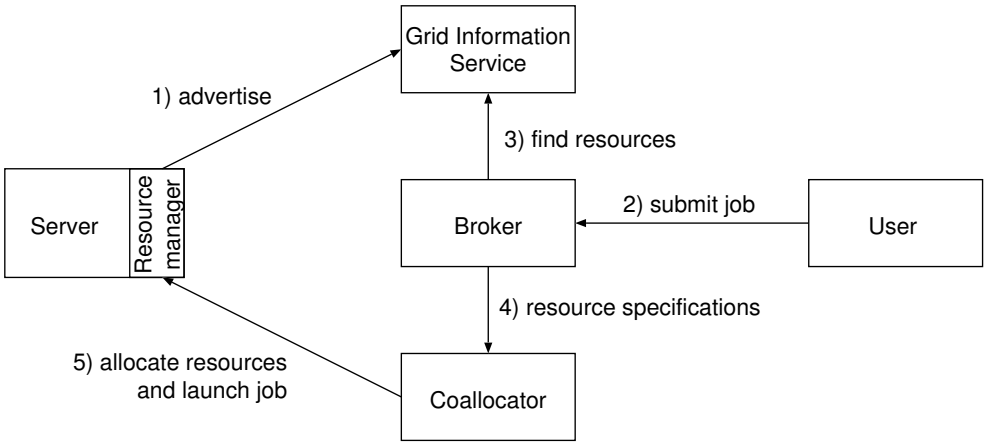

Figure 2.1: Resource discovery and service deployment in Globus

cations for requesting and using resources on Globus nodes. Uniform descriptions, formatted in Globus' Resource Specification Language (RSL) [Glo00], are then translated to a form that local operating systems can comprehend at each Globus node. The types of resources that can be requested using RSL are the number and type of the machines to be selected, the memory needed, and the type of network connectivity required. Other execution parameters may be included, such as the working directory and command-line arguments for the job execution, as well as the maximum CPU time to be allocated and the maximum wall-clock execution time for the job.

The *information service* is used for publication and discovery of available resources in Globus Grids. The information service uses the Metacomputing Directory Service, which builds on the Lightweight Directory Access Protocol (LDAP) [HM02, vLF98] to provide facilities for advertising and retrieving resource information. Hosts advertise their resource availability to the information service — operation 1 in Figure 2.1.

Job submission in Globus can be done either by contacting the individual nodes directly, or by submitting a job description to a *resource broker* — operation 2 in Figure 2.1. Resource brokers contact the information service to find resources — operation 3 in Figure 2.1. Then, they translate resource requirements to *resource specifications*.

Service deployment. The resource broker then passes the resource specifications produced to a *coallocator* — operation 4 in Figure 2.1. This breaks the

specifications down to components, and passes each component to each of the nodes on which jobs are to be deployed for the subsequent *resource allocations* to be made — operation 5 in Figure 2.1. It also provides a means for *monitoring* the jobs' status as well as limited *management* facilities, such as interfaces for terminating jobs. Resources are allocated optimistically based on current availability, as no support for QoS guarantees is provided in the core Globus architecture.

Globus uses coallocation [CFK99] to allow simultaneous allocation of resources on more than one Globus node. To enable this, the coallocator operates in two distinct stages: *reservation* and *allocation* of resources. In the first stage, a reservation is created on the node to provide assurances that a subsequent allocation request succeeds. In the second stage, when reservations on all nodes involved have succeeded, an allocation request allows services to obtain the resources. This technique bears similarity to the *two-phase commit* technique devised several years earlier in the context of distributed databases [Gra78, LS79].

The local *resource managers* on each Globus node, which all run the Globus Resource Allocation Manager (GRAM) protocol, process the component specifications passed to them by the coallocators and decide whether to admit or deny the allocation requests. If a request is accepted, the resource manager proceeds to launch the job required by invoking a *job manager* process. This ensures that everything the new job needs to run is made available to it at execution time — such as arguments and files. Feedback from the job's execution can be passed back to a user-specified URL. Resource managers also periodically update the information service with their current resource availability and capabilities.

For convenience, the launching of complex distributed services is facilitated by *service factories*, which launch new instances of predefined distributed services. Factories are themselves normal services that run on Globus nodes. Each factory is specific to the type of service to be created; in other words, if a factory is used, a service creation request does not need to specify the type of service to be created or how it is to be launched — this is part of the factory itself.

Factories then create instances of the desired services by using the resource discovery, reservation, and coallocation mechanisms provided. Factories also provide the services with delegated proxy credentials that allow them to perform operations on behalf of the users who created them.

Each service is associated with a limited *lifetime*. To keep it alive, either the user who created it or some other service acting on behalf of the user needs to keep sending `keepalive` messages to the server on which the service runs. Once

the service has finished its operation, its creator stops refreshing its lifetime and the service terminates subsequently.

Third-party services can also provide a facility for *discovering* existing services and *composing* new services by plugging together existing service components. *Registry* services allow other services to submit descriptions including their identifier, name, type, interfaces provided, and time-to-live information. They also export interfaces for discovering suitable service components.

As discussed in [FKNT02], jobs running on individual Grid nodes are executed in *hosting environments*. The interaction models specified by Globus are independent of the nature of the underlying technology used to support hosting environments, as long as the common interfaces used by the higher-level Globus services are exported. Simple Unix processes are used at the time of writing, and JVM environments are suggested as a possible hosting technology.

Resource management. A scheme for defining simple *policies* on resource usage, stating which categories of users are entitled to access a resource, is suggested in [KWL+03], although this feature is not yet present in the Globus implementation at the time of writing. When a resource reservation request is made, it is evaluated against both local and global resource allocation policies to decide whether access to resources should be granted.

The language used for describing resource allocation policies consists of *actions* associated with *job descriptions*. Actions define what should happen when a resource allocation request is made for a job that matches the job description. Job descriptions are in this context collections of attribute-value pairs, forming subsets of the resource objects used to represent Globus resources — of the type of "create and populate a local directory, and start the job, if the job's owner is a specific user".

Each local resource manager runs a *gatekeeper* module, which has ultimate control over which jobs are admitted for execution. It also has the responsibility to authenticate users and carry out operations required to launch jobs.

2.4.2 Condor

The Condor project [LLM88, LBRT97, TL03, TTL04] is one of the first platforms developed for the deployment of distributed services. It mainly aims to support high-throughput computing for scientific applications by integrating distributively owned computing resources in resource pools and providing mechanisms to applications for shared use.

In overview, "users submit their serial or parallel jobs to Condor, Condor places them into a queue, chooses when and where to run the jobs based upon a policy, carefully monitors their progress and ultimately informs the user upon completion"[8].

Condor-G [FTF+02] represents the marriage of technologies from the Condor and Globus projects. It is effectively a Condor-compliant client — or job submission portal, which can interface with Globus Grids.

Resource description and discovery. Central to the description of resources in Condor is the scheme of *classified advertisements* (`classads`) [CRLS03]. These are data models used for representing machines and jobs, including declarative descriptions of their characteristics, constraints, and preferences. They have no fixed schema and consist of a set of attribute-value pairs. Classad expressions are strongly but dynamically typed, with the supported types being integer, floating point, boolean, string, timestamp, and time interval.

Applications to be executed in Condor environments are termed *jobs*. To execute a Condor job, resources required for its execution need to be located first. Initially, a user contacts the *customer agent* — also referred to as `schedd`, which provides interfaces for submitting, querying and removing jobs — operation 1 in Figure 2.2. The customer agent manages a persistent queue of application descriptions for the jobs to be run and is responsible for making sure that user requirements for the jobs are met by the resource allocations to be made.

Each Condor server is managed by a *resource owner agent* — also referred to as `startd`; this is an application responsible for discovering jobs to be executed on the machine, within the constraints placed on its resources by its owner. The resource owner agent is responsible for making sure these constraints are met, as well as for monitoring the machine and for cleaning up after jobs are finished.

[8] According to the Condor project web site (`http://www.cs.wisc.edu/condor/`).

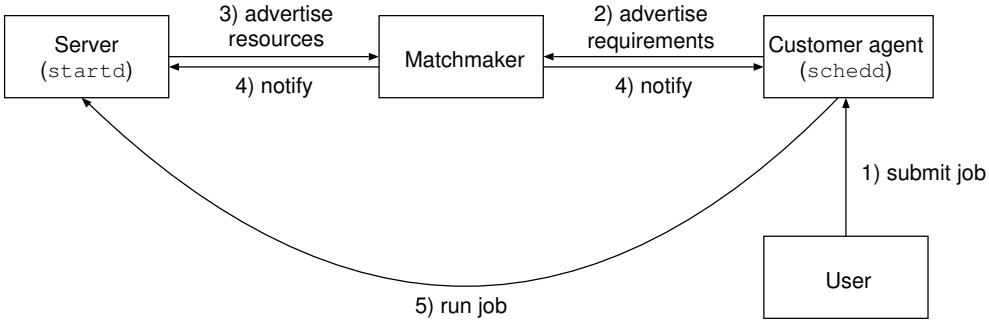

Figure 2.2: Resource discovery and service deployment in Condor

The *matchmaker* [RLS98, RLS00, RLS03] is a central component used in Condor infrastructures to provide functionality for matching resource requests with availability as well as for enforcing system-wide resource allocation policies. Customer and resource owner agents represent their jobs and machines as `classads`, and *advertise* these descriptions to the matchmaker — operations 2 and 3 in Figure 2.2. The matchmaker maintains a soft state, as resource owner agents submit classads, advertising their machines, periodically.

Then, the matchmaker *scans* through all advertised `classads` and creates pairs that satisfy each other's constraints and preferences. For two `classads` to match, both their corresponding *constraints* must evaluate to true. The *rank* attribute of classads must evaluate to a floating point number, which is used to select one of all compatible matches.

Once a match is found the matchmaker *notifies* both parties about the details of their potential cooperation — operation 4 in Figure 2.2. As stale information may lead to bad matches, no resource allocation is made at this stage.

Service deployment. The final step is that of *claiming*; the matched consumer and resource owner agents contact each other, negotiate the terms of the deal, and agree to cooperate to execute a job — operation 5 in Figure 2.2.

While the matchmaker does notify the parties about successful matches, it is up to the customer and resource owner agents to decide whether to proceed

with their cooperation for the execution of a job. If both parties agree, the job is deployed on the server. Even after the match is agreed by both parties and the job is under execution, either side may abort the operation at any time.

Once a successful match is found and agreed between the customer (`schedd`) and resource owner (`startd`) agents, the latter launches a *starter* process for creating an execution environment to host the job. It creates a working directory, sets up standard I/O streams, and monitors the job's progress and exit status.

On the customer side (`schedd`), after agreeing on a match the agent spawns a *shadow* process, which is responsible for making sure that everything the new job needs to run is made available to it at execution time — such as arguments and files — similar to the job manager in Globus. When the job terminates, the shadow process examines the exit code and output data to determine whether the job has been executed successfully and terminated normally.

Resource management. The *rank* attribute of `classads` can be used to allow users to represent preferences in terms of which servers they wish to get resources on. Similarly, it can be used by server owners to define which users get priority in accessing a resource. The expression used to calculate the value of the rank attribute is included in the `classad` to denote how preference should be determined. For example, if the `classad` is representing a job, preference could be given to machines that have more memory than others; if the `classad` is representing a machine, preference may be given to jobs that are shorter than others.

The owner of a machine or a job can further annotate the classified advertisement with *constraints*, which define the conditions that the other side must satisfy for a match to occur. For instance, a machine owner may specify the set of users allowed to claim the machine's resources, as well as other conditions under which this may happen, such as low current load or absence of local activity — i.e. the keyboard having been idle for several minutes. The owner of a job may require that the machine to be used is of a particular architecture, runs a specific operating system, or has a minimum amount of memory available.

It is important to note that *policies* on resource management are enforced at several distinct points. Consumer agents enforce policies by making sure the constraints of their jobs are met before jobs are deployed. Resource owner agents implement the policies associated with the machines' `classads`. The matchmaker itself may implicitly implement global policies by favouring particular matches.

2.4.3 Global Grid Forum

The Global Grid Forum (GGF) encompasses a number of research and working groups, whose collective aim is to provide the specifications of an open global Grid infrastructure. The GGF is not building any particular Grid; its ultimate goal is to *standardise* the functionality that different Grid components are to provide and the APIs exported by Grids and used by Grid-enabled programs, to allow for interoperability.

At the same time, Globus is becoming the *de facto standard* in Grids, through its popularity and relative maturity. The OGSA [FGK04] and OGSI [TCF+03] working groups represent efforts within the GGF to embrace Globus' approaches, in order to accelerate standardisation.

Work within the GGF in the areas of *discovering* and *administering* Grid resources is carried out in several research and working groups in parallel. The Distributed Resource Management Application API (DRMAA) working group is developing an API specification for the *submission and control of jobs* [RBC+04]. In the model proposed by DRMAA, each job is submitted to a Distributed Resource Management System (DRMS), which exports interfaces for individual and bulk job deployment. DRMS also supports management and control operations for suspending, resuming, killing, and releasing jobs, as well as for checking their status or exit code. It should be noted that the job management API proposed by DRMAA is different from the one proposed by OGSI and OGSA.

Advance resource reservations are the subject of the Grid Resource Allocation Agreement Protocol (GRAAP) working group. GRAAP aims to devise a protocol for resource reservation and negotiation between nodes and users or other entities, like super-schedulers[9], initially only for computational resources (CPU time). In [Glo03a], GRAAP is considering a number of scheduling algorithms that support advance reservations, and proposing a set of interfaces to be exported to users or super-schedulers for supporting such functionality. However, as identified in [Glo03b], the group has not yet addressed issues related to the representation of Grid resources at the time of writing.

The Grid Economic Services Architecture (GESA) working group considers providing the enabling infrastructure for Grid platforms where resources will be *priced* and users *charged* according to usage of such resources [New03]. GESA

[9] A *super-scheduler* — or *Grid scheduler* — manages the distribution of workload among nodes in Grids for optimising resource utilisation efficiency.

has identified the need to be able to sell the same physical resource in several different fashions, under different QoS parameters and pricing schemes, as well as the necessity of a scheme to define how users are given access to resources.

To facilitate charging in Grids, it is necessary to provide *accounting* mechanisms. The Usage Record (UR) working group proposes a common format for representing resource consumption information in natural language [Mac04] and XML [LJ03]. The Resource Usage Service (RUS) working group devises the interfaces of a service for storing and retrieving such accounting records [New04].

The Policy research group aims to standardise the expression of policies to control the distributed resources available in Grid environments [WS03]. Policies are expressed as *policy rules* using the *policy management tool* and stored in a central *policy repository*. *Policy consumers* — software components on the machines that will enforce the policies — retrieve rules from the repository, perform translations if necessary and instruct the *policy targets* accordingly. These are the entities that carry out the policy definitions, such as the scheduler for CPU resources. The *security system* is an access control component performing identification and authentication, and there is a *system administrator* in charge of provisioning configuration and has ultimate control of the Grid system.

Most of the proposals of the various GGF working and research groups are still at a draft stage at the time of writing. Furthermore, the GGF's effort to abstract from the diverse Grid implementations and suggest common and interoperable standards, the overlapping interests and responsibilities of the various research groups, and the considerable lack of communication between them, make the proposals often vague [WS03, Pul03] and inconsistent [FGK04, RBC+04].

2.4.4 PlanetLab

The PlanetLab project [PCAR02, BBC+04] substantiates an overlay network testbed for deploying large-scale services. It consists of a number of machines, termed *nodes*; these run a common software package, which includes a modified Linux kernel, mechanisms for bootstrapping nodes and distributing software updates, as well as a collection of management tools performing functions such as monitoring node state and managing user accounts and authentication keys.

The PlanetLab software allows authenticated users to obtain computing resources on a number of nodes and deploy services by executing tasks on each of

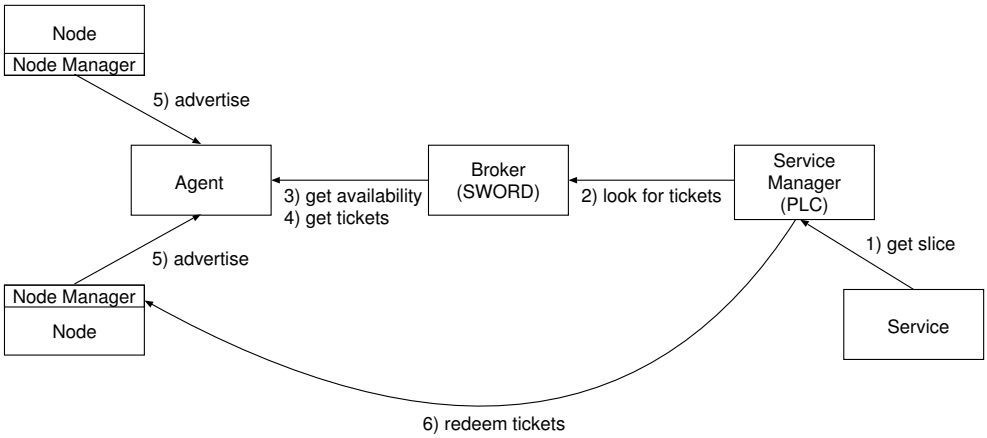

Figure 2.3: Resource discovery and service deployment in PlanetLab

these nodes. Some degree of resource isolation between the different services deployed on a node is achieved by running each one of these in a *sandboxed* "Virtual Server on Linux" environment [Gel03] and using a modified version of the Linux kernel to enforce some protection between these environments.

The key components of the PlanetLab resource management and service deployment infrastructure are *slices*, *service managers*, *resource brokers*, *tickets*, *agents*, and *node managers*, as shown in Figure 2.3.

PlanetLab introduces the abstraction of a *slice* to denote an allocation of resources spread across a number of machines [PCAR02]. A slice is essentially a horizontal cut of the global PlanetLab resources, allocated to a particular service. It typically binds together several types of computing resources such as processing time, memory, storage and network resources across several machines.

Resource description and discovery. To obtain a slice, the following process needs to be followed, described in detail in [CCR+03, PV02]. Each service needs to contact a *service manager* — operation 1 in Figure 2.3. The manager is typically running outside PlanetLab to allow the deployment of new services without requiring prior acquisition of a slice. Its main responsibility is to locate the *tickets* needed to obtain the required distributed resources.

PlanetLab's implementation is tied to a single service manager, called *PlanetLab Central* (PLC). This comprises a database containing information about all nodes, users, their keys, and active slices. It exports interfaces for *creating* and *managing* slices, for instance to allow associating users and additional nodes with slices. PLC takes care of user authentication, by pushing user keys to nodes at slice creation.

Service managers obtain tickets by contacting one of the existing *resource brokers*, which may themselves run as PlanetLab services or exist outside PlanetLab — operation 2. Service managers specify the resource requirements and environmental expectations for the slice to be created. A proposed resource broker is SWORD [OAPV04], a distributed resource discovery system. SWORD is based on an implementation of the Bamboo DHT [KRRS04]. As the implementation of SWORD is distributed, PlanetLab is hoping that the resource discovery mechanisms scale respectively.

Then, resource brokers contact one or more *agents* to discover if appropriate tickets can be collected and returned to the service managers — operations 3 and 4. Agents are software components that collect resource availability information from a set of nodes and are authorised to issue tickets that can be used to obtain resources on these nodes. Agents receive information about resource availability directly from the nodes in the form of advertisements — operation 5 in Figure 2.3.

Resource brokers can perform two kinds of queries on an agent. First, they can ask for current resource availability on one or more nodes. Agents in this case return descriptions of tickets held — sets of *advertisements*. Brokers can then combine those advertisements — representing currently available resources on the nodes — with the services' requirements, as described by the service manager, to build *slice specifications*; these are submitted to the agents, which in turn return the corresponding tickets.

The process through which agents acquire tickets is as follows. Each individual PlanetLab node manager — control and management software that runs on each PlanetLab node — holds information about the resources available on the node, as well as mappings between sets of these resources and what is termed `rcap` data structures [CS03]. These are long (128-bit) opaque values referencing specific sets of resources on a node. Knowledge of an `rcap` value implies access to the associated resources, as the space of `rcap` values is large enough for guessing to be considered improbable.

Planetlab *agents* can acquire `rcap` values from a node by presenting the node manager with a specification of a set of resources to be reserved and the reservations to be made, called `rspec`. These include a start time, an end time, and a list of privileges or limits to be imposed. The privileges or limits that can be used — such as upper bounds on inbound and outbound IP traffic, the number of TCP or UDP ports to be used, and limits on CPU percentage, memory, and disk space — can be selected from a static list available on the node manager. Provided that `rspec` is a subset of the currently available resources on the node, the node manager creates, stores and returns an `rcap` value to be used for referencing and obtaining the resources.

Service deployment. Once a service manager has received the tickets needed to instantiate a slice, it can try *redeeming* these for real resources by directly contacting the node managers of individual nodes — operation 6 in Figure 2.3. A node manager takes a set of tickets as input and, according to its local admission control policy, decides whether to honour the tickets. If the admission control decision is positive, the node manager creates a new Virtual Server execution environment for the service and allocates the requested resources to it. Node managers are individually responsible for deciding how many tickets to issue and for honouring tickets or not.

Transferring files and data required for service deployment to the nodes is a responsibility of users. They transfer data to each individual server involved either manually or using automated tools. Similarly, services need to be individually configured and launched on each server. For easing the task of distributing experimental software to a set of PlanetLab nodes, the CoDeploy service has been developed — see Section 5.2.1.

Resource management. In terms of defining and applying resource management policies, a node manager exports functionality to the infrastructure services to *limit* and *unlimit* the usage of a particular resource on the node by a specific user in the context of a given slice. At the time of writing, PlanetLab does not offer resource management functionality as part of its core platform, other than launching best-effort sandboxed execution environments.

2.4.5 Utility computing

Most of the leading IT services companies have announced initiatives in the area of utility computing services under different business names. As a few examples, HP is providing "Infrastructure and Management Solutions for the Adaptive Enterprise" [HP03], Sun is proposing the "N1 Grid: Managing N Computers as 1" [Sun02], while IBM is offering "On demand business" [IBM02] and developing the Océano project [Gol00]. Furthermore, Ensim Virtual Private Servers [Ens03] and Akamai EdgeComputing [Aka03] provide worldwide service hosting facilities.

Detailed technical descriptions of these initiatives are not available, partly because these are commercial projects, therefore protected by the companies that launch them, and partly because some are in an early development stage. The rest of this section focuses on the high-level design requirements and goals that utility computing architectures are hoping to meet.

Utility computing infrastructures build on Grids to provide a "plug and pay" model for executing tasks. Instead of maintaining expensive infrastructures themselves, companies will be able to submit their tasks to the utility computing infrastructure. The infrastructure will then make sure that tasks get the resources they need from a pool of available distributed resources, execute, and finish. Charges will be passed on to the tasks' sponsors accordingly.

Utility computing presents attractive potential benefits for businesses. Estimating the computing resources needed to meet potential *demand surges* — caused by, for instance, the launch of a new product or a popular article — is complicated. At the same time, *business continuity* is crucial; an hour of downtime costs half of U.S. corporations more than $1,000, while in major organisations it ranges from nearly $100,000 per hour in retail, to millions per hour in banking and brokerage [Jac04, Ide04, Pat02].

In traditional service hosting, such as co-location facilities, companies need to provision for the worst-case scenario; sufficient resources to meet the highest possible demand need to be constantly available. To meet such potential demand peaks, most companies *overprovision* the capabilities of the machines they use to host the services they provide, and pay significantly more than is technically necessary.

At the same time, overprovisioning of resources incurs *indirect costs*; maintaining more machines usually requires more management, thus higher expenses

in IT administration staff. Also, even if overprovisioning is standard practice one can never be sure that a demand surge will not still exceed availability. In case the provisioning has not been accurate, or demand grows unpredictably, acquiring additional resources is highly manual, slow, and can cost a lot in terms of money, time, and effort.

Utility computing promises to provide businesses with greater flexibility and resilience, and at the same time more efficient utilisation of resources at lower operating and maintenance costs.

While this model of distributed computing encompasses significant similarities with the XenoServer vision, it also exhibits important differences; the *administrative model* in utility computing is very simple, as all resources are owned by a single organisation — the company that owns the servers. Thus, control of servers is centralised, and resource management decisions are trivial. At the same time, utility computing server farms do not envisage *scaling* to great numbers, as computing resources provided are usually physically *colocated*.

2.4.6 Putting the pieces together

Distributed deployment infrastructures have offered important, but piecemeal, solutions. Different systems are solving different parts of the problem, and only in the context of the needs of the user community of each system.

In this section the main technical challenges on the way towards global public computing are identified, within the framework outlined in Section 1.3. The specific challenges met by each one of the deployment platforms presented earlier are also discussed. Wherever possible, comparisons to related problems and solutions found in peer-to-peer systems, scientific computing applications, and active networks are included. My findings are summarised in Table 2.1.

Ease of deployment. The deployment model followed by all of the aforementioned distributed deployment platforms is largely inconvenient for realistic global-scale service deployment; users have to transfer the data required to launch services, configure, and launch the services individually on every server involved. Even if automated tools can be developed to assist bulk transfers of data to all servers, scalability is compromised; transferring large amounts of data to each individual server incurs prohibitive network traffic figures. A more detailed discussion of deployment models is provided in Section 5.2.1.

Challenges	PlanetLab	Condor	Globus	Utility
Ease of deployment	×	×	×	×
Non-cooperative users	×	×	×	√
Untrusted code	×	×	×	√
Out-of-the-box applications	×	×	×	depends
Self-financing	×	×	×	√
Short timescales	√	√	√	×
Incremental scalability	×	×	√	×
Flexible server selection	√	×	√	×
Resource-oriented	×	√	√	√
Location-oriented	√	×	×	×
Resource description coordination	√	×	×	×
Resource management	some	×	some	×
Policy-based management	some	some	some	×
Convenient management	×	×	×	×
Local control	√	×	depends	×
Federated policies	×	×	×	×
Overlap resolution	×	×	×	×

Table 2.1: Main research challenges for global public computing and which of those are met by previous distributed deployment platforms.

This might be adequate for Grid computing, where applications are largely immobile and maintain a long presence in the system. Global public computing must support services that migrate often, run only for short periods of time, and need to be deployed on large numbers of machines around the world in parallel. More *efficient* mechanisms for *deployment of global-scale services* are needed to allow customers to quickly obtain resources and run services without requiring the transfer of excessive amounts of — largely replicated — data.

Non-cooperative users. Aside from utility computing schemes, existing distributed deployment platforms, as well as active networks, rely on one main assumption: that resource owners make their resources available to remote users without receiving any direct reward. This may be a reasonable hypothesis in cooperative scientific communities; in the Condor pools or PlanetLab, where users of resources are usually owners of PlanetLab nodes too. It may also be an affordable assumption in cases where executing the code brings benefits to the hosts that run it, as in active networking, where nodes have the incentive of obtaining up-to-date version of the protocols they are running. However, it is not clear how this hypothesis can be extended to general-purpose global public computing.

The cooperative scheme will not be incentive-compatible if applied to the global community of *non-cooperative* users, as self-interested resource owners rarely consent to provide resources for free. For the platform to be open to any user, it is necessary that users get charged for the amount of resources they consume, and paid for the amount of resources they provide. Also, if users are to pay for the resources they acquire on the servers, stronger protection and resource isolation is needed between the services running on each server.

Untrusted code. If servers are to execute tasks belonging to competing, *untrusted* services, then it is necessary to ensure that services are not able to adversely influence each other's execution or performance in any way.

In most distributed deployment platforms, attempts to provide resource isolation are concentrated either on restricting the kinds of applications that can be accommodated, or on modifying the CPU scheduler to provide some kind of per service, rather than per process, fairness. Utility computing platforms often lease dedicated servers to users at a much higher cost, to avoid dealing with protection and isolation issues. Active networks address the issue by requiring that code be signed by a trusted compiler, accompanied by a safety proof, or written in a restrictive, safe language; this, however, does little to prevent potential performance interference between competing tasks.

Condor does not support jobs that spawn multiple processes or threads, perform interprocess communication, require long-term network connectivity, use timers and alarms, or memory-mapped files. PlanetLab attempts to provide basic resource isolation by hosting services in sandboxed Virtual Servers [Gel03]. However, as argued in [BDF+03a], such protection is inadequate in an uncooperative environment aiming to accommodate experimental or harmful tasks. A malicious service cannot be prevented, for instance, from taking up all available file descriptors. In order to accommodate any service, including potentially untrusted or experimental ones, stronger protection mechanisms are required.

Out-of-the-box applications. The service deployment infrastructure of PlanetLab only accommodates services running on a specific version of Linux, as the required protection functionality is provided by Virtual Servers which require modifications in the operating system kernel. It also does not allow running custom operating system kernels, including components such as device drivers or modules that users may need and that may not be part of the PlanetLab

Linux kernel. In a general-purpose public computing environment, it is necessary that services running on any of the popular operating systems, including custom kernels, are accommodated.

Globus mandates that applications to be deployed are "gridified"; to be executable in Globus Grids, applications need to be rewritten or modified to comply with its API. Globus' API is available only for some programming languages — namely C, Java, Perl, and Python. Applications run on Globus nodes need to be executable in Linux.

Condor supports both Windows and Linux — but not other operating systems, and does not need compliance to an API. It does require, though, that jobs are recompiled to be executable on one of a number of "universes" — types of execution environments, such as PVM or JVM. Active networks often mandate that programs are written in safe, restricted languages, or run on specific safe interpreted environments.

While mandating that applications be written in a specific language, be executable on a particular version of an operating system, or comply with a common middleware may be acceptable for the applications that Grids envisage hosting, it generally raises the cost of entry for users significantly. It requires that users have access to the source code of the services they wish to deploy, programming knowledge, and time to modify it.

To maintain a low cost of entry, it is necessary that global public computing is able to run *out-of-the-box* applications, without requiring modifications or recompilation.

Self-financing. In the world of PlanetLab and Grids, machines are provided and maintained by universities and research institutes to promote research and scientific collaboration. Funds for server acquisition, upgrades, and maintenance, as well as transatlantic network traffic come from government funding, donations, and research grants.

To ensure the sustainability of global public computing, it is necessary for the deployment platforms to be *self-financing*. The cost of running a platform, including development and maintenance of the various software and hardware components, should be distributed, and covered by applying charges on the resource consumers.

Short timescales. In most distributed computing systems, participation over *short timescales* is possible; users can obtain resources for a short time and then release them. In the Grid computing case this is a relatively easy task partly due to the assumption of trusted, cooperative, and well-behaved user communities, and partly because of the absence of any form of resource pricing and payments.

On the other hand, utility computing and traditional server rental, which do not anticipate collaborative users and rely on charging for resources, unsurprisingly do not support resource acquisition over short timescales; users manually contact utility computing providers and engage iN long-term agreements.

For global public computing, it is necessary that the cost of entry for the user is low. Requiring that resources are leased by the week or month would discourage exploratory users and harm the efficiency of resource utilisation. Resource acquisition over short timescales must be supported, and charging needs to be done in a fine-grained fashion for resource consumption.

Incremental scalability. The resource discovery process in Condor pools is carried out by a single, central matchmaker. All `classads`, representing machines willing to host services or expressing the requirements of services to be hosted, are submitted to the matchmaker. There, availability is matched against requests.

This affects the *incremental scalability* of Condor pools. As the numbers of machines, users, and services rise, so does the load that the matchmaker needs to handle, thus making it a potential bottleneck in the resource discovery process. While this may not be a problem in cases where the maximum expected growth can be handled by the matchmaker, it is not a solution that can apply to general-purpose global public computing.

Also, the central matchmaker is a single point of failure; even if the machine hosting the matchmaker is relatively reliable and well-configured, an unanticipated hardware fault could render the entire Condor resource discovery system unusable for some time. A multiple-matchmaker model, even by simple replication of the matchmaker, can provide better resilience against unexpected faults.

PlanetLab relies on the PlanetLab Central (PLC) database, which contains information about all users, slices, resource allocations, and policies. This single, central database presents a single point of failure and potential scalability bottleneck. Scientific computing applications normally distribute and recombine the scientific problem in one — or a few — central servers, therefore their scalability

is questionable. Utility computing, due to its physically centralised nature — i.e. server farms, does not envisage scaling to great numbers, thus its matchmaking and resource reservation operations are often highly manual. Globus follows a more scalable approach, using distributed directories for resource discovery.

General-purpose global public computing requires the service deployment infrastructure to be incrementally scalable on demand; performance of the service deployment mechanisms should not deteriorate significantly as the number of participating users and servers increases.

Flexible server selection. Global public computing does not aim to optimise resource utilisation or balance load distribution. Users need to be allowed to select the servers they wish to use to deploy their services, according to their requirements and the servers' capabilities.

Condor relies on a single, central matchmaker for discovering distributed resources. Whilst this approach simplifies design, maintenance, and debugging, it has a number of implications on the extensibility and openness of Condor platforms. First, the single matchmaker model impairs competition, and prevents the evolution of more complex economic relationships and institutions.

Condor's approach also fails to allow specialisation; most users need only fairly simple functionality from the matchmaker of their choice, namely to find servers that can provide the required resources. However, particular groups may wish to impose specialised criteria in the server discovery procedure; for instance, "find a set of servers, such that the total round-trip time between those servers and a fixed network point is minimised" or "find the server that will provide the best value for money". It is not practical for a single matchmaker to support this level of specialisation in its functionality in all potential dimensions. Furthermore, it is not possible to anticipate all server selection criteria that different user groups may require in the future.

The strategy that Globus uses for discovering and selecting servers is more flexible. In Globus, advertising resources is separated from searching through the advertisements, and is carried out by different entities. Advertisements are stored in the information service, which can be implemented in a distributed or replicated way as this functionality is provided by LDAP, on which the information service is built. Then, multiple independent resource brokers can discover resources in the information service and, using a matchmaking algorithm of their choice, they can then match those with the resource requests they receive.

PlanetLab aims to use SWORD, a resource discovery tool for wide-area distributed systems. This provides a distributed mechanism for storing and searching through resource availability advertisements from the different PlanetLab nodes. Its distributed implementation can provide scalability advantages.

Searching for files in peer-to-peer file sharing applications shares some goals with resource discovery in global public computing. However, resources are only of one type, files, so no complex resource descriptions are needed. Resources are not "consumed" in the same sense as in global public computing; using up a millisecond of CPU makes it unavailable to other users, while downloading a file does not. Instead, an additional replica of it becomes available in the system. Also, the number of participants in peer-to-peer networks can grow extremely large — in the order of tens of millions, influencing system design accordingly.

Global public computing requires *flexible server selection*; users have to be given full control over which servers their services are to be deployed on. If server discovery and selection components are provided to assist users, these need not be part of the core deployment infrastructure, but rather third-party services themselves. This allows for multiple, specialised, and competing matchmakers to coexist, compete, and complement each other.

Resource-oriented vs. location-oriented. Condor, Globus, the Grid infrastructures envisaged by the GGF, utility computing initiatives, as well as scientific computing projects and peer-to-peer systems are all *resource-oriented* — resources being computational, such as CPU and memory, or files. Their main goal is to provide users with access to raw resources; whether the servers that provide those resources are next door or at the other side of the world generally makes little difference to the applications deployed on these platforms. Functionality for location-based resource acquisition is either non-existent or limited.

PlanetLab differs in that it is built for *location-oriented* applications, where it is the location of resources that matters much more than the resources themselves. This is because PlanetLab was built as a testbed for distributed systems that usually need to be widely distributed.

Distributed services often need to obtain resources at key network locations to minimise network latencies between the services and their clients, or to reduce long-haul network traffic and the associated charges. Global public computing needs to provide facility to allow users to decide where to deploy their services using any type of criteria they consider appropriate.

Resource description coordination. The resource representation scheme has to comprise mechanisms to avoid having an overabundance of different names and pricing schemes — e.g. "P4 CPU 2 GHz", "Pentium4 processor at 2GHz", and "2GHz P4", or "10MB of RAM" and "10MB of main memory space". At the same time, it has to be sufficiently flexible to accommodate a wide range of different resources while avoiding the use of a central taxonomy.

In peer-to-peer systems, where no coordination of resource descriptions takes place, multiple instances of the same file under different names do exist and significantly complicate the resource discovery and selection process.

Most distributed deployment platforms either do not tackle the naming coordination problem, or address it by allowing only specific resources to be declared, usually defined in a static list. Globus' GRAM protocol only allows describing two types of resources: CPU time and memory. The General-purpose Architecture for Reservations (GARA) [FKL+99] extension for Globus allows describing network bandwidth, but more advanced types of resources cannot be represented.

This limits its applicability to general-purpose global public computing, as it does not allow for representing and advertising other important resources, such as IP addresses and ports, nor does it facilitate extensibility by allowing new types of resources to be defined. This also prevents describing and sharing exotic resources that some servers may comprise. Global public computing needs to provide mechanisms for *resource description coordination*, which will achieve consistency of naming common resources and still allow for uncommon resources to be represented.

While distributed ontologies [MMS03] can be used, it is necessary that no global agreement is required — to allow servers to independently define exotic resource types or pricing schemes. At the same time, a simpler, more lightweight scheme than ontologies may be adequate, as resource categorisation does not create the deep, complex hierarchies present in the Semantic Web [BHL01].

Resource management. Although PlanetLab provides some degree of isolation of CPU time and network bandwidth between the different services by running them in sandboxed Virtual Server environments, it does not facilitate resource reservations — both in higher-level tools for describing resources, requests, and reservations, and in underlying isolation mechanisms to enforce them.

GRAM, Globus' resource allocation module, does not provide advance reservation functionality either; even if a Globus node has the mechanisms to provide QoS guarantees — for instance, a QoS-enabled CPU scheduler — no higher-level support is provided by GRAM. Globus can be extended using GARA to support limited reservation functionality for processing time and network bandwidth, but not more advanced types of resource reservations.

Condor does not provide a facility for reserving resources in advance of service deployment. However, Condor-G, the job submission portal that allows the deployment of Condor jobs on Globus Grids, supports this kind of functionality to the same extent as Globus' GARA.

All distributed deployment platforms either do not provide a facility to support advance reservations at all, or provide very limited support. This has often not been the result of an absence of mechanisms to enforce reservations and limitations — e.g. operating system schedulers, Virtual Machine Monitors, resource-managed JVMs — but rather of an absence of higher-level facilities for expressing how allocations are to be made. Furthermore, a fine-grained resource usage accounting facility for charging for resource usage is not provided.

Global public computing needs comprehensive *resource management* solutions to allow resource reservation, isolation and protection. *Accounting* facility is required to allow fine-grained control and charging. While these mechanisms need to be provided by the platform, their use must not be compulsory; in some cases, a best-effort service may be adequate.

Policy-based management. Allowing users to manage resources on servers by defining high-level *policies*[10] provides convenient abstractions; users and operators can administer resources by describing simple, general goals and restrictions — for instance, "limit use of CPU by remote users to X%" — instead of directly communicating with low-level mechanisms — such as CPU schedulers — to modify resource limitations for individual tasks.

Policy languages have been devised in other fields, such as role-based access control [LPL+03] — explored in more detail in Chapter 4. A language for specifying authorisation and quality of service policies for Web Services is under consideration at the time of writing [And04]. In the past, policy languages have been

[10] "A high-level overall plan embracing the general goals and acceptable procedures", from Merriam-Webster On-line Dictionary.

used for flexible control of programmable networks [MZE02] and differentiated services networks [LLS02].

PlanetLab aims to allow infrastructure services to deploy simple restrictions on access of resources by particular users; interfaces are provided to `limit` and `unlimit` access on a specific resource. These restrictions are defined using `rspecs` data structures, as described in Section 2.4.4. Individual node managers' ability to influence how resources are apportioned on their machines is very limited.

For this mechanism to be applicable in global-scale federated settings, improvements need to be made. The `rspecs` structures allow exclusively infrastructure services to declare limitations selected from a static, predefined list. While the exhaustive list of supported resource restrictions may be adequate for the PlanetLab setting, where machines are relatively *homogeneous*, it would not work as well in a global-scale heterogeneous environment with diverse resources and policy requirements.

Condor allows resource owners to define simple resource allocation policies, but these are implicitly interleaved with "unwritten" policies in the matchmaker; the algorithm that the matchmaker is running can essentially *override* policies included in the `classads`.

In peer-to-peer systems, a peer's owner often has some limited means to restrict the number of simultaneous downloads, or the total bandwidth that they consume. No mechanisms for more fine-grained control of resources, such as making different parts of a peer's bandwidth available to different peers or peer groups, are usually provided. Supporting such mechanisms, however, is harder than in public computing, because of the anonymity requirements of users.

Existing distributed deployment systems offer limited or zero support for *policy-based management*. Policy languages allow only a few types of restrictions to be expressed, and decisions on resource allocation are sometimes based in static, hard-coded policies, or out-of-band agreements. In global public computing, a flexible and open policy-based management facility is needed.

Convenient management. In the Condor, Globus, and PlanetLab resource management schemes, there is no mechanism for grouping users; the users on which policy-based restrictions apply need to be enumerated, which is rather impractical — especially if it is not the identities of users that resource management policies are based on but rather their attributes.

Additionally, there is no way of quantifying access to resources; while access to a resource can be controlled, it is not possible to specify how much of a resource a user or user group should be given. Furthermore, in Condor, as the granularity of the current `classads` and matchmaking scheme is per entity — machine — rather than per resource, a single policy needs to be applied to all resources of a machine; there is no clear way to apportion different resources in different ways.

Global public computing needs to make resource management *convenient*, *flexible*, and *comprehensible* for server operators and other stakeholders, to support the goal of maintaining a low cost of entry in terms of money and effort.

Local control. Another important limitation of the Condor scheme is that resource allocation policies have to be placed in `classads`, along with resource representations. Policy evaluation is merged with resource discovery, as it is in the matchmaker that the ranking and constraints in resource representations are checked and evaluated against resource requests. This design decision ultimately allows the central matchmaker to decide to what extent policies are taken into account, and requires that policies are known before resources are advertised to the matchmaker. It also makes updating submitted resource descriptions' policies expensive, as classads advertised in the remote matchmaker need to be changed.

Other systems use super-schedulers or other central components that claim control of the deployment infrastructure and attempt to enforce decisions on resource allocation and management to individual servers [BM02, MBHJ98]. Globus Grids can employ super-schedulers such as Nimrod/G [BAG00], in the form of resource brokers. In the GGF proposals, when individual server owners attempt to obtain control over their resources by failing to comply with the Grid infrastructural authority's global policies they are checked and possibly penalised or ultimately removed from the Grid.

Ensuring that servers delegate full control over their resources to a central super-scheduler is technically complicated; as server owners have exclusive physical console access to their servers, it ultimately requires signed operating systems [Mic03] or tamper-proof hardware [Ken81]. It is also intuitively unrealistic to expect that all individual server owners will be willing to fully resign control over their resources to a third party in a non-cooperative environment.

Global public computing needs to allow nodes to retain direct *local control* of their resources. Allowing evaluation to take place on the servers themselves also has potential scalability and performance benefits, due to better load distribution.

Federated and overlapping policies. Creating a PlanetLab slice involves both global and local admission control decisions; service managers and resource brokers agree to satisfy users' requests by creating slices, while at the same time individual node managers are able to decide whether to honour tickets. For global infrastructure services, it is necessary to ensure that slices running valued or other infrastructure services receive sufficient resources, and that misbehaving or greedy services are restricted — particularly since users are not charged for resource consumption on servers. Local components define how resources on each node are apportioned between the different services, for instance, how many resources are reserved for non-PlanetLab use.

In PlanetLab, only infrastructure services are explicitly allowed to influence resource allocation; other stakeholders, such as network administrators, ISPs, or even node managers themselves, are ignored. While node managers can indeed affect those policies, as they have ultimate control over the nodes, for instance by refusing to honour a ticket, this process is implicit and not a result of a specific policy defined by the managers. As explicit policies are effectively an exclusive privilege of the infrastructure services, no mechanism for resolving overlapping policies is provided. The need for allowing node owners, administrators, and all other stakeholders to define their resource allocation policies independently has been identified in several PlanetLab documents [PCAR02, PV02, CCR+03].

In PlanetLab, parties that donate nodes may wish to impose specialised usage restrictions on the equipment, including limits on slice behaviour or blacklists. Node managers may enforce temporary resource restrictions, such as disciplinary actions, to restrict the activity of misbehaving users. Because of the fact that PlanetLab resources are provided for free, and therefore are prone to abuse, other parties in the same organisational domain as a PlanetLab node may wish to enforce usage limitations; for instance, ISPs or local network administrators may need to enforce tighter control on international network traffic and the associated charges. This gives rise to the need for a flexible system to allow expressing and applying *federated resource allocation policies* defined by multiple entities. Similar requirements are found in Grid computing infrastructures.

Another factor making PlanetLab a particularly interesting resource management problem is that the administrative control over the nodes is separate from the institutions at which the nodes are hosted; while each node's owner is the institution providing the node, the administration of nodes is a responsibility of the PlanetLab infrastructural authority. In federated systems, *overlaps* between goals and policies that node owners and administrators express are common.

The `classads` scheme of Condor provides basic functionality in terms of representing resources and simple resource management policies, and matching resources with requests. No support for federated and potentially overlapping policies is provided. As all policies are evaluated at the central matchmaker, introducing a module for resolving overlaps would be reasonably straightforward; essentially it would only require modifying the matchmaking algorithm. However, federated policies cannot be supported, as there is only a single, globally-valid, applicable policy for each resource: the one defined in its `classad`.

In Globus, policy statements are separate entities, and are not carried in the resource descriptions, which simplifies supporting dynamic and federated policies. A request is evaluated against global and local policies after it is placed, instead of at the matchmaking stage. Policy evaluation is clearly separated from resource discovery. Limited support for federation of policies has been proposed in the form of local and global policies [KWL+03]. However, this does not accommodate cases where entities other than the machine's direct owner or the infrastructural or organisational authority need to influence the way resources are apportioned.

The scheme for supporting resource allocation policies proposed by the Global Grid Forum's Policy research group exhibits significant limitations [WS03, Pul03]. As the current design only supports globally valid policies, federation of policies is not supported. Furthermore, it is unrealistic to expect that allowing only infrastructure services to define policies will be adequate for general-purpose global public computing infrastructures.

The possibility of overlapping rules is recognised by the Policy research group. Overlaps are to be detected in a two-stage process; first, static overlaps are detected off-line — for instance if there are two rules restricting access to the same resource by the same user. Other rule overlaps that happen dynamically — for example the ones that depend on dynamic properties of the user requesting resources — can be detected by the policy consumer before instructing the policy target to apply the policies. No mechanisms are suggested, however, for resolving these overlaps.

As discussed in Chapter 4, a flexible and expressive scheme is needed in global public computing. Ultimate control is bound to be given to individual nodes, and mechanisms to allow the representation of potentially *overlapping interests* and *federated control* are necessary. A generic resource management framework could also be applicable to Grids, PlanetLab, and other deployment infrastructures.

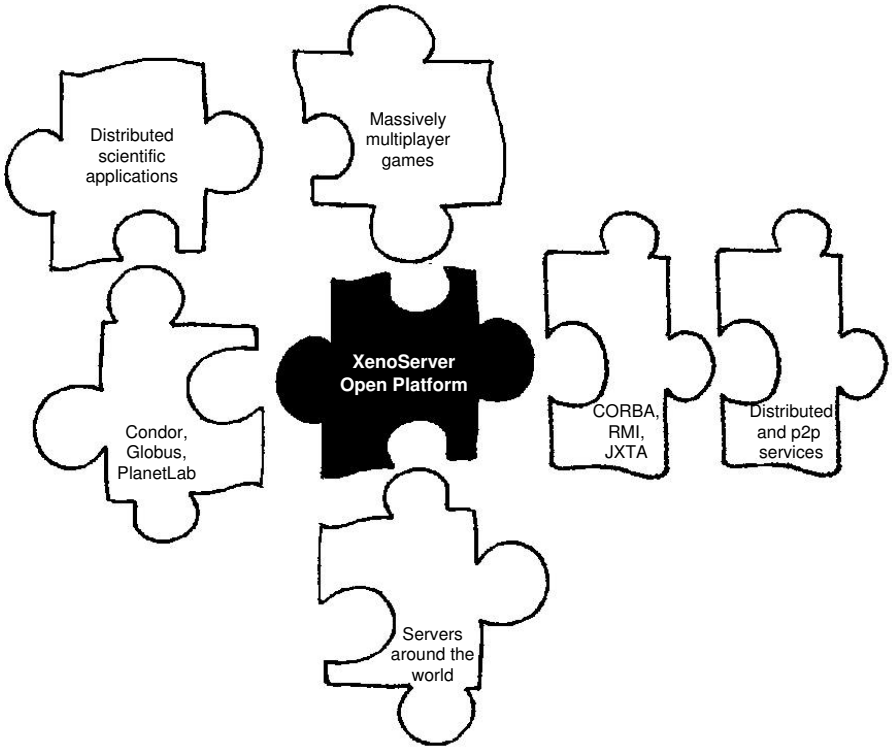

Figure 2.4: The XenoServer platform as a generic mechanism for acquiring resources on machines scattered around the world

2.5 Global public computing

Grids and other distributed deployment platforms that have been developed have been successful in addressing the needs of the individual and closed user communities they have aimed to serve. For global-scale public computing, however, most of the problems discussed previously remain open and challenging. An infrastructure is required to put the pieces together and provide resources on servers around the world to distributed services and deployment platforms, as shown in Figure 2.4. In the rest of this document, I propose a general-purpose global public computing platform, comprising generic reusable mechanisms for addressing common problems, including the ones described in the previous section.

The XenoServer Open Platform, an implementation of the global public computing vision, is a self-contained *service deployment* infrastructure. It provides a

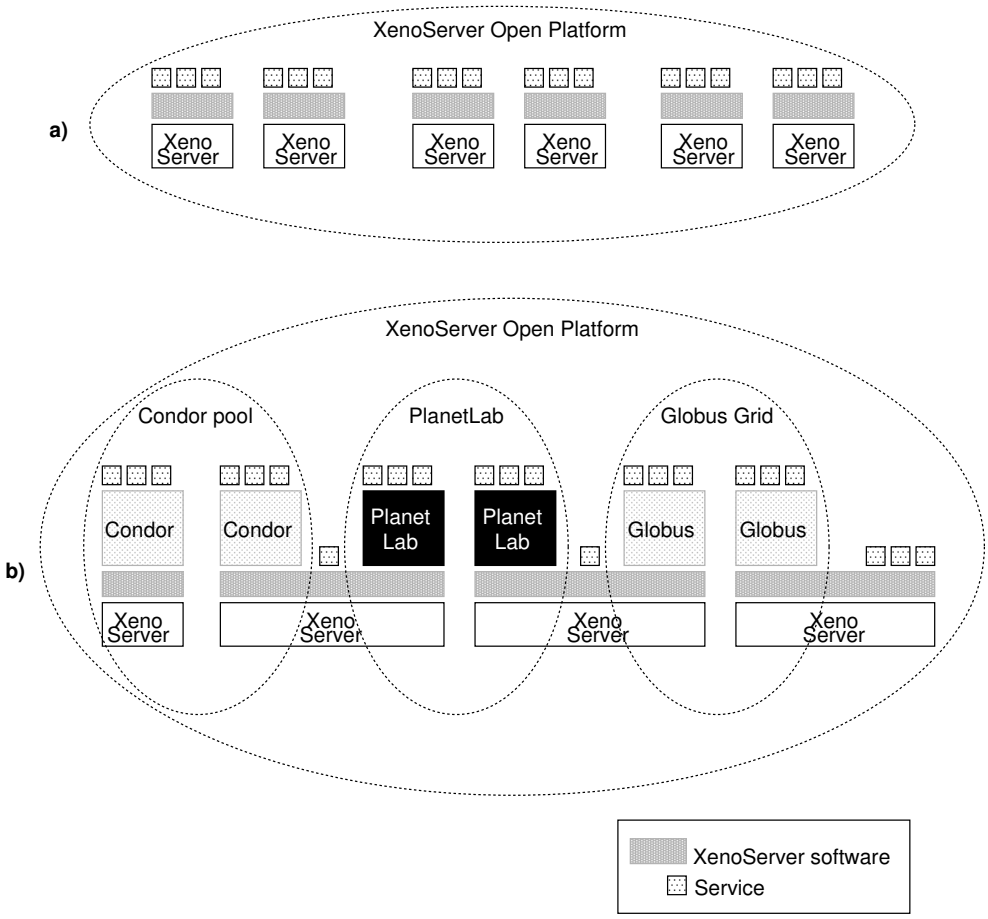

Figure 2.5: The XenoServer platform (a) as an autonomous global-scale service deployment infrastructure and (b) as a common substrate for deploying Grids and existing distributed computing systems

comprehensive range of services required to roll out global-scale services, such as server discovery and selection, convenient policy-based management, federation of control, service deployment at a low cost, fine-grained accounting, billing, and charging. Users' services can be deployed directly on XenoServers — as shown in Figure 2.5a.

At the same time, the platform is providing a *common substrate* for existing distributed deployment infrastructures. CORBA and Java RMI middleware, as well as Grids and other distributed computing infrastructures, can themselves be deployed as services on XenoServers. By doing so, they can take advantage of the resource discovery, global-scale resource acquisition, policy-based management,

protection, and the accounting and charging mechanisms provided[11]. Existing Grid services can then run on Grids, as they normally would — as shown in Figure 2.5b.

Essentially, the XenoServer platform allows users — or higher-level services, such as Grids — to obtain protected and isolated computing resources on a number of participating machines. Any code, out-of-the-box, trusted or not, can be executed; charges are made according to the amount of resources consumed.

[11]PlanetLab is, at the time of writing, working towards using the XenoServer platform as its resource acquisition and protection substrate.

Chapter 3

The XenoServer Open Platform

The aim of the global public computing vision is to allow non-cooperative and competing members of the public to dynamically locate, select, and obtain computing resources on servers around the world, run any applications, and pay for the resources that their applications consume. As we have seen in the previous chapter, none of the conventional distributed computing systems and deployment platforms are sufficient.

The XenoServer Open Platform [HHKP03, KS03] is the product of the XenoServers project [RPM+99, FHH+03], building a global public infrastructure for service deployment. This chapter describes the design of the platform, focusing on the operations supported and the control plane that facilitates the coordination of these operations. In particular, I describe the *entities* that exist, the *functionality* that each entity is expected to carry out, and the *interfaces* and *interactions* between these entities. Details about the internal architecture of the entities — in other words, *how* they deliver the expected functionality — are provided in Chapter 5.

3.1 Overview

This section describes the main components of the distributed XenoServer Open Platform for global public computing, the functionality that each one delivers, and the interactions between them to facilitate service deployment. Figure 3.1 provides an overview of the system; Section 3.2 analyses the system's operation in detail.

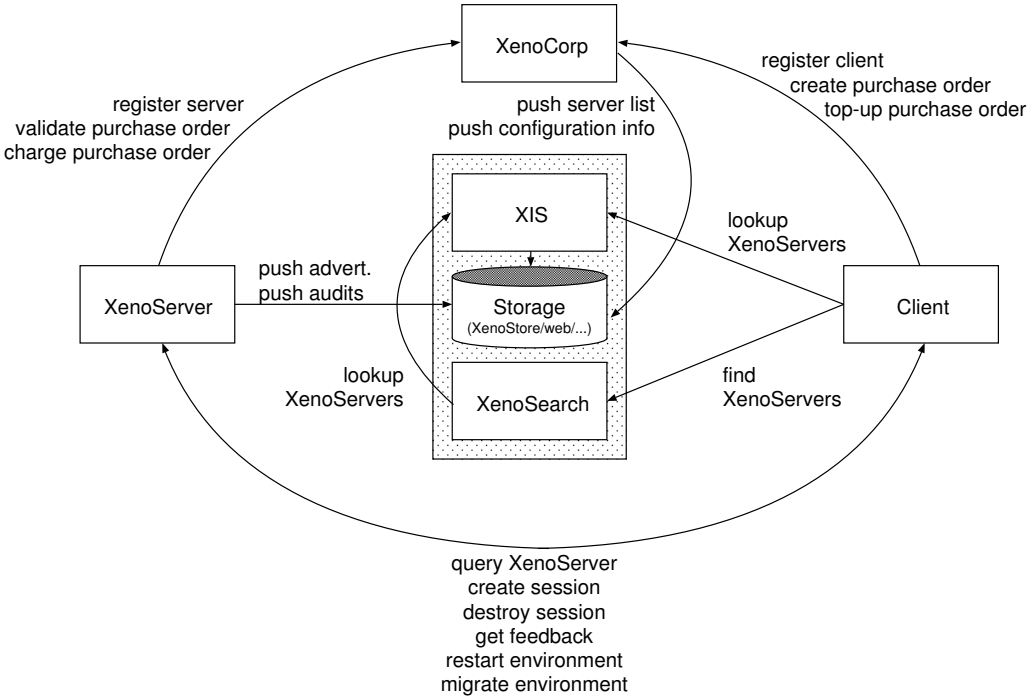

Figure 3.1: Overview of entities and interactions in the XenoServer Open Platform

The entity on the left in Figure 3.1, called a *XenoServer*, is a server that undertakes the safe execution of client tasks in exchange for monetary rewards. A XenoServer's high-level structure is shown in Figure 3.2. Tasks are hosted in *execution environments*; an environment encompasses a set of reserved resources, and hosts one or more tasks belonging to the same service. Resource isolation and protection are enforced between the execution environments, such that several can coexist without being able to adversely affect one another. A privileged execution environment contains the software required for participation in the XenoServer platform, allowing the creation and management of other (client) execution environments on the XenoServer.

A *client* is an entity that deploys services on XenoServers and pays for the resources its services consume. Clients need to be able to locate suitable servers — after describing what "suitable" means in each case, negotiate with XenoServers to reserve the desired resources, and deploy their tasks. Clients expect to receive feedback from the servers regarding the progress of service deployment, as well as information about resource usage by their services and the associated costs. They may also perform management operations on the deployed services, such as stopping and restarting a service, or migrating it to a different XenoServer.

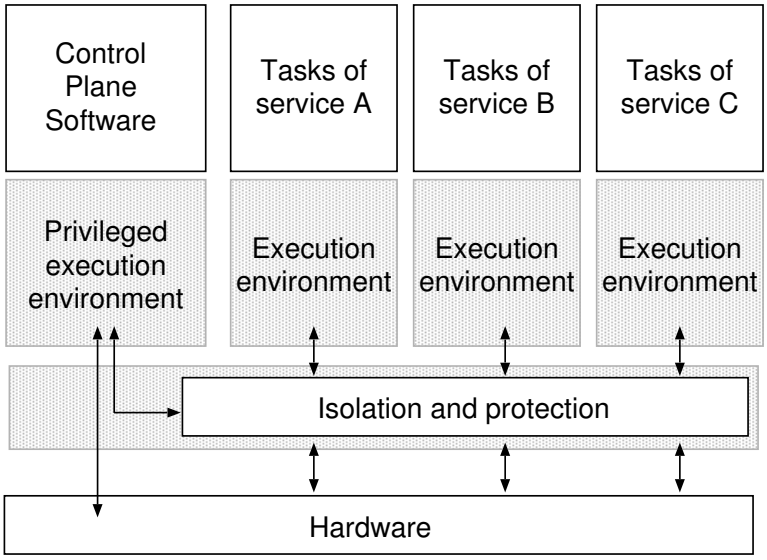

Figure 3.2: Abstract view of a XenoServer's design

The entity at the top in Figure 3.1 is *XenoCorp*, the trusted third party between clients and XenoServers. Its existence is necessary in an inherently untrusted and uncooperative environment, like the one anticipated for the XenoServer platform; as clients and XenoServers do not initially know or trust each other, a trusted broker is required to "guarantee" that XenoServers provide the expected service and clients pay for the charges incurred.

XenoCorp issues authentication credentials for clients and XenoServers when they join the platform, stores details about servers' ownership and administration, provides configuration information and handles charging and payments. Although logically central, XenoCorp may be implemented in a replicated and distributed fashion for fault tolerance and scalability.

The shaded box in the middle of Figure 3.1 represents extensions to this core architecture, comprising additional services provided by third parties. Clients are responsible for choosing which of these services, if any, they use.

The *XenoServer Information Service* (XIS) is a service that helps the process of server selection. Servers periodically advertise their resource availability and the XIS provides the basic functionality required for clients to perform simple searches through those advertisements, in order to locate a number of servers that are suitable for hosting a particular service. Its operation is analogous to that of Globus' Metacomputing Directory Service, described in Section 2.4.1.

XenoSearch builds on the XIS to provide advanced, specialised search functionality, such as searching for a set of servers that minimises the total cost of deploying a particular service or the total round-trip time between the servers and a given set of clients. The use of the XIS is not mandatory, but simplifies the development of XenoSearch services, as it avoids the need for each XenoSearch to communicate with each XenoServer directly. It also allows for easier updates or additions of basic, common search algorithms used by many XenoSearch services.

For service deployment convenience and efficiency, universally-accessible *storage* can be used to allow clients and XenoServers to store and retrieve files. This simplifies parallel service deployment, as it removes the need for clients to manually transfer all the data required for deployment to all XenoServers involved in the process. A detailed discussion about the benefits of using external storage services is provided in 5.2.

Storage services are not part of the core infrastructure of the platform, and may be provided by trusted — by XenoServers and clients — third-party organisations or individuals. These services are termed *XenoStore* services. As described in 5.2.4, XenoServers and clients may decide not to use one of the trusted XenoStore services, but instead run and administer their own, untrusted, independent storage services, such as private NFS [NFS89] or web servers. The platform allows this; the only requirement is that storage locations are made globally-accessible to authorised clients and servers for reading and/or writing.

Analogies can be drawn between the XenoServer platform and aspects of every-day life. XenoCorp's operation is similar to that of VISA, removing the need for direct trust between merchants and customers. XenoServers are merchants; they provide resources in exchange for money. The XIS is analogous to the yellow pages or on-line shop directories, as it contains a structured list of merchants and their capabilities, and provides basic indexing functionality. XenoSearch is similar to high-level searching services, such as on-line search engines, supporting sophisticated and multidimensional merchant discovery.

Note that designing the XenoServer platform and defining the interfaces and interactions between its components effectively determines the *rules of the game*. Previous research on *mechanism design* [HL73] has pointed out the relationship between design decisions made and expectations about the form of the market in which the system operates. In one example, the use of pricing as a mechanism for achieving quality of service [RBTD99, BDH$^+$03] has been investigated, shifting responsibility for accounting and billing to customer systems.

3.2 Operation

The general operation of the XenoServer Open Platform consists of four successive stages, which will be analysed in detail in the following sections. First, clients and XenoServers need to *register* with the platform, in order to be able to participate and trade resources for money.

Then, servers *advertise* themselves and clients *select* the servers on which their services are to be deployed. To do so, they may use the server discovery and selection functionality provided, or they may directly select servers that are known or trusted by them.

Once the servers are selected, clients can proceed with service *deployment*. Clients submit the deployment specifications of their tasks to each one of the selected servers. Tasks may or may not be accepted for hosting according to the local admission control decisions, based on resource requirement, availability, and resource management policies. If accepted, tasks are launched in execution environments on the servers.

After a service is started up in an execution environment, further *management* actions may be taken to, for example, stop, restart, or migrate execution environments to other XenoServers. Servers account for resources consumed and claim payment from XenoCorp.

3.2.1 Registration

Registration is the process of establishing an identity and obtaining credentials allowing participation in the platform. Both XenoServers and clients must be registered with a XenoCorp before they can host or deploy tasks. Figure 3.3 shows the entities that participate in registration operations and the interactions between them.

XenoCorp is the trusted third party that handles registration, authentication, charging and payments. It is discovered using external mechanisms — such as advertising or word-of-mouth — or by accessing a list of available XenoCorp services found at a globally accessible location, such as a well-known web site — operation 1 in Figure 3.3. The coexistence of multiple XenoCorps is discussed in Section 3.4.

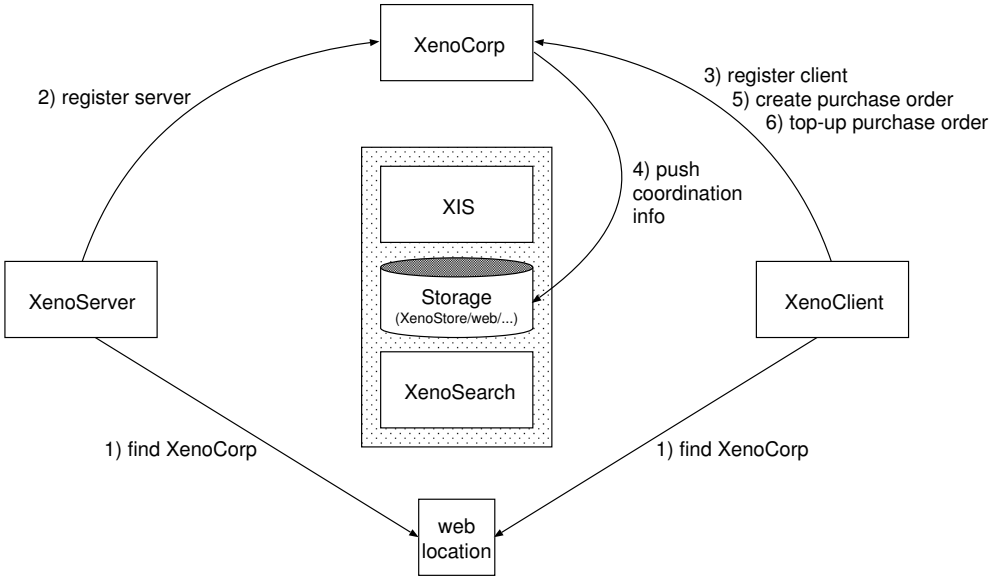

Figure 3.3: Registration of XenoServers and clients

XenoServers must be registered with a XenoCorp, in order to be eligible to claim payment for hosting jobs. Registration of XenoServers works as follows. After selecting a XenoCorp, a XenoServer proceeds to provide information about itself. It submits information about its ownership and administration, details about how payments for resource consumption are to be carried out — for instance, credit card or bank account information. It also submits a URL where its server advertisement describing current status and resource availability is to be found, and a URL where configuration and coordination information for the server is to be stored — operation 2 in Figure 3.3.

XenoCorp decides whether to admit the servers to the platform and endorse their credentials. XenoCorp may at this point also explain its policy to the server and require that the XenoServers' operators enter into a *contractual relationship* with it; this may, for instance, require that servers accept XenoCorp's payments policy — paying each XenoServer according to resource consumption, at a flat rate, or by proportion of XenoCorp's profit — or mandate that servers agree to correctly host the jobs placed on them. They may also need to agree to accept other aspects of XenoCorp's charging policy; for instance, the policies that different XenoCorps may have for dealing with resources reserved but not consumed, as discussed in Section 3.4.

Clients register with a XenoCorp in order to be able to deploy their tasks on XenoServers and to set up an account for the charges incurred — operation 3 in Figure 3.3. Apart from contact information and address, a client must also present some means of settling these charges. For example, in a public wide-area deployment, by providing a credit card number to bill, or a valid bank account for direct debit payments[1].

For ease of server start-up and configuration, and service deployment, as well as for achieving uniform service, XenoCorp provides *configuration* and *coordination information* to the XenoServers at registration time and at infrequent intervals after that. The coordination information includes common resource kinds and pricing units, as described in the next chapter. This information is pushed to locations specified by each server, such as web or XenoStore locations for which authenticated XenoCorps have write permissions and authenticated XenoServers have read permissions — operation 4 in Figure 3.3.

The set of servers and clients cooperating with a particular XenoCorp, including XenoCorp itself, forms this XenoCorp's *domain*. As servers and clients can cooperate with more than one XenoCorp, there is no architectural barrier preventing different domains from overlapping.

Registration is an *infrequent* operation; it is undertaken when a company launches a new XenoServer or when a client wishes to use the platform for the first time — or cooperate with a particular XenoCorp for the first time; thus, the involvement of a potentially central component, such as XenoCorp, does not prevent the incremental scalability of the XenoServer platform, as shown in Chapter 6.

Purchase orders. A purchase order represents a client's commitment to reserving a particular monetary amount for funding resource consumption on XenoServers. This identifies the amount to be reserved, and may contain limits imposed by the XenoCorp or constraints made by the client such as on the type of execution environments that may be requested, the XenoServers on which the purchase order may be spent, and the frequency at which the purchase order is to be charged for resource consumption.

[1]Note that associating users of the XenoServer platform with credit card numbers or bank accounts binds them to a *semi-permanent identity*; while a person may have several credit cards, she cannot easily create fresh ones indefinitely. This is important for avoiding Sybil attacks [Dou02].

The *creation* of a purchase order — operation 5 in Figure 3.3 — involves XenoCorp checking the credit-worthiness of the client, ring-fencing the portion issued as a purchase order from the client's credit card, and endorsing the order with restrictions that must be met in order for it to honour payment — for instance that the purchase order must be properly *validated* before the XenoServer selected to run the tasks starts work.

Each purchase order is associated with a *balance*, representing the funds that the order contains, and an *available balance*, denoting the funds that are not currently reserved for funding resource consumption on another server. *Validation* of a purchase order checks if the order exists, if its balance is higher than a certain available amount, and subsequently reserves that amount by deducting that from the available balance.

Clients may add funds in existing purchase orders at any point, by requesting a *top-up* — operation 6 in Figure 3.3. This deducts a particular amount from the client's credit card or bank account.

As purchase orders are signed by XenoCorp, they are *unforgeable*; given that all components in the platform have a copy of the public keys of all XenoCorps with which they cooperate, it is straightforward to determine whether a purchase order is genuine.

3.2.2 Server selection

The process of choosing the XenoServers to be used for the deployment of a distributed service is termed *server selection*. This can be carried out directly, if clients happen to know which XenoServers are suitable for their requirements.

However, in most cases *server discovery* needs to be carried out. This allows competing XenoServers to publicise their capabilities, available resources and pricing schemes, and clients to find servers that match their requirements. Figure 3.4 shows the entities and interactions involved in the server discovery process.

Prior to discovery, resources need to be *named* and *described* by individual XenoServers. The next chapter analyses the proposed resource description framework and presents mechanisms for representing resources and pricing units. Then, XenoServers *advertise* resource availability and clients *search* through the advertisements.

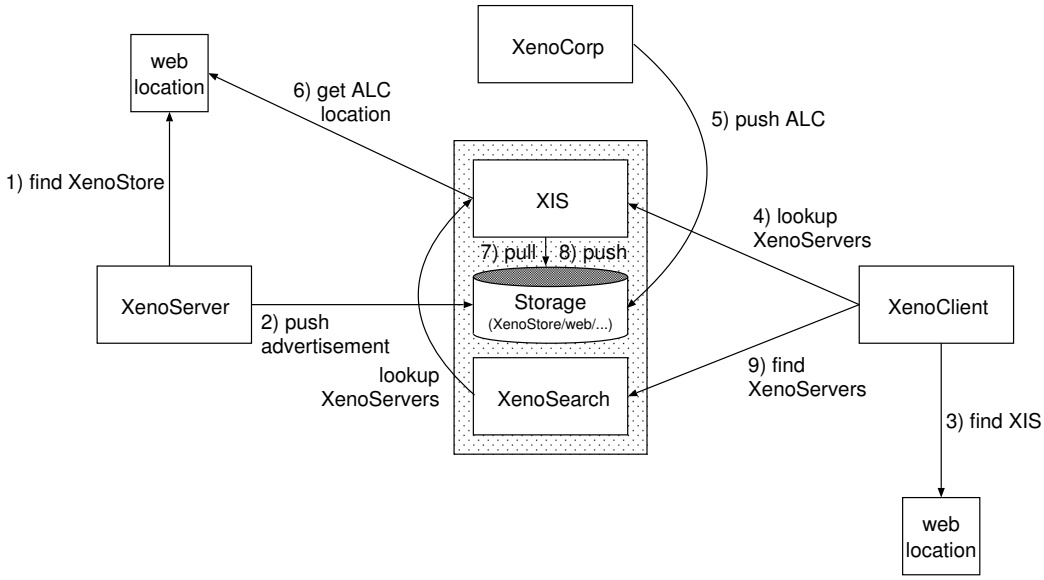

Figure 3.4: Advertisement and discovery of resources

The platform follows the *end-to-end principle*; rather than attempting to have "the system" infer and optimise for the user's goals, clients are directly responsible for selecting servers to use based on their experiences, requirements, preferences, and their willingness to pay for resources.

3.2.2.1 Advertisement

Once a server has represented its available resources, it makes this information available to users who may be interested in purchasing some of its resources. To achieve that, each server periodically produces a server *advertisement*, which is a digest of its current status and resource availability. It includes information about the server's location, ownership, administration, status and average load, as well as a list of the currently available resources and how these are priced. Advertisements are *signed* and *timestamped* for non-repudiation [DH76].

The advertisements that XenoServers periodically produce are stored in globally readable locations. Such locations may be, for example, on web servers running on the advertising XenoServers themselves or elsewhere, or in storage XenoServers obtain in one of the available XenoStore services. Such services can be discovered at server start-up by accessing a list of available services found at a globally accessible location, such as a web site — operation 1 in Figure 3.4.

Then XenoServers independently advertise their resource availability by periodically pushing their advertisements to locations of their choice — operation 2 in Figure 3.4.

3.2.2.2 Discovery

The *XenoServer Information Service* (XIS) indexes advertisements and provides simple lookup functionality. The operation of the XIS is similar to the one of yellow pages; it periodically collects and stores a number of independently produced advertisements in a structured manner, in order to make searching convenient for clients. For more advanced search operations, including multi-dimensional searching, *XenoSearch* can be used.

XenoServer Information Service. Clients can discover the XIS by reading a list of available services advertised on the web — operation 3 in Figure 3.4. The XIS exports interfaces for lookup to the clients, providing basic search functionality that allows searching for advertisements that present values inside a predefined range for a specific token — operation 4 in Figure 3.4. For example, the following query returns all servers connected to the network 128.232.0.0/16.

```
{
  Token    = IPAddress;
  MinValue = 128.232.0.0;
  MaxValue = 128.232.255.255;
}
```

In more detail, the operation of the XIS is as follows. First, at registration time, each XenoServer provides to XenoCorp a URL pointing to the storage location — XenoStore, web, or any other type — where its server advertisements are to be stored. Using this information, XenoCorp periodically produces the *Advertisement Locations Catalogue* (ALC); this is a list containing URLs to the locations where all registered servers' advertisements are to be found. The ALC is then itself pushed to one or more globally-readable storage locations, such as web or XenoStore locations — operation 5 in Figure 3.4. The XIS obtains the URLs of these locations from a well-known web server — operation 6 in Figure 3.4.

The XIS reads the ALC, periodically *pulls* fresh — recently written — advertisements from the locations listed in the ALC — operation 7 in Figure 3.4, and

stores them in a structured, searchable manner — operation 8 in Figure 3.4. Older advertisements are ignored, as they represent servers likely to be down, thus not pushing fresh advertisements to their storage locations. To prevent repudiation, advertisements not properly signed by the advertisers are also ignored.

XenoSearch. Although there is nothing to stop users from using the XIS to select appropriate servers, or from contacting prospective servers directly without introduction, it is anticipated that many will make use of one of a number of available *XenoSearch* resource discovery systems — operation 9 in Figure 3.4. These support server discovery, receiving sophisticated specifications of user and task requirements and using search algorithms to identify a number of suitable servers.

XenoSearch queries are at a much higher level than those supported by the XIS, for instance corresponding to "find a server for a networked game of Quake that is suitable for users A, B and C", "find a server that will minimise the total maximum round-trip time between the server and a given set of clients", or "find a group of servers to minimise the total cost of execution of a particular distributed service".

The way such search requirements are described depends on the implementation of individual XenoSearch services. Languages for description of distributed services and workflows can be employed [CAD+03, AAF+02] to express more complex resource and server requirements. At the same time, the way server advertisements are obtained is also left open, and depends on the implementation of each XenoSearch service — discussed in more detail in Section 3.4.

It is important to note that, since server discovery is not part of the core XenoServer Open Platform's infrastructure, any alternative discovery services — even following different interaction models — may be developed and provided as third-party services alongside XIS and XenoSearch.

3.2.3 Service deployment

Once suitable XenoServers have been selected using the mechanisms described in the previous section, a client can proceed with the *deployment* of a service on those servers, as shown in Figure 3.5. Service deployment is carried out through a direct interaction between the client and the selected XenoServers.

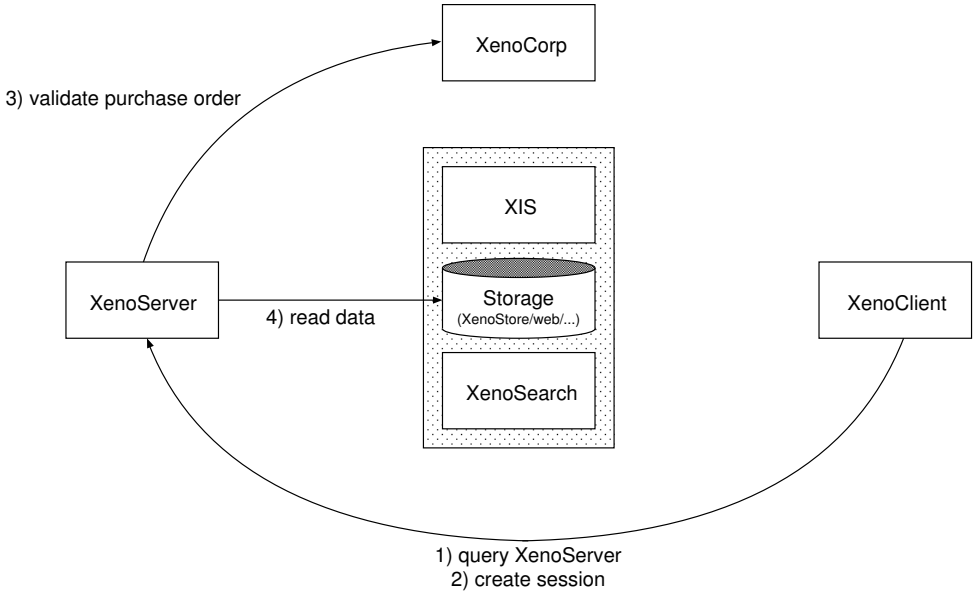

Figure 3.5: Service deployment operations

Before proceeding with service deployment, clients typically *query* the XenoServers on which they wish to deploy services, in order to obtain up-to-date information about resource availability and pricing schemes — operation 1 in Figure 3.5. This is a sensible strategy because a server's resource availability may have changed since its last advertisement was produced.

The client then requests a *session creation* on each of the XenoServers chosen to host the service — operation 2 in Figure 3.5. A session substantiates an agreement between the XenoServer and the client, with the server agreeing to reserve and provide the requested resources and the client promising to pay for the resources his or her service reserves and consumes. Parameters of resource provision, such as an optional expiration time, or charges to be made for resources that are reserved but in the end not consumed are to be agreed upon at this stage. At the time of session creation, resource prices may need to be altered and agreed upon, as the quantity of requested resources may affect the price a server charges per unit, for instance if a very large quantity of a resource is requested.

Clients are *authenticated* before requesting a session creation or any other operation on a XenoServer or XenoCorp. Authentication uses the credentials issued by XenoCorp, is implementation-specific, and takes place ahead of session creation, as described in Section 5.1.3. In the rest of this chapter, all entities are assumed to be authenticated to other entities before interactions take place.

A session creation request has to be accompanied by a valid *purchase order* containing funds adequate to sponsor the envisaged resource consumption, created with a XenoCorp that cooperates with the server. A session creation request also needs to present to the server the *list of resources* required for the service's execution. What kinds of resources, and how much of each resource is required for a service's execution can be manually defined by the user who is launching the service, or automatically estimated [DI89, GMC+01, MC04]. The way resources and resource reservation requests are represented in the prototype implementation is analysed in Chapter 4.

Along with the purchase order and required resources, clients need to pass the *deployment specification* of the service to be launched to the XenoServer at session creation. This includes the type of execution environment needed, which has to be one of the types supported by the server on which the service is to be deployed. Supported environment types may be guest operating systems running on a Virtual Machine Monitor — as in the prototype XenoServer, described in more detail in Section 5.1.1 — or simple UNIX processes, Virtual Servers on Linux environments, or sandboxed JVM environments [BTS+98, BLT98, vD00]. If the guestOS model is used for multiplexing, the deployment specification also needs to describe the execution environment's boot parameters, such as kernel image and root file system to be used.

At this point, the server decides whether or not to allow creation of the requested session and provide the resources required for the service's execution. Session creation and allocation of resources is done in two stages; *reservation* and *claiming*. The main function of the former stage is *admission control*, which may involve checking the *validity* of the purchase order — operation 3 in Figure 3.5 — if necessary, and investigating whether the resource reservation that the client has requested can be met[2].

Whether the validity of the purchase order to be used to fund the session is checked at session creation depends on the arrangement between XenoCorp and the affiliated XenoServers, and on the type of the purchase order. If a purchase order is annotated with the restriction that it may be cashed only on a particular XenoServer, validation is not required at the admission control stage; as XenoCorp reserves the funds required to pay for resource consumption up to the value indicated in the purchase order at order creation, if the order can only

[2]More details about how admission control concludes how to handle a resource reservation, according to current resource availability and resource usage policies defined by the various stakeholders, are given in Chapter 4.

be redeemed on one specific XenoServer then it is guaranteed that the funds are available when charging is to be performed. Multiple usage of the same order on the same server can be locally detected by the server software.

If, however, the order is not associated with a specific server, validation may be required, as other XenoServers may be charging for resource consumption on the same order simultaneously. Validation can be done at session creation, or at other times, according to the type of the purchase order and the agreement between XenoCorp and its affiliated servers — see Section 3.4.

If the admission control decision suggests that the request can be satisfied by the server, or if an alternative reservation suggestion can be made, the outcome of the first stage is a *tentative reservation* of resources that expires after a specific amount of time. A reservation indicates that a particular amount of resources has been ring-fenced and is not considered available on the XenoServer until the reservation expires.

To obtain the set of tentatively reserved resources, a client needs to *claim* the reservation. The result of a successful claim is the launch of a new *execution environment* on the XenoServer — or the resumption of a previously suspended one, which hosts the service to be deployed and encompasses the reserved resources. Resource isolation and protection may be enforced between the environments at a lower layer — as in Figure 3.2. If the reservation is not claimed before it expires, a new reservation needs to be pursued.

Once the reservation is claimed and the execution environment is launched, the client can use any means available in the environment — e.g. SSH [Ylo96], RPC [BN84], or other application-specific service management tools supported by the environment [BBR+97, CKR+01, Tch04] — to connect and launch the service. For convenience and efficiency, the client can use XenoStore to store and access files and data required for launching and running the service — operation 4 in Figure 3.5.

Resource coallocation and negotiation. The two-stage approach employed for resource acquisition allows for the *coallocation* of resources. This ensures that, when necessary, a service can request that a set of required resources is allocated on all or none of the selected XenoServers. Resource coallocation is optional; clients can proceed and claim some or all of their tentative reservations regardless of the failure of allocating resources on some of the selected servers if they so wish.

Supporting resource coallocation mechanisms is important where there are dependencies between resources needed on different machines. Let us consider an example where a task has reserved resources on server A and is waiting for input from another task (belonging to the same service), which is trying to obtain resources on a fully utilised server B. Deadlock situations may be reached, as resources on A are allocated to an inactive task.

Apart from coallocation, the two-stage session creation approach allows for simple *negotiation* to take place between the server and the client. If the server cannot satisfy the original resource reservation request, it can counterpropose a different reservation that may be adequate for the client — see Section 4.3.4.4. Also, it allows the server to modify the pricing schemes to be used if required — for instance, if the client is purchasing a very large quantity of a resource.

3.2.4 Management

Operations that may need to be carried out on a session that is already running are termed *management* operations. A running session comprises an execution environment, hosting the tasks of the service being deployed. Management of a running session can involve control-plane operations on the execution environment itself, or on the service's tasks running in it.

Both the client who is deploying a service and has requested the creation of the session, and the XenoServer that is hosting the service's tasks, may need to perform some form of ongoing management. Clients may need functionality for administering the environment on which their tasks are running, as well as for management of the tasks themselves. XenoServers need functionality to allow them to monitor the running environments and the resources they consume, and at some point claim the corresponding payments. This is shown in Figure 3.6.

Clients. Clients need to perform *service management*, which is application-level administration of the tasks they have launched in the execution environment provided to them at session creation. Such management may include starting or stopping tasks, changing their execution parameters, changing or replacing data or files used by a service, or passing messages to running tasks that cause them to alter their behaviour in some way. It may also include any other types of operations specific to the functions that tasks are carrying out in the context of the distributed service they belong to.

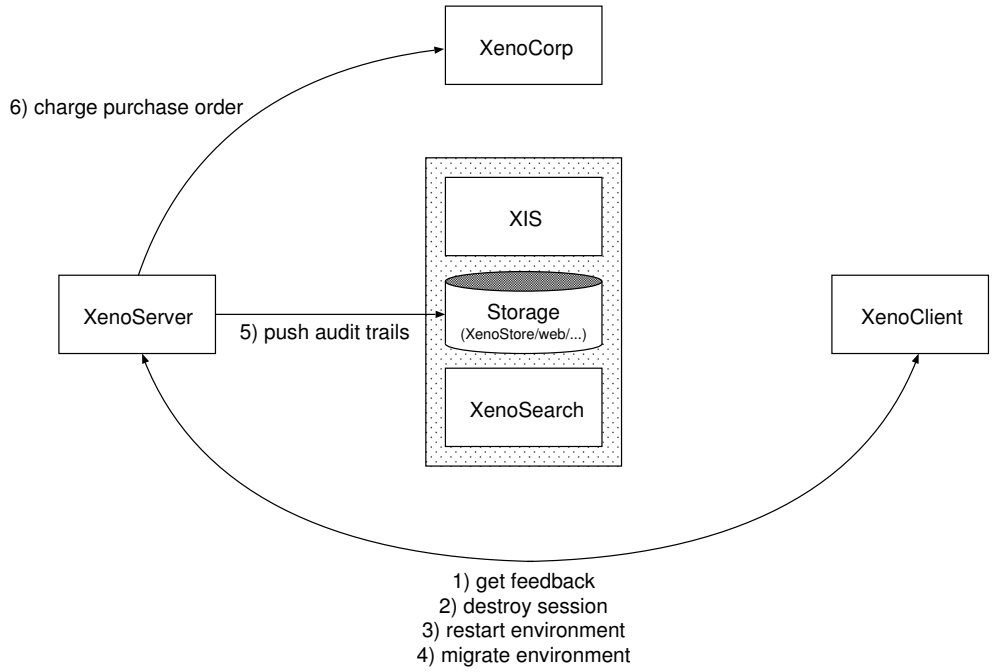

Figure 3.6: Environment management operations

This type of management is *service-specific*; it is expected that clients will perform administration of their tasks in the way that is most convenient for them in each service's case. Clients may use SSH, RPC, or any other protocols supported by the execution environment, to start, stop, and modify the tasks executed. More sophisticated applications that allow advanced management or orchestration of management operations across several servers [CCMN04] may also be deployed as services or tasks, alongside the tasks they are meant to help manage. No support from the XenoServer platform is necessary for application-level service management to be undertaken successfully.

However, clients may also need to undertake management operations on the *execution environment* itself, therefore requiring that the underlying XenoServer platform provides mechanisms to facilitate such operations. The management requests that a client may need to place, as shown in Figure 3.6, are:

- **Get feedback.** It is useful to a client to receive feedback from the XenoServer, regarding the initialisation and operation of an environment — operation 1 in Figure 3.6. This includes the initialisation messages produced as the environment starts, as well as usage statistics and information about resource consumption incurred by the environment and the associated costs.

- **Destroy session.** A client can stop a session at any time — operation 2 in Figure 3.6, typically when the service is finished. This shuts down the corresponding execution environment permanently or temporarily, and terminates the associated resource provision and payment agreement between the client and the XenoServer. If supported by the XenoServer, clients may request that the memory image of the environment to be stopped is saved at a storage location, to allow resumption at a later stage.

- **Restart environment.** In some circumstances a client may need to restart an environment — operation 3 in Figure 3.6. For instance, to boot it using a different operating system kernel — if such environments are supported by the XenoServer — or in cases of environment failure. Restarting an environment does not affect the corresponding session; existing resource allocations remain in place and the new environment provides the same resource availability guarantees as the one that was terminated.

- **Migrate environment.** Where possible — depending on whether the type of the execution environment permits it — the client may request the migration of an environment, along with the tasks running over it, to a different session on another XenoServer — operation 4 in Figure 3.6. Before requesting a migration, the client needs to have successfully created a new "container" session on the server to which the environment is to be migrated. This session does not contain an execution environment, but comprises adequate resources to accommodate the environment to be migrated. Then, the XenoServer on which the environment is running is requested to perform migration of the environment. The environment migrates to the container session and starts running on the target server.

- **Change resource reservations.** A session represents a fixed agreement for resource provisioning by the XenoServer and according payment by the client. While allowing changing the session's parameters — such as reserved resources and pricing schemes — after it has been created would be technically feasible, it would violate the nature of a session as a permanent agreement and complicate its semantics. To alter the reservations made within a session, a client launches a new session comprising the desired resources, uses the environment suspend/resume or live migration mechanisms — if provided by the underlying XenoServer technology — for uninterrupted execution of services, and destroys the old session. This may not involve the user; high-level resource brokers, responsible for long-term management of services, can be employed.

XenoServers. It is necessary for charging and billing, as well as fault tolerance, that the XenoServer control software *monitors* the state of existing environments and the resource consumption associated with each one. Additional management operations that a XenoServer needs to carry out are:

- **Account.** With help from the underlying accounting mechanisms provided — as in Figure 3.2 — XenoServers keep track of the environments that are running and the resources that each one has used. Information available by the underlying layer may be in the form of total resources used by each environment, such as total CPU time and memory.

- **Log and audit.** For security and legal reasons, it is necessary that XenoServers record information about service activity, to be used if a service is engaged in malicious or unlawful actions. Logging and auditing allow the service responsible for such actions to be traced, and the sponsor of the service to be identified. It is also important for resolving incidents of payment contention; clients may dispute resource consumption claims put forward by XenoServers. Audit trails are signed for non-repudiation, and stored in XenoStore or any other storage locations specified by the XenoServers' administrators — operation 5 in Figure 3.6.

- **Charge and bill.** Using the accounting statistics that have been collected, XenoServers request payments from XenoCorp for the resources consumed by the execution environments running — operation 6 in Figure 3.6. Charging may happen *in advance*, in which case charging is based on reservations not consumption, *periodically*, or *at session destruction*. Charging may be made for resource reservations, consumption, or both, depending on the parameters agreed between the servers and XenoCorp at registration.

3.3 Interfaces

In this section I present the interfaces exported by each component and the black-box functionality that each entity delivers when the corresponding interfaces are invoked. As discussed earlier, authentication of entities takes place ahead of the interactions described here; the way it is performed in the prototype platform implementation is described in Section 5.1.3.

3.3.1 XenoCorp

XenoCorp exports the following interfaces for registration of clients and servers, charging, billing, and payments:

1. `register_client(personal_info,address,charging_info,temp_creds)`

   `personal_info:` name, contact details of client
   `address:` address, city, postcode, country where client is located
   `charging_info:` credit card or bank account to be used
   `temp_creds :` credentials that XenoCorp will endorse upon registration

   XenoCorp endorses the client's `temp_creds` and returns unique `credentials` for the client. These are used for authentication of the client to XenoServers and XenoCorp.

2. `register_XenoServer(owner_info,admin_info,payment_info,`
   `adv_loc,config_loc,temp_creds)`

   `owner_info:` details of XenoServer's owner
   `admin_info:` details of XenoServer's administrator
   `payment_info:` credit card or bank account to be used
   `adv_loc:` URL where the server's advertisement is stored
   `config_loc:` URL where the server's configuration is stored
   `temp_creds:` credentials that XenoCorp will endorse upon registration

   XenoCorp endorses the server's `temp_creds` and returns unique `credentials` for it. These are used for authentication of the server to clients and XenoCorp. XenoCorp also pushes configuration and coordination settings to `config_loc`, and adds `adv_loc` to the ALC.

3. `create_purchase_order(amount,<constraints>)`

   `amount:` amount to be reserved
   `constraints:` optional additional restrictions

   Constraints may specify for instance that the order may only be cashed on members of a particular set of XenoServers. Upon successful completion of this operation, XenoCorp returns a new, signed `order` containing the requested amount.

4. `validate_purchase_order(order,amount)`

   `order:` purchase order to be validated
   `amount:` amount the order must contain

   This checks if the specified `amount` is available in the order and reserves — but does not charge — the `amount`. It returns `true` or `false`.

5. `topup_order(order,amount)`

   `order:` purchase order in question
   `amount:` amount that to be added to the order

   This reserves the specified `amount` on the client's credit card or bank account, and adds it to the `order`. It returns `true` or `false`, depending on whether the operation succeeds. If not, an additional `message` is returned explaining the reason.

6. `charge_purchase_order(order,accounting)`

   `order:` purchase order to be charged
   `accounting:` resource usage information

   This calculates the amount to be charged on the specified purchase order (`order`) from the resource usage information provided (`accounting`). Subsequently, it returns `true` or `false`, depending on whether charging the amount on the purchase order succeeds. If not, an additional `message` is returned explaining the reason.

   XenoCorp resolves payment disputes and complaints using out-of-band mechanisms, such as examining purchase orders, audit trails, resource consumption logs, and session creation agreements. Charging involves communication with a credit card system or direct-debit payment infrastructure.

3.3.2 XenoServer

XenoServers export the following interfaces for creating and managing sessions and execution environments:

1. `query_XenoServer()`

   This returns `server_info`, containing information about the server's current status, resource availability, and pricing — effectively, an up-to-date copy of its advertisement.

2. `create_session(order,depl_spec,resources,<res_img>,<parameters>)`

`order:`	purchase order to fund the session
`depl_specs:`	deployment specification
`resources:`	list of resources to be reserved
`res_img:`	optional, file from which to resume memory image
`parameters:`	optional, e.g. expiration time

   This operation ring-fences but does not allocate the requested resources. Upon success, it returns a `rsv_handle` (reservation handle) to be used for referencing the tentative reservation, and a set of `resources` tentatively reserved and the associated prices. This is required as the server may propose an alternative reservation or pricing scheme, if the requested one is not possible — see Section 4.3.4.4.

   For the resources to be allocated, `create_session` has to be followed by a reservation claim `claim_rsv(rsv_handle)`. If the reservation has not expired, this operation launches a new execution environment or resumes one from its saved memory image (`res_img`). it returns a `session_handle`, which is only valid on the server on which the session is deployed and is used for management operations on the session.

   The client also receives a globally-unique, hierarchical `session_id`. This can be used for off-line platform-wide identification of sessions, such as in cases of payment disputes.

3. `get_feedback(session_handle)`

`session_handle:`	session in question

   This operation returns `session_info`, a description of the session's status, total resource usage, and the associated costs to that point.

4. `destroy_session(session_handle,<sus_img>)`

`session_handle:`	session to be terminated
`sus_img:`	optional, file where memory image is to be saved

This causes the execution environment to shut down, and any existing resource reservations to be cancelled. Final charges for resource consumption may be made. If supported by the underlying mechanisms, clients may ask for memory image of the execution environment to be saved at a storage location of their choice (`sus_img`). It returns confirmation of session destruction, and a final set of resource usage information (`accounting`), describing the resources consumed and charges incurred.

5. `restart_env(session_handle,depl_spec)`

`session_handle:`	session containing the environment in question
`depl_specs:`	deployment specification

   This operation invokes lower-level mechanisms to terminate the existing environment associated with the session, launch a new one, and associate it with the same session. It returns `true` or `false` to denote success or failure. On failure, an additional `message` is returned explaining the reason.

6. `migrate_env(session_handle,tgt_server,cont_handle)`

`session_handle:`	session whose environment is to be migrated
`tgt_server:`	target server
`cont_handle:`	container session on the target server

   The environment in question is moved to the target XenoServer and associated with the specified container session there. This operation returns `true` or `false` to denote success or failure. On failure, an additional `message` is returned explaining the reason.

3.3.3 XenoServer Information Service (XIS)

The XenoServer Information Service (XIS) provides basic resource discovery functionality. It collects individual XenoServers' advertisements and allows clients to search through these advertisements, in order to locate a number of servers that are suitable for hosting a particular service.

The interface that XenoServer Information Services export is the following:

1. `lookup_XenoServers(token,min_value,max_value)`

`token:` dimension of search, e.g. network address
`min_value:` minimum value for the token
`max_value:` maximum value for the token

This operation returns a `server_list`, containing the names and addresses of XenoServers that satisfy the above criterion.

3.3.4 XenoSearch

Building on the functionality provided by the XIS is XenoSearch, which supports more complex, multi-dimensional search requests for server discovery. The generic interface that XenoSearch-type entities need to conform to is given here.

1. `find_XenoServers(expression)`

   `expression:` XenoSearch-specific search criteria

   This operation returns `server_list`, containing the names and addresses of XenoServers that satisfy the above criteria.

3.4 Openness

The XenoServer platform can be classed as *open*, based on the *low cost of entry* it presents to users and servers when compared with previously available facilities. This involves a low cost in terms of effort, as existing programming environments and out-of-the-box applications are supported, and mechanisms for easy large-scale service deployment and management are provided. The incurred monetary cost also remains low, due to support that is provided for fine-grained, on-demand resource acquisition over flexible timescales.

The openness of the XenoServer platform is also expressed through its *flexible* and *extensible* design, encouraging the customisation, specialisation, competition, and ultimately evolution of platform components. The system is structured in a way that while the "rules of the game" are defined — which functions each type of entity should undertake and how entities are to interact — it allows for variety and diversity in the *mechanisms* used to carry out these functions, as well as in

the *parameters* of the interactions. Also, the *coexistence* of multiple entities of each kind is envisaged.

The following sections discuss aspects of the open design of the XenoServer platform and present trade-offs that cannot be resolved at design time, simply because no single global optimal solution can be assumed. Like modern market economies, the platform relies on the principle of *disciplined pluralism* [Kay03]. In this book, John Kay observes that *"it is often true that coordination is more effectively achieved through mechanisms of spontaneous order than central direction.....Market economies did not succeed because business people were cleverer than politicians. They succeeded because disciplined pluralism is more innovative and more responsive to customer needs than centralised decision-making"*. Instead of devising an "optimal grand plan" the XenoServer Open Platform is based on many little individual self-interested decisions that weave a net of spontaneous order.

3.4.1 Multiple XenoCorps

The design of the XenoServer Open Platform allows and encourages the coexistence of multiple XenoCorps. While it is possible that even with one XenoCorp the system will be able to cope with high numbers of servers, the fact that multiple XenoCorps may exist if necessary is reassuring for *scalability*; as the number of servers increases, so can the number of XenoCorps. Growth of the platform does not necessarily result in increase in each XenoCorp's workload. At the same time, different XenoCorps compete and provide *diversity of services*, by following different strategies in the level of anonymity and privacy provided, purchase order validation, and charging and payment arrangements.

Here I present some of the alternative approaches that XenoCorps may follow in different settings, and discuss the associated trade-offs that exist. Each XenoCorp informs clients and XenoServers about its *policy* in all matters in detail before registration, and only proceeds with providing them with authentication credentials if they explicitly agree on the terms and conditions for participation in this XenoCorp's domain.

Anonymity. XenoCorps may differ in the degree of *privacy guarantees* they provide to their clients. A reputable XenoCorp, cooperating only with highly respected servers, may require its clients to prove their ability to pay for resource

consumption in advance. An anonymous XenoCorp, cooperating with cheaper low-quality servers, may be willing to take the risk of letting clients deploy services and pay for resource consumption at a later time. XenoCorps whose policy is to require association of clients and XenoServers with real-world identities may be able to provide better guarantees of security and service reliability, as action may be taken against individuals who engage in malicious activities or consistently do not meet the agreed standards of service. On the other hand, XenoCorps that allow clients and XenoServers to register without providing their owners' real-world identity may be popular where anonymity is required; in oppressive regimes, users may trade high standards of service for privacy.

Purchase order validation. Another parameter that can be tuned differently by different XenoCorps is the strategy towards *purchase order validation*. Service deployment on XenoServers is accompanied by purchase orders, which represent client funds reserved to pay for resource consumption on XenoServers. The point in time when XenoServers request XenoCorp's help to check the validity of purchase orders is to be decided by individual XenoCorps on a per purchase order basis, and relates to the estimated degree of risk and expected level of scalability.

Some orders may be validated at the admission control stage, before a session is created on the XenoServer; this eliminates the risk of having a purchase order, which is invalid, non-existent, or does not contain sufficient funds, being used for service deployment on a server. However, this places XenoCorp in the service deployment path. In this scenario, XenoCorp could become a bottleneck; the time required to complete a service deployment operation depends on the time required for purchase order validation, as all deployment requests on all servers incur a validation operation on XenoCorp. As XenoCorp's load is increasing, validation may take ever longer.

One way to deal with the problem is to move validation services out of the XenoCorp server, and replicate them at several network locations. While logically part of XenoCorp, validation components can themselves run on XenoServers, allowing the incremental scalability of the system on demand. Another solution is to implement a fully-distributed XenoCorp; this, however, introduces additional complexity in terms of security and trustworthiness of servers running XenoCorp nodes.

A different approach is to disassociate purchase order validation from service deployment. When a service deployment request arrives on a XenoServer

along with a purchase order, it is initially assumed that the order is valid and the server proceeds with the deployment operations. Purchase orders are then validated periodically, either individually — where the period of validation can be agreed between XenoCorp and the client at purchase order creation — or in batches — where the period is fixed, as a result of agreement between XenoCorp and the affiliated servers. While this does not ease the load of XenoCorp — if the numbers of clients and XenoServers increase, it still needs to handle an ever increasing load, it does speed up service deployment, as requests do not have to wait for validations. This provides better responsiveness and faster deployment, but incurs a risk of invalid purchase orders being used. Other XenoCorps may mandate that all purchase orders are tied to specific XenoServers, to ensure purchase order validity and avoid the validation step altogether.

The choice of approach to be followed is to be made by individual XenoCorps, and depends largely on the parameters of the environment in which different XenoCorps are to operate. In trusted or controlled environments, or to further limit XenoCorp's involvement in service deployment, XenoCorps may agree with their associated servers on a loose purchase order validation model, or on a probabilistic model where only a sample of orders are validated and where perhaps additional charges or penalties are brought to the clients whose orders are found to be invalid — similar to the approach followed for ticket checks in public transport. The proportion of validated orders can adapt to the frequency of offences. The implications of the different validation strategies are examined in Chapter 6.

Charging. Related to this is the point in time when *charging* is to be performed. XenoCorps and their associated servers may agree that charging be carried out at session creation, at session destruction, periodically while the session is running, or after it is finished. Charging at session creation implies charging for resource reservations rather than usage, as no consumption has been made by the service yet. While this assures that no resources are reserved and not paid for, it does not allow charging for "ms of best-effort CPU time"; users who prefer to pay for usage even if no guarantees are to be made may find it impractical. Also, it places charging in the session destruction path and may turn XenoCorp into a bottleneck.

If charging happens at session destruction, complete information of resource usage is available, thus fine-grained charging for consumption is possible. However, this strategy, too, places charging in the session destruction path and may turn XenoCorp into a bottleneck. Periodic charging, individually or in batches,

can help mitigate this effect, as for purchase order validation above. Separating charging components from XenoCorp and dynamically replicating them on XenoServers can provide an implementation-specific way around the problem, but complicates the trust relationships, as discussed earlier.

The extent to which XenoServers affiliated with different XenoCorps charge for resources reserved by a client but not subsequently consumed is again left to XenoCorps to determine. Some XenoCorps may charge for reservation, where resources that are reserved but not consumed are fully paid for by the users, while others may charge for consumption, allowing resources reserved but not consumed to be wasted. Compromise models, where users pay a proportion of the originally agreed price for resources reserved but not consumed — similarly to cancellation of hotel bookings — are also envisaged.

Payments. XenoCorps may also differ in the mechanisms employed for transferring *payments* to XenoServer operators. Bank accounts, credit cards, and untraceable digital cash [CFN90] are only a few of the payment methods that can potentially be used.

At the same time, the way payments are determined may vary among XenoCorps too. In one scenario, clients pay XenoCorp for resource consumption, and XenoCorp transfers payments to XenoServers accordingly, after deducting a proportion of each payment. In another scenario, XenoServers receive payments from XenoCorp not based on resource usage but according to a fixed, pre-agreed scheme — for instance, a flat monthly payment, or a proportion of XenoCorp's operating profit. This approach reduces XenoCorp's initial risk — as its expenses, in terms of payments to XenoServer operators, are guaranteed to be lower than its profit — and provides incentives for new XenoCorps to join the platform. Also, it can provide benefits for XenoServer owners in risky, untrusted environments, as flat monthly payments mask the exposure to fraud by clients not paying for resource consumption, and remove the need for XenoServer operators to ensure clients they choose to sell resources to are trustworthy.

3.4.2 Multiple XenoServers

The presence of large numbers of XenoServers is central to the design of the platform. Apart from multiple XenoServer machines, the platform also supports a wide range of alternative XenoServer designs.

The structure of the developed XenoServer prototype is discussed in detail in Chapter 5. It is based on a low-level Virtual Machine Monitor, which virtualises the physical resources of the machine, apportioning them between the various environments that it hosts, by creating a Virtual Machine for each one.

However, there is no architectural restriction prohibiting the implementation of *different types of XenoServers*, as long as the interfaces exported, described earlier, are preserved. In some cases, clients may wish to use execution environments other than complete operating system instances over a Virtual Machine Monitor. For instance, an alternative XenoServer could be running a resource-managed JVM, deploying services in protected JVM environments; another could be a XenoServer accepting .NET [Par00] components using existing application-server packages. Other implementations may use Virtual Servers [Gel03], User-Mode Linux [Dik01], Virtual PC [Con00], or VMWare [VMW99] for resource isolation.

Openness is also expressed in the way individual XenoServers *describe* their resources and pricing schemes. While a list of supported resources is advertised by XenoCorp to coordinate descriptions and control arbitrary naming of types of resources and pricing units, individual XenoServers are allowed to produce and advertise resource descriptions that are not included in that list. This allows XenoServers to sell resources on potentially less common hardware, and to use exotic or custom pricing units.

This, together with the resource management module described in Chapter 4, provides significant flexibility to servers to manage and sell their physical resources. Servers can sell different parts of the same physical resource in different packages; for instance, a XenoServer may split its CPU in three parts, then sell the first part in a "ms of CPU time per wall-clock second" fashion as an expensive guaranteed-QoS option, the second as "best-effort CPU access" at a lower price, and reserve the third for its owner's use.

The openness of the XenoServer platform allows individual servers to independently set their pricing schemes, but also to use different approaches to set prices on the resources. Some XenoServer owners may decide to manually define the prices themselves, while others may employ a high-level *dynamic pricing* component, perhaps negotiating with adaptation-aware applications and services [NSN+97, KHG00, BKR98], either for automatically adapting to the balance of supply and demand of different resources and increasing profit, or for regulating resource congestion [SM99].

3.4.3 Multiple clients

The client-side software may provide a graphical interface to support convenient service deployment and management by human users. Alternative clients may comprise different user interfaces or may not even be human-operated; autonomous agents may request on-demand resources when the load on a server they monitor increases above a certain threshold. Transient mobile devices may automatically obtain resources on XenoServers around them to maintain uninterrupted network presence. A wide variety of internal designs and user interfaces of client applications can be admitted, given that the standard interfaces for communicating with other entities are used and the general interaction models are maintained.

3.4.4 Multiple XIS and XenoSearch services

There are no architectural restrictions barring the coexistence of multiple XISs, perhaps providing different lookup mechanisms. However, competition in this area may not necessarily be beneficial; as with DNS, while technically possible to have more than one systems, the market is pushing towards a single infrastructure to maximise exposure of advertisements and ensure clients draw on the largest possible information base when performing lookups.

There may be multiple XenoSearch systems, either for simple *competition* — as exists between on-line search engines — or for *specialisation* to particular kinds of user, server or task. The algorithm with which the mapping is performed is entirely dependent on the particular implementation of the XenoSearch mechanisms; different XenoSearch systems may offer different algorithms, tailor-made to meet needs of distinct user communities. The XenoServer team has developed two prototype XenoSearch systems, described in [SH03, SHH04].

The exact format of the parameters to be passed to XenoSearch through the search interface, such as the language to be used for expressing service requirements, can differ between different XenoSearch systems. Different types of XenoSearch services may also provide additional interfaces; for instance, pay-per-search services may require that clients provide a purchase order that is used for paying for searches that clients carry out using XenoSearch. Different XenoSearch systems may also follow different interaction models; some implementations may employ publish/subscribe middleware, such as Hermes [PB02], to provide *explicit*

event notification functionality, to let clients know when the results of a search query they have placed change.

The way XenoSearch services obtain the data on which they perform searches on behalf of the clients is implementation-specific; XenoSearch services can divide complex search expressions to simpler ones and use XIS to perform these. Alternatively, they can obtain the advertisements directly; this can be done either by simply fetching the list of all advertisements from XIS, or by collecting advertisements directly from the storage locations where individual XenoServers place them. It is envisaged that most XenoSearch systems will find it convenient to use XIS. This simplifies the design and operation of XenoSearch, as the XIS implements functionality for collecting advertisements from individual XenoServers' storage locations and storing these in a structured manner. At the same time, it enables the use of common underlying mechanisms, such as explicit event notification; the XIS implementations utilising a publish/subscribe module can notify XenoSearch services of changes in the status or resource availability of sets of selected XenoServers, which in turn may notify the subscribed users, removing the need for users to subscribe to each XenoSearch individually.

The control-plane of the XenoServer Open Platform coordinates the operation of the various entities involved, but allows for a significant degree of freedom of choice and flexibility, encouraging competition and promoting the evolution of components and services of the platform.

3.5 Summary

This chapter has proposed the XenoServer Open Platform to substantiate a public infrastructure allowing any user to deploy any code anywhere, building on the requirements from a global public computing system and the functionality provided by existing distributed deployment platforms outlined in Chapter 2.

It has introduced the concept of a *XenoServer*, a server that undertakes the execution of potentially untrusted code on behalf of members of the public. Users who run code on XenoServers ultimately get charged for resources consumed by their tasks through *XenoCorp*, a trusted third party. Extensions to the core architecture, such as the *XenoServer Information Service* that assists users in finding suitable XenoServers, have been presented.

This chapter has analysed the operations that the platform supports and the envisaged usage scenarios. Clients and XenoServers *register* with XenoCorp to obtain authentication credentials. Clients *select* the servers on which they wish to deploy services; to do so, they may use the XIS or any other high-level discovery service that collects server advertisements and implements search algorithms. Services can be *deployed* by direct communication of clients and XenoServers, following positive admission control decisions. Users can carry out ongoing *management* of services and execution environments on the XenoServers. Servers *audit* service activity for security purposes, *account* for resources consumed by services they execute, and submit payment claims to XenoCorp which *charges* the corresponding sponsors.

The system does not attempt to optimise resource utilisation using a platform-wide super-scheduler, enforce a single, compulsory matchmaking strategy, or rely on a particular server discovery mechanism. Instead, it has been designed for *openness*, as discussed at the end of this chapter. Allowing the coexistence of a variety of implementations of entities such as XenoCorps, XenoServers, the XIS, and XenoSearch services, allows for low cost of entry, customisation, competition, and the evolution of the platform to suit the diverse needs of different user communities.

The XenoServer platform allows for *local control*. Each user selects servers independently, perhaps using a server discovery service of his or her choice. Also, each XenoServer agrees to comply with the general guidelines its affiliated XenoCorp sets at registration, but ultimately manages itself and carries out operations such as admission control, scheduling, and resource allocation, on its own, without any direct XenoCorp involvement. This is similar to the way franchising works; franchisees agree to comply with the franchise agreement, setting out the franshisor's general requirements and policy, but then make and apply "little" decisions locally, such as the seating allocation or serving order in a restaurant.

A challenging problem that emerges is how the interleaved interests of XenoCorp, server owners, and other stakeholders may be expressed as *resource management policies* and applied on independent XenoServers in a way that does not assume or require central control. The following chapter analyses the problem in more detail, and presents a flexible policy-based resource management framework to address it.

Chapter 4

Resource management

Resource management for global public computing is a challenging area which encompasses problems of how to describe resources, how to advertise their availability, and how to control the allocation or consumption of resources.

There are a number of specific problems which emerge. Firstly, the *heterogeneity* of nodes which make up the systems; nodes have different hardware with different performance characteristics. Devising a methodology for describing resources easily, coordinating descriptions to make sure common resources are described uniformly, and at the same time allowing for unusual and exotic resources to be declared, poses interesting research challenges. Secondly, since XenoServers are charging for resource consumption, a facility for *describing pricing schemes* is also required, allowing the fine-grained association of different schemes with different portions of the same resource.

Thirdly, the potential *global scale* of such systems means that the control software which manages them must be designed with scalability in mind; solutions based on centralised administration are unlikely to be acceptable from a technical view point. Furthermore, *decentralisation* and *federation of control* are necessary in terms of who can define resource allocation policies. A single, central point of control cannot be assumed, as machine owners and other stakeholders must be allowed to influence how resources are to be apportioned between different users or user groups. At the same time, mechanisms for *coordinating* the operation of servers, for instance to achieve naming consistency for resources and to resolve potential conflicts between different entities' interests, are necessary.

Most of the shortcomings of existing deployment platforms do not arise, in this context, from a lack of underlying management mechanisms, such as low-level protection for allowing untrusted code, Virtual Machine Monitors to allow unmodified applications, or QoS-aware schedulers to facilitate resource reservations. They result from a lack of facilities for describing resources and pricing schemes in a fine-grained, coordinated and extensible way, and *expressing* and *combining resource allocation policies* when individual organisations become federated. In this chapter, I present techniques developed for addressing these problems as part of a comprehensive resource management framework for global public computing, reusable in other service deployment platforms.

Section 4.1 provides an example that is used for exposition in the rest of this chapter. Subsequently, Section 4.2 sets out the facilities provided for describing resources and for advertising their pricing and availability. Finally, Section 4.3 introduces *role-base resource management* as a framework for defining resource allocation policies. Examples are provided to show how individual entities describe their constraints on resource usage, and how policies can be defined to specify the reaction of the system to overlapping constraints.

4.1 Running example

The following setting will be used in the rest of this chapter as a consistent source of examples for several aspects of policy-based resource management that will be presented.

Server Sergei is a participant in a public computing platform and provides access to some of its resources to platform users. Sergei's owner wishes to supply its resources in different forms: for instance, he or she wishes to charge a modest amount for best-effort storage in main memory, and to charge a higher amount for reserved access. Also, Sergei's owner wishes to be able to reserve some portion of the machine's resources for his own use and may also want to influence how its resources are apportioned between different platform users or user groups to favour friends or colleagues.

The local administrator of the network where Sergei is connected, called Lou, needs to be able to impose restrictions. In particular, Lou is concerned about excessive non-local network traffic generated by Sergei's participation in the global computing platform because this incurs charges from the network provider. To

limit this, Lou wishes to impose a restriction on the total network bandwidth made available to Sergei's non-local users.

Indy, an infrastructural authority of the public computing platform — a XenoCorp in the XenoServer Open Platform setting, or PLC in PlanetLab — is another entity that wishes to control resource allocation. Indy is interested in providing incentives to users to behave well and to differentiate its services in favour of users who pay higher-fee subscriptions. Users that pay higher fees to Indy, pay on time, and never abuse the platform's resources should be given better services than others, and users who misbehave or are not paying their bills promptly should be penalised.

4.2 Resource description

In this section I introduce the mechanisms used for *naming* and *describing* resources, availability, and pricing schemes. While it is necessary to allow resource description to be carried out in an *independent* and *decentralised* fashion by the individual XenoServers that provide resources, it is also crucial to *coordinate* the way resources are represented.

I break down the resource description process into three parts; firstly, in Section 4.2.1 I examine the naming of individual resources, such as a particular storage device or CPU pool. Then, in Section 4.2.2 I show how different classes of resources are identified — such as a particular kind of CPU — and how their availability and pricing is described. Finally, in Section 4.2.3 I show how decentralised resource descriptions can be coordinated to enhance naming uniformity in the system.

4.2.1 Naming individual resources

In inherently large, federated, and not centrally controlled systems, such as distributed deployment platforms and global public computing infrastructures, the decision about how resources are to be named needs to be completely decentralised. Each of the machine owners needs to be able to name and represent its resources independently from others, without involving any authority that has central control over that process. Thus, a flat and static naming scheme would prove inefficient. A *hierarchical* naming scheme is necessary.

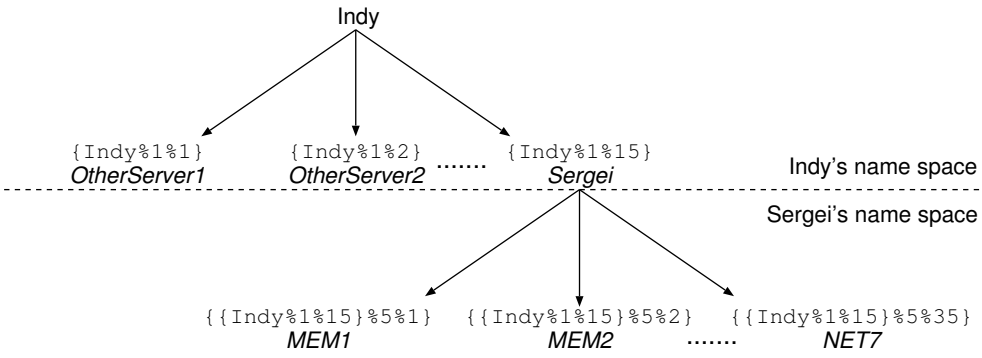

Figure 4.1: Hierarchical resource naming scheme

An example of how the proposed scheme works is shown in Figure 4.1. All entities and items are named using *hierarchical tuples* of the form {`namer%type%ID`}. Unique naming of top-level entities — infrastructural authorities, such as `Indy` — is achieved by using their credentials (e.g. public keys) as their names. Infrastructural authorities assign names to servers and other entities or items, by placing themselves in the `namer` part. In the running example, Indy names Sergei as {`Indy%1%15`}, where 1 indicates that the entity named is of type "server", and 15 is a unique ID for Sergei in Indy's domain, and Sergei's owner as {`Indy%2%38`}, where 2 signifies a human user.

Sergei can decide on the identifiers that he assigns to his resources and sessions independently, by placing {`Indy%1%15`} in the `namer` part. For instance, he names his `MEM1` resource — a portion of memory sold in a particular pricing unit — as {{`Indy%1%15`}`%5%1`}, where 5 denotes a resource instance. This hierarchical structure achieves decentralisation of resource descriptions, as each server can manage its own name-space without requiring communication with a central coordinator.

4.2.2 Describing resources

As shown above, each server can name its resources individually and independently. A more challenging issue is the one of *describing* resources. Servers need to be able to express, for each resource they can provide, its type, any QoS guarantees associated with it, the unit in which it is sold, pricing information, and current availability.

It is necessary that resource description mechanisms are *flexible*, yet *comprehensible*. The size and complexity of descriptions has to be controlled, avoiding an unnecessarily complicated description language. Descriptions need to be *decentralised*, as servers must be allowed to describe their resources independently.

Each resource in a public computing environment carries QoS specifications regarding the fashion in which it is provided by the server. For instance, Sergei could provide access to its memory as best-effort, or with QoS guarantees such as N MB guaranteed throughout the session, or N MB $\times$ sec guaranteed.

Instead of associating a separate set of QoS guarantees with each resource description, these are incorporated in the description itself. A resource is described by its *kind* as well as *pricing unit*. The following parameters are necessary to describe a resource and the fashion in which it is made available by the server:

- *Name* identifies the particular resource.
- *Owner* is the entity that owns the resource.
- *Administrator* is the entity that is technically responsible for administering the machine where the resource is located.
- *Provider* is the entity that provides — i.e. declares and advertises — the resource.
- *Kind* refers to the type of the resource, like "Pentium4 CPU at 2GHz".
- *Pricing Unit* is the unit in which the resource is made available. For instance, such units can be "1 ms of CPU time per wall-clock sec" or "10 MB of memory".
- *Cost per unit* indicates how much each unit of that resource is priced by the server.
- *Availability* represents the number of units of that resource that are available on the server.

The tuple {`name, owner, administrator, provider, kind, pricing unit, cost per unit, availability`} is termed a *resource description*. The resource descriptions used are simple and generic enough to not be bound to a particular language; it is easy to envisage descriptions being expressed in XML or any other

suitably generic description language. The rest of this dissertation uses a semi-structured notation, borrowed from the Condor `classads` scheme, as introduced in Section 2.4.2.

For example, suppose that Sergei wishes to sell his 512MB memory in three different ways. Firstly, he wishes to provide 200MB on a purely best-effort basis. Clients who purchase this resource get access to a 200MB memory pool, but no guarantees of how much of that memory will be available at any time. Sergei wishes to sell this resource at a fixed charge of $2 and with a cap of 10 tasks allowed to access this pool at any time. This is represented as:

```
{
  Type          = Resource;
  Name          = MEM1;
  Owner         = Sergei's owner;
  Administrator = Sergei's owner;
  Provider      = Sergei;
  Kind          = memory;
  PricingUnit   = 200MB best-effort access;
  CostPerUnit   = 2;
  Availability  = 10;
}
```

Also, Sergei wishes to sell a further 150MB in a per 10MB fashion at a price of $6 per 10MB unit, guaranteeing that this amount of memory will be available to the task throughout the session. To express this resource, Sergei can use:

```
{
  Type          = Resource;
  Name          = MEM2;
  Owner         = Sergei's owner;
  Administrator = Sergei's owner;
  Provider      = Sergei;
  Kind          = memory;
  PricingUnit   = 10MB guaranteed;
  CostPerUnit   = 6;
  Availability  = 15;
}
```

Finally, Sergei is to represent the sale of the last 160MB in a "per 10MB $\times$ minute" fashion at a cost of $1 for an allocation of 10MB of main memory for

one minute — or any combination, resulting in the same total allocation, e.g. 1MB for ten minutes or 2MB for five minutes. Supposing he is advertising his resource availability every five minutes, then he can provide $5 \times 160/10 = 80$ units of 10MB $\times$ minute in that time period. Thus, Sergei describes the resource as:

```
{
  Type          = Resource;
  Name          = MEM3;
  Owner         = Sergei's owner;
  Administrator = Sergei's owner;
  Provider      = Sergei;
  Kind          = memory;
  PricingUnit   = 10MB x min guaranteed;
  CostPerUnit   = 1;
  Availability  = 80;
}
```

4.2.3 Coordinating descriptions

While it is necessary that no central authority is involved in the resource discovery path, allowing each server operator to independently define the kinds and pricing units of his or her servers' resources using arbitrary names would lead to inconsistencies. It is necessary to avoid having an overabundance of different names and pricing schemes — e.g. "P4 CPU 2 GHz", "Pentium4 processor at 2GHz", and "2GHz P4", or "10MB of RAM" and "10MB of main memory space", as that would significantly increase the complexity of searching. At the same time, it is equally desirable that the system accommodates a wide range of different resources, while avoiding requiring a central taxonomy.

The approach that I take is to define a set of common resource kinds and pricing units, whose naming can be coordinated, and assign *identifiers* and *human-readable names* to each one. Then, each resource declared in server advertisements includes not the human-readable descriptions of its resource kind and pricing unit, but the identifiers that correspond to these. The three memory resources introduced earlier can now be expressed as shown in Figure 4.2. Figure 4.3 shows the resulting style of server advertisement containing a list of resource descriptions.

```
{ Type          = Resource;
  Name          = Indy%1%15%5%1;  # MEM1
  Owner         = Indy%2%38;      # Sergei's owner
  Administrator = Indy%2%38;      # Sergei's owner
  Provider      = Indy%1%15;      # Sergei
  Kind          = Indy%6%9900;    # memory
  PricingUnit   = Indy%7%100;     # best-effort
  CostPerUnit   = 2;
  Availability  = 10; }

{ Type          = Resource;
  Name          = Indy%1%15%5%2;  # MEM2
  Owner         = Indy%2%38;      # Sergei's owner
  Administrator = Indy%2%38;      # Sergei's owner
  Provider      = Indy%1%15;      # Sergei
  Kind          = Indy%6%9900;    # memory
  PricingUnit   = Indy%7%101;     # 10MB guaranteed
  CostPerUnit   = 6;
  Availability  = 15; }

{ Type          = Resource;
  Name          = Indy%1%15%5%3;  # MEM3
  Owner         = Indy%2%38;      # Sergei's owner
  Administrator = Indy%2%38;      # Sergei's owner
  Provider      = Indy%1%15;      # Sergei
  Kind          = Indy%6%9900;    # memory
  PricingUnit   = Indy%7%102;     # 10MB x min guaranteed
  CostPerUnit   = 1;
  Availability  = 80; }
```

Figure 4.2: Resource descriptions using the proposed naming coordination scheme

To identify resource kinds and pricing schemes I use the same form of hierarchical names used to identify servers and individual resources. Indy uses {`Indy%6%X`} and {`Indy%7%Y`} to identify a resource kind and pricing unit respectively, where 6 and 7 represent that the items represented are of type "resource kind" and "pricing unit" respectively. Identifiers allocated by infrastructure providers are anticipated to be common ones used by all of their associated servers, while the servers themselves are still able to represent fresh kinds of resources if they have unique facilities.

This way, advertisements become shorter, simpler, and easier to process. References to resources and pricing schemes are readable by machines — as they follow the common hierarchical tuple format — and can be easily translated to

```
{
  Type          = Server;
  Name          = Indy%1%15;
  Owner         = Indy%2%38;
  Administrator = Indy%2%38;
  IPAddress     = 128.232.35.170;
  City          = Cambridge;
  Area          = Cambridgeshire;
  Country       = UK;
  Resources
  {
    { Type          = Resource;
      Name          = Indy%1%15%5%6;   # CPU2
      Owner         = Indy%2%38;
      Administrator = Indy%2%38;
      Provider      = Indy%1%15;
      Kind          = Indy%6%401;      # Pentium4 - 1400
      PricingUnit   = Indy%7%2;        # CPU ms per wall-clock sec
      CostPerUnit   = 10;
      Availability  = 110; }

    { Type          = Resource;
      Name          = Indy%1%15%5%3;   # MEM3
      Owner         = Indy%2%38;
      Administrator = Indy%2%38;
      Provider      = Indy%1%15;
      Kind          = Indy%6%9900;     # memory
      PricingUnit   = Indy%7%102;      # 10MB x minute guaranteed
      CostPerUnit   = 1;
      Availability  = 80; }
    ...
  }
}
```

Figure 4.3: An example server advertisement.

be human-readable. Consistency is improved, as clients can be sure that the same kind is used to characterise instances of the same resource, and the same unit to indicate resources that are priced in the same way. Moreover, it allows infrastructural authorities to easily rename, remove, and coordinate platform-wide support for specific resources and pricing units.

The infrastructural authority — XenoCorp, in the XenoServers' case — or some other entity or service needs to provide the predefined sets of resource

```
0    = Pentium2
1    = Pentium2-233
...
100  = Pentium3
...
200  = Celeron
...
300  = Itanium
301  = Itanium-733
...
400  = Pentium4
401  = Pentium4-1400
...
9900 = Memory
9901 = Local Storage
9902 = Network interface
9903 = IPv4 Address
9904 = IPv6 Address
...
```

(a) Resource kinds map

```
0   = best-effort CPU access
1   = ms of CPU time
2   = ms of CPU time per wall clock sec
3   = proportion of CPU
...
100 = best-effort memory access
101 = memory 10MB
102 = memory 10MB x minute
103 = proportion of memory
...
200 = best-effort network access
201 = network MB
202 = network MB/sec
203 = proportion of network bandwidth
204 = network GB per month
205 = network GB per month on port 80
...
300 = IPv4 address and full port range
...
```

(b) Pricing units map

Figure 4.4: Example maps advertised by the infrastructural authority. Although small integer numbers are used for illustration here, secure hash values based on textual descriptions are used in the implemented prototype to avoid confusion if one of the descriptions is updated

kinds and pricing units required, and the mappings between hierarchical names and descriptions. Such mappings, associating the ID part — `X` and `Y` — of the hierarchical tuples with resource kinds and pricing units are shown in Figure 4.4. To disseminate these mappings, XenoCorp pushes them in clients' and XenoServers' storage locations — as was shown in Figure 3.3, operation 4. This happens at registration, and at infrequent intervals after that.

A trade-off between the level of detail in resource descriptions and ease of searching can be identified here. Defining a few fixed resource descriptions is beneficial for consistency and ease of searching, but imprecise and inflexible, as descriptions have to be coarse-grained and generic. On the other hand, defining a large number of detailed resource descriptions makes searching much harder, as there are many different identifiers for the same — or very similar — resources. The coordination scheme proposed provides a flexible and adjustable mechanism that allows consistent naming of common resources and pricing schemes while permitting the representation of exotic ones.

4.3 Role-based resource management

In the previous section I described how individual resources can be named and described. I now turn to mechanisms for defining policies over resource usage and for combining policies expressed by different administrators, central authorities or other entities. In outline, I propose a *role-based resource management* (RBRM) scheme, in which the owner of the resources and other stakeholders can express which users or groups of users can be allowed access to which parts of the resources. Early work on this subject has been presented in [KH03].

The approach followed can be seen as a development of role-based access control (RBAC). As with RBAC, I use the concept of a *role* as a method for organising users into groups to which common policies should be applied. RBAC allows entities to specify which users should be allowed to enter which role, as well as which roles should be granted access to which resources. The nature of RBAC decisions is binary; one can either be granted access or not. Thus, when role entry conditions are overlapping, the most common approach is that if there is one that denies access then it simply overrides the others, in application of the "least privilege" principle [SS75].

When considering applying the same concepts on distributed resource management, a key technical difference that emerges is that, in contrast to access control, which is binary, resource management is *quantitative*; the question then becomes how much access to grant a user to a resource, rather than simply whether to grant access or not.

Moreover, in global-scale systems there is usually no notion of a central authority controlling the distributed resources. Therefore, one can expect that *federated policies* and roles may need to be defined by a number of heterogeneous entities that coexist under separate administrations. Without a notion of a central authority controlling the resources one can anticipate *overlapping policies*, where entities have imposed different constraints that relate to the same user and resource, as described in Section 4.3.4.3.

A problem that emerges as a result of these challenges is *overlapping resolution*. In the running example, Sergei restricts access to the network bandwidth to X% for users other than its owner, while Lou restricts access to Y% for remote users and Indy guarantees access to Z% for good customers. In the event of a user being a non-local good customer, would he be given X%, Y% or Z%? Or maybe the minimum, maximum or average of the three?

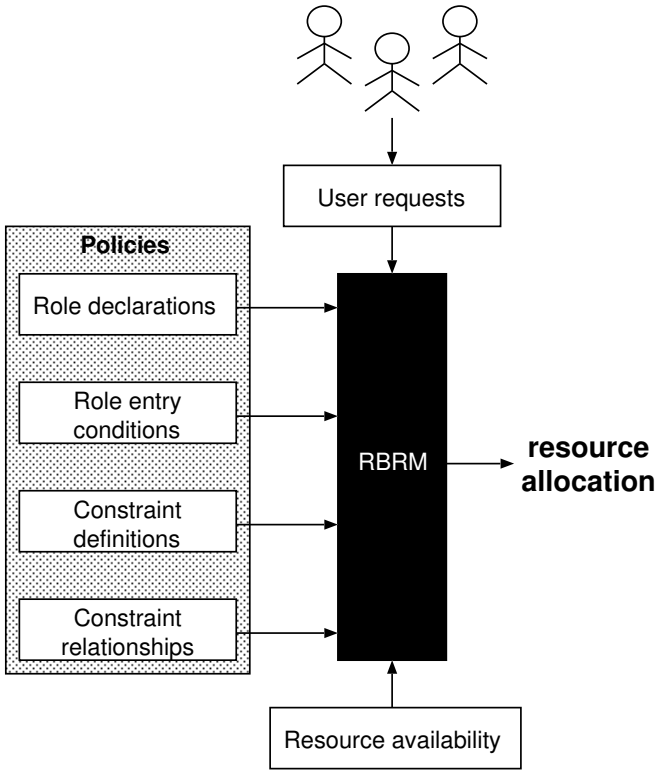

Figure 4.5: High-level view of the RBRM architecture

This style of problem can be resolved more easily in classical RBAC systems, usually by denying an access request if any one constraint rules against it. In the resource management case, where policies are quantitative, an explicit *resolution* step is introduced to provide more flexible alternatives.

A flexible, expressive and comprehensible system, able to combine role entry and resource allocation constraints, is required to allow effective resource management in global-scale public computing infrastructures.

4.3.1 Overview

The aim is to allow the server owner and other entities to define policies that dictate how resources of the server are to be apportioned between different users or user groups.

The main components of the proposed role-based resource management architecture, shown in Figure 4.5, are the following. There are *users*, who request resources from a server. Entities declare *roles* on servers, which identify classes of users for which they wish to define policies, and *role entry conditions* which determine which users are members of which roles. To define how resources can be allocated to users based on their properties and role memberships, entities specify *constraints*. Finally, *constraint relationships* can be used to indicate how to resolve some kinds of overlaps in policies. Each server decides on whether to accept or deny resource reservation requests based on policies and current *resource availability*. Roles, entry conditions, constraints, and constraint relationships are collectively referred to as *policy elements*.

Server owners and other entities can *describe policies* for resource allocation on servers. Policy elements are *declared* on servers on which they apply. For the policies to reach the servers, *deployment* of policy elements is performed. Finally, each server independently *evaluates* the policies declared on it to determine how to handle resource allocation requests.

The following section examines how policies are declared on a server, and the authentication and authorisation decisions involved. Section 4.3.3 introduces the types and format of policy elements that can be used. Section 4.3.4 discusses how a server reaches admission control decisions based on the policies declared on it. Section 4.3.5 describes example deployment settings. Finally, Section 4.3.6 presents example applications of the proposed policy description scheme.

4.3.2 Declaration of policies

All policy enforcement takes place in a decentralised manner on individual servers. The *scope* of policy elements is local; an element is only applicable on the server on which it is declared. This section explains how federated entities declare policy elements on a server. The process of transferring the policies to be declared on the servers that are to apply them is termed *policy deployment* and discussed in Section 4.3.5.

Each server comprises an *authentication* layer, a *tentative declarations* set, a *policy filter*, and a *confirmed declarations* set, as shown in Figure 4.6. The tentative declarations set is accessible by all authenticated entities, and contains all policy element declarations attempted on a server. The confirmed declarations set is not accessible to any entities other than the server itself and contains the

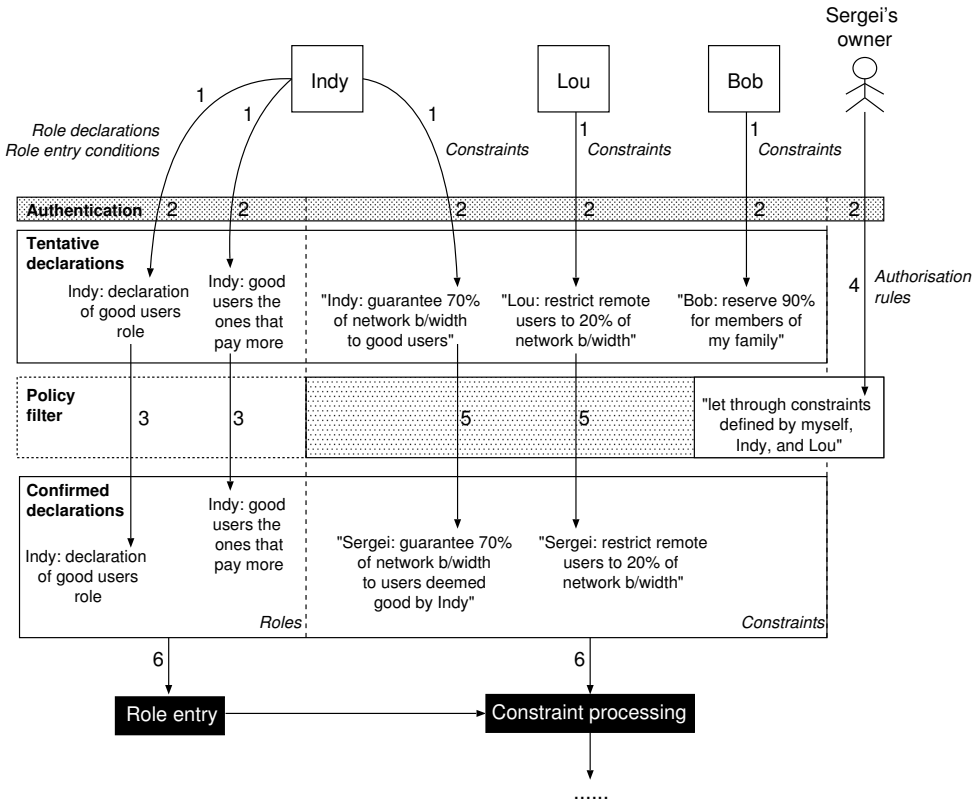

Figure 4.6: Authentication and filtering of deployed policies

policy elements that are taken into account when considering resource allocations on the server. The policy filter controls which of the tentative declarations are to be copied to the confirmed set.

A policy element is *declared* on a server when it is placed in its tentative declarations container — operation 1 in Figure 4.6. All entities registered with the public computing platform are allowed to independently declare policy elements on servers, subject to a constraint: policy elements need to be properly *signed* by the entities to be non-repudiable and unforgeable. The authentication layer checks the signatures on policy elements to ensure that the *Declarer* part of role declarations, the *Elector* part of role entry conditions, and the *Constrainer* part of constraint definitions and relationships (explained in the next section) correspond to the identities of the entities who attempt the declarations — operation 2 in Figure 4.6.

Role declarations and role entry conditions declared by arbitrary entities offer a federated view of the role memberships but cannot directly affect the resource

allocations taking place on a server. Thus, it is safe to subsequently copy all properly authenticated role declarations and role entry conditions to the confirmed declarations set — operation 3 in Figure 4.6.

The association of role declarations and entry conditions with *constraints* does affect resource reservations. Hence, entities are allowed to deploy constraints and constraint relationships on servers only subject to a further *authorisation* check, carried out by the policy filter. The server owner can define simple authorisation rules — operation 4 in Figure 4.6. Authorisation rules are accepted by the filter only if they are properly signed by the server itself. These allow policies from a few entities that the server trusts or has long-term relationships with — such as XenoCorps — through the filter — operation 5 in Figure 4.6.

Role declarations and role entry conditions contained in the confirmed set may be declared by entities other than the server itself, to offer a federated view of the role memberships. This allows constraints and constraint relationships to take the views of different entities into account to devise the applicable policy. On the other hand, constraint definitions and constraint relationships have to be signed — i.e. declared — by the server itself when in the confirmed set, as they affect resource allocations. When the filter lets through a constraint or constraint relationship declared by an entity other than the server itself, the constraint is *endorsed*: the server places itself as the *Constrainer*, and signs the constraint.

The set of confirmed declarations is then passed on to the RBRM policy evaluation procedure — operation 6 in Figure 4.6, as described in Section 4.3.4.

4.3.3 Policy description

This section describes the syntax and usage of the policy elements that can be declared. I first describe the inference notation used in the rest of this chapter. Then, I explain how roles, constraints, and constraint relationships can be defined and combined to express complex resource allocation policies in a flexible manner.

For simplicity, this section uses human-readable names (e.g. Indy, Sergei, NET3, CPU2) instead of hierarchical tuples — as introduced in Section 4.2.1 — to identify entities and resources.

Notation. I use `this` font to denote defined entities, roles, constants, and methods. I use *this* font to refer to variables and to describe the generic format

of role declarations and entry conditions, constraint definitions and constraint relationships. The notation of inference used denotes that if the condition that is defined above the inference line holds, then the action defined below the inference line is taken. Variables are always bound to the value they get above the inference line. For instance, let us consider the following example.

$$\frac{\texttt{A}(X) \quad X > \texttt{10} \quad \texttt{B}()}{\texttt{C}(X)}$$

Suppose `A(15)` and `B(2)` are true. Then the variable X is bound to the value 15, and the result of the inference is `C(15)`. If $X \leq$ `10` or `B()` does not hold, then no inference can be made about $\texttt{C}(X)$ using the above rule.

The wildcard $*$ is used to denote "any entity" or "any value". Conditions referring to parameterised objects without including a value for the parameter denote that any value leads to a match — i.e. $\texttt{A}() \equiv \texttt{A}(*)$.

Also, note that the $*$ does not signify in practice "any arbitrary entity" as the policy filtering process only allows *authenticated* and *authorised* policy elements to be declared on a server.

The `User` and `Server` objects represent properties and methods exported by the user in question and the server on which the resource reservation request is placed. Such properties may include the network address, location, or ownership.

4.3.3.1 Role declarations.

In an open, large-scale system, it would be impractical and restrictive to enforce a single, flat, system-wide name space for roles. Instead, the chosen approach is to name roles *hierarchically*, so that each entity that defines roles can have its own role name space.

To define a role, a *role declaration* has to be used, identifying a role called *Name* created by the entity called *Declarer*. The format of *RoleDeclaration* statements is:

$$Declarer : Name(Parameter1, Parameter2, ...)$$

It is required that the *Declarer* signs the statement to prove its identity — in other words, entities are not able to declare roles in the name of other entities. *Parameter1*, *Parameter2*, ... are bound when users enter the role.

In the running example, Sergei, Lou and Indy declare the roles on which they impose resource restrictions or reservations:

$$\texttt{Sergei} : \texttt{Authenticated}()$$

$$\texttt{Sergei} : \texttt{NotOwner}()$$

$$\texttt{Lou} : \texttt{Local}()$$

$$\texttt{Lou} : \texttt{Remote}(\texttt{RTT})$$

$$\texttt{Indy} : \texttt{Bad}()$$

$$\texttt{Indy} : \texttt{Good}()$$

The `Remote` role is parameterised; the `RTT` parameter is bound to the round-trip time between the server and the user when the user enters the role.

4.3.3.2 Role entry conditions

Entry conditions specify what conditions a user has to meet to be deemed a member of a given role. These conditions can be in terms of membership of other roles, user properties, or in reference to external sources such as the time of day or a user-reputation service. A generic form of a *RoleEntryCondition* is:

$$\frac{\mathit{RoleMemberships} \quad \mathit{Expressions}}{\mathit{RoleMembership}}$$

To express *RoleMembership* statements, the notation $\mathit{Elector} \rightarrow \mathit{RoleDeclaration}$ is used to indicate that the specified elector entity asserts that the user in question is a member of the specified role. It is required that electors of the resulting role memberships *sign* election statements to prevent forging.

For flexibility, role membership is designed to be *subjective*. A user's membership of a role is not global truth, but truth according to the *Elector*. This

notion is borrowed from earlier work in role-based access control [Hay96]. In other words, different electors may indeed have different views of role memberships, and present these views to other entities by declaring the corresponding entry conditions on them. How much these other entities value each elector's view is entirely up to them, in accordance with the *local control* principle introduced in Chapter 2.

For example, Sergei defines that users connected to any network apart from his own should enter the `Remote` role, as defined by Lou.

$$\frac{\texttt{User}.\mathit{Network} \neq \texttt{128.232.0.0/16}}{\texttt{Sergei} \rightarrow \texttt{Lou} : \texttt{Remote}(\texttt{Server.ping}(\texttt{User}.\mathit{IPAddress}))}$$

Sergei allows entry to the `Local` role to any user connected to the local network that is not its owner:

$$\frac{\texttt{User}.\mathit{Network} = \texttt{128.232.0.0/16} \quad \texttt{Sergei} \rightarrow \texttt{Sergei} : \texttt{NotOwner}()}{\texttt{Sergei} \rightarrow \texttt{Lou} : \texttt{Local}()}$$

Using the above statement, Sergei elects users to roles defined by Lou. Any entity is allowed to elect users to any role, as this allows for a federated view of role memberships to be formed — different entities may have different views of role memberships — and does not directly affect resource allocations on the servers; to attach policies on how resources are to be allocated to roles, constraints need to be declared.

4.3.3.3 Constraint definitions

To express a reservation or usage limitation on a resource, *constraint definitions* are used. A constraint definition is associated with a role, applies to all members of the role, and limits or guarantees the amount of a resource that members of a role can get. A *ConstraintDefinition* is of the form:

$$\frac{\mathit{RoleMembership}}{\mathit{Constrainer} \rightarrow \mathit{Constraint}}$$

where *Constrainer* is the entity that is imposing the constraint. As before, the

Constrainer has to *sign* the constraint definition, to ensure unforgeability. The format of the *Constraint* itself is

$$\mathit{ConstraintKind}(\mathit{Resource}, \mathit{Parameters})$$

ConstraintKind is an identifier that describes what kind of limitation or reservation the constraint is meant to indicate, such as `limEach` (limit each member), `limGrp` (limit all members collectively), or `rsvGrp` (reserve for all members collectively). *Resource* identifies the resource that the constraint applies to, either a specific one or a resource kind. *Parameters* indicate the extent of the limitation or reservation. Note that reservations are applied on a per group basis rather than allowing "reserve for each member" as a kind of constraint — that latter kind of reservation is not possible without being able to enumerate the group's membership, which is difficult in a decentralised, federated model.

Constraints can only influence resource allocations on the *Constrainer* entity; to declare constraints on servers, an *authorisation* step is required, as explained in Section 4.3.2. In the example, Sergei has long-term agreements with Lou and Indy and explicitly authorises constraints declared by them.

Let us consider the running example again. Lou wishes to restrict access to Sergei's network bandwidth by the group of `Remote` users to 20%, and he intends to impose a more strict limitation of 10% on users connected to the particularly greedy network 139.91.0.0/16. Also, if the round-trip latency between the user and the server is more than one second, Lou wishes to impose a limitation such that the bandwidth restriction is inversely proportional to that latency. Assuming `NET3` denotes a network bandwidth resource:

$$\frac{\texttt{Sergei} \rightarrow \texttt{Lou} : \texttt{Remote}()}{\texttt{Lou} \rightarrow \texttt{limGrp}(\texttt{NET3}, 20\%)}$$

$$\frac{\texttt{Sergei} \rightarrow \texttt{Lou} : \texttt{Remote}() \quad \texttt{User}.\mathit{Network} \;=\; \texttt{139.91.0.0/16}}{\texttt{Lou} \rightarrow \texttt{limGrp}(\texttt{NET3}, 10\%)}$$

$$\frac{\texttt{Sergei} \rightarrow \texttt{Lou} : \texttt{Remote}(R) \quad R \geq \texttt{1}}{\texttt{Lou} \rightarrow \texttt{limGrp}(\texttt{NET3}, \frac{10}{R}\%)}$$

Sergei's owner wishes to reserve some of the network bandwidth for himself, so he limits access by members of the `NotOwner` to 90%:

$$\frac{\texttt{Sergei} \rightarrow \texttt{Sergei} : \texttt{NotOwner}()}{\texttt{Sergei} \rightarrow \texttt{limGrp}(\texttt{NET3}, 90\%)}$$

Indy, the infrastructural authority, imposes a network bandwidth reservation of 70% for the `Good` customers group, while restricting access by each of the `Bad` users to 2%:

$$\frac{\texttt{Indy} \rightarrow \texttt{Indy} : \texttt{Good}()}{\texttt{Indy} \rightarrow \texttt{rsvGrp}(\texttt{NET3}, 70\%)}$$

$$\frac{\texttt{Indy} \rightarrow \texttt{Indy} : \texttt{Bad}()}{\texttt{Indy} \rightarrow \texttt{limEach}(\texttt{NET3}, 2\%)}$$

Sergei subsequently *endorses* constraints — i.e. places itself as the *Constrainer* and signs the constraints — declared by the authorised (by him) entities Lou and Indy for them to be applicable, as described in Section 4.3.2.

Note that the above constraints can be *overlapping*; which restriction should apply to a user who happens not to be the owner of Sergei, to be Remote, and Good? The next section describes a scheme to explicitly define policies on how such overlaps are to be handled.

4.3.3.4 Constraint relationships

In order to allow defining policies on how overlapping constraints are resolved, I introduce *constraint relationships*. Each relationship gives a series of pattern-matches for existing constraints, and then a replacement constraint to be generated in their place. The format of a *ConstraintRelationship* is:

$$\frac{\mathit{ConstraintDefinition}(s) \quad \mathit{Expression}(s)}{\mathit{ConstraintDefinition}}$$

A constraint relationship has to be signed by the *Constrainer* of the replacement constraint.

For instance, suppose that in the example we need to define what should be done when a user is a member of both the `Good` and `Remote` roles. Sergei wishes to express that if the round-trip time between the server and the user is greater than 2 seconds then the overlapping constraint definitions should be replaced by a new constraint, limiting access to the extent dictated by the constraint associated with the `Remote` role. Otherwise, the (endorsed) constraints are to be replaced by one making a reservation in accordance with the constraint associated with the `Good` role. Then, the following constraint relationship statements can be used:

$$\dfrac{\dfrac{\texttt{Sergei} \rightarrow \texttt{Lou} : \texttt{Remote}(R)}{\texttt{Sergei} \rightarrow \texttt{limGrp}(\texttt{NET3}, Y\%)} \quad \dfrac{\texttt{Indy} \rightarrow \texttt{Indy} : \texttt{Good}()}{\texttt{Sergei} \rightarrow \texttt{rsvGrp}(\texttt{NET3}, Z\%)} \quad R > 2}{\dfrac{\texttt{Sergei} \rightarrow \{\texttt{Lou} : \texttt{Remote}(R), \texttt{Indy} : \texttt{Good}()\}}{\texttt{Sergei} \rightarrow \texttt{limGrp}(\texttt{NET3}, Y\%)}}$$

$$\dfrac{\dfrac{\texttt{Sergei} \rightarrow \texttt{Lou} : \texttt{Remote}(R)}{\texttt{Sergei} \rightarrow \texttt{limGrp}(\texttt{NET3}, Y\%)} \quad \dfrac{\texttt{Indy} \rightarrow \texttt{Indy} : \texttt{Good}()}{\texttt{Sergei} \rightarrow \texttt{rsvGrp}(\texttt{NET3}, Z\%)} \quad R \leq 2}{\dfrac{\texttt{Sergei} \rightarrow \{\texttt{Lou} : \texttt{Remote}(R), \texttt{Indy} : \texttt{Good}()\}}{\texttt{Sergei} \rightarrow \texttt{rsvGrp}(\texttt{NET3}, Z\%)}}$$

Note that the replacement constraint may be associated with any or both of the `Remote` and `Good` roles; the new constraint applies to the user anyway, as he or she is a member of both roles — otherwise the overlap would not exist.

To specify that when a user is a member of the `Remote` and `Bad` roles — according to any authorised entity — the minimum ought be taken, the following two relationships can be used:

$$\dfrac{\dfrac{* \rightarrow \texttt{Lou} : \texttt{Remote}(R)}{* \rightarrow \texttt{limGrp}(\texttt{NET3}, Y\%)} \quad \dfrac{* \rightarrow \texttt{Indy} : \texttt{Bad}()}{* \rightarrow \texttt{limGrp}(\texttt{NET3}, L\%)}}{\dfrac{\texttt{Sergei} \rightarrow \{\texttt{Lou} : \texttt{Remote}(R), \texttt{Indy} : \texttt{Bad}()\}}{\texttt{Sergei} \rightarrow \texttt{limGrp}(\texttt{NET3}, \texttt{min}(Y, L)\%)}}$$

Constraint relationships can also express more general resolution strategies such as taking the minimum of a set of `limEach` constraints given on a particular resource:

$$\dfrac{\dfrac{\dfrac{* \rightarrow * : R1}{* \rightarrow \texttt{limEach}(K, X\%)} \quad \dfrac{* \rightarrow * : R2}{* \rightarrow \texttt{limEach}(K, Y\%)}}{* \rightarrow \{* : R1, * : R2\}}}{\texttt{Sergei} \rightarrow \texttt{limEach}(K, \texttt{min}(X, Y)\%}$$

4.3.3.5 Timed policies

Some entities may need to define policies that incorporate a time element. To do so, an optional $|Starttime|Stoptime$ expression can be used, where times are expressed in the form $|w, yyyymmdd, hhmm|$ and refer to absolute times. The w parameter refers to the day of the week, and can take values from 1 to 7 — 1 representing Sunday and 7 denoting Saturday. Time expressions can be attached to policy elements, such as role entry conditions and constraints, to denote when elements are to start being considered in deriving resource management decisions, and when they are to be withdrawn from the system.

The $*$ wildcard may be used to denote that a policy should start and stop at particular times every day of the week, day, month, or year[1]. Note that there has to be at least one $*$ in each time expression, as a w parameter and a full $yyyymmdd$ expression are mutually exclusive.

For instance, Indy, the infrastructural authority, wishes to declare that the reservation he imposes on network bandwidth for the `Good` customers group should only be active for 9 September 2009:

$$\dfrac{\texttt{Indy} \rightarrow \texttt{Indy} : \texttt{Good}()}{\texttt{Indy} \rightarrow \texttt{Indy} : \texttt{rsvGrp}(\texttt{NET3}, 70\%)| *, 20090909, 0000 \mid *, 20090909, 2359}$$

In another example, if Indy wishes to declare that the reservation he imposes on network bandwidth for the `Good` customers group should only be active from 10am to 10pm every Monday of year 2009:

$$\dfrac{\texttt{Indy} \rightarrow \texttt{Indy} : \texttt{Good}()}{\texttt{Indy} \rightarrow \texttt{Indy} : \texttt{rsvGrp}(\texttt{NET3}, 70\%)| 2, 2009 * *, 1000 \mid 2, 2009 * *, 2159}$$

[1] One may note the similarity of this syntax with the one used in Unix `crontab` (`cron` table) files.

Similarly, Lou may wish to limit the restriction on network bandwidth usage by remote users further to 10% between 3-5pm on Mondays and Wednesdays — peak periods, as network traffic charges are higher at those times. Such a policy can be defined using two timed policy elements:

$$\frac{\texttt{Sergei} \rightarrow \texttt{Lou} : \texttt{Remote}()}{\texttt{Lou} \rightarrow \texttt{limGrp}(\texttt{NET3}, 10\%) \mid 2, * * *, 1500 \mid 2, * * *, 1659}$$

$$\frac{\texttt{Sergei} \rightarrow \texttt{Lou} : \texttt{Remote}()}{\texttt{Lou} \rightarrow \texttt{limGrp}(\texttt{NET3}, 10\%) \mid 4, * * *, 1500 \mid 4, * * *, 1659}$$

4.3.4 Policy evaluation

In the previous sections I introduced the policy description language used in the system, and explained how policies consisting of roles and constraints can be defined, managed and combined. This section examines how the system uses the set of roles and constraints declared, in order to determine how to handle a resource allocation request. I term this process *policy evaluation*.

The evaluation process, as shown in Figure 4.7, takes place every time a user places a resource allocation request. If the decision is positive, then the user is allocated the requested resources for the duration of his or her session, or until one of the conditions based on which the decision was made expires. In cases where a resource allocation has to be urgently recalled, an asynchronous revocation mechanism can be used — see Section 4.3.4.5.

For exposition, let us consider a remote user Uma, who is well-behaved and pays a high monthly fee to Indy, and a remote user Randy, who constantly delays payments. Uma and Randy both wish to reserve 5% of the network bandwidth on Sergei — resource `NET3`.

The policy elements — roles, role entry conditions, constraints, and constraint relationships — to be taken into account when considering Uma's request are taken from Sergei's *confirmed declarations* set, as described in Section 4.3.2.

4.3.4.1 Role entry

The first step to reach an admission control decision is to decide which roles a user is a member of — operation 1 in Figure 4.7. This process requires as input

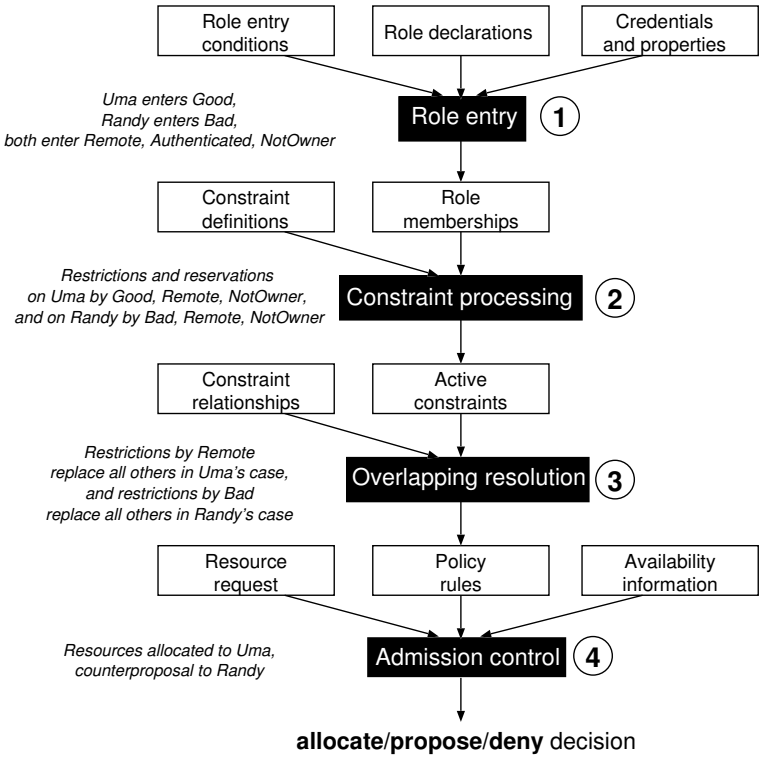

Figure 4.7: Policy evaluation process

the credentials and properties of the user as well as the role declarations and entry conditions.

Entry conditions have to be examined and checked against the user's properties and credentials for the role memberships to be determined. In the running example, Uma enters the `Good` and Randy enters the `Bad` role according to Indy, as he consults the information he keeps about user behaviour and subscription. Both Uma and Randy enter the `Authenticated` role according to Indy. Sergei asserts that Uma and Randy enter the `Remote` role after realising that their `Network` properties do not match his. By checking their credentials, Sergei also asserts that both users enter the `NotOwner` role.

The output of this process is the set of *role memberships* that the user shares. Developing an algorithm to determine role memberships is straightforward, as there are no role entry conditions that allow membership in a role on the condition that there is no membership of another one. The algorithm that derives which of the defined roles a user shall enter works as follows.

I term the roles whose entry conditions are exclusively based on the credentials and properties of the user — not membership of other roles — as the *first-level* roles — for instance, the *Remote* role in the running example. The algorithm initially checks the properties of the user in question against each first-level role's entry conditions and determines whether to allow entry to each of these roles.

Then, once all first-level role memberships have been determined, the algorithm can proceed to *second-level* roles, which are the roles whose entry conditions may be based on user properties and membership of first-level roles. After second-level role memberships are finalised the algorithm proceeds to *third-level* roles, which comprise entry conditions based on user properties, first-level role membership and second-level role membership. The algorithm continues operating in this manner until membership decisions have been made for all existing roles.

4.3.4.2 Constraint processing

Once the role memberships for the user that requests resources have been determined, they are associated with the constraints that apply to them. Constraints that are not associated with any of these roles are ignored further on, since they are unable to affect the admission control decision. This allows the reduction of the initial set of constraints to a set of — potentially overlapping — *active constraints* — operation 2 in Figure 4.7.

In the example, Uma's membership of the `Good` role imposes a reservation of 70% on the total bandwidth used by members of the same group, while her memberships of `Remote` and `NotOwner` impose restrictions of 20% and 90% on the same resource respectively.

4.3.4.3 Overlap resolution

Allowing multiple authorised entities to define policies may result in overlaps, when policies defined by different entities refer to the same resource and apply to the same user. Lou applies a restriction on network bandwidth usage on Sergei by remote users to 20%, while Indy guarantees 70% of the same resource to good customers and restricts access by each bad customer to 2%.

In Randy's case, where a bad customer happens to be remote as well, then a default, hard-coded policy, defining that the minimum of the restrictions imposed by Indy and Lou should be taken, is sufficient. However, if a good customer, like Uma, happens to be connected to a remote network, which of the two constraints should apply to her — or both? Or maybe a new constraint, granting access to the average of 20% and 70%, should be applied? The decision is related to the real-world relationships of the entities involved, it is not trivial and here, unlike RBAC, a simple default policy such as "choose minimum" is inadequate.

The proposed role-based resource management framework allows entities to define explicitly how overlaps should be resolved, by declaring *constraint relationships*, as described in Section 4.3.3.4. The set of active constraints is checked against the constraint relationships, and sets of overlapping constraints are replaced by single constraints and resolved — operation 3 in Figure 4.7.

In the example, a constraint relationship has been defined — in Section 4.3.3.4, which states that in overlaps between the restriction imposed by `Remote` and the reservation attempted by `Good`, the two constraints should be replaced by a new one restricting access as dictated by `Remote`, if the round-trip time between the user and the server is greater than two seconds. Assuming that this applies to Uma, and that a similar constraint relationship is defined to resolve overlaps with the `NotOwner` role, in the end Uma will only be subject to the restriction imposed by `Remote`.

The algorithm that resolves overlaps, given a set of overlapping constraints and a set of constraint relationships, works as follows. The algorithm first derives the sets of overlapping constraints by looking for active constraints attempting to impose different reservations or limitations on access to the same resource, for each resource. Then, for each of these sets, the algorithm uses any constraint relationships that are applicable to resolve overlaps, until there are no more sets of overlapping constraints. Situations where none or more than one constraint relationships are applicable are discussed later.

When this stage is finished, a set of unambiguous *policy rules* — a set of non-overlapping constraints applicable to the user in question, in accordance with the terminology introduced in [WSS+01] — is produced.

Conflicting constraint relationships or unresolvable overlaps. If more than one constraint relationships are applicable to a set of overlapping constraints, and the replacement constraints proposed by the applicable constraint relation-

ships differ, then there is a *conflict* between constraint relationships; there is no single replacement constraint to substitute a set of overlapping constraints.

It is possible to devise several techniques that would automatically select one of the constraint relationships that conflict, such as choosing one at random or ordering entities that define constraint relationships in a hierarchy and selecting the one defined by the more important entity. For the benefit of simplicity, the approach taken is to either select the one that imposes the minimum reservation or restriction or — if that cannot be clearly determined — request manual intervention. Similar actions are taken if an overlap is *not resolvable* because no constraint relationship is applicable on the set of overlapping constraints.

The policy on how to apportion resources of a machine — expressed using roles and constraints — may change often, as it may depend on a number of frequently-changing parameters, such as current load, current arrangements between the entities involved, or the time of day. As changes in constraints and roles may be frequent, overlaps may be frequent too, and having an automatic mechanism to resolve such overlaps provides important benefits.

On the other hand, it is anticipated that defining constraint relationships will be a significantly *less frequent* operation than defining constraints. The "meta-policy" on how constraint overlaps are to be resolved — as expressed by constraint relationships — is inherently more permanent. The decision on what to do when Lou's limitations overlap with Indy's reservations depends on long-term relationships between the entities involved. Constraint relationships denote the "order of authority" of the entities in each particular overlapping case; it is not anticipated that these relationships will be changing often enough to require an automated way of dealing with potential conflicts.

4.3.4.4 Admission control

The admission control module checks the current resource availability and usage, and the set of unambiguous policy rules, to determine the *maximum allowed allocation* that can be made on each resource for the user in question — operation 4 in Figure 4.7. If the attempted allocation does not request more than the maximum allowed allocation, the request is *granted*. Otherwise, the system *proposes* an alternative allocation, up to the maximum amount allowed.

A user request may be *denied* if no policy rules exist for the resource in question, and the system's default behaviour is to deny access to resources if not explicitly allowed by a policy rule. In order to specify the general default behaviour of the system, a simple parameter can be set — on a per server basis — to specify either that access to resources can be allowed if not explicitly prohibited by a rule, or prohibited if not explicitly permitted by a rule.

In the example, if there is enough network bandwidth available, and if the current total usage by members of the Remote group is lower than 20% — the limit set by the constraint that was the outcome of the overlapping resolution stage — then Uma is allocated the resource she requested. Otherwise, the system proposes an alternative allocation that results to the group using up to as much as the maximum allocation allows. In Randy's case, the system does not accept the attempted allocation and counterproposes an allocation of up to 2% of the network bandwidth, in accordance with the policy rule associated with the `Bad` role.

Admission control decisions take *timing* of policies into account. If a policy rule (based on which a positive admission control decision has been made, and a session created) has an expiration time, the session that has been created is assigned an expiration time identical to that of the rule. At that point, a new session creation request needs to be issued, leading to a new evaluation.

This "admission control – session creation – session expiration" cycle does not necessarily need to involve the user. High-level services, such as resource agents or brokers, can be developed that will receive long-term resource allocation requests from users and automatically apply for re-evaluations on their behalf.

4.3.4.5 Asynchronous revocation

In most cases, resources allocated at the admission control stage are available to the client throughout the duration of a session or until they run out — if a fixed amount is purchased, e.g. "10 minutes of CPU time". However, there can be situations where it is necessary to restrict access to resources immediately, for instance if a credit card is reported to be stolen or a user is abusing server resources to perform illegal activities, such as denial of service attacks.

To allow entities to deal with unexpected situations, RBRM provides an *asynchronous revocation* mechanism. This is accessible by authorised entities, sus-

pends access to a specific resource by a particular client or role, and adds a role entry condition that places the client in question in a `Blacklisted` role. To avoid future allocation of resources of that kind to that client, a constraint is associated with the `Blacklisted` role, restricting the allocation of resources to members of that role to 0%.

4.3.5 Policy deployment

In a policy-based resource management system, policies need to reach the *policy decision points* and *policy enforcement points*. A policy decision point is where the policies are evaluated and the results of this evaluation are used for enforcing the policies. A policy enforcement point is the entity that makes sure the policy decision made by the policy decision point is enforced — for example, a QoS-aware scheduler.

As policies are only applied locally on the servers on which they are declared, the *policy domain* of each policy is defined as the set of servers on which the policy is declared. In the settings introduced in Chapters 2 and 3, policy domains may be established through out-of-band mechanisms. For instance, in the XenoServers case, the registration of a XenoServer with a given XenoCorp may involve the server agreeing to accept policy definitions from that XenoCorp.

In the following sections I demonstrate how the proposed role-based resource management framework can be used and deployed in the XenoServer Open platform and Condor. I chose to integrate RBRM with Condor because it is a popular distributed deployment platform that facilitates some degree of basic policy-based resource management, as described in Section 2.4.2.

4.3.5.1 The XenoServer Open Platform

Resource management policies in the XenoServer platform can be deployed as follows. Each XenoServer runs an RBRM module, which allows entities to declare resource allocation policies using roles, constraints, and constraint relationships, as discussed in Section 4.3.2. While entities are authenticated and authorised, no other control is enforced on the policies they define, which users they elect in which roles, or what constraints they associate with these roles. A XenoServer's owner may choose which entities' policy elements to take into account and how much, by defining appropriate constraints and constraint relationships.

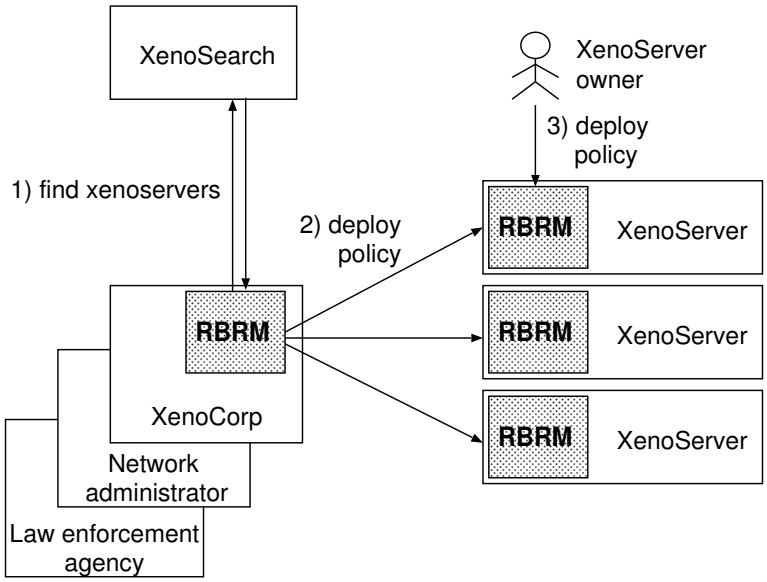

Figure 4.8: Example policy deployment case in the XenoServer Platform. XenoCorp's RBRM module uses XenoSearch to locate the servers on which the policy is to be applied and then deploys it directly

Agreeing to endorse policy elements declared by XenoCorp may be part of the contractual agreement that a XenoServer enters when it registers. For instance, Sergei may have to accept Indy's authority to authenticate users — barring the ones that are blacklisted by Sergei. Assuming the existence of a `NotBlacklisted`() role, this can be expressed as:

$$\frac{\texttt{Indy} \rightarrow \texttt{Sergei} : \texttt{Authenticated}() \quad \texttt{Sergei} \rightarrow \texttt{Sergei} : \texttt{NotBlacklisted}()}{\texttt{Sergei} \rightarrow \texttt{Sergei} : \texttt{Authenticated}()}$$

The RBRM modules running on the entities that wish to deploy policies contact XenoSearch services to discover servers on which the policies are to be deployed — operation 1 in Figure 4.8 — and then proceed to deploy policies directly — operation 2 in Figure 4.8. XenoServer owners may also define resource management policies as well as authorisation rules — endorsing constraints and constraint relationships declared by other entities — by contacting their XenoServers' RBRM modules directly — operation 3 in Figure 4.8.

The constraint relationships defined on each server specify how these federated policies are to be combined. Policies are evaluated locally on each server by

the RBRM module, and enforced by the lower-level resource management infrastructure provided for resource reservation and isolation between environments, as described in Chapter 5.

Local control. Individual servers are given significant freedom, as they substantiate both the *policy decision* and *policy enforcement* points. Sergei evaluates locally the set of policies deployed by himself, Indy and Lou, and it is the result of Sergei's evaluation of these policies that is used by Sergei to enforce a resource allocation scheme. There is no way to control that Sergei's policy decision will be in accordance with the policies deployed by Indy and Lou, nor is there a mechanism to make sure that an accurate decision will be enforced properly by Sergei's underlying enforcement mechanisms, such as the scheduler.

Giving control on which policies to accept and how to enforce them to the individual servers may seem to compromise the consistency and uniformity of the platform's behaviour; some servers may wish to ignore policies defined by the infrastructural authority or a local administrator. Sergei may define a constraint relationship that effectively neutralises restrictions imposed by Indy.

However, this is not really a compromise but rather a realisation of how federated systems are organised. Systems that encompass entities owned by a variety of organisations, and administered by a number of unrelated individuals are inherently not centrally controllable. Even if a central policy management system is used that attempts to prevent server owners from influencing policies defined on their machines, they can, for instance, subvert the CPU scheduler and achieve any apportioning of resources they wish, if the cost of doing so is relatively small compared to the potential gain [SPM04]. Node owners ultimately have total control of their machines, as they have exclusive physical console access, and there is little that can be done about it — in the extreme case, signed operating systems or tamper-proof hardware may need to be used.

Even if controlling servers centrally is technically possible, I believe that it limits the openness, scalability and manageability of the platform, and effectively impairs any chance of long-term sustainability. The operation of federated systems is based on mutual *trust relationships* formed between the participants. Sergei trusts Indy for defining policies on which users should be considered as authenticated and on which users should get privileged service, and enforces these policies. Indy trusts Sergei for enforcing his policies and providing reasonable service, and allows him to participate in the platform.

The approach I take is based on a combination of *logging* and *auditing*, and *reputation management*. Servers are granted full control over their resources, but client and server activity is recorded. XenoCorp can resolve disputes by examining the corresponding audit trails, and may take action when a node is found to be seriously non-conformant, such as ejecting it from the platform or even initiating legal proceedings. At the same time, external reputation management services [DKHP03] can be used to allow servers and clients to express their opinions about other clients and servers, based on their past interactions. Selfish or non-conformant behaviour will result in the server getting a bad reputation and therefore fewer users willing to deploy their code there and fewer infrastructural authorities willing to cooperate with it.

4.3.5.2 Condor

Condor, described in Section 2.4.2, uses `classads` as a mechanism for representing resources and resource requests, employing a matchmaking system for finding suitable machines to run jobs. Simple policies indicating which users are most eligible to claim the resources of a machine can be included in the `classad` in the form of ranking — for example, rank requests by length of job and select the shortest — and constraints — for instance, the current load average is less than 30%.

However, there are several problems that arise with respect to applying it on large-scale federated systems. For instance, there is no mechanism for supporting *federated* and potentially *overlapping* policies; no one, other than the machine's immediate owner, can influence the policies used to manage a resource. As argued previously, allowing federated policies is necessary for global scale, general-purpose public computing systems.

Moreover, there is no convenient mechanism for *grouping* users; to allow or restrict access to a resource by users belonging to a group these users need to be enumerated. Additionally, there is no way of *quantifying* access to resources; while access to a resource can be controlled, it is not possible to specify how much of a resource a user or user group should be given. Finally, as the *granularity* of the current `classads` and matchmaking scheme appears to be per entity (machine) rather than per resource, a *single policy* needs to be applied to all resources of a machine; there is no clear way to specify that different resources may be apportioned in different ways.

Most of these shortcomings can be easily mitigated by combining Condor `classads` with the role-based resource management system I propose. `Classads` provide a way for describing and matching resources and requests. The RBRM policy description and management framework supports federation of policies, overlapping resolution, and convenient and flexible grouping of users.

Let us consider the running example once more. As in Section 4.3.4, a remote user Uma, who is well-behaved and pays a high monthly fee to Indy, wishes to reserve 5% of the network bandwidth — resource NET3. As a `classad`[2], this request can be represented as:

```
{ Type                    = "resource_request";
  Provider                = "Uma";
  Quantity                = "5%";
  Constraint =
       other.Type         = "resource" &&
       other.Name         = "NET3" &&
       other.Availability = "5%" }
```

The server declares, using the following two `classads`, that remote users can only get up to 20% of the network bandwidth, while no restriction is placed for local users. Since there is no explicit distinction between limitations and reservations in Condor, a limitation needs to be expressed as a pair of `classads`:

```
{ Type                    = "resource";
  Name                    = "NET3";
  Availability            = "20%";
  Constraint =
       other.Type         = "resource_request" &&
       other.Network      != "128.232.0.0/16" }

{ Type                    = "resource";
  Name                    = "NET3";
  Availability            = "80%";
  Constraint =
       other.Type         = "resource_request" &&
       other.Network      = "128.232.0.0/16" }
```

[2]Only the parts of the classified advertisements that contain properties that are useful for exposition are shown in this section.

Whereas reserving 70% for good users and limiting bad users to 2% can be represented as:

```
{ Type                    = "resource";
  Name                    = "NET3";
  Availability            = "70%";
  Constraint =
       other.Type         = "resource_request" &&
       xenocorp.isGood(other.name); }

{ Type                    = "resource";
  Name                    = "NET3";
  Availability            = "2%";
  Constraint =
       other.Type         = "resource_request" &&
       !xenocorp.isGood(other.name); }
```

To preserve the model of interaction of Condor, where resource management policies are advertised along with the resource descriptions before a request is directed to a server, overlaps would need to be resolved in advance of submitting the `classads` to the matchmaker. This is possible in this case — the overlapping resolution scheme would, according to the constraint relationships defined, replace the above overlapping constraints with:

```
{ Type                    = "resource";
  Name                    = "NET3";
  Availability            = "20%";
  Constraint =
       other.Type         = "resource_request" &&
       other.Network      != "128.232.0.0/16" &&
       xenocorp.isGood(other.name); }

{ Type                    = "resource";
  Name                    = "NET3";
  Availability            = "70%";
  Constraint =
       other.Type         = "resource_request" &&
       other.Network      = "128.232.0.0/16" &&
       xenocorp.isGood(other.name); }
```

```
{ Type                    = "resource";
  Name                    = "NET3";
  Availability            = "5%";
  Constraint =
       other.Type         = "resource_request" &&
       other.Network      = "128.232.0.0/16" &&
       !xenocorp.isGood(other.name); }

{ Type                    = "resource";
  Name                    = "NET3";
  Availability            = "2%";
  Constraint =
       other.Type         = "resource_request" &&
       other.Network      != "128.232.0.0/16" &&
       !xenocorp.isGood(other.name); }
```

However, such *off-line resolution* of overlaps, required if the constraints are to be finalised and placed in `classads` in advance of resource advertisement, may not always work well. It complicates matchmaking, imposes potential scalability problems, and makes run-time policy changes expensive. If new roles or constraints that may overlap with some of the existing ones are declared dynamically, then existing `classads`' constraints may need to be revisited. Also, this model requires that server owners resign full control over their resources to the central matchmaker, which can favour particular matches according to its own implicit policies.

A better approach in terms of combining Condor `classads` with RBRM is to use `classads` for advertising resources — perhaps annotated with a few simple and fairly static constraints — and then *delegate* more complex policy management to the RBRM system. As some policy decisions need not be made at resource advertisement time, this approach allows for better scalability, flexibility and dynamicity.

The Condor matchmaking component can then request the higher-level RBRM system's recommendation to decide which resource to allocate to a request, as shown below.

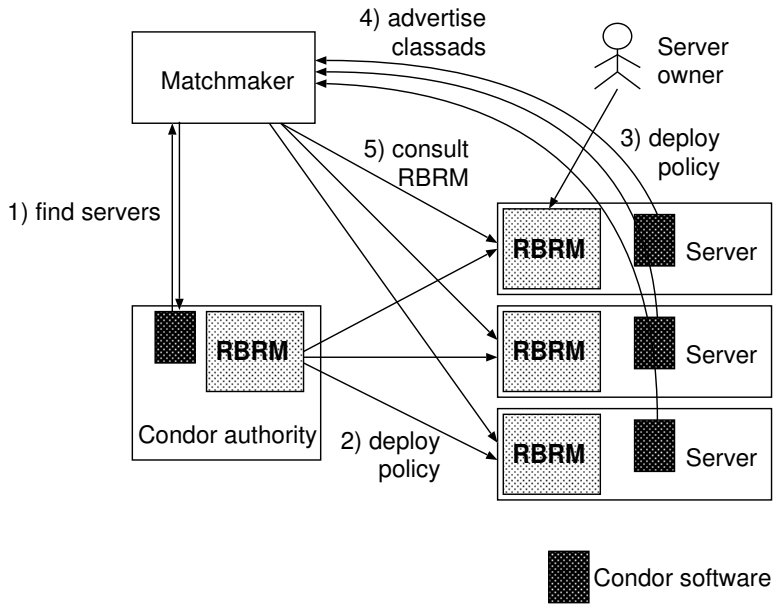

Figure 4.9: Example policy deployment case in Condor. The `classads` mechanism is combined with RBRM to support federated policies, dynamic policy decisions, and overlapping resolution

```
{ Type                    = "resource";
  Name                    = "NET3";
  Availability            = other.Quantity;
  Constraint =
       other.Type         = "resource_request" &&
       isAuthenticated    = "true" &&
       RBRM.admit(other); }
```

In Condor's setting, a suitable deployment scenario of an RBRM system would be similar to the one shown in Figure 4.9. Similar to the XenoServers deployment, RBRM modules are running on each server involved, along with the modules supporting the `classads` mechanisms.

First, the entity wishing to deploy a policy — in this case, an infrastructural authority of Condor — uses RBRM to define its policy, and contacts the matchmaker to locate servers on which the policy is applicable — operation 1 in Figure 4.9. Then the high-level RBRM policies are pushed to the individual servers involved — operation 2 in Figure 4.9. Server owners can use their local

RBRM modules to define policies on their servers conveniently — operation 3 in Figure 4.9.

Resource availability is advertised to the matchmaker using `classads`, as before, annotated with simple constraints — operation 4 in Figure 4.9. When a resource request arrives, the matchmaker evaluates the constraints defined in `classads` and then delegates more complex resource allocation decisions and overlap resolutions to the local RBRM module of the servers involved — operation 5 in Figure 4.9.

This is a suitable deployment strategy because in Condor, as in the XenoServer platform, there is a need to support decentralised resource usage policies. Constraints are independently defined by server owners and carried in the `classads` that servers submit to the matchmaker. At the same time, the proposed scheme exhibits scalability and incentive compatibility benefits, as complicated policy-based decisions need not consume computing resources on the central matchmaker, but rather on the individual servers involved. The proposed approach combines the functionality of Condor `classads` for resource description and discovery with the mechanisms for federating policies and resolving overlaps provided by RBRM.

4.3.6 Expressing realistic policies

In this section, I return to the running example and examine various realistic resource allocation policies, of the kind suggested on public computing mailing lists and fora, such as the PlanetLab architecture mailing list[3] and the GGF's Policy Research Group[4].

Limit bandwidth to X Gigabytes per month to remote users.
First, resource `NET4` representing "GB per month on a network interface" needs to be described. In accordance with the mappings introduced in Section 4.2.3, this pricing unit can be identified as `Indy%7%204`. Thus, the resource description to be defined is as follows.

[3] `arch@lists.planet-lab.org`
[4] `https://forge.gridforum.org/projects/policy-rg/`

```
{ Type          = Resource;
  Name          = NET4;
  Owner         = Indy%2%38;    # Sergei's owner
  Administrator = Indy%2%38;    # Sergei's owner
  Provider      = Indy%1%15;    # Sergei
  Kind          = Indy%6%9902;  # network interface
  PricingUnit   = Indy%7%204;   # network GB per month
  CostPerUnit   = 2;
  Availability  = 2000; }
```

Then, Lou, the local network administrator, can define the following constraint — if authorised by Sergei, limiting access to that resource by remote users:

$$\frac{\texttt{Sergei} \rightarrow \texttt{Lou} : \texttt{Remote}()}{\texttt{Lou} \rightarrow \texttt{limGrp}(\texttt{NET4}, \texttt{X})}$$

Limit bandwidth of traffic on TCP port 80 to X Gigabytes per month to all users.
Assuming that the resource `NET5` represents "GB per month on TCP port 80" (can be declared, as above), Lou can define the policy:

$$\frac{* \rightarrow *}{\texttt{Lou} \rightarrow \texttt{limGrp}(\texttt{NET5}, \texttt{X})}$$

Allow dedicated use of only Y IP addresses by each local user.
Using the *Local* role as defined earlier, and assuming $\texttt{IPv4} - \texttt{FULL}$ refers to a resource denoting an IP address with its full port range, Sergei can use the constraint:

$$\frac{\texttt{Sergei} \rightarrow \texttt{Lou} : \texttt{Local}()}{\texttt{Sergei} \rightarrow \texttt{limEach}(\texttt{IPv4} - \texttt{FULL}, \texttt{Y})}$$

Guarantee bandwidth to X% of total bandwidth to users of a particular XenoCorp named FooCorp.
First, a new role needs to be defined that groups together users of FooCorp. The role is declared by Indy, but FooCorp is the one that elects users to the role — as FooCorp knows who its subscribers are:

$$\frac{\texttt{User}.\mathit{XenoCorp} \ = \texttt{FooCorp}}{\texttt{FooCorp} \rightarrow \texttt{Indy} : \texttt{FooCorpUsers}()}$$

Secondly, Sergei associates the role with a constraint definition, guaranteeing access to X% of the bandwidth:

$$\frac{\texttt{FooCorp} \rightarrow \texttt{Indy} : \texttt{FooCorpUsers}()}{\texttt{Sergei} \rightarrow \texttt{rsvGrp}(\texttt{NET3}, \texttt{X}\%)}$$

Reserve X% of CPU for users connected to the local network.
Assuming that `CPU1` is a resource referring to the CPU, Sergei can use the `Local` role as defined earlier and the following constraint definition to reserve the CPU proportion required:

$$\frac{\texttt{Sergei} \rightarrow \texttt{Lou} : \texttt{Local}()}{\texttt{Sergei} \rightarrow \texttt{rsvGrp}(\texttt{CPU1}, \texttt{X}\%)}$$

4.4 Related work

In this section I discuss previous research in the area of role-based access control. Ferraiolo and Kuhn [FK92] outlined the ideas of RBAC and provided a formal description of role definition and membership. Several role-based systems were devised over the subsequent ten years. [NO95] provides a framework for the administration of roles and access rights, and focuses on the organisation of roles by allowing the explicit declaration of relationships between them. [LMSY96] defines roles as sets of rights and duties, which is similar to RBRM's distinction of roles from constraints. Relationships between roles are considered, as well as meta-policies for resolving conflicts.

[JD96] combines roles and policies applied by different sources to assemble a global layer for the interoperability of heterogeneous databases. [HBM98] takes it even further by escaping from the "central authority" model and understanding the challenges imposed by applying RBAC to open, large-scale systems, while also providing a flexible and comprehensible role description language.

A generalised version of RBAC is proposed in [CMA00]. This approach goes beyond the common subject-centric approach to role management, by allowing object-centric or environment-centric policies to be defined. Object-centric roles are similar to RBRM's concept of constraint definitions, as presented earlier in this chapter.

The Ponder language [DDLS01] provides a means of specifying security policies associated with roles, allowing the declaration of positive and negative authorisation policies, as well as meta-policies for conflict resolution and role inheritance. The problem of conflict resolution in the context of role-based access control has been explored [LS99, RT04]. The RT framework [LMW02] combines role-based access control with hard security trust management, for efficient access control in large-scale systems.

The proposed role-based resource management framework draws on some of the techniques developed in role-based access control, but differs fundamentally in its use of *quantitative policies*, its emphasis on *federated control*, and the introduction of constraint relationships to control the way in which *overlapping policies* are combined.

4.5 Summary

Open public computing platforms are anticipated to comprise highly diverse sets of servers, in terms of specifications, hardware, and performance. Moreover, the ways in which different servers may wish to sell their resources, and the pricing schemes they may wish to apply accordingly, are practically unlimited. This chapter has presented mechanisms to allow servers to independently *name* and *describe* their resources and the pricing schemes associated with each. A *coordination* scheme has been proposed for achieving naming consistency of common resource descriptions.

This chapter has presented a *role-based resource management* framework to allow expressing policies on how resources on servers are to be apportioned between different users or user groups. Using an example, it has been demonstrated that the assumption of central control is untenable; XenoCorp, server owners, network administrators, and other authorised stakeholders need to influence resource allocation decisions. To support that, *policy federation* has been proposed, allowing entities to maintain subjective views of role memberships and declare policy

elements on servers. As interests of stakeholders may not always be compatible, mechanisms for automatic *overlapping resolution* have been developed.

Resource allocation requests, submitted by users to XenoServers, are *evaluated* against the declared resource management policies. The operations carried out and the algorithms employed in each of the stages of policy evaluation have been described. The operation of *deploying* policies, involving transferring a policy to all servers that it applies on, has been discussed in two practical example environments: the XenoServer Open Platform and Condor.

To investigate the effectiveness of the proposed resource management framework, policy requirements have been collected from mailing lists and discussion fora of distributed deployment platforms. RBRM has been examined with respect to its applicability to provide solutions for expressing *realistic policies*, and shown to be successful. Finally, facilities provided by role-based access control have been discussed, and RBRM positioned in the context of related research.

The following chapter departs from the platform design, and analyses the prototype implementation of the XenoServer platform. It focuses on the internal structure of components, discusses issues presented while building the infrastructure, and presents mechanisms developed for efficient global-scale service deployment.

Chapter 5

Implementation

Previous chapters have offered solutions to design challenges for global public computing, proposing a distributed infrastructure that allows non-cooperative users to deploy untrusted distributed services on servers around the world in exchange for money. Accounting and charging mechanisms supported by a trusted third party have been proposed for supporting non-cooperative users and server owners. An open and extensible server selection infrastructure has been designed to assist resource-oriented and location-oriented server selection. Finally, a role-based resource management framework has been devised to allow independent servers to flexibly and extensibly describe their resource availability, support coordination of common resources' descriptions, allow federated stakeholders to define policies on resource allocation, and provide the facility for dynamically combining those policies for flexible admission control.

However, important implementation challenges remain; the platform needs to provide support for easy, efficient, and quick *global-scale service deployment*. Launching services on large numbers of servers around the world would incur the transfer of potentially prohibitive amounts of data, which needs to be avoided. The platform components need to be implemented in an *incrementally scalable* way, to allow high client and XenoServer participation. Low-level mechanisms are needed for supporting *resource reservations, protection,* and *isolation* between the different untrusted and non-cooperative distributed services. Execution environments that allow the deployment of unmodified, *out-of-the-box* distributed applications are required.

This chapter describes the prototype implementation and deployment of the XenoServer Open Platform and proposes solutions to the corresponding chal-

lenges. Wherever possible, existing software components, such as Virtual Machine Monitors, distributed file systems, or infrastructures for authentication and secure communication, have been used. This has a number of advantages, such as higher software reliability and maturity, faster development, and easier upgradeability.

5.1 Component implementation

In this section, I describe the internal prototype implementation of the various entities that compose the XenoServer Open Platform for global public computing. As explained in Chapter 3, the development of alternative implementations of the platform's components is not only allowed but encouraged.

Prototype components have been developed in Java 1.4.2 and use RMI for execution of remote methods. Java has been chosen because it supports portability of code, allowing components to run in a variety of computing platforms without requiring recompilation. RMI has been selected because it allows faster development and helps maintaining code simplicity.

5.1.1 XenoServer

The architecture of prototype XenoServers is based on the *Xen* Virtual Machine Monitor [BDF+03a, BDF+03b]. Xen has been a collaborative project in the Systems Research Group, Computer Laboratory, University of Cambridge, and does not constitute part of this dissertation.

It securely divides the resources of a machine among a set of resource-isolated Virtual Machines (VMs) running software on behalf of users. A special *Management Virtual Machine* (MVM) is used for the administration and control of the XenoServer. The architecture of the prototype XenoServer is shown in Figure 5.1.

The owner of a Virtual Machine has complete freedom to select the guestOSs of his or her choice, and run any applications on it. For performance benefits Xen does not fully virtualise commodity x86 hardware. Operating systems ported to a modified *x86-Xen* architecture are used. Porting an operating system to run on Xen typically involves replacing privileged processor instructions with explicit calls to Xen, and using new network and block-device drivers to perform I/O via a virtualised interface. In most cases, it is not required to modify anything other

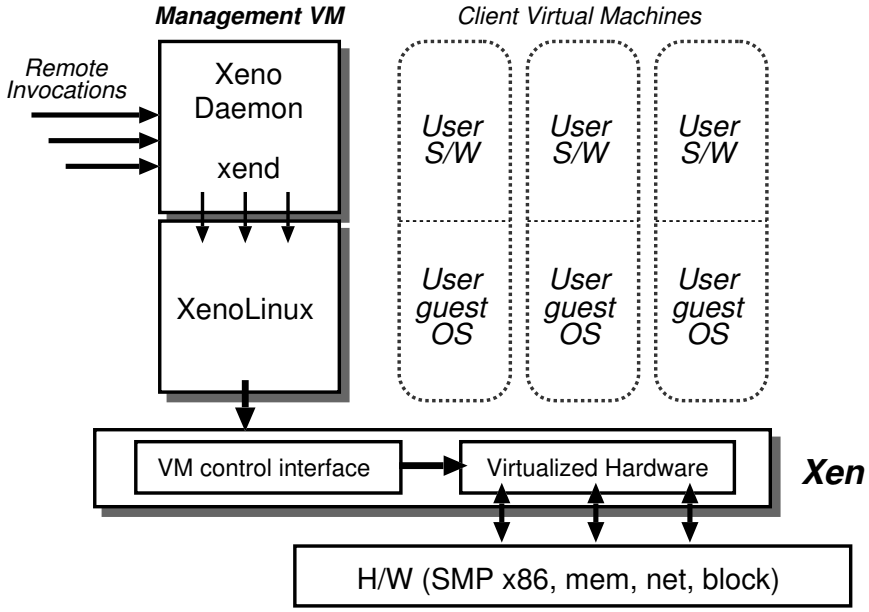

Figure 5.1: Architecture of a Xen-based XenoServer

than architecture-dependent portions of the operating system. Applications run *out-of-the-box* without recompilation on the modified guestOSs.

5.1.1.1 VM Control Interface

The internal architecture of Xen is described in detail in [BDF+03a], along with the protection and isolation it facilitates. The purpose of this section is to describe the interfaces exported by Xen for VM management and the way these are used by the higher-level tools that carry out communication with other platform components. Aside of that, the approach that this chapter adopts towards Xen is a black-box one.

The Management Virtual Machine (MVM) is booted at the start of the day and allows the owner of a XenoServer to create and manage VMs via the *VM control interface* (VCI). All configuration and control is performed via this interface, and all policy decisions, such as admission control and yield management, are made by software running within the MVM. Xen itself is responsible only for the *mechanisms* of facilitating resource multiplexing between VMs.

The VCI is a rather low-level interface, comprising a set of privileged entry points into Xen that may be invoked from a suitably privileged Virtual Machine. The administrative interface takes the form of a set of user-space control tools (called `xend`) that use a set of appropriate calls to the VCI to achieve some higher-level goal. Broadly speaking, VCI calls fall into one of the following three categories:

- **Virtual Machine Management:** Allows Virtual Machines to be created, started, stopped and destroyed, and to have the contents of their memory inspected or modified. These functions are used by higher-level control software such as the XenoDaemon — to be described in the next section — to boot guest operating systems, and implement VM suspend/resume and dynamic migration.
- **Virtual Machine Configuration:** Enables querying and setting of allocation and scheduling parameters for CPU, memory, disk and network. These are typically set at session creation — prior to starting a particular VM for the first time. While it is technically possible that they are adjusted dynamically, it does not fit well with the session-based model, where a session is an agreement for the provision of a particular set of resources at a specific pricing scheme. Higher-level resource descriptions, such as the ones proposed in Chapter 4, are translated to specific scheduling parameters passed to Xen.
- **Virtual Device Configuration:** Allows the administrator to configure virtual block devices — dynamic partitions — and virtual network interfaces.

The VCI is accessible only by Virtual Machines that are created with special privileges — such as the MVM; by default, only certain introspection aspects of the interface are accessible to other (client) VMs.

5.1.1.2 XenoDaemon

The `xend` tools are sufficient for local XenoServer administration, but do not allow interaction with the rest of the XenoServer platform. This role is handled by *XenoDaemon*, a network daemon process that runs within the MVM. *XenoDaemon* is responsible for interfacing with both clients and XenoCorp, and its internal architecture is shown in Figure 5.2.

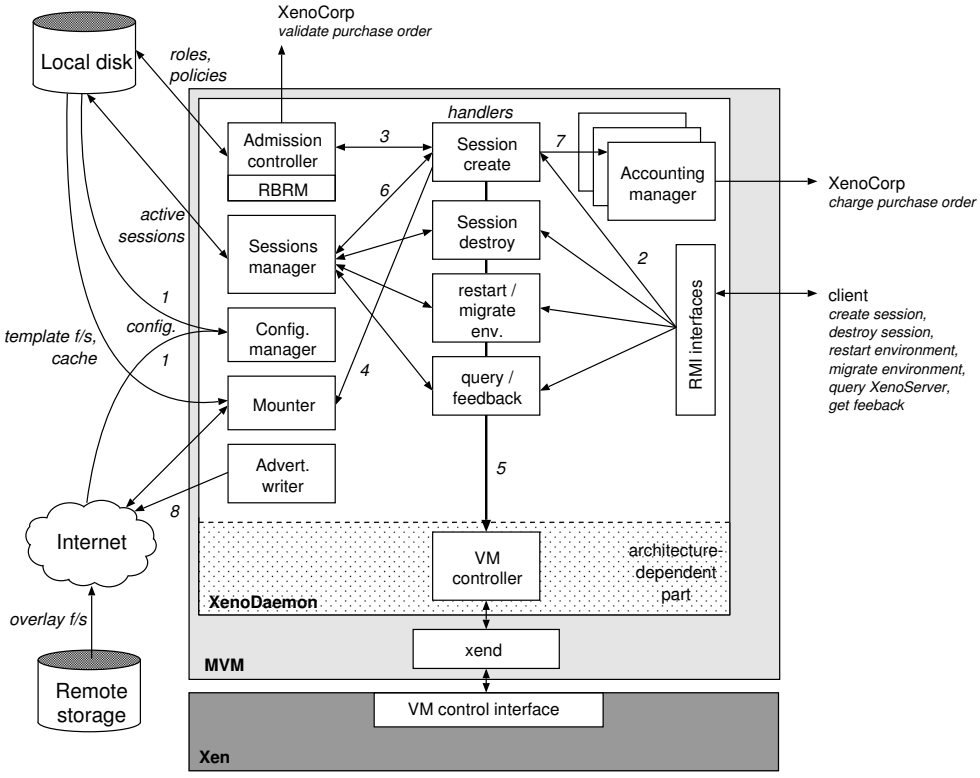

Figure 5.2: Control-plane architecture of a XenoServer

At the start of the day, the *configuration manager* module of the XenoDaemon is called; this loads configuration information from the storage location that has been chosen to store that information, such as a local hard disk, a web site, or the server's XenoStore location — operation 1 in Figure 5.2. It reads configuration parameters such as the server's own ID and credentials, the location where server advertisements — described later — are to be stored, and the location where audit trails are to be saved. It also reads the list of resources initially available on the server, which can be defined by the server operator using external tools provided.

The manager also reads the list of XenoCorps with which the server is registered, and the parameters of the server's agreement with each one, such as the purchase order validation and charging policy, the authentication infrastructure to be used, and the way the server is to receive payments from each XenoCorp — for instance, proportional to resource consumption, fixed monthly payment, or a proportion of the XenoCorp's profit.

The XenoDaemon allows remote clients to place requests, subject to *authentication*; this is performed ahead of the interactions described here using digital certificates, as explained in Section 5.1.3.

A user requests creation of a session — i.e. deployment of a Virtual Machine — on a particular XenoServer by using a client program to invoke the RMI interfaces exported by XenoDaemon. To do so, a *deployment specification* and a valid *purchase order* need to be submitted to the XenoServer. A deployment specification describes which operating system kernel image and file system to use when booting a Virtual Machine, and is accompanied by a set of resources to be reserved, which describes the required amount of CPU time, physical memory, and so on. A session creation request may also specify that, instead of creating a new VM, a previously suspended (to a file) VM should be *resumed*.

At this point, a *handler* thread is created to serve the session creation request — operation 2 in Figure 5.2. Using new threads to serve requests by other components of the platform allows for the operation of XenoDaemon to continue while the request is served.

The handler for a session creation in turn requests the help of the *admission controller* module to decide whether to agree to provide the requested resources — operation 3 in Figure 5.2. The admission controller performs *purchase order validation* and *admission control* at this point.

The controller may proceed — according to the arrangement between the XenoServer and its affiliated XenoCorp — with validating the purchase order with XenoCorp. If the order is valid, the controller invokes a prototype version of the role-based resource management (RBRM) framework described in Chapter 4 to decide whether the requested resources can be allocated. According to the policies defined by the federated stakeholders, RBRM makes a positive or negative recommendation regarding the allocation of the requested resources. External tools have been developed for declaring and deploying RBRM policies.

The file system to be used as the root file system of the guestOS to be launched may be stored locally on the XenoServer, fetched from a remote storage location, or may be an overlay comprising both local and remote elements — as described in more detail in Section 5.2. Provided that the admission controller's recommendation is positive, the handler requests the help of the *mounter* module to mount the remote and local storage locations as required — operation 4 in Figure 5.2. This decodes the URLs it is given and, according to the scheme portions — such as `nfs://` or `afs://` — determines how mounting is to be performed.

The handler then calls the *VM controller* to translate deployment specifications to low-level deployment parameters — operation 5 in Figure 5.2 — and determine whether a new VM is to be created, or a suspended one is to be resumed. The controller then uses `xend`, a user-space tool that provides interface wrappers around the low-level VCI, to communicate with Xen and have a VM created, configured and started. The new session's specifications and resource reservations are stored in the *sessions manager*, which keeps track of all live sessions and handles persistent storage of that information to the disk — operation 6 in Figure 5.2.

Clients receive console output during boot and throughout the operation of the VMs associated with the sessions they have created; subsequently, users interact with their VMs in whichever way they wish, such as via SSH if they have started the appropriate server-side daemon, in order to start, configure, and manage tasks that are running on the VM. In guestOS execution environments, they may also specify the operations to be performed for their tasks to be started in boot-time scripts — such as the `/etc/rc.d` scripts in *NIX guestOSs (e.g. Unix, Linux, BSD).

XenoDaemon also creates an *accounting manager* thread for each VM started, and schedules its periodic execution — operation 7 in Figure 5.2. While a VM is running, the accounting manager periodically checks and accounts for resource consumption, records activity in audit trails, and sends the corresponding accounting and billing information back to XenoCorp. Clients, if so configured, may also receive updates regarding the resource usage and charges incurred by their VMs.

When a client requests the *destruction* of a session, the corresponding execution environment is terminated, and the resource reservations released. If so requested, the VM's image can be *suspended* — saved to a file at a user-defined location, such as a virtual disk on the XenoServer, a private storage server of the client, or a XenoStore location. This allows for the VM to be *resumed* at a later stage using a quick, lightweight operation.

Clients can also submit requests to *get feedback* about existing sessions and invoke management operations, such as to *restart*, or *migrate* execution environments. Xen's support for live migration of execution environments is explained in detail in [CFH+04]. A client uses the *session handle* he or she has been given at session creation to reference a session and request management operations on the VM associated with it.

The *advertisement writer* module of XenoDaemon is part of the server selection and resource discovery infrastructure. Based on initial resource availability read by the configuration manager, and current resource usage, it periodically produces a *server advertisement* and stores it in a user-defined location, such as a web or XenoStore location — operation 8 in Figure 5.2. An advertisement is a digest of the server's current status, location, resource availability, and pricing schemes as described in Chapter 4. The XIS and XenoSearch services can then collect advertisements and allow clients to perform searches on those, thus increasing a XenoServer's exposure to its customer base.

The existence of two levels of *indirection* — handlers and managers or VM controllers — serves the important goal of the platform openness. Techniques specific to particular XenoDaemon implementations, such as the way persistent storage is handled (files, databases of a particular type, or XenoStore objects), how XenoServer configuration is represented, or how VMs are configured, started, and managed (using the low-level control interfaces) can be altered easily by replacing the controller and managers, without requiring changes to the rest of the XenoDaemon software.

5.1.2 XenoClient

The software running on users' machines is termed *XenoClient*. It is responsible for communicating with XenoCorp for registration and purchase order creation, with the XIS and XenoSearch for server discovery and selection, and with XenoServers for service deployment and management. This section describes the implementation of the prototype XenoClient and its interface. Its architecture is shown in Figure 5.3.

There are no architectural restrictions mandating that the prototype version of XenoClient is exclusively used. Alternative client applications can be developed and used for communication with the other components of the XenoServer platform, as long as they comply with the interfaces defined in Chapter 3.

At the start of the day, the *configuration manager* retrieves settings and coordination information, such as XenoCorps that the client is registered with, and the XIS or XenoSearch services that may be used — operation 1 in Figure 5.3.

The prototype XenoClient provides a graphical interface for user convenience. The client is designed to allow *disconnected operation*. For each operation that

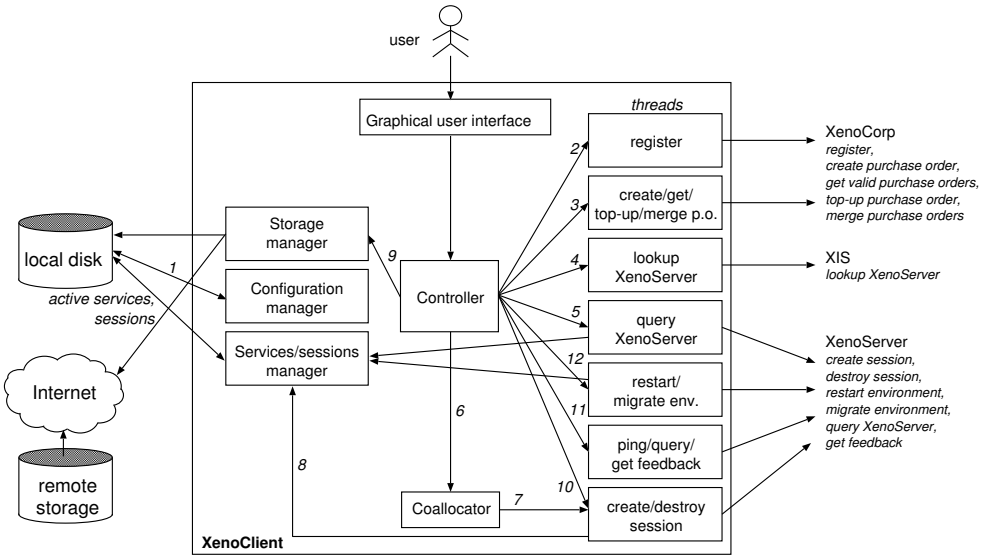

Figure 5.3: Architecture of XenoClient

users request, which involves interaction with other components of the XenoServer platform, a new thread is launched to handle it; this allows for the uninterrupted operation of the graphical environment, and permits running several operations concurrently.

The first operation a user typically undertakes is that of *registering* with XenoCorp — operation 2 in Figure 5.3; the interface provided to clients for performing this process is shown in Figure 5.4. Registration is carried out the first time that XenoClient is launched, and after that every time the user wishes to join a new XenoCorp's domain.

Before deploying services, the user needs to *create a purchase order* with XenoCorp — operation 3 in Figure 5.3. In a wide-area deployment, this would ring-fence an amount of money in the users' bank account or credit card to be used for funding resource consumption on XenoServers. As the prototype implementation is not linked to a real credit card or direct debit system, the amount is reserved from a credit balance that each user has on XenoCorp.

When creating a purchase order a user may also specify additional *constraints*, for instance to require that the order is tied to a particular XenoServer, or to set the frequency at which the order is charged by the server — inside a predefined range.

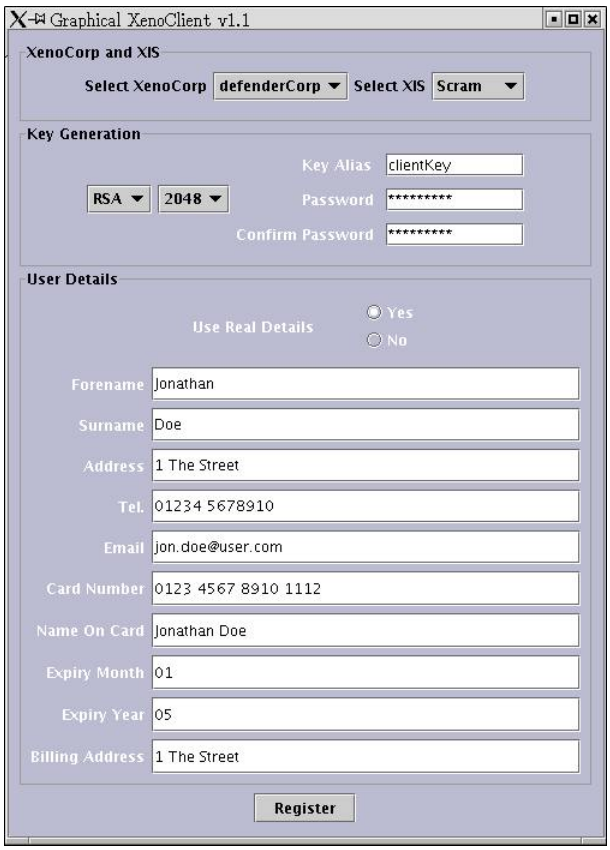

Figure 5.4: Interface for user registration

The user can request a list of his or her *valid purchase orders* from XenoCorp, *top-up* an existing order, and *merge orders* to add one order's funds to another to reduce fragmentation of funds. The interface for creating and managing purchase orders is shown in Figure 5.5.

The next step is typically *server selection*; this may include server discovery, which can be carried out using the XIS directly, or through one of the available XenoSearch components — operation 4 in Figure 5.3. Server selection does not necessarily require server discovery though; the user can select servers directly from the cache of recently used XenoServers that XenoClient maintains, or just type in the host names of XenoServers already known to the user. The interface for server discovery and selection is shown in Figure 5.6.

When servers are selected, the user can proceed with selecting the *resource reservations* he or she wishes to make on the servers; this ensures that the resources required to run the service to be deployed are available on the XenoServers

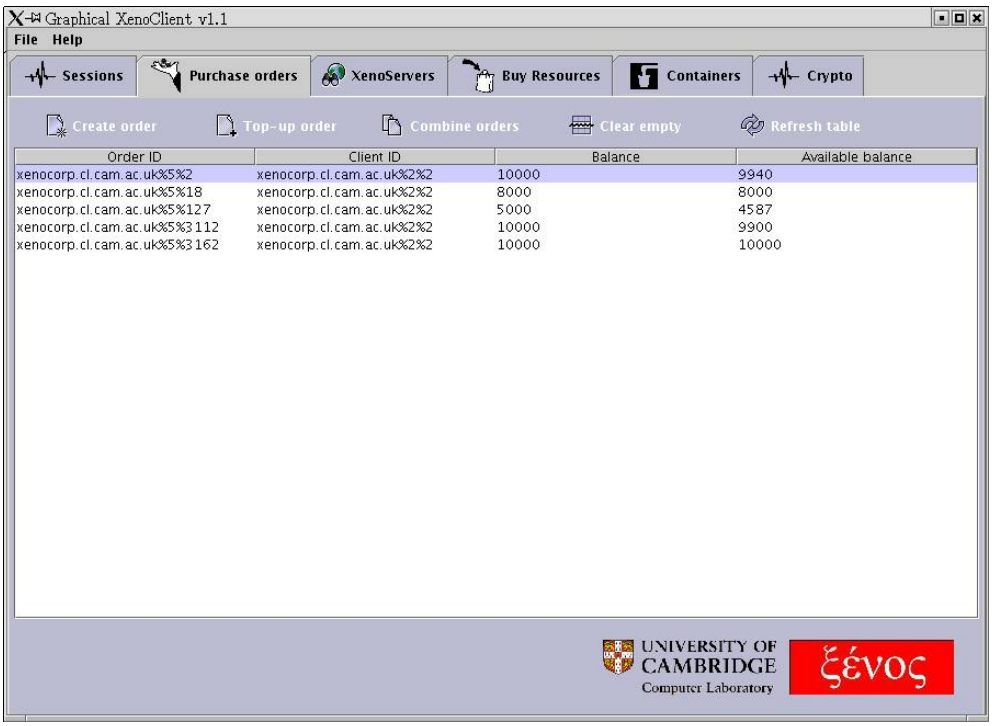

Figure 5.5: Interface for purchase order creation and management

throughout the course of its execution. The interface for resource reservations is shown in Figure 5.7. XenoClient automatically *queries* the selected XenoServers — operation 5 in Figure 5.3 — to retrieve up-to-date status and resource availability information, subsequently displayed in the table on the left in Figure 5.7. For user convenience, if more than one server is selected at a time, XenoClient can automatically determine and display the *common resource availability* to the user — i.e. the maximum set of resources available on all selected XenoServers.

Resources are internally represented as described in Section 4.2.1, but displayed to the user in human-readable form. The translation is performed using the resource kinds and pricing units maps provided by XenoCorp at registration, as described in the same chapter.

XenoClient comes with a number of template *resource envelopes*, which are lists of resources required for the execution of some common services — such as an Apache web server or a Quake3 game server. Generic envelopes are also provided, for instance allowing the execution of different types of average-sized distributed applications requiring memory, CPU time, and network connectivity. More fine-grained resource envelopes for specific applications can be calculated

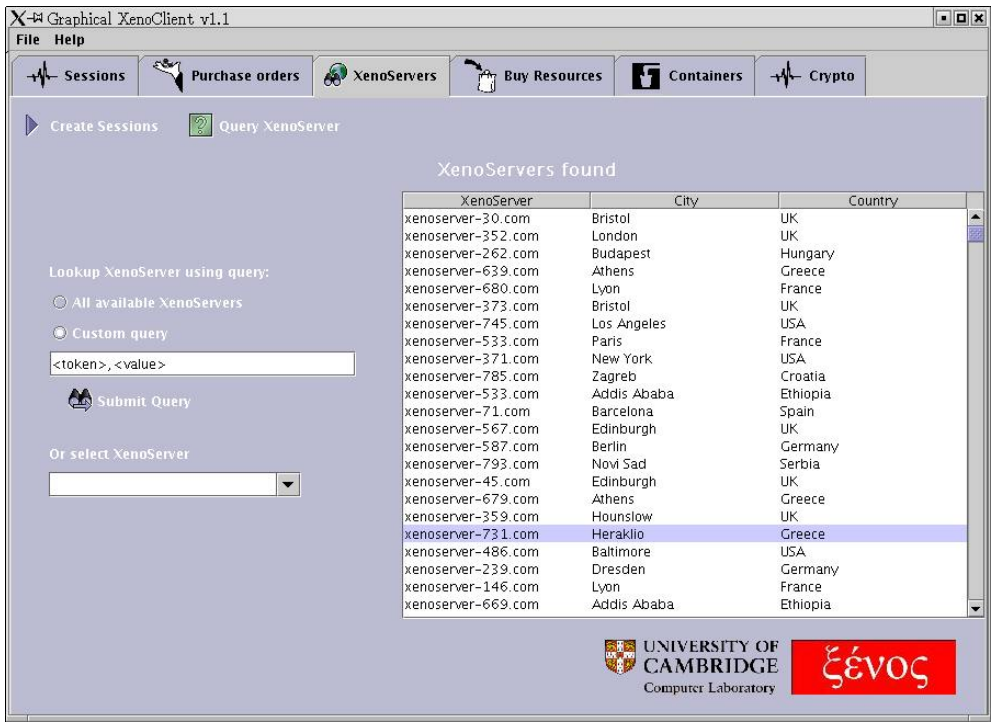

Figure 5.6: Interface for XenoServer discovery and selection

using techniques under development at the time of writing by the XenoServer project team, which do not constitute part of this work.

Resource envelopes can be loaded, altered — in terms of resources to be reserved and quantity of each resource to be requested — and saved to the disk for later use. The list of resources to be purchased is shown on the right, together with resource pricing information and the estimated costs.

When the list of resources to be purchased is finalised, the user can request for *sessions* to be created on the selected XenoServers — operation 6 in Figure 5.3. At this point XenoClient negotiates with XenoServers, proceeds with *coallocating* resources on all selected XenoServers — operation 7 in Figure 5.3 — if so requested, and informs the user about the success or failure of the process. Information about the newly created sessions is passed on to the *sessions and services manager* — operation 8 in Figure 5.3, which handles permanent storage of the client's state. Storing information about the currently active services and sessions on disk is necessary to allow the client to continue operating when XenoServers are not reachable and to recover from a fault or a software restart.

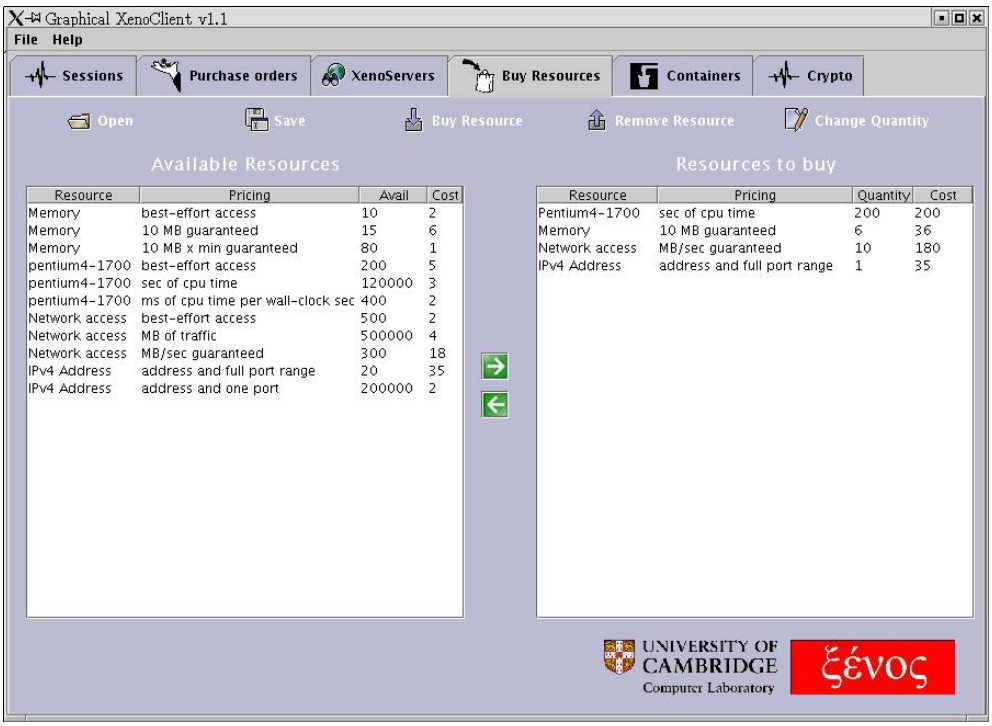

Figure 5.7: Interface for purchasing resources on a XenoServer

When requesting the creation of a session, the client needs to build a *deployment specification*, selecting the root file system and type of execution environment required — for instance, a VM on Xen, the guestOS kernel to be used — such as XenoLinux 2.4.26. The user can also choose whether overlaying functionality is to be used. If so, the user selects the overlay and persistently cached reference file system to be used — as described in Section 5.2. The user can also select whether a new VM is required, or whether one that has been previously suspended to a file is to be resumed. The interface for building deployment specifications is shown in Figure 5.8.

Although it is not necessary that the XenoClient is used for that purpose, facility is provided for managing external storage, such as local disks, XenoStore, AFS [SS96], or NFS directories, web locations, and so on — operation 9 in Figure 5.3. This allows the user to easily transfer the data required to deploy services wherever is convenient, and perform simple data management operations.

XenoClient provides an interface for monitoring and managing sessions and distributed *services* — sets of sessions on one or more XenoServers, shown in Figure 5.9. On request of a session *destruction* — operation 10 in Figure 5.3,

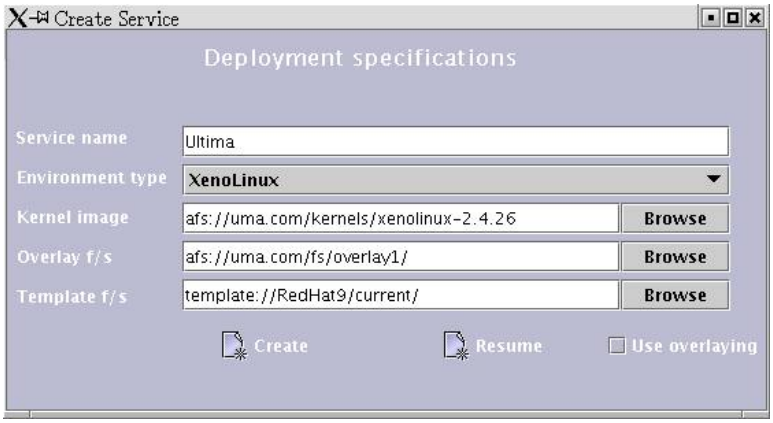

Figure 5.8: Interface for building deployment specifications

the XenoServer stops the tasks that are running on the corresponding VM, shuts down the VM, terminates resource reservations, and submits the final payment claim to XenoCorp for resources consumed by that session. If so requested by the user, the VM's image is stored to a file, to allow resumption at a later stage.

Requesting *feedback* — operation 11 in Figure 5.3 — returns up-to-date information about a session's current status — alive, dead, or unknown — and its resource consumption, which is displayed in detail in the table at the bottom in Figure 5.9. Resource pricing details and current charges are also shown. *Querying* a server returns information about its current resource availability and pricing. *Restarting* execution environments is a useful feature in cases of environment failure, or when a new operating system kernel is to be booted in an existing session. *Migration* of environments allows for the VM to be relocated to another session on the same or a different XenoServer — operation 12 in Figure 5.3.

5.1.3 XenoCorp

XenoCorp is the trusted third party that handles authentication and registration of clients and servers, billing, and payments. As XenoCorp is envisaged to be run, administered, and maintained by expert users or qualified administrators rather than home users, a graphical user interface has not been developed as part of the prototype implementation. The internal architecture of XenoCorp is shown in Figure 5.10. Communication between XenoCorp and other components of the platform is done through the RMI interfaces exported.

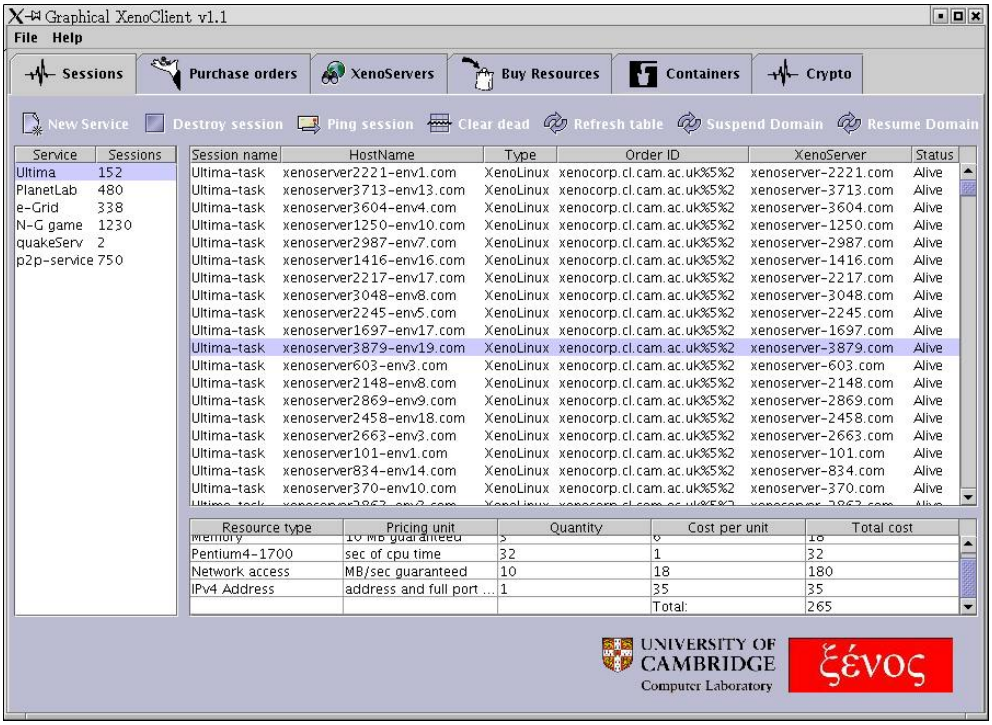

Figure 5.9: Interface for service and session management

Clients and XenoServers *register* with XenoCorp to join the platform. After that, clients can *create*, *renew*, and *get* their *valid purchase orders*. XenoServers contact XenoCorp for *validating* purchase orders or requesting *charging* on resource consumption — accompanying the request with billing details. As the prototype implementation is not linked to a real credit card or direct debit system, charges for resource consumption are simply deducted from purchase orders. Creation of a purchase order deducts an amount from the credit balance that a user has on XenoCorp.

When any of these RMI interfaces is called, XenoCorp invokes an appropriate *handler* thread to carry out the request — operation 1 in Figure 5.10.

Servers may also specify at registration time a location where they wish to receive *configuration* and *coordination* information, such as the list of available resource kinds and pricing units in this XenoCorp's domain. XenoCorp places the requested data at a remote location of the XenoServer's choice — which may be a directory on a XenoStore server, its own AFS server, or a publicly accessible web location — operation 2 in Figure 5.10. XenoCorp also stores the Advertisement Locations Catalogue (ALC) at a storage location of its choice.

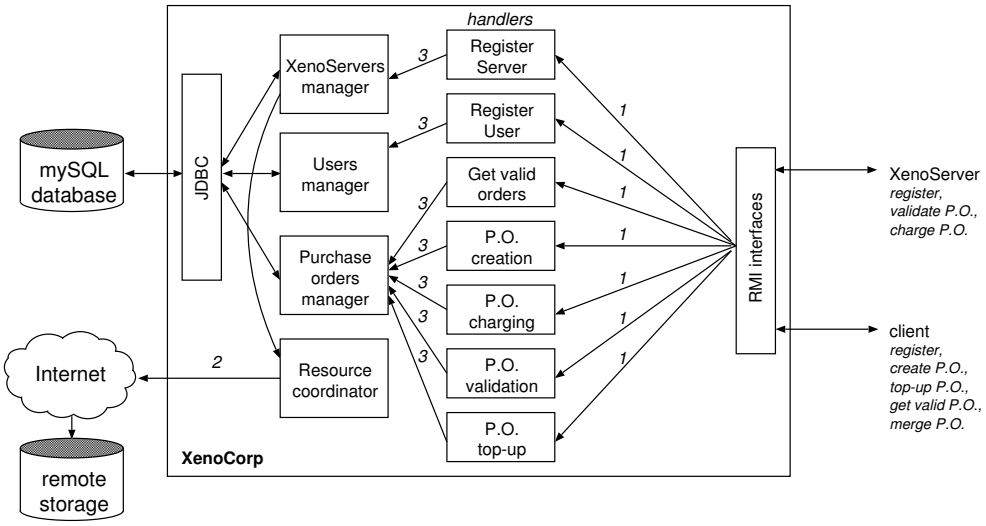

Figure 5.10: Architecture of XenoCorp

Permanent storage of information about registered clients and XenoServers, purchase orders, as well as charging and payment history, is handled by *managers*. In the prototype implementation, information is stored in a MySQL database, with which managers communicate using a JDBC driver. Managers are contacted — operation 3 in Figure 5.10 — to insert or delete entries from the database, to retrieve information on behalf of the handlers, and to administer run-time state — such as caching entries in memory data structures for performance.

As before, the existence of two levels of *indirection* — handlers and managers — enhances the platform's openness; XenoCorp can easily be ported to run on a different type of database or database communication driver, as this only involves changing the manager modules.

Alternative implementations of XenoCorp may be distributed; by splitting its functionality between several distributed components, and running each component on a different XenoServer, the scalability and fault-tolerance of XenoCorp can be improved. However, as this is the trusted third party that handles authentication, and charging, distributing it over more than one machine complicates trust relationships. For simplicity, a centralised prototype implementation of XenoCorp has been chosen; incremental scalability can still be achieved by allowing the coexistence of multiple XenoCorps.

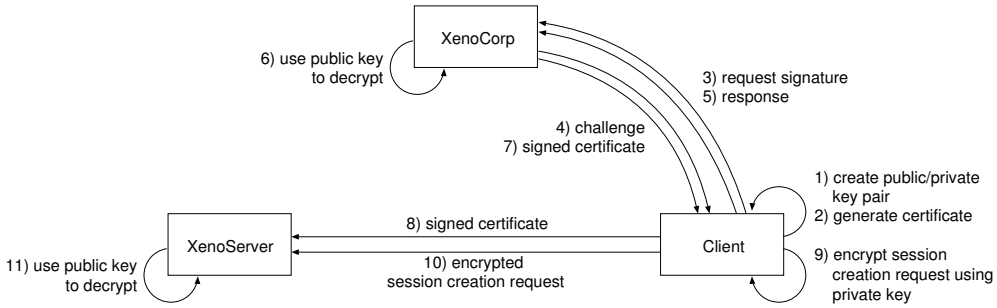

Figure 5.11: Authentication in the prototype XenoServer platform

Authentication. Authentication of XenoCorp to clients and XenoServers takes place using digital certificates on which XenoCorp has obtained a signature from a well-known certification authority. Authentication of clients to XenoServers is achieved using XenoCorp as a certification authority, as shown in Figure 5.11 — authentication of XenoServers to clients is carried out similarly. First, a client generates a public/private key pair, and creates a digital certificate that includes its public key and unique client identifier — operations 1 and 2 in Figure 5.11.

Then, the client requests to be registered with XenoCorp — operation 3 in Figure 5.11; this involves XenoCorp signing the client's certificate, as a form of endorsing its credentials. Authentication of clients (and XenoServers) to XenoCorp when they first contact it to register uses a challenge/response mechanism; XenoCorp asks the invoker to encrypt a random message using its private key — operation 4 in Figure 5.11. The client returns the encrypted result — operation 5 in Figure 5.11, and XenoCorp checks whether the encrypted result can be decrypted properly using the invoker's public key — operation 6 in Figure 5.11.

Upon completion of the challenge/response phase, XenoCorp signs, stores, and returns the client's certificate — operation 7 in Figure 5.11. The signed certificate is passed on to a XenoServer on which the client wishes to create sessions — operation 8 in Figure 5.11. The client then submits a session creation request, which is encrypted using its private key — operations 9 and 10 in Figure 5.11; the XenoServer uses the client's public key, contained in the certificate, to decrypt the session creation request — operation 11 in Figure 5.11. If the request can be decrypted successfully then the server can be sure that the client is who it claims to be. At the same time, the deployment specification and purchase order that a client submits are non-repudiable, as they are encrypted using the private key that only the client possesses.

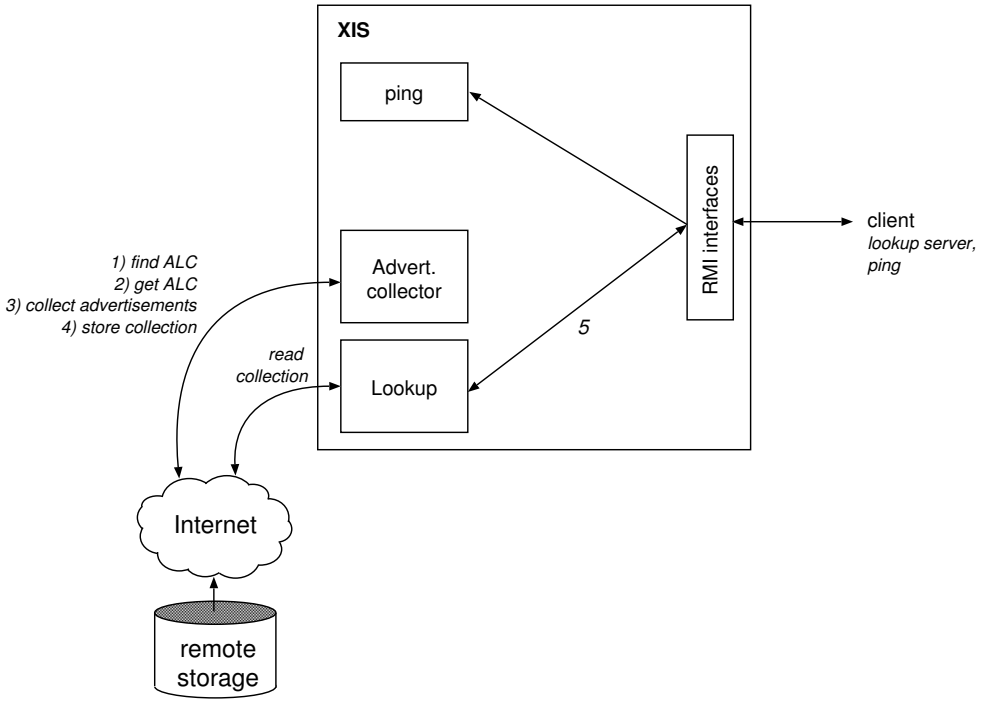

Figure 5.12: Architecture of a XenoServer Information Service node

5.1.4 XenoServer Information Service

The XIS provides a server advertisement aggregation and lookup service; it periodically collects advertisements from individual XenoServers' storage locations and stores them in a structured, searchable manner. Basic lookup functionality that allows searching for advertisements that present values inside a predefined range for a specific token is provided. Its internal structure is shown in Figure 5.12.

Initially, the XIS reads from a well-known web server the storage location where the Advertisement Locations Catalogue (ALC) can be found, as described in Chapter 3 — operation 1 in Figure 5.12. This is a list maintained by XenoCorp, and may be advertised on a web server or stored in a remote storage location.

The *advertisement collector* thread is invoked periodically, reads the ALC — operation 2 in Figure 5.12, and collects all fresh advertisements from the locations specified — operation 3 in Figure 5.12. It then stores advertisements in another storage location in a structured manner — operation 4 in Figure 5.12. Old advertisements, or ones that are not properly signed by the advertisers, are

ignored. The XIS exports a `lookup` interface to clients and returns the list of servers matching the queries — operation 5 in Figure 5.12.

This *indirection* is important: servers and XenoCorp respectively do not have to be queried every time their advertisements or ALC are to be retrieved; instead, they periodically perform a low-cost push operation to storage locations from where these can be accessed by clients or search systems. Moreover, the ALC and advertisements can be replicated on demand to balance load.

For better load distribution, performance and scalability, the XIS is deployed as a *replicated* rather than single, central service; there are several instances of the XIS running simultaneously, termed *nodes*, which themselves may be deployed on XenoServers, with each one providing the same facility of indexing and retrieval of server advertisements.

5.1.5 Storage

In the prototype implementation of the XenoServer Open Platform, an external XenoStore storage system is provided. The difference between XenoStore and other types of external storage, such as private NFS or web servers maintained by individual XenoServers or clients, is that the former is *trusted* by XenoServers and clients; it is run and administered by reputable organisations on well-maintained machines, and has long-term presence in the infrastructure.

XenoStore is not required for service deployment, and is not part of the core infrastructure of the XenoServer Open Platform. Clients and XenoServers can use alternative storage locations of their choice. Storage space is useful for:

- **Automatic configuration.** XenoCorp can automatically push configuration parameters to clients' and servers' XenoStore quotas or other storage locations.
- **Resource advertisement.** XenoServers can periodically store their digests of status and resource availability in their XenoStore quotas or other storage locations, and the XIS or XenoSearch services can pull these from there.
- **Resource description coordination.** XenoCorp pushes the list of common resource types and pricing units, and mappings to their human-readable names to XenoServers' and clients' storage locations.

- **Efficient deployment.** As analysed in more detail in 5.2, storage provides users a convenient and efficient mechanism for parallel deployment of services to large numbers of servers around the world.

For providing a simple prototype XenoStore service the distributed Andrew File System (AFS) has been chosen, since its design makes it more suitable for a wide-area deployment compared to other mature off-the-shelf distributed file systems, such as NFS. AFS has been designed for wide-area scalability [Sat92], and its interaction model allows relatively efficient bulk data transfer even when the round-trip delay is high, as in realistic wide-area deployment scenarios. At the same time, its persistent caching mechanism provides greatly increased performance relative to NFS when used in the wide-area.

Using AFS is sufficient to validate this approach and, as shown in Chapter 6, performs more than adequately well. The XenoServer team is, at the time of writing, looking to implement XenoStore as an entirely new distributed file system (Xest), incorporating ideas from SFS [MKKW99], Pond [REG+03] and Pasta [MPH02] to produce a file system which seamlessly combines the 'push' model of the distribution templates with the 'pull' model of demand caching. The design and development of Xest does not constitute part of my work or of this dissertation.

To provide secure communication between XenoStore and the XenoServers and clients that are using it secure Virtual Private Networks based on the Internet Protocol Security (IPSEC) [KA98, IBM98] are used. This creates a permanent safe IP-layer link between XenoStore and XenoServers, and ensures all information sent on it is properly encrypted to protect clients and servers from eavesdropping.

AFS uses Kerberos [NT94] to handle authentication. Clients and XenoServers are initially authenticated on XenoStore using their digital certificates signed by XenoCorp. Then they are provided with their Kerberos ticket that they can use to access their data from that point on. Note that different implementations of XenoStore services can follow different authentication approaches.

This dissertation focuses on building the core XenoServer platform. It has not been a purpose of this work to develop a production-quality implementation of XenoStore or any other high-level, third-party service. The sole purpose of using the simple XenoStore service described here is to demonstrate the benefits of external storage for server discovery and service deployment. Mechanisms

that realistic XenoStore services should comprise, such as quota management, accounting, and perhaps charging for usage of storage resources, have not been investigated or implemented.

5.1.6 XenoSearch

It is necessary that XenoSearch implementations can scale to large numbers of XenoServers and users scattered around the globe. Although a semi-centralised solution might be possible — like the one employed by google[1], it is sensible to aim for a distributed one, provided that communication between search nodes can be minimised, and no central point of failure is introduced. Additionally, XenoSearch is anticipated to be collecting updates about XenoServers' current resource availability much more frequently than web content is updated.

Considering these factors, the XenoServer team has designed and implemented *XenoSearch II* [SHH04]. This replaces *XenoSearch I* [SH03] and provides a distributed topologically-aware search service that supports complex queries, which combine spatial (location) constraints with multi-dimensional resource availability requirements. XenoSearch II comprises a set of approximately 100 nodes distributed around the globe, which periodically harvest resource availability information from the XIS or from individual XenoServers' storage locations. Each XenoSearch II node holds at least imprecise information about the resource usage and location of each XenoServer, and so can produce an approximate query result independently of any other node.

The goal of XenoSearch II is not to obtain a single "best" result for a given user's query. The system is designed to produce a set of results, which are presented to the user in an order determined by a heuristic, much like a page-ranking system in a web-search engine. However given that information may be out of date, and the client's utility function is unknown, the choice is ultimately left to the end user.

The implementation of the prototype XenoSearch II component has not been part of my work, and is not part of this dissertation. It is described in more detail in [SHH04].

[1] `http://www.google.com`

5.2 Service deployment

This section focuses on *service deployment*; this is the step where users, after having selected a number of XenoServers on which the service is to be deployed, proceed to contact the XenoServers to configure and start the Virtual Machines that accommodate the service components, and to launch the service components themselves.

Here, I describe deployment models followed by other distributed deployment platforms, discuss why global public computing needs more than the conventional ad hoc means used to deploy services in the wide-area, and propose an effective solution to address these problems. Work on service deployment mechanisms for the XenoServer Open Platform has been presented in [KMP+04].

5.2.1 Other deployment models

Distributed deployment infrastructures comprise *deployment models* — solutions for storing and transferring the data required to deploy distributed services to the servers involved — that are often adequate for the needs of the environments they are designed to serve, but unsuitable for general-purpose global-scale service deployment.

Denali [WSG02] and Grid computing projects [FK97, TTL04, FGK04] all follow the same model, where users have to transfer the data required for service deployment to every server involved, and configure the machines individually. Grid services are deployed using the APIs provided by the Grid infrastructure, which usually employs mechanisms such as GridFTP [LGT+01] for data distribution. The configuration and execution of the services on each individual server can be done either manually, or with the help of an automated tool. This represents a *push* approach in service deployment.

The PlanetLab project, as described in Section 2.4.4, only offers basic support for service distribution, requiring users to connect over SSH to each node individually to copy, configure, and control the custom service, a process that may be tedious when deploying to hundreds of nodes. More recently, the CoDeploy[2] service has been developed. This considerably eases the task of distributing experimental software to a set of PlanetLab nodes, and operates efficiently by using

[2] `http://codeen.cs.princeton.edu/codeploy/`

the CoDeen [WPP02] CDN. It is not aimed at distributing operating system kernels or entire file system images, however, where higher commonality of files — most users request a few popular operating systems' kernels and distribution file systems — allows for persistent caching on the servers. Our system exploits this to allow efficient and parallel deployment, and mobility of services.

System imaging is a technique that enables archiving and copying disk images, usually containing operating system distributions and applications. Images can be used to clone servers by automating image deployment and configuration. Partition Image[3] generates disk images and uses domain-specific data compression techniques, while Frisbee [HSL+03] employs local-area multicast for efficient distribution of images in local networks.

Imaging focuses on the replication of entire disks' — and sometimes memory — contents to other machines in the local network for ease of configuration. The proposed system is different in that it aims at global-scale data distribution at deployment time and support for per node configuration, parallel deployment and Virtual Machine migration.

VMatrix [AR02] follows a similar concept to that of disk imaging, facilitating the imaging of the run-time state of the machine along with files on the disk, and distributes such "Virtual Machines" on servers for easier configuration. The Internet Suspend/Resume project [KS02] allows users to capture and transfer the state of machines through the network. It targets the movement of a single Virtual Machine between two points, and does not address parallel deployment or per node customisation.

5.2.2 Deployment requirements

In global public computing deployment models followed by other distributed platforms are inadequate due to the following challenging requirements:

- **Ease of deployment:** It is necessary that the cost of deploying large-scale distributed services on XenoServers remains low, both in terms of money and effort. Offering users mechanisms to "configure once, deploy anywhere" is necessary; after preparing their VM configurations, launching services on large numbers of servers should be trivial.

[3] `http://www.partimage.org`

- **Efficiency:** To provide users full and flexible control of the configuration of their Virtual Machines, each new VM is specified from the ground up in terms of a file system image and kernel. In a naive deployment model, this would incur transfers of several gigabytes to each selected XenoServer for each service deployment, and would raise the cost of deployment to potentially prohibitive heights.

- **Support for migration:** Services are likely to be location sensitive, meaning that service instances may need to be migrated as search tools determine better deployment positions within the network according to service-specific criteria. For instance, it may be decided that migrating a service to a new network location may reduce the total round-trip time between the service component and its clients, or the hosting costs. For performance and convenience, it is necessary that the deployment architecture allows services to move around at a low cost. The platform must not require that large amounts of service data be transferred to the new deployment position at every service migration.

- **Parallel deployment:** Location-sensitive services may be widely replicated and deployed simultaneously on large numbers of machines around the world. Transferring large volumes of data required to launch a complex service to large numbers of remote servers incurs significant overhead in terms of time, as long-haul network transfers can be slow. It also requires more funds for the additional storage space required for storing the service replicas on each server involved. Furthermore, it increases costs incurred by international traffic. At the same time, configuration of each replica individually must be allowed, for the required customisation to be supported.

5.2.3 Deployment in global public computing

The deployment model proposed in this dissertation is a *pull* one; instead of mandating that users transfer data required to deploy a service to the servers involved, servers pull that data from a location specified by the client whenever that is needed. In order to avoid the transfer of several gigabytes for each deployment, *overlaying* techniques are used; stock distributions of commodity operating systems that most clients are anticipated to use are persistently cached at XenoServers as *template images*. Users describe *tailored images* in terms of modifications to these templates, called *overlays*. This greatly reduces the amount

of data that must be shipped to a XenoServer during deployment, reducing the setup time for a new Virtual Machine. This further enables dynamic replication or migration of services to proactively respond to offered load.

Although template images are stored locally at the XenoServer, client overlays are remotely accessed across the network. This extra level of indirection means that clients may configure their overlay *independently of where their VM is instantiated*. Since the overlay is remote, it may be shared between multiple VMs running on a set of XenoServers — for example, to facilitate replicated server instances — and easily accessed by migrating services. Several layers of stacking are also supported, hence allowing per service instance customisations and writable directories as required. Example customisations might include new start-up scripts, SSL certificates, or software packages.

5.2.4 Deployment configurations

The prototype platform implementation supports two different configurations for deployment, in terms of where an overlay is to be fetched from. Both use a copy-on-write (CoW) stacking file system to merge a read-only template file system image stored on the XenoServer's local disk and an overlay file system stored remotely, though in slightly different fashions.

The first configuration, shown in Figure 5.13, assumes a XenoStore service that is trusted by XenoServers and clients. Mounting inside the privileged Management Virtual Machine (MVM) a storage location that is exported by an untrusted remote server can be risky, as disconnected operation is not entirely supported, thus server failure — or malicious servers — may cause problems to client operation in several distributed file system protocols such as NFS or AFS [HH93, She99]. Since XenoStore is trusted, it is reasonable for the MVM to use client-provided credentials to mount parts of their storage area.

In this configuration, the template file system, fetched from the local disk — operation 1 in Figure 5.13, is merged with the overlay, fetched from the XenoStore server — operation 2 in Figure 5.13, by a copy-on-write NFS server[4]. The NFS server then re-exports the resulting, overlaid file system to VMs — operation 3 in Figure 5.13, which boot from `NFS-root` over the machine-local virtual network. A CoW NFS server was chosen as NFS is supported by a wide range of guestOSs.

[4]The copy-on-write NFS server used was developed by the XenoServer team. Its design and implementation do not constitute part of this dissertation.

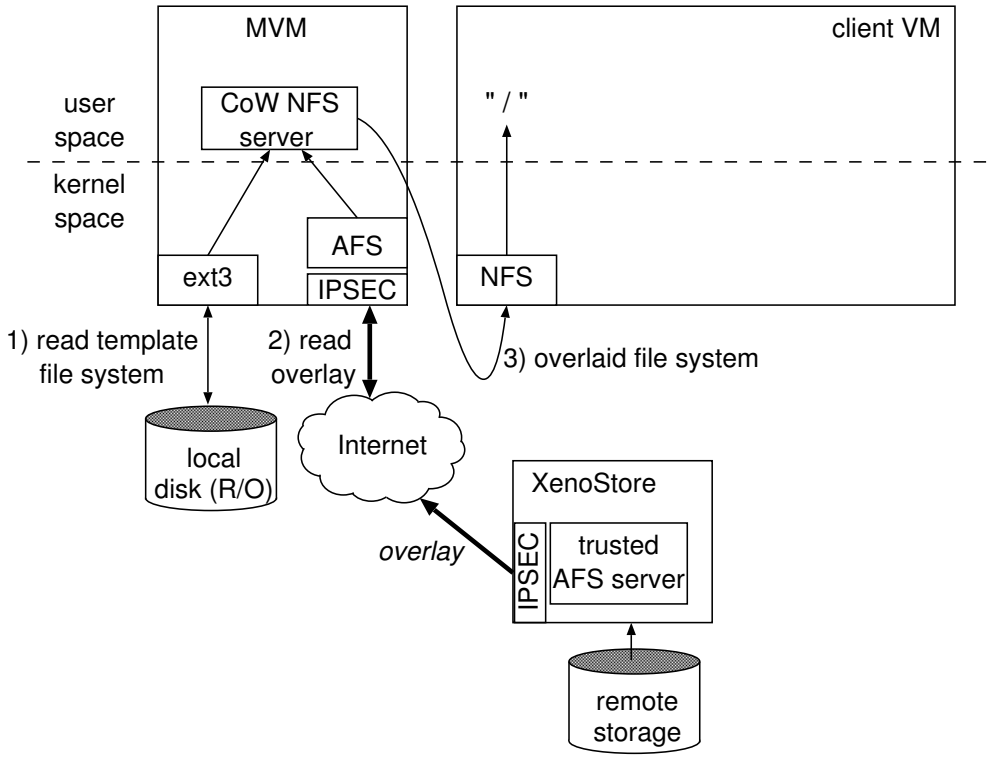

Figure 5.13: Service deployment from XenoStore

The second configuration, shown in Figure 5.14, does not require XenoStore: clients implement the copy-on-write functionality within their VMs directly. They use a copy-on-write component that again merges the template file system stored locally — operation 1 in Figure 5.14 — with the overlay stored in the untrusted storage — operation 2 in Figure 5.14. They can employ their own copy-on-write NFS server to re-export the overlaid file system to themselves — operation 3 in Figure 5.14. In the case of Linux guests, an implementation of a block-level copy-on-write device that may achieve better performance can be used instead of the CoW NFS server. This allows for untrusted storage locations to be mounted, as they can only affect the operation of unprivileged, client VMs.

Each of the XenoStore and private storage configurations have their benefits and drawbacks. XenoStore services run on *well-provisioned* and *well-connected* servers, and so access latency should be low and data availability high. However it does require that users "buy in" to XenoStore. Potential performance benefits from sharing template file systems within the buffer cache of the MVM are counterbalanced by Xen's capabilities for inter-VM buffer cache sharing. The private

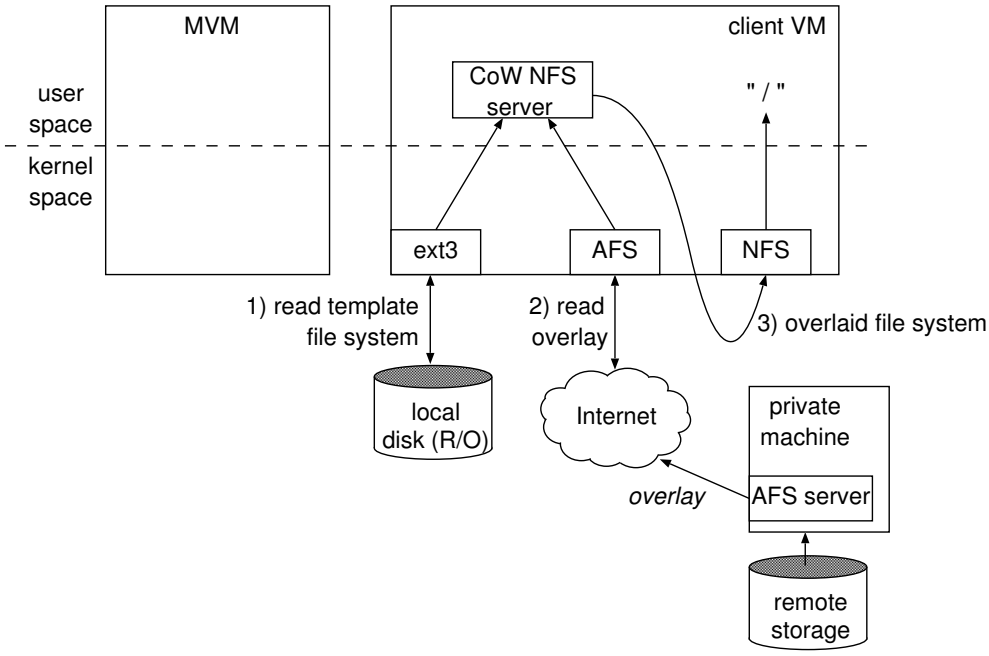

Figure 5.14: Service deployment from private remote storage

storage approach has a potentially *lower barrier to entry*; any user with a file server may remotely access this to populate their overlay file system. However it requires some additional complexity at VM boot time, as overlaying has to be done while the VM is booting — discussed in more detail in the next section.

In both configurations, Virtual Machines are able to lease *virtual disks* allocated on sections of the XenoServer's local disk drives, using the interfaces exported by Xen. Virtual disks may be employed, for instance, to provide fast predictable access to a local copy of a file system or database, or to implement a large file system cache to improve the performance of a network file system.

5.2.5 Prototype implementation

A key component is the stacking CoW NFS server, designed and implemented by the XenoServer project team, which overlays template images with one or more user-provided file systems to construct the root file system for a client VM.

The CoW server interprets a `.mount` file in any directory to specify a list of file systems to overlay at that subtree; this is presented as a unified namespace

in which the order of the mounts specified determines which of two identically-named objects overrides the other, reminiscent of union directories[5] in the file system of the Plan 9 operating system [PPT+92]. Modifications are written through to the first listed writable file system on a per file copy-on-write basis.

When using the XenoStore model shown in Figure 5.13, the user-provided deployment specification, submitted by the client to the XenoServer at session creation, includes a URL identifying the overlaid file system along with a XenoStore-specific *storage access certificate* signed by the user that allows the XenoServer access to that storage for a specified period. XenoDaemon parses the URL, and uses the scheme portion — such as `nfs://` or `afs://` — to determine the file system type. It then mounts the remote overlay so that it is accessible by the MVM at a path chosen according to the new VM's identifier, and notifies the copy-on-write NFS server. This subsequently exports that locally-accessible path as `/` such that it may only be mounted by the user's Virtual Machine; the link-local address Xen assigns to each VM is unforgeable, and so is used for this purpose.

Subtrees in multiple file systems can be overlaid at any point in a path, and hence mounting may be required on-demand. The stacking file system server invokes the *mounter* module of the XenoDaemon to mount any such remote storage systems. This gives a clear separation between the manipulation of the overlaid namespace — performed by the copy-on-write file system — and the mounting of templates and remote file systems — carried out by XenoDaemon's mounter.

By convention, XenoDaemon uses the convention `template://` to name read-only operating system distribution templates. Immutable naming schemes provide a guarantee to a client using the template that the contents underneath it will not change. Mappings are also maintained from well-known names, such as `template://RedHat/9/current`, to these immutable identifiers, allowing 'default' distributions to be updated or have security patches applied. Choice of template identifier allows a client to specify the degree to which a service's file system is subject to any template maintenance process.

[5] An interesting feature of Plan 9 was union directories, allowing directories across different media or across a network to be bound to other directories transparently. For example, another computer's `/bin` directory can be bound to one's own, and then this directory will hold both local binaries and the remote binaries and can use both transparently. Under Unix, mapping directories in this fashion would make the original disappear, one "covered" the other. Using the same system, under Plan 9 external devices and resources can be bound to `/dev`, making all devices network devices without additional code (from `http://www.wikipedia.org`).

If a VM is being deployed using the private storage model any overlaying must be performed by the VM itself; in this case the VM has to mount the remote user-tailored file system, run the copy-on-write file system server, and overlay the remote file system and a local template. The VM has to perform these operations as it boots. This is achieved by generating an initial `ramdisk` image that starts the file system servers necessary to construct the root file system — including the optional creation of a persistent cache on virtual disk — and then performs a `chroot` to hand control to `/sbin/init` to let the normal boot process proceed. Tools to automatically construct an appropriate `ramdisk` are provided as part of the XenoDaemon installation package.

5.3 Summary

This chapter has focused on practical issues related to the implementation and deployment of the prototype XenoServer Open Platform. The first part of the chapter presented the internal architecture of platform components, and described how they deliver the required functionality.

The prototype XenoServer consists of two key components: the first one is the low-level *Xen Virtual Machine Monitor*, used to achieve effective protection and resource isolation of execution environments. The second one, the high-level *XenoDaemon*, handles server configuration, user authentication, session creation, configuration of Virtual Machines to be initialised, admission control, resource management, accounting, auditing, submission of payment claims, and server advertisement.

The facilities provided by the prototype *XenoClient* tool have been presented, allowing users to conveniently register with XenoCorp, create purchase orders, discover and select XenoServers, purchase resources on servers, create, monitor, and manage sessions on XenoServers, and administer storage locations. The implementation of *XenoCorp* over a MySQL database has been described. It stores information about registered clients and XenoServers and details of existing purchase orders, and provides configuration and coordination information to its affiliated servers and clients. For supporting discovery of XenoServers, a prototype *XenoServer Information Service* has been developed as a service running on the platform. It collects server advertisements and performs simple searches on those on behalf of users.

In the second part of this chapter, a *service deployment model* suitable for global public computing settings has been presented. Inadequacies of existing deployment models have been discussed, and an alternative based on *overlaying* techniques has been suggested. The proposed model reduces the amount of data to be transferred over the network, as demonstrated in the next chapter, while allowing users to launch customised execution environments on large numbers of machines around the world. Two practical implementations of the proposed model have been described.

The platform has been designed and implemented to meet the goals for global public computing, set in Chapter 2. The following chapter assesses, through experiments and discussion, the efficiency, scalability, and effectiveness of the design decisions made and the mechanisms employed.

Chapter 6

Evaluation

This section evaluates the design and mechanisms implemented for building the XenoServer Open Platform for global public computing, by carrying out two types of practical experiments on the prototype deployment of the platform. *Performance* evaluation assesses the overhead imposed by the XenoServer platform in terms of time and network traffic. *Scalability* tests measure the ability of the platform to cope with increasing numbers of participants.

The principal goal of this chapter is to demonstrate the overall *effectiveness* of the architecture's design and implemented mechanisms. Given the facility provided by the prototype XenoServer platform and the results of the performance and scalability experiments, I examine in detail how each of the design requirements that have been set in Chapter 2 is met.

6.1 Experimental setup

All experiments were performed using machines connected to the Systems Research Group's local Ethernet. XenoServers were Dell 2650 machines configured with dual 2.4GHz Xeon processors, 2GB RAM, Broadcom Tigon 3 Gigabit Ethernet NICs, and Hitachi DK32EJ 146GB 10k RPM SCSI disks. XenoCorp was a Compaq ProLiant DL360 with two 1.4GHz P4 processors, 4GB RAM, Compaq NC7780 Gigabit Ethernet NICs, and 36.4GB 10k RPM SCSI disks. The machine used as XenoStore, storing clients' overlays, was a dual processor 2.4GHz with 1GB RAM, and was running a stock Andrew File System (AFS) server. XenoSearch II was deployed on 100 PlanetLab nodes around the world, as ex-

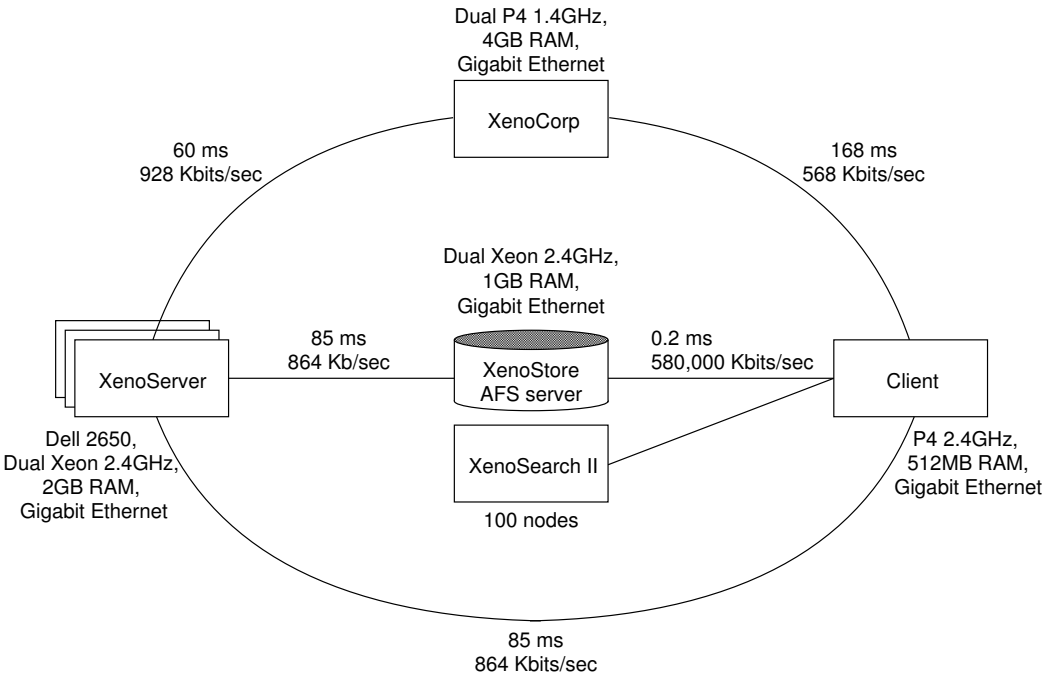

Figure 6.1: The experimental evaluation setup

plained in [SHH04]. This setup is shown in Figure 6.1, and was used in all experiments presented in this chapter.

In order to measure wide area network effects under controlled conditions, the NISTNet [DBCF95] emulator was deployed on a machine configured as an IP router. NIST allows a single PC set up as a router to emulate a wide variety of network conditions. Delay and bandwidth limits were specified as shown in Figure 6.1. This configuration was illustrative of an arrangement in which the client and the AFS server were in Cambridge, UK, the three XenoServers in New York, NY, and the XenoCorp in Berkeley, CA. Round-trip times were distributed normally with a standard deviation of 5% of the mean shown.

6.2 Performance

In this section I evaluate the process of service deployment. I focus on two applications that are representative of the types of service that are expected to be common on the XenoServer platform — an Apache web server, and a Quake 3 game server. I divide the service deployment process into the specific

Service	Overlay (KB)	Total F/S (KB)	Proportion
Apache	23,695	2,318,937	1%
Quake3	533,671	2,828,913	18.8%

Table 6.1: Size of copy-on-write overlays

steps identified in Section 3.2.3 and measure the cost of each step, using the XenoStore deployment model described in the previous chapter.

Preliminary results for service deployment using the private storage model indicated almost identical performance to that of the XenoStore deployment model. As there is no technical reason to suggest that a significant performance difference should be expected, the performance of the private storage model is not investigated further in this dissertation.

6.2.1 Overlay size

Before measuring the deployment process for the two services, *overlays* have been prepared for them using the *copy-on-write* file system server mounted loopback by a local NFS client over an immutable template.

The template Red Hat 9 file system was persistently cached on the XenoServers' local disks, and the overlays were stored and fetched from XenoStore. The copy-on-write NFS server combined overlays with template images, and exported the resulting file system to the client VMs to be launched, as described in Section 5.2.4.

Table 6.1 shows that the total size of modified files required to support the services is a *small fraction* of the total file system size. This result demonstrates the importance of the overlaying functionality; most of the data required for launching services can be persistently cached on the servers, and overlays that need to be fetched over the network are small.

Therefore, it can be expected that overlaying will provide efficiency benefits in terms of reducing network traffic and speeding up service deployment. Recent studies also confirm the significant commonality found in Linux file systems [PP04].

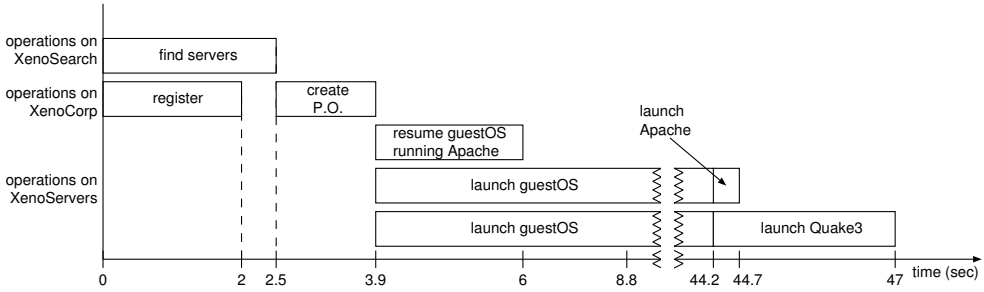

Figure 6.2: Service deployment timeline, showing the individual operations taking place and the time needed for each one

6.2.2 Deployment timeline

Measurements of the constituent phases of a service deployment, as shown in Figure 6.2, were conducted. Each experiment was repeated 100 times; times are measured as the UK user perceives operations to complete, including processing time on the user-side for submitting and receiving requests, and network latency between the components involved. Think times of the users are not measured or included in the calculations.

Before using the platform, a new user must *register* with XenoCorp to establish credentials. This takes around 2 seconds in the prototype implementation; in a real world deployment this would be dominated by credit card processing delays. The time recorded as *find servers* measures the time that XenoSearch II requires in order to answer a query to locate three XenoServers satisfying a complex resource constraint set by the user. The query returns a ranked set of suggestions meeting the criteria. Using XenoSearch II does not require that users be registered with XenoCorp; thus, registration and server selection can be done in parallel.

The next step is for the user to ask XenoCorp for *purchase order creation*. The user then builds the deployment specification, loads the corresponding resource envelope, and contacts the XenoServers directly. The XenoServers perform admission control and, upon acceptance of the job, configure and instantiate Virtual Machines (*launch guestOS* in Figure 6.2). The guest operating systems — Linux 2.4.26 in this example — boot and then deploy the target applications (*launch Apache* or *Quake3* in Figure 6.2).

Components	New	Resumed
XenoCorp - XenoServer	132 (15KB)	
User - XenoCorp	158 (19KB)	
User - XenoServer	255 (37KB)	35 (6.5KB)
User - XenoStore	25193 (25.6MB)	none
XenoServer - XenoStore	1055 (731KB)	714 (720KB)

Table 6.2: Messages exchanged during deploying a service from a new guestOS and a previously suspended one (total amount of data exchanged in parentheses)

The results of this experiment are summarised in Figure 6.2. A new user can join the XenoServer platform, locate a number of suitable XenoServers, and deploy a new web server in *less than 45 seconds* and a new Quake3 server in about *47 seconds*. Most of the time spent in service deployment is in booting the guestOS to host the new service. The timeline also shows an example of using Xen's capability to *suspend* and *resume* entire guest operating system images. By restoring an operating system image from a previously saved snapshot the application deployment time is reduced substantially, to just over *6 seconds* in this example. Most of this time is spent in registering and creating a purchase order; a previously registered user can resume sessions in just above *2 seconds*.

Given that all measured times include wide-area deployment latencies, as described in 6.1, this is a more than satisfactory result. This is partly due to the overlaying infrastructure, and partly due to AFS's caching mechanisms, both of which significantly reduce the amount of data that have to be fetched from remote locations.

6.2.3 Network traffic

Table 6.2 shows the average network traffic generated, in terms of size of data transferred and messages (packets) exchanged, from the *register* phase through to the completion of the *deploy software* stage for an Apache service. The experiment was carried out in two settings: (a) launching a completely new VM for deploying Apache and (b) resuming a previously suspended VM.

The user–XenoStore figure accounts for the transfer of the Apache overlay to the XenoStore AFS server. This operation can be performed *off-line*, thus does not delay service deployment. Furthermore, since this is only performed *once*

for each service deployment — or more rarely, if overlays can be shared — and also since overlays are small compared to entire file systems, this represents a significant saving on the PlanetLab and Grid models. These models require a transfer per Virtual Machine instantiation, and at the same time do not provide the flexibility that overlaying techniques support. This shows the efficiency of the service deployment mechanisms, and emphasises the ease of service migration and redeployment using the suspend/resume mechanism.

Note that only approximately 720KB of data were transferred from XenoStore to XenoServers. This may seem intuitively unexpected, as the overlay size for a file system that includes the Apache web server is around 23.5MB. This is partly due to the fact that not all files in the Apache distribution are required immediately when the web server starts up, and partly due to AFS's persistent caching mechanisms; as the experiment was repeated 100 times, it is likely that the AFS caches on XenoServers were warm in most deployment operations.

Files required by Apache are transparently cached on the XenoServers by AFS, and need not be fetched over the network from the remote XenoStore at the time of service deployment. This is an important feature for allowing efficient and rapid service deployment. AFS writes cached files back to the their permanent storage location (XenoStore) when they are closed, and this happens after the service is deployed, producing around 9MB of traffic from each XenoServer to XenoStore.

6.3 Scalability

As part of the evaluation of the design of the XenoServer Open Platform and its prototype implementation, it is important to assess its ability to work well under increasing load.

In this section, I first examine the platform design and main operations from the point of view of potential scalability concerns raised. Then, I describe experiments conducted to observe the platform's behaviour as the numbers of participating XenoServers and clients registered with a XenoCorp increase.

Finally, I employ techniques from operational analysis to measure the components' relative utilisation, and determine thresholds of scaling in a XenoCorp's domain, above which bottlenecks emerge to inhibit its further growth.

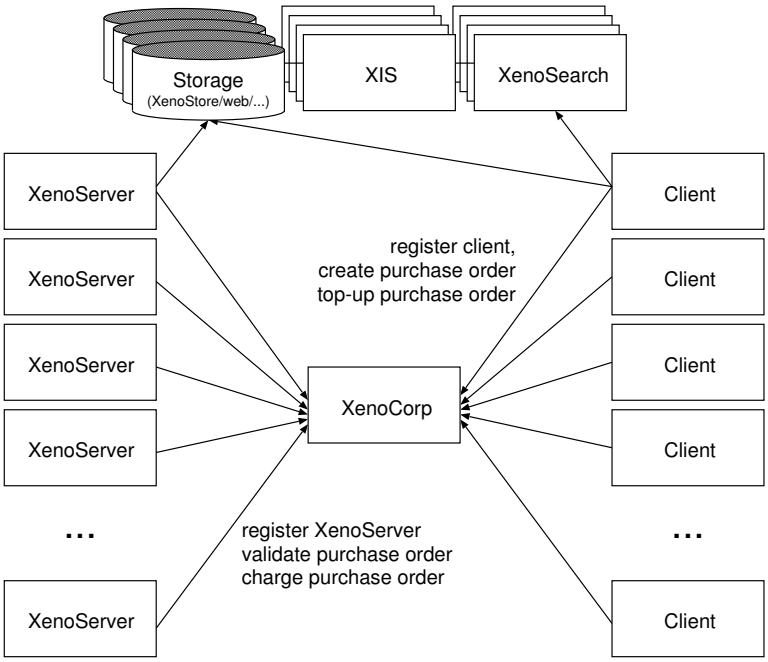

Figure 6.3: XenoCorp as a potential scalability bottleneck in its domain

6.3.1 Domain scalability

This section discusses scalability issues in the proposed system design, raised by the different types of participating components and operations carried out between them. Let us consider a XenoCorp domain, shown in Figure 6.3, consisting of one XenoCorp, a number of XenoServers and clients, and a number of external services, such as storage, the XIS, or XenoSearch components.

The XenoServer platform scales with increase in the numbers of *clients* and their activity. Thus, clients themselves do not present scalability concerns, as it is their activity that stimulates scaling in the first place.

XenoServers do not pose any potential scalability concerns either; if the numbers of clients and XenoServers increase in proportion, then the average workload of each XenoServer will remain constant. If clients increase faster than XenoServers, placing more workload on each server, some servers will get more utilised than others. XenoServer owners are given the flexibility to set the prices of their own resources themselves. Together with the full control that users are allowed over which servers and resources they select, this ensures that the platform operates as a "free market", where load may naturally be distributed towards servers

that offer better deals. At the same time, if load on many servers increases as a result of growing customer demand, the business potential may attract more new XenoServers in the platform. XenoServer owners can price their resources higher than before, thus generating additional profit to acquire more machines.

Scalability of components that are not in the core infrastructure, such as the *XIS, external storage,* or *XenoSearch*, is not tied to the scalability of the platform itself, mainly because the use of such components is optional and not on the critical path of platform operation. This is analogous to the scalability of on-line search engines not affecting the growth of the Internet; as there can be an unlimited number of such third-party services, competing and coexisting, it is expected that load will be naturally distributed, and client demand will determine the number of support services in the platform.

The only potential inhibiting factor for platform growth is *XenoCorp*: it is only one in each domain, it is on the *critical path* of platform operation, and its use by both XenoServers and clients is mandatory. At some point of domain expansion, it may not be able to cope with the increasing number of participants. I demonstrate this behaviour by conducting experiments described in Section 6.3.4.

As XenoCorp is a potential impediment to platform scalability, it should be limited to serving as few frequent operations as possible; this is one of the reasons why *server discovery* has been chosen to be a third-party service rather than functionality provided by XenoCorp as part of the core infrastructure. This also avoids accusations that XenoCorp may favour particular XenoServers, supports local control, and allows for competition and specialisation of search service providers.

The following are the main operations in the XenoServer platform, and the potential scalability concerns for each one:

- **Registration.** Clients and XenoServers contact XenoCorp to obtain their credentials. This operation is infrequent, as it only occurs when a new client or XenoServer joins the domain, and does not present any potential scalability problems.
- **Purchase order creation.** Clients request that XenoCorp creates purchase orders to be used for funding sessions on XenoServers. This operation is also relatively infrequent, as clients are anticipated not to create a separate order per session but to use a single purchase order to create several sessions on XenoServers.

- **Server discovery.** XenoServers independently store their advertisements to their individual storage locations. Clients directly contact a third-party, such as the XIS or a XenoSearch system, to retrieve information about available servers.

- **Service deployment.** Clients contact XenoServers directly to deploy services. As XenoCorp does not participate in the deployment process, this operation does not present any scalability risks.

- **Purchase order validation.** XenoServers communicate with XenoCorp to check if a purchase order contains an adequate amount to fund a session, and ring-fence that amount. This operation is more lightweight than server discovery. Whether it is frequent or not depends on the validation strategy that each XenoCorp applies according to current conditions, on a per purchase order basis; they may not validate some orders, or none at all if their sponsors are trusted, or if XenoCorps are confident that it is possible to track them down, should they try to avoid paying. XenoCorps may also demand that orders be validated in advance but at a lower frequency, or only on a random sample, if load is high and increasing.

- **Service and environment management.** Clients contact XenoServers to administer services they have launched or to perform management operations on the execution environments hosting the tasks — such as stopping or restarting. No potential scalability issues exist here, as XenoCorp is not involved.

- **Charging purchase order.** XenoServers submit accounting and billing information to XenoCorp to request that charges be made against a particular purchase order that has been used for funding a session. Similarly to purchase order validation, XenoCorps may adjust charging strategies according to their current load — for instance, buffer and carry out charging in batches when load is low.

6.3.2 Experiments

To assess the *scalability* of the prototype XenoServer platform, experiments were conducted to measure properties of the system in conditions that resemble its normal operation under increasing numbers of participating XenoServers and clients. The same experimental setting as in previous sections, shown in Figure 6.1, was used to run a XenoCorp and a number of XenoServers and clients. All experiments focused on a single XenoCorp's domain.

Note that a difference between the previous set of experiments and the scalability tests presented here is that in the latter case session creations were *"dry"*; no VMs were launched on the XenoServers as a result of session creations. This allowed running relatively large numbers of XenoDaemons on a single machine, which would not have been possible otherwise due to hardware — mainly memory — limitations combined with inefficient use of memory by Java RMI.

Three scenarios of system operation were considered: in the first one, called *draconian*, XenoServers make sure they do not lose any profit; before each session creation, they validate the purchase order to be used and ring-fence the amount corresponding to the resource reservations to be made in the sponsor's account to guarantee it is available for charging at any later stage.

In the second, *easygoing* scenario, XenoServers do not validate purchase orders in advance; they trust that clients generally pay, and if not they are confident they can trace them and pursue payments. An analogy can be drawn between this model and the way restaurants operate; a diner's ability to pay is rarely verified before the meal. In both the draconian and easygoing cases XenoServers charge clients after sessions are finished and destroyed, in batches of 50.

Finally, in the *lazy* scenario XenoServers neither validate purchase orders nor charge clients soon after they destroy their sessions. In this case, XenoCorp has agreed with XenoServers that payment claims may only be submitted in batches at times of low load, for example during the night.

Note that XenoCorps can select the approach that they follow — draconian, easygoing, lazy, or any custom policy — on a *per purchase order* basis; they can apply different policies to different clients, according to their history and credibility, and the associated estimated risk. In the experiments, I looked at the "extreme" cases where the same policy applied to all clients, both for simplicity and for determining a domain's scalability bounds.

$\mathbf{N_{servers}}$	$\mathbf{N_{clients}}$	$\mathbf{N_{sessions/client}}$	$\mathbf{N_{sessions}}$
5	15	2	30
10	30	3	90
15	45	5	225
20	60	6	360
25	75	8	600
30	90	10	900
35	105	11	1155
40	120	13	1560
45	135	15	2025

Table 6.3: System parameters used for domain growth reconstruction

The *system parameters* that were varied to reflect domain growth from 30 to 2025 active sessions are:

- $N_{servers}$ is the number of active XenoServers in the domain
- $N_{clients}$ is the number of active XenoClients in the domain
- $N_{sessions/client}$ is the number of sessions each XenoClient launches
- $N_{sessions}$ is the approximate number of active sessions in the domain at any time

For each scenario and stage of domain growth — combination of system parameters, represented by a row in Table 6.3 — the following iteration was executed to reconstruct the platform operation:

1. XenoCorp starts
2. $N_{servers}$ XenoServers start
3. $N_{clients}$ XenoClients start
4. Each XenoClient selects $N_{sessions/client}$ XenoServers at random
5. Each XenoClient registers with XenoCorp and creates a purchase order after a random amount of time $t_{register}$, where $0 < t_{register} < 20$ sec

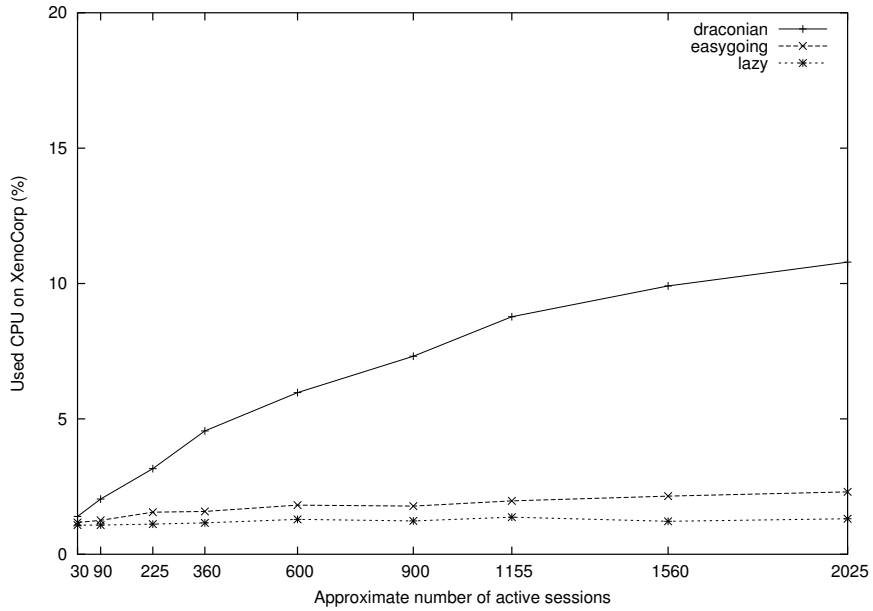

Figure 6.4: CPU utilisation on XenoCorp as its domain expands (graph scaled to a maximum of 20%)

6. Each XenoClient creates $N_{sessions/client}$ sessions, one on each of the randomly selected XenoServers, after time t_{deploy}, where $0 < t_{deploy} < 60$ sec.

7. Each XenoClient destroys the sessions it created after time $t_{destroy}$, where $0 < t_{destroy} < 120$ sec

8. Each XenoClient returns to stage 6 and repeats.

The algorithm was run for a 20-minute period for each combination of scenario and system parameters. Aspects of the system's behaviour, such as the *number of requests* each component received, its average *CPU* and *memory usage*, and *network traffic* generated, were measured in all the aforementioned scenarios and increasing load conditions.

6.3.3 Performance and network traffic effects

I describe the effects of domain growth on XenoCorp in terms of CPU and main memory utilisation, and network traffic to and from XenoCorp. CPU and memory

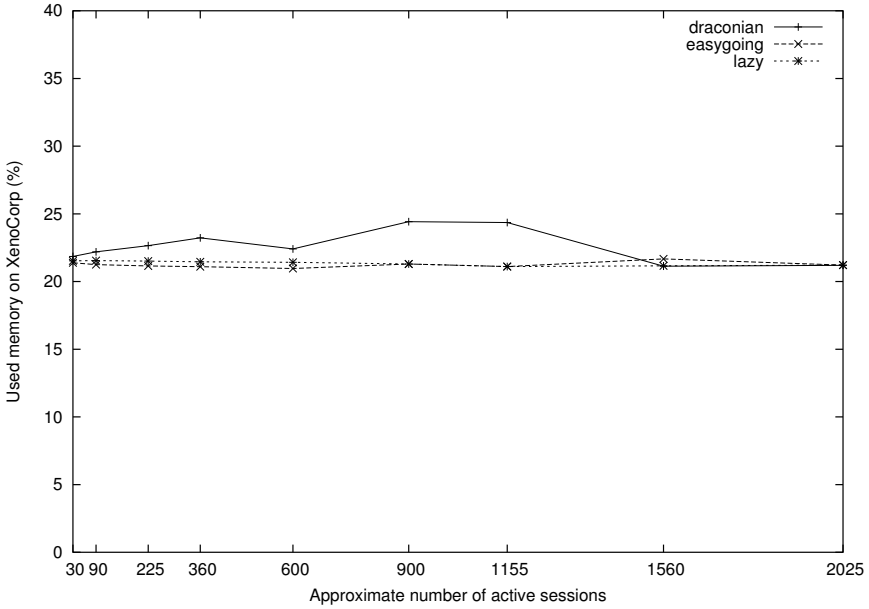

Figure 6.5: Memory utilisation on XenoCorp as its domain expands (graph scaled to a maximum of 40%)

utilisation figures were obtained using Linux's `top` command[1], and network traffic measurements using the Ethereal [OR04] network protocol analyser. Results of this experiment are shown in Figures 6.4, 6.5, and 6.6.

The obtained results show that CPU utilisation on XenoCorp increases slightly in the lazy and easygoing scenarios and more steeply in the draconian case as the numbers of participating servers and clients increase, as shown in Figure 6.4. Memory usage on XenoCorp remains largely unaffected by domain growth, as shown in Figure 6.5.

This behaviour can be explained by considering that most of the part of memory that appears as "used" is the part obtained by the Java Virtual Machine at start-up and reserved for the applications to be deployed on it, called the *heap*. Management of the heap and memory allocation to threads is handled by JVM itself; the heap appears as a reserved memory portion of a fixed size to the operating system.

Moreover, the heap is cleaned-up periodically by the garbage collector, which removes unused — not referenced — objects to free up space. As the domain size

[1] `http://sourceforge.net/projects/unixtop`

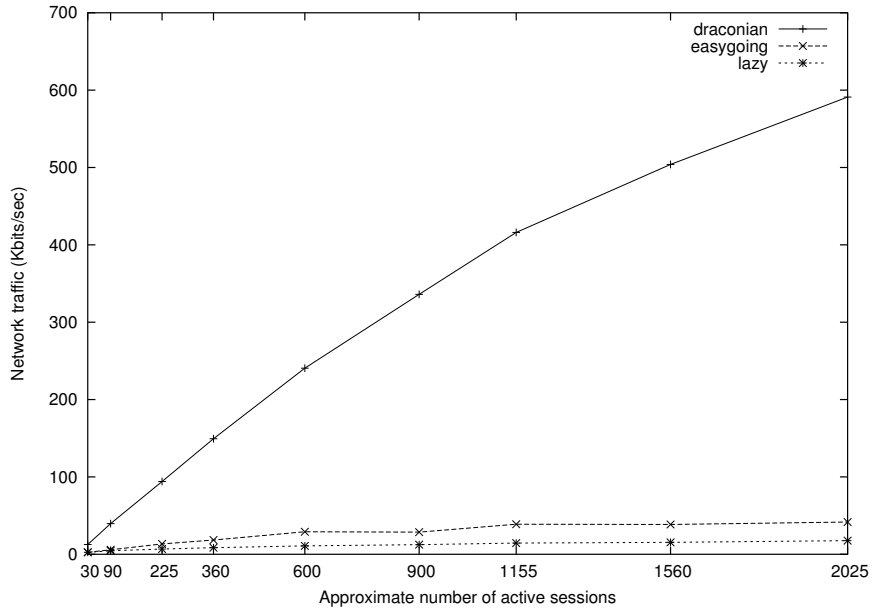

Figure 6.6: Network traffic to/from XenoCorp as its domain expands

increases, XenoCorp has to carry out more and more operations per time unit. This results in higher CPU utilisation because of both request processing and garbage collection; under increased load, XenoCorp's heap fills up quicker, and the garbage collector has to run more often, which consumes increasing amounts of CPU cycles.

Regarding the network traffic generated to and from XenoCorp — as shown in Figure 6.6 — results are very close to what was expected. The draconian approach requires much more communication with XenoCorp per created session than the easygoing and lazy scenarios. The latter two scale considerably better, generating less than 42 and 18 Kbits/sec respectively for 2025 active sessions, compared to approximately 591 Kbits/sec under draconian tracking. This is because of the looser relationship between the number of sessions created and operations involving XenoCorp under easygoing and lazy tracking.

The above numbers impact the cost for XenoCorp; hence unreliable users that need "draconian" tracking may be requested to pay more to cover for the additional resource usage their tracking incurs. Assuming that the $2480 machine used as the prototype XenoCorp is to be replaced in two years, each second of CPU time costs 0.004¢. Assuming also that a permanent guaranteed 100 Mbits/sec connection to the Internet costs $2000 — at a standard co-location

Scenario	CPU cycles	Traffic	Total
Draconian	0.011	0.00400	0.01500
Easygoing	0.004	0.00028	0.00428
Lazy	0.003	0.00012	0.00312

Table 6.4: Cost in ¢ (US$ cents) for XenoCorp handling one client for 20 minutes in each scenario

facility, at the time of writing — the estimated costs incurred by one client in each of the three scenarios are shown in Table 6.4. An unreliable customer may be requested to pay up to approximately 0.011¢ more per 20 minutes than customers for which easygoing tracking is applied. This can be implemented as an agreement between XenoCorp and XenoServers that would see XenoCorp retaining a higher proportion of payments for resource consumption for unreliable users. XenoServers would cover that by applying an additional surcharge to those users or pricing their resources differently for such user groups — this can be achieved using the flexible resource management scheme proposed in Chapter 4.

6.3.4 Bottleneck analysis

A *bottleneck* is one process in a chain of processes, such that its limited capacity reduces the capacity of the whole chain[2]. *Saturation* in a communications system is the condition in which a component of the system has reached its maximum traffic-handling capacity[2].

These two concepts are often confused, but differ fundamentally; being a bottleneck is an *inherent* property of a component in a particular system, as it is determined by its capacity to handle workload in relation to other components' capacity, and is not related to its current load. Saturation is a *temporary* condition, in which the component is fully utilised. A bottleneck component is a bottleneck regardless of whether it is currently saturated or not. A bottleneck component is the one to saturate first in a system, and a saturated bottleneck slows down the rest of the system.

In the following sections, I examine the behaviour of the prototype XenoServer Open Platform to detect bottlenecks at different stages of domain growth.

[2]Definitions from `http://www.wikipedia.org`.

6.3.4.1 Background

Operational analysis offers a methodology for identifying bottlenecks in a simple interactive system. For simplicity, the system is assumed to be load-independent[3] — the throughput of components is not affected by the queue length at each component.

The following quantities are defined:

- C is the *number of completions* of requests in the system, and C_i is the number of completions at component i.
- V_i is the *visit count* for component i, indicating the number of completions of requests at that component for every completion in the entire system:

$$V_i = \frac{C_i}{C}$$

- S_i is the *average service requirement* for component i. This denotes the mean time that the component spends for serving each request.
- X is the system *throughput* — the mean number of requests the system serves per time unit — and X_i is the throughput of component i. This can be expressed as

$$X_i = V_i X$$

- U_i is the *utilisation* of component i, indicating the proportion of the time that the component is busy. Utilisation can be expressed as

$$U_i = X_i S_i$$

As V_i are intrinsic properties of system design and setup in a simple interactive system, and S_i are independent of the queue length at i — because the system is load-independent — the ratio of utilisation of components i and j

[3]This assumption may not be accurate in a real deployment; if the memory utilisation on XenoCorp increases beyond a specific threshold, intensive garbage collection will require additional processing resources.

$$\frac{U_i}{U_j} = \frac{X_i S_i}{X_j S_j} = \frac{V_i X S_i}{V_j X S_j} = \frac{V_i S_i}{V_j S_j} \tag{6.1}$$

indicates which component of i and j will be the first to achieve 100% utilisation; if the ratio is greater than one then i will saturate first, otherwise j will. This can be used to determine the system *bottleneck*, as this will be the component that saturates first. The following section uses this metric to examine the system's behaviour under increasing load.

6.3.4.2 Utilisation ratio measurement

In the case of the XenoServer Open Platform, each XenoCorp is a single central component in its domain, which has to cope with potentially ever increasing numbers of participating servers and clients. Intuitively, it is very likely that each XenoCorp will be a bottleneck at some point of domain growth.

This experiment aimed to determine the threshold of domain size, above which XenoCorp becomes a bottleneck in its domain. Note, however, that the existence of a bottleneck does not indicate saturation; the utilisation of a saturated bottleneck component i is $U_i \to 1$.

In this experiment, I measured the visit count V_i for XenoCorp and each XenoDaemon for all combinations of parameters of domain growth and scenarios in the experimental setting described in Section 6.3.2, using counters added in the prototype platform code. The average service requirement S_i for each component was calculated using the times required for performing each platform operation, which were measured earlier in Section 6.2.2. Note that, since S_i are based on times measured in a setting where session creation did launch VMs on the servers, results of this section do not correspond to a "dry" session creation scenario.

I used the V_i and S_i values obtained to calculate the ratio of utilisation of XenoCorp to XenoDaemon — as in Equation 6.1 — as a function of domain growth, and plotted the results in Figure 6.7.

The figure shows that for all scenarios and domain sizes measured the utilisation ratio was well below one, which means that the bottlenecks were XenoDaemon instances, not XenoCorp. For fewer than 2025 active sessions, XenoDaemon instances will saturate first; thus, even if they all worked at capacity, XenoCorp would be able to cope comfortably.

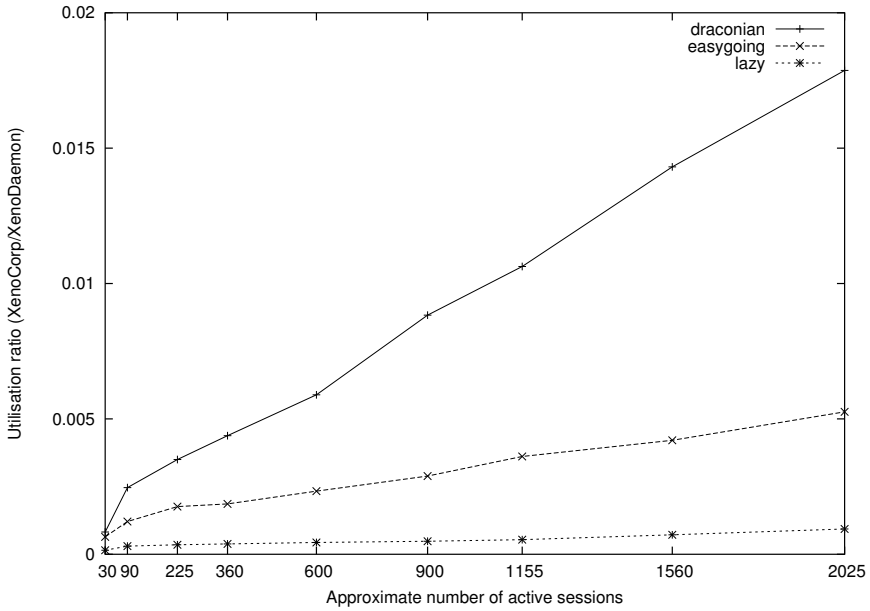

Figure 6.7: XenoCorp/XenoDaemon utilisation ratio

Scenario	$\mathbf{N_{servers}}$	$\mathbf{N_{clients}}$	$\mathbf{N_{sessions/client}}$	$\mathbf{N_{sessions}}$
Draconian	359	1077	119	129,000
Easygoing	664	1992	221	441,000
Lazy	1464	4392	488	2,142,000

Table 6.5: Estimated threshold above which XenoCorp becomes a bottleneck in the system in all three operation scenarios

However, the ratio, while low, is linearly increasing. It can be safely assumed that the ratio increase will remain linear for higher numbers of active sessions as the system is assumed to be load-independent — thus S_i is constant for each component — and the proportions of the number of servers to that of clients and the number of servers to that of active sessions are constant — thus V_i will increase linearly. Then the graph can be extrapolated to determine the threshold above which XenoCorp will become a bottleneck in its domain — the point at which the ratio will surpass one. Results of this projection are summarised in Table 6.5.

As expected, the lazy scenario is the most scalable, as XenoCorp is effectively only registering clients and XenoServers when they first join the platform and create purchase orders for them. After that, the number of participating clients

and servers makes no difference to XenoCorp's workload, as it is not involved in purchase order validation or payment claims.

Each lazy XenoCorp can cope, without being a bottleneck in the system, with *more than 2.1 million* concurrently active sessions; this corresponds to 4392 clients, each one maintaining sessions on 488 of the 1464 XenoServers present. XenoCorp may be able to cope with higher numbers, depending on the utilisation of XenoServers, but will be the first component to saturate.

This is an encouraging result, given that the chosen experimental parameters generate conditions of *exceptionally high load*; in the experiments undertaken, clients create sessions on a third of all available XenoServers at most every three minutes. Such conditions would be hardly realistic in a real-world platform deployment, where most clients would need fewer machines and would deploy more long-lived sessions. Also, *multiple, competing XenoCorps* will exist in a real-world deployment, allowing for much higher participation figures. Furthermore, real XenoCorps are anticipated to comprise more expensive, *high-end hardware*, and thus be able to scale to much higher numbers.

6.4 Effectiveness

Previous sections in this chapter have shown that the implementation of the XenoServer platform provides an efficient and scalable service deployment infrastructure. Here, I discuss in detail how each of the requirements for global public computing — as identified in Section 2.4.6 — is met through a combination of design decisions, mechanisms provided, and implementation choices followed.

Ease of deployment. Mechanisms for efficient and easy global-scale service deployment allow customers to dynamically obtain computing resources on demand, and run services on a number of XenoServers around the world.

Service deployment is made *easy* as users do not have to push their services' data on to the servers where the services are to be deployed, but can instead specify the location from which all necessary data is to be pulled by the servers.

The approach of using overlays significantly eases *management* of *replicated services*. Since overlays can be constructed ahead of time and may be applied to more than one deployed instance, no administrative intervention is required when

an instance migrates. This is in contrast to PlanetLab or Grids which require users to actively transfer files to servers.

Common distribution file system templates and operating system kernels are cached persistently on XenoServers for *efficiency*. The copy-on-write architecture used allows users to maintain customised views of these, and ensures that only what is required to allow this customisation is fetched over the network. Parallel deployment allows users to deploy services on several XenoServers concurrently. The efficiency of the developed mechanisms is evident from the results of the performance evaluation experiments, which show that users can deploy complex services on multiple XenoServers in *45 seconds*. Also, experiments have shown that, using the XenoServer platform's overlaying deployment mechanisms, much smaller amounts of data had to be transferred to the servers, than if the conventional "push" deployment model was followed — up to 99% less if an Apache web server was to be launched.

The overlaying functionality provided allows users to *trade off* the degree of *customisation* they require against the ensuing impact on performance, network traffic, and the associated real-world costs. As demonstrated in Section 6.2.1, the relative size of an overlay for realistic service deployments is generally small; however there is nothing to prevent a user from specifying an entirely bespoke file system should they require it.

Non-cooperative users. The XenoServer platform does not assume a cooperative user community, unlike PlanetLab, Grids, and scientific computing applications. Server owners are given explicit *monetary rewards* for providing resources on their machines.

XenoServers *account* for resources consumed by the various services and request payments from XenoCorp, providing billing and accounting details. XenoCorp then passes the *charges* on to the users, and retains some proportion of these for funding its own operation. The accounting, charging, and billing infrastructure has been described in Chapter 3.

Untrusted code. The XenoServer Open Platform anticipates *competing users* running potentially *unsafe* and *untrusted* services. For that reason, it has been chosen to provide harder resource isolation and protection at a layer below the operating system using the Xen Virtual Machine Monitor.

Xen securely multiplexes several operating system instances on a single physical machine. As shown in [BDF+03a], Xen does protect VMs effectively; extensive tests, presented in that paper, have shown that even intentionally malicious code running in a VM has not been able to adversely affect other VMs running on the same machine in any way.

However, there are some cases where the inter-VM protection provided by Xen is not adequate; services may use resources on XenoServers not to directly harm services running in other VMs, but to perform illegal activities, such as spread viruses, perform port-scans, or participate in distributed denial of service (DDoS) attacks.

This problem is tackled by a combination of three techniques: first, all users of the platform can be tied to *real-world identities*, for instance through credit card or bank account details provided; this does not allow the comfort of full anonymity and provides the threat of prosecution. Second, XenoServers record information about service activity in *audit trails*, to allow tracking down the sponsor of malicious actions. Third, external services such as XenoTrust [DHH+03] can be employed to allow clients and XenoServers to *exchange opinions* about other servers and clients according to their past experiences. In a free-market environment, such as the one of the XenoServer Open Platform, it is envisaged that servers and clients who get consistently bad reputation will naturally get isolated, not only if XenoCorp pursues disciplinary action against them, but as a result of the unwillingness of others to pursue business with them.

Out-of-the-box applications. In the prototype implementation, services are hosted in VMs running on Xen, which are defined from the ground up at deployment time, in terms of the operating system kernel image and root file system to be used. Additionally, the XenoServer platform does not require that services to be deployed are programmed to use a platform-wide API or middleware. The platform can run unmodified, *out-of-the-box* services, written and compiled to run on normal, commodity operating systems. The interfaces that VMs present to applications are not affected by the underlying presence of Xen.

The advantage of using entirely customisable guestOSs as execution environments presents another important benefit; *experimental* kernel code can be safely deployed on the servers without the possibility of harming any other service. This allows using the platform's wide-area deployment facilities for operating systems or networking research.

However, as Xen does not perform full virtualisation of physical resources for performance reasons, it is necessary that guestOSs are themselves *ported* to Xen. While the modifications required are not significant, as shown in [BDF+03a], this presents a limitation of the VMM-based approach; it is not possible to run off-the-shelf operating systems.

In the case of global public computing, it is much more important to accommodate unmodified applications than unmodified guestOSs; common guestOSs are few, and can be modified relatively easily, whereas user applications are practically infinite, and modifying those requires significantly higher amounts of effort. It is important that the cost of entry to the end users is low; a self-financing public computing platform can afford employing a few trained software engineers to modify popular guestOSs whenever a new major version is released[4].

Self-financing. The business model behind the XenoServer platform differs significantly from those found in other deployment infrastructures; those are usually built, funded, and maintained by universities or research institutes. Resources are provided for free in the interest of science.

The XenoServer platform aims to be *self-financing*. Corporations owning XenoServers meet the costs of maintaining and upgrading servers by profit generated by selling their servers' resources to users. XenoCorp retains a portion of payments to XenoServers, which funds its own operation and potentially that of the XIS, XenoSearch, and XenoStore, if these services are provided by XenoCorp.

According to the arrangement between XenoCorps and XenoServers in their domain, XenoServer payments can be proportional to resource consumption on the servers, a flat monthly payment, or proportional to XenoCorp's profit. Other custom payment options may also be devised and employed.

The built-in mechanisms that the XenoServer platform provides for supporting competing untrusted users, and facilitating accurate accounting, charging, and billing, are reusable for implementing utility computing functionality. The framework for flexible and open description of resources and pricing schemes for the coordination of common descriptions, allows partitioning and selling resources easily. The role-based resource management facility provides support for policy-based fine-grained control over resources to be sold.

[4]Barring potential legal concerns related to licensing for porting commercial operating systems on Xen, which are not investigated by this dissertation.

Short timescales. As shown by the results of the performance evaluation experiments, it is possible to acquire resources on several XenoServers around the world and deploy complex distributed services in entirely new Virtual Machines in *less than 45 seconds*. Thus, there is nothing to prevent users from reserving resources even only for a few minutes if this suits their services' needs to keep hosting costs down and achieve greater flexibility in terms of relocating their resources according to geographical changes in demand.

A number of mechanisms are employed to enhance the efficiency of service deployment; cached XenoSearch results at the client side, AFS's caching mechanisms, persistently cached template file systems and operating system images stored locally on XenoServers, and the overlaying functionality provided. When *rapid deployment* is required, this can be achieved by storing an environment's memory image on disk when it is stopped and then resuming it, rather than building a new VM from scratch when a new session is started. This allows resuming execution environments over very short timescales, in just above *2 seconds*.

Incremental Scalability. As shown by experiments described in Section 6.3, the prototype deployment of the XenoServer Open Platform is able to scale to large numbers of participating components — *more than 2.1 million*[5] simultaneous sessions in each XenoCorp's domain under exceptionally heavy load — without any component presenting a performance bottleneck.

At the same time, as the operation and coexistence of multiple, competing XenoCorps and third-party services — such as the XIS, server discovery services, and external storage services — is allowed and encouraged, the system can grow to accommodate much greater numbers of participants on demand.

Flexible server selection. The XenoServer platform comprises open and extensible resource discovery mechanisms; servers advertise their resource availability at globally accessible locations, and then the XenoServer Information Service (XIS) collects and stores those in a structured manner. Other higher-level services, such as XenoSearch, can be employed for more advanced searching facility.

No single matchmaker component or matchmaking algorithm is assumed in the XenoServer platform. Multiple different server discovery services (such as XenoSearch, XIS) can coexist, providing different specialised searching algo-

[5]This result is accurate only if the system is load-independent.

rithms, or simply competing with each other. At the same time, as resource discovery is not part of the core infrastructure of the XenoServer platform, alternative components may be developed that carry out matchmaking in entirely different ways from the XIS and XenoSearch. Users are free not to use any server discovery mechanisms at all if they so wish, and have ultimate control on which servers they select.

This has a number of important benefits; first, it enhances *openness* and *extensibility* of the platform, as it allows the evolution of matchmaker components to meet specialised needs of the user community that cannot be anticipated at the time the XenoServer platform is designed; new distributed services that may emerge once the enabling global-scale service deployment technology is in place may require searching features that are not obvious at the time of platform design.

Furthermore, it allows for *incremental scalability*, as decentralising matchmaking removes the matchmaker bottleneck found in distributed deployment platforms. Also, it removes single points of failure, as if one XenoSearch system is unavailable users can simply consult another one.

Resource-oriented vs. location-oriented. As discussed in the previous paragraph, different XenoSearch systems may implement any kind of searching algorithms the customer communities they target require. Some may support traditional *resource-oriented* server discovery — for instance, answering queries such as "suggest a group of servers that will carry out a specific distributed job in less than five minutes". Others may also provide search functionality based on different metrics, such as *location-based* resource discovery — answering queries such as "suggest a server that will be located at a point in the network which minimises the total round-trip time among the server and a particular set of clients". No architectural decision has been made to tie the platform to either model, or any other specific model of server discovery.

Resource description coordination. While XenoServers can independently describe and advertise their resources, the platform provides *coordination* mechanisms to limit the use of arbitrary names that would lead to inconsistencies — as described in Section 4.2.3. Common resource types and pricing units are given uniform platform-wide names, while servers retain the ability to independently describe new types of resources or pricing schemes, to allow for unusual hardware or exotic charging set-ups.

Resource management. In the XenoServers' case, resource management is carried out using two different mechanisms. An implicit mechanism for influencing the way resources are allocated, which is not present in Grids and PlanetLab, is that of *pricing*. XenoServer owners can adjust the prices they associate with the different resources according to demand and availability, thus influencing future demand and trying to ensure costs of upgrades and acquisition of additional hardware when required are covered.

Explicit resource management can also be achieved using the *role-based resource management* framework presented in Chapter 4. VCI interfaces exported by the Xen Virtual Machine Monitor are used to enforce restrictions on the resources that each user VM is allowed to use, according to the recommendations of the admission controller module of the XenoDaemon.

Policy-based management. The role-based resource management framework allows easy and convenient management of resources on XenoServers. It provides expressive and flexible mechanisms to participating entities for defining *role-based policies*, influencing how resources are to be apportioned between different users or user groups on the servers.

Convenient management. The role-based approach employed for managing resource allocation provides significant *management conveniences*, and offers an intuitive and flexible, yet sufficiently expressive and powerful, technique for expressing resource management policies. Users can be *grouped* according to their properties or membership of other roles, and enumeration of principals is not required.

Access to resources is *quantified*; RBRM uses the mechanisms provided for policy federation and overlapping resolution to determine how much of each resource a user is entitled to get. It also allows much more *fine-grained control* of a server's resources than most resource management architectures found in distributed deployment platforms; for example, it supports defining different allocation policies for different resources on the same server, or even for different parts of the same resource that are priced differently.

Local control. In the XenoServer platform, the evaluation of resource management policies is carried out at an explicit *admission control* stage on the XenoServers themselves — when users contact servers directly to deploy services.

This is in contrast to previously available deployment infrastructures, which carried out policy evaluation on the matchmaker at the resource discovery stage — when users are searching for suitable servers.

The approach employed by XenoServers provides significant *flexibility*; as policies are self-contained objects, not carried in the resource advertisements, changing policies on a server does not require changing the server advertisements that have been submitted. Also, it provides a clearer *separation of control* and *functionality*; evaluating policies locally means that no remote matchmaker has to be trusted, and ultimate control naturally remains on the server owners.

At the same time, individual XenoServers are allowed to define their own resources, decide on the pricing schemes to be used, and advertise their resource availability accordingly. *No central component* is responsible of ensuring that they do not present false information about their capabilities; in cases of misbehaviour, reputation systems, audit trails, and information about their owners maintained by XenoCorp can be used for penalising them.

Furthermore, the XenoServer platform does not aim to provide platform-wide optimisation of resource usage, as some distributed deployment platforms do. It provides the mechanisms required for resource discovery and server selection, and allows *end-to-end decisions* to be made; clients can request any resources on any servers — provided that they are able to pay — and servers may accept or deny any such request, according to the federated resource management policies defined. The end-to-end principle simplifies the business model, allows competition of services, and independent control of XenoServers.

Federated and overlapping policies. The proposed resource management framework allows multiple entities to independently define *federated* resource allocation policies. Server owners, XenoCorp services, local network administrators, affiliated organisations, law enforcement agencies, and other entities are allowed to express and maintain subjective views on role memberships as well as whether and to what extent to take policies expressed by other entities into account. The ability to express federated policies is crucial for global public computing because of the inherently federated (not centrally controlled) nature of such systems. Different stakeholders have different interests and potentially different views on how resources are to be apportioned on the servers.

Federated policies expressed by independent entities can be *overlapping*: when more than one policy rule matches exist for the same resource and user group on

the same machine. In conventional resource management systems, such overlaps are either quietly ignored, or require manual intervention to be resolved. The proposed framework provides mechanisms for specifying explicitly how such overlaps are to be resolved and removes the need for manual intervention. This is another important feature of the XenoServer platform; distributively-owned systems are likely to comprise entities with distinct goals and interests, thus with different and possibly overlapping resource allocation policies. At the same time, due to the global scale in which the platform is envisaged to operate, the availability of manual intervention when required cannot be assumed.

Chapter 7

Conclusion

Change is inevitable - except from a vending machine.
— Robert C. Gallagher

XenoServers are not about changing the world but rather are building the infrastructure that allows it to change by providing a compelling platform for general-purpose public computing. Challenges in service deployment, resource description and pricing, server selection, resource management, accounting, and charging have been addressed so as to substantially *lower the barrier to entry* for deploying new services and applications on a global scale, allowing the participation of *uncooperative, competing,* and *untrusted* users.

This has been achieved by a combination of novel ideas — such as the proposed *role-based resource management* framework for expressing federated policies — with off-the-shelf components — such as the Xen Virtual Machine Monitor, and AFS. New *deployment models* have been devised to allow users to launch their applications on large numbers of machines easily and quickly. At the same time, the use of the Xen VMM for low-level multiplexing of VMs has been instrumental in maintaining a low cost of entry, as it allows *out-of-the-box applications* to run on client VMs, without requiring that they be rewritten to comply with a particular platform-wide API or middleware or to be executable on a specific operating system.

A prototype implementation of the XenoServer Open Platform has been developed, deployed, and shown to be functional on a number of machines in the local network of the Systems Research Group. Performance and scalability evaluation results — in a simulated wide-area setting — have shown that the system

can operate *efficiently* and facilitate *rapid service deployment*, allowing users to launch complex services on large numbers of XenoServers around the world in *less than 45 seconds*. At the same time, it has been shown that, even in the modest experimental setting on which the experiments were attempted — real-world XenoCorps would possess more advanced and expensive hardware — and under exceptionally heavy load, each XenoCorp's domain can scale to *more than 2.1 million* active sessions without presenting performance bottlenecks.

7.1 Contributions

This dissertation makes the following contributions:

- Introduces the concept of general-purpose, commercial, *global public computing*, and the main research challenges that need to be tackled by systems providing such functionality.
- Describes the design and implementation of the *XenoServer Open Platform* for global public computing, and demonstrates that the mechanisms and techniques employed work efficiently and scalably.
- Provides reusable public computing mechanisms, such as a comprehensive and flexible *role-based resource management* framework, and an efficient *global-scale service deployment* architecture.

The XenoServer Open Platform presented in this dissertation has employed the Xen Virtual Machine Monitor, which securely isolates execution environments on a XenoServer, XenoSearch, which supports high-level server discovery, and the copy-on-write NFS server, to perform overlaying. All three components have been designed and developed by other members of the XenoServer team and are not part of the work presented in this dissertation.

7.2 Future work

Work presented in this dissertation has demonstrated that building an infrastructure to allow the deployment of untrusted global-scale services by competing

users is possible. The internal roll-out of the XenoServer platform has been described, and initial experimental results have been analysed. The next step will be to deploy the platform on larger numbers of machines outside the Computer Laboratory network, enabling the development of meaningful next-generation distributed services.

In parallel with that, the autonomy of servers and clients raises important trust management challenges, for instance requiring mechanisms for the dissemination of participants' opinions about others. Security aspects, such as automated recognition and interception of malicious service activity, need to be investigated. Furthermore, the applicability of dynamic pricing can be examined, both as a means of maximising revenue for XenoServer owners and as a mechanism for regulating server congestion.

Large-scale deployment. Having tested the platform on an internal deployment setting, a *large-scale deployment*, outside the internal Computer Laboratory network, is planned. This will provide valuable feedback for potential refinements of platform implementation aspects, and open the way for a *beta trial*; this will make the platform open to the general public, and will use credit cards for authentication and for billing nominal sums for resource credits on the platform.

The next stage will be to see whether the XenoServer platform prototype can be transitioned into a sustainable self-financing infrastructure which both individuals and commercial organisations are prepared to trust for their application deployment needs.

Next-generation services. The existence of an infrastructure that allows users to deploy any kind of global-scale distributed services easily, efficiently, and at a low cost, will give birth to *new types of on-line services* that do not exist at the time of writing because of limitations of the existing service deployment technology. Instead of maintaining a static network of server mirrors and requiring that the user communicates with the closest one, next-generation distributed services will themselves move closer to the users to maintain low latency, reduce long-haul network traffic, and maintain predictable quality of service.

Distributed massively multi-player games may be developed, whose infrastructure of game servers will dynamically relocate to adapt to geographical changes in player demand. Next-generation mobile agent systems may allow traditionally centralised and static services — such as video streaming — to be implemented

distributively [GMS+01], actively migrate closer to the users, and follow their movement. Competing high-level resource brokerage systems may achieve discounts by purchasing XenoServer resources in bulk, and provide users of Grids or utility computing infrastructures transparent access to incrementally large groups of resources — similar to the way competing telephone companies often share the same physical network but provide different services and pricing schemes.

Trust management. In the real world, servers and clients operate autonomously. Servers may be unreliable or untrustworthy, trying to overcharge clients or not run programs faithfully. Clients may attempt to abuse the platform by avoiding paying their bills, or by running programs with nefarious, anti-social or illegal goals.

The problem may initially seem to be a security one; mechanisms need to be designed to ensure that only properly authenticated components can participate in the platform. If a component — client or server — misbehaves, a complaint to the XenoCorp with which it is registered can result in its credentials being invalidated. Problems can arise, however, in a number of ways. Firstly, it is unrealistic to expect all of the participants to agree on common standards of behaviour — even if a single "acceptable use policy" could be enforced by XenoCorp it would most likely be written in a natural language. Secondly, the volume of complaints may be large and the distinction between trivial and important ones unclear. Finally, issuing a complaint is difficult when the source of misbehaviour is not straightforward to determine. As an analogy to the real-world, a customer can complain to VISA if a restaurant overcharges him or her, but not if the food is of unsatisfactory quality.

Reputation management systems such as the eBay[1] and amazon marketplace[2] rating schemes have very limited expressiveness, as they assume that all participants agree on the criteria on which such reputations are based, and that they attach equal credence to everyone's statements. Moreover, reputation in those systems is usually unidimensional, and their scalability is restricted, as the central repository or agent that stores the reputation information is a single point of failure, and has to grow in proportion to the number of participants.

Initial work on an *event-based trust management* architecture that is being developed for use in the XenoServer Open Platform is described in [DHH+03,

[1] `http://www.ebay.com`
[2] `http://www.amazon.com/`

DKHP03]. It allows XenoServers and clients to exchange views about others' performance in several different areas based on their past interactions. To encourage clients and servers to provide information about each other, initial work on a system that provides incentives for honest participation in trust management infrastructures has been undertaken [FKOD04]. The system aims to improve the quality of information contained in trust management systems by reducing free-riding and encouraging honesty.

Security and privacy. The XenoServer platform allows untrusted users to obtain computing resources on large numbers of servers around the world based on their ability to pay. This enables malicious users with access to monetary resources to, for instance, easily build large networks of agent hosts to launch distributed denial of service attacks, or to use resources on XenoServers to engage in other antisocial or illegal activities.

Such problems have been tackled using auditing in the prototype development of the XenoServer platform. However, this may often require manual intervention by a XenoCorp administrator or other authorities to discover illegal activity and track down the sponsor of the corresponding task. For a global-scale deployment, additional *automated tools* may be used for discovery of malicious activity and automatic interception of such behaviour — for instance, by immediately restricting the network resources allocated to the suspicious service's execution environment.

Another important research challenge is safeguarding *user privacy*; work will be undertaken on a methodology that will protect users from intrusive XenoServer owners. By using purchase orders for authentication of clients on XenoServers, their actions can be completely disassociated from their identity on the XenoServer. The server will only know which order may be used to charge for resources consumed by a service, but will have no information about its owner or sponsor. In cases of misbehaviour, the sponsor will be traceable by XenoCorp.

Dynamic pricing. Software components can be developed, which will adjust the resource pricing schemes on a XenoServer according to resource demand. For instance, a *dynamic pricing component* could decide to increase the price of network bandwidth if the overall network load on a server increases. This approach may be applied on groups of more than one server where, for example, the owner of a number of machines elects to run a dynamic pricing component to

make sure her revenue is maximised. This bears similarity to research on dynamic road pricing based on the time of day and traffic [SLDI01].

This approach imposes a number of challenges, ranging from accurately measuring — or estimating — resource demand and supply, disseminating this information between other servers in the same pricing group, and using this information to adjust prices. Communication with smart "dynamic pricing-aware" applications could be facilitated to allow applications to adjust their resource requirements according to the current pricing scheme — for instance, a smart application might decide to consume less CPU and more memory if the former becomes more expensive than a specific threshold.

Bibliography

[AAF+02] Assaf Arkin, Sid Askary, Scott Fordin, Wolfgang Jekeli, Kohsuke Kawaguchi, David Orchard, Stefano Pogliani, Karsten Riemer, Susan Struble, Pal Takacsi-Nagy, Ivana Trickovic, and Sinisa Zimek. Web Service Choreography Interface (WSCI) 1.0. W3C Note, World Wide Web Consortium, August 2002. Available from `http://www.w3.org/TR/wsci/`.

[ABCC66] R. J. Adair, R. U. Bayles, L. W. Comeau, and R. J. Creasy. A Virtual Machine System for the 360/40. Tehnical Report 320-2007, IBM Corporation, Cambridge Scientific Center, May 1966.

[Aka03] Akamai Technologies, Inc. Akamai EdgeComputing, 2003. Brochure, available from `http://www.akamai.com/`.

[AMK98] Elan Amir, Steven McCanne, and Randy H. Katz. An Active Service Framework and Its Application to Real-Time Multimedia Transcoding. In *Proceedings of the ACM Annual Conference of the Special Interest Group on Data Communication (SIGCOMM 1998)*, pages 178–189, Vancouver, Canada, 1998.

[And04] Anne H. Anderson. An Introduction to the Web Services Policy Language (WSPL). In *5th IEEE International Workshop on Policies for Distributed Systems and Networks (POLICY '04)*, pages 189–192, Yorktown Heights, NY, USA, June 2004. IEEE Computer Society.

[AR02] Amr Awadallah and Mendel Rosenblum. The vMatrix: A Network of Virtual Machine Monitors for Dynamic Content Distribution. In *Proceedings of the 7th International Workshop on Web Content Caching and Distribution (WCW 2002)*, Boulder, CO, USA, August 2002.

[BAG00] R. Buyya, R. Abramson, and D. Giddy. Nimrod/G: An Architecture of a Resource Management and Scheduling System in a Global Computational Grid. In *Proceedings of the 4th International Conference on High Performance Computing and Grid in Asia Pacific Region*, Beijing, China, May 2000. IEEE Computer Society Press.

[BBC+04] A. Bavier, M. Bowman, B. Chun, D. Culler, S. Karlin, S. Muir, L. Peterson, T. Roscoe, T. Spalink, and M. Wawrzoniak. Operating System Support for Planetary-Scale Network Services. In *Proceedings of the 1st Symposium on Networked Systems Design and Implementation (NSDI '04)*, pages 253–266, San Francisco, CA, March 2004.

[BBR+97] Michael A. Bauer, Richard B. Bunt, Asham El Rayess, Patrick J. Finnigan, Thomas Kunz, Hanan Lutfiyya, Andrew D. Marshall, Patrick Martin, Gregory M. Oster, Wendy Powley, Jerome A. Rolia, David Taylor, and C. Murray Woodside. Services Supporting Management of Distributed Applications and Systems. *IBM Systems Journal*, 36(4):508–526, 1997.

[BDF+03a] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and The Art of Virtualization. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles (SOSP19)*, pages 164–177, Bolton Landing, NY, USA, 2003. ACM Press.

[BDF+03b] Paul R. Barham, Boris Dragovic, Keir A. Fraser, Steven M. Hand, Timothy L. Harris, Alex C. Ho, Evangelos Kotsovinos, Anil V.S. Madhavapeddy, Rolf Neugebauer, Ian A. Pratt, and Andrew K. Warfield. Xen 2002. Technical Report UCAM-CL-TR-553, University of Cambridge, Computer Laboratory, January 2003.

[BDH+03] Bob Briscoe, Vasilios Darlagiannis, Oliver Heckmann, Huw Oliver, Vasilios A. Siris, David Songhurst, and Burkhard Stiller. A Market Managed Multi-Service Internet (M3I). *Computer Communications*, 26(4):404–414, 2003.

[BHL01] T. Berners-Lee, J. Hendler, and O. Lassilia. The Semantic Web. *Scientific American*, May 2001.

[BKR98] Jonathan Bredin, David Kotz, and Daniela Rus. Market-based Resource Control for Mobile Agents. In *Proceedings of the Second Inter-*

national Conference on Autonomous Agents, pages 197–204. ACM Press, 1998.

[Bla01] Muriel Blanchier. Video games. White Paper, Societe Generale Group, June 2001.

[BLT98] P. Bernadat, D. Lambright, and F. Travostino. Towards a Resource-safe Java for Service Guarantees in Uncooperative Environments. In *Proceedings of the IEEE Workshop on Programming Languages for Real-Time Industrial Applications*, pages 101–111, Madrid, Spain, December 1998.

[BM02] L. Boloni and D. Marinescu. Robust Scheduling of Meta-Programs in a Nondeterministic Environment. Technical Report PDN–02–005, Department of Computer Sciences, Purdue University, CSD-TR #98-003, October 2002.

[BN84] Andrew D. Birrell and Bruce Jay Nelson. Implementing Remote Procedure Calls. *ACM Transactions on Computer Systems*, 2(1):39–59, 1984.

[BPS98] Tim Bray, Jean Paoli, and C. M. Sperberg-McQueen. Extensible Markup Language (XML) 1.0. W3C Recommendation REC-XML-19980210, World Wide Web Consortium, February 1998. Available from `http://www.w3.org/TR/REC-xml/`.

[BSP+95] B. Bershad, S. Savage, P. Pardyak, E. Sirer, M. Fiuczynski, D. Becker, S. Eggers, and C. Chambers. Extensibility, Safety and Performance in the SPIN Operating System. In *Symposium on Operating Systems Principles (SOSP '95)*, pages 267–284, Copper Mountain Resort, Colorado, December 1995.

[BTS+98] Godmar Back, Patrick Tullmann, Leigh Stoller, Wilson C. Hsieh, and Jay Lepreau. Java Operating Systems: Design and Implementation. Technical Report UUCS-98-015, University of Utah, Department of Computer Science, August 1998.

[Bul80] G. M. Bull. *Dartmouth Time-Sharing System.* Halsted Press, January 1980.

[CAD+03] F. Curbera, T. Andrews, H. Dholakia, Y. Goland, J. Klein, F. Leymann, K. Liu, D. Roller, D. Smith, S. Thatte,

I. Trickovic, and S. Weerawarana. Business Process Execution Language For Web Services, version 1.1. White paper, 2003. Available from `http://www-106.ibm.com/developerworks/webservices/library/ws-bpel/`.

[Cau00] Brian Caulfield. Searching for Something New - Gnutella, Freenet, Napster Swap Files and Meet the RIAA, November 2000. In the Internet World magazine. Available from `http://www.internetworld.com`.

[CCMN04] G. Chafle, S. Chandra, V. Mann, and M. G. Nanda. Decentralized Orchestration of Composite Web Services. In *Proceedings of the 13th International World Wide Web Conference*, New York City, USA, May 2004.

[CCMW01] Erik Christensen, Francisco Curbera, Greg Meredith, and Sanjiva Weerawarana. Web Services Description Language (WSDL) 1.1. W3C Note, World Wide Web Consortium, March 2001. Available from `http://www.w3.org/TR/wsdl`.

[CCR+03] Brent Chun, David Culler, Timothy Roscoe, Andy Bavier, Larry Peterson, Mike Wawrzoniak, and Mic Bowman. PlanetLab: An Overlay Testbed for Broad-Coverage Services. *ACM SIGCOMM Computer Communication Review*, 33(3):3–12, July 2003.

[CDD62] F. J. Corbató, M. M. Daggett, and R. C. Daley. An Experimental Time-Sharing System. In *Proceedings of the Spring Joint Computer Conference (SJCC)*, pages 335–344, San Francisco, CA, 1962.

[CFH+04] Christopher Clark, Keir Fraser, Steve Hand, Jakob Gorm Hansen, Eric Jul, Christian Limpach, Ian Pratt, and Andy Warfield. Live Migration of Virtual Machines, October 2004. Under review, submitted to a refereed international conference.

[CFK+98] Karl Czajkowski, Ian T. Foster, Nicholas T. Karonis, Carl Kesselman, Stuart Martin, Warren Smith, and Steven Tuecke. A Resource Management Architecture for Metacomputing Systems. In *Proceedings of the Workshop on Job Scheduling Strategies for Parallel Processing*, pages 62–82, Orlando, Florida, USA, 1998. Springer-Verlag.

[CFK99] Karl Czajkowski, Ian Foster, and Carl Kesselman. Resource Co-Allocation in Computational Grids. In *Proceedings of the The Eighth*

IEEE International Symposium on High Performance Distributed Computing, page 37, Redondo Beach, California, August 1999. IEEE Computer Society.

[CFN90] D. Chaum, A. Fiat, and M. Naor. Untraceable Electronic Cash. In *Proceedings on Advances in Cryptology*, pages 319–327, Santa Barbara, CA, 1990. Springer-Verlag New York, Inc.

[CKR+01] Michael J. Carey, Steve Kirsch, Mary Roth, Bert Van der Linden, Nicolas Adiba, Michael Blow, Daniela Florescu, David Li, Ivan Oprencak, Rajendra Panwar, Runping Qi, David Rieber, John C. Shafer, Brian Sterling, Tolga Urhan, Brian Vickery, Dan Wineman, and Kuan Yee. The Propel Distributed Services Platform. In *The International Journal on Very Large Databases (VLDB)*, pages 671–674, 2001.

[CMA00] M. Covington, M. Moyer, and M. Ahamad. Generalized Role-Based Access Control for Securing Future Applications. In *Proceedings of the 23rd National Information Systems Security Conference (NISSC)*, Baltimore, MD, October 2000.

[Con00] Connectix. The Technology of Virtual PC. White Paper, 2000. Available from `http://www.connectix.com/downloadcenter/pdf/vpc5w_whitepaper.pdf`.

[Cox00] Tony Cox. Online Gaming, October 2000. Talk, in the Visualization and Virtual Environments Community Club (VVECC). Available from `http://www-ais.itd.clrc.ac.uk/VVECC/proceed/entertain/materials/cox/`.

[CRLS03] Nicholas Coleman, Rajesh Raman, Miron Livny, and Marvin Solomon. Distributed Policy Management and Comprehension with Classified Advertisements. Tehnical Report UW-CS-TR-1481, University of Wisconsin - Madison Computer Sciences Department, April 2003.

[CS03] Brent Chun and Tammo Spalink. Slice Creation and Management. Technical Report PDN–03–013, PlanetLab Consortium, July 2003.

[CV65] F. J. Corbató and V. A. Vyssotsky. Introduction and Overview of the Multics System. In *Proceedings of the Fall Joint Computer Conference (AFIPS)*, pages 185–196, New York, 1965. Spartan Books.

[DA99] T. Dierks and C. Allen. The TLS Protocol — Version 1.0. RFC 2246, Internet Engineering Task Force (IETF), January 1999. Available from `http://www.rfc-editor.org/rfc/rfc2246.txt`.

[DBCF95] N. Davies, G.S. Blair, K. Cheverst, and A. Friday. A Network Emulator to Support the Development of Adaptive Applications. In *2nd USENIX Symposium on Mobile and Location Independent Computing*, Ann Arbor, MI, April 1995.

[DDLS01] Nicodemos Damianou, Naranker Dulay, Emil Lupu, and Morris Sloman. The Ponder Policy Specification Language. In *Proceedings of the Workshop on Policies for Distributed Systems and Networks (Policy2001)*, Bristol, UK, January 2001.

[Den64] J. B Dennis. A Multiuser Computation Facility for Education and Research. *Communications of the ACM*, 7:521–529, September 1964.

[DH76] Whitfield Diffie and Martin E. Hellman. New Directions in Cryptography. *IEEE Transactions on Information Theory*, IT-22(6):644–654, 1976.

[DHH+03] Boris Dragovic, Steven Hand, Tim Harris, Evangelos Kotsovinos, and Andrew Twigg. Managing Trust and Reputation in the XenoServer Open Platform. In *Proceedings of the 1st International Conference on Trust Management*, pages 59–64, Heraklion, Crete, Greece, May 2003. Also published in Springer-Verlag Lecture Notes in Computer Science (LNCS), Volume 2692, pp. 59-74.

[DI89] M. V. Devarakonda and R. K. Iyer. Predictability of Process Resource Usage: A Measurement-Based Study on UNIX. *IEEE Transactions on Software Engineering*, 15(12):1579–1586, 1989.

[Dik01] Jeff Dike. User-mode Linux. In *Proceedings of the 5th Annual Linux Showcase and Conference*, pages 3–14, Oakland, CA, November 2001.

[DKHP03] Boris Dragovic, Evangelos Kotsovinos, Steven Hand, and Peter Pietzuch. XenoTrust: Event-Based Distributed Trust Management. In *Proceedings of the Second IEEE International Workshop on Trust and Privacy in Digital Business (DEXA Workshop)*, pages 410–414, Prague, Czech Republic, September 2003.

[Dou02] John R. Douceur. The Sybil Attack. In *Revised papers from the 1st International Workshop on Peer-to-Peer Systems (IPTPS '02)*, pages 251–260, Cambridge, MA, USA, March 2002. Springer-Verlag.

[DS68] Daniel S. Diamond and Lee L. Selwyn. Considerations for Computer Utility Pricing Policies. In *Proceedings of the 23rd ACM National Conference*, pages 189–200, Washington, DC, 1968.

[Dun99] L. Dunn. The Internet2 Project. *The Internet Protocol Journal*, 2(4), December 1999.

[Ele97] Electronic Arts Inc. Ultima Online, 1997. On-line Multiplayer Computer Game. Web Site at `http://www.uo.com`.

[Ens78] P.H. Enslow Jr. What Is a 'Distributed' Data Processing System? *Computer*, 11(1):13–21, January 1978.

[Ens03] Ensim Corporation. Ensim Virtual Private Server. Datasheet, 2003. Available from `http://www.ensim.com/products/materials/datasheet_vps_051003.pdf`.

[Fan65] R. M. Fano. The MAC System: The Computer Utility Approach. *IEEE Spectrum*, pages 56–64, January 1965.

[FC90] R. F. Freund and D. S. Conwell. Superconcurency: A Form of Distributed Heterogeneous Supercomputing. *Supercomputing Review*, October 1990.

[FFK+97] S. Fitzgerald, I. Foster, C. Kesselman, G. von Laszewski, W. Smith, and S. Tuecke. A Directory Service for Configuring High-Performance Distributed Computations. In *Proceedings of the 6th IEEE Symposium on High Performance Distributed Computing (HPDC6)*, pages 365–375, Portland, Oregon, 1997. IEEE Computer Society Press.

[FGK04] I. Foster, D. Gannon, and H. Kishimoto. Open Grid Services Architecture (OGSA) - Version 019. Technical Report, Global Grid Forum, July 2004. Available from `www.ggf.org/ogsa-wg/`.

[FHH+03] Keir Fraser, Steven Hand, Tim Harris, Ian Leslie, and Ian Pratt. The Xenoserver Computing Infrastructure. Technical Report UCAM-CL-TR-552, Computer Laboratory, University of Cambridge, January 2003.

[Fit01] Steven Fitzgerald. Grid Information Services for Distributed Resource Sharing. In *Proceedings of the 10th IEEE International Symposium on High Performance Distributed Computing (HPDC10)*, page 181, San Francisco, CA, 2001. IEEE Computer Society.

[FK92] D. Ferraiolo and R. Kuhn. Role-Based Access Controls. In *Proceedings of the 15th NIST-NCSC National Computer Security Conference*, pages 554–563, Baltimore, MD, 1992.

[FK97] I. Foster and C. Kesselman. Globus: A Metacomputing Infrastructure Toolkit. *The International Journal of Supercomputer Applications and High Performance Computing*, 11(2):115–128, 1997.

[FKL+99] I. Foster, C. Kesselman, C. Lee, R. Lindell, K. Nahrstedt, and A. Roy. A Distributed Resource Management Architecture that Supports Advance Reservations and Co-Allocation. In *Proceedings of the International Workshop on Quality of Service (IWQoS '99)*, London, UK, June 1999.

[FKNT02] I. Foster, C. Kesselman, J. Nick, and S. Tuecke. The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration. GGF Working Draft, Global Grid Forum, Open Grid Service Infrastructure (OGSA) Working Group, June 2002.

[FKOD04] Alberto Fernandes, Evangelos Kotsovinos, Sven Östring, and Boris Dragovic. Pinocchio: Incentives for Honest Participation in Distributed Trust Management. In *Proceedings of the 2nd International Conference on Trust Management (iTrust 2004)*, pages 63–77, Oxford, UK, March 2004. Also published in Springer-Verlag Lecture Notes in Computer Science (LNCS), Volume 2995, pp. 63-77.

[FKT01] Ian Foster, Carl Kesselman, and Steven Tuecke. The Anatomy of the Grid: Enabling Scalable Virtual Organization. *The International Journal of High Performance Computing Applications*, 15(3):200–222, 2001.

[Fly66] M. J. Flynn. Very High-Speed Computing Systems. In *Proceedings of the IEEE*, volume 54, pages 1901–1909, December 1966.

[Fre89] R. F. Freund. Optimal Selection Theory for Superconcurrency. In *Proceedings of Supercomputing '89*, pages 699–703, Reno, Nevada, November 1989.

[FTF+02] James Frey, Todd Tannenbaum, Ian Foster, Miron Livny, and Steve Tuecke. Condor-G: A Computation Management Agent for Multi-Institutional Grids. *Cluster Computing*, 5:237–246, 2002.

[Gel03] Jacques Gelinas. Virtual Private Servers and Security Contexts, 2003. Available from `http://www.solucorp.qc.ca/miscprj/s_context.hc`.

[GGKS02] K. Gottschalk, S. Graham, H. Kreger, and J. Snell. Introduction to Web Services Architecture. *IBM Systems Journal*, 41(2):170–177, 2002.

[Gho02] Atanu Ghosh. Towards the Rapid Network-Wide Deployment of New Application Specific Network Protocols, Using Application Level Active Networking, January 2002. PhD dissertation, University College London.

[Glo00] The Globus Alliance. *The Globus Resource Specification Language RSL v1.0*, February 2000. Available from `http://www.globus.org/gram/rsl_spec1.html`.

[Glo03a] Global Grid Forum, Distributed Resource Management Application API (DRMAA) Working Group. Advance Reservations: State of the Art. GGF Working Draft ggf-draft-sched-graap-2.0, June 2003. Available from `http://www.fz-juelich.de/zam/RD/coop/ggf/graap/graap-wg.html`.

[Glo03b] Global Grid Forum, Grid Resource Allocation Agreement Protocol (GRAAP) Working Group. Presentation in the GGF7 meeting, March 2003. Available from `http://www.fz-juelich.de/zam/RD/coop/ggf/graap/graap-wg.html`.

[GMC+01] V. Galtier, K. Mills, Y. Carlinet, S. Bush, and A. Kulkarni. Predicting and Controlling Resource Usage in a Heterogeneous Active Network. In *Proceedings of the Third Annual International Workshop on Active Middleware Services (AMS)*, page 35, San Francisco, CA, August 2001. IEEE Computer Society.

[GMS+01] E. Gialama, E. Markatos, J. Sevasslidou, D. Serpanos, E. Kotsovinos, and X. Asimakopoulou. DIVISOR: Distributed Video Server For Streaming. In *Proceedings of the 5th IEEE/WSES International Conference on Circuits, Systems, Communications and Computers (CSCC)*, pages 4531–4536, Rethymno, Crete, Greece, June 2001.

[Gol00] Germán Goldszmidt. The Océano Project - A Multi-Domain Cluster for a Computing Utility. In *Proceedings of the 2nd IEEE International Conference on Cluster Computing (CLUSTER 2000)*, Chemnitz, Germany, November 2000.

[Gon02] Li Gong. Project JXTA: A Technology Overview. White Paper, Sun Microsystems, Inc., October 2002. Available from `http://www.jxta.org/`.

[Gra78] Jim Gray. Notes on Data Base Operating Systems. In *Operating Systems, An Advanced Course*, pages 393–481. Springer-Verlag, 1978.

[Gri03] Grid.org. The PatriotGrid - A Global Effort to Combat Bioterrorism, February 2003. Web site at `http://www.grid.org/projects/patriot.htm`.

[GS92] G. A. Geist and V. S. Sunderam. Network-Based Concurrent Computing on the PVM System. *Concurrency: Practice and Experience*, 4(4):293–311, 1992.

[Hay96] Richard Hayton. An Open Architecture for Secure Interworking Services. Technical Report UCAM-CL-TR-399, University of Cambridge, Computer Laboratory, June 1996.

[HBM98] R. J. Hayton, J. M. Bacon, and K. Moody. Access Control in an Open Distributed Environment. In *Proceedings of the 1998 IEEE Symposium on Security and Privacy*, Oakland, CA, May 1998.

[HCK95] Colin G. Harrison, David M. Chess, and Aaron Kershenbaum. Mobile Agents: Are They a Good Idea? Tehnical Report, IBM Research Division, T. J. Watson Research Center, Yorktown Heights, New York, 1995.

[HH93] L. B. Huston and P. Honeyman. Disconnected Operation for AFS. In *Proceedings of the USENIX Mobile and Location-Independent Computing Symposium*, pages 1–10, Cambridge, MA, 2–3 1993.

[HHKP03] Steven Hand, Timothy L Harris, Evangelos Kotsovinos, and Ian Pratt. Controlling the XenoServer Open Platform. In *Proceedings of the 6th International Conference on Open Architectures and Network Programming (OPENARCH)*, San Francisco, CA, April 2003.

[HKM+98] Michael Hicks, Pankaj Kakkar, Jonathan T. Moore, Carl A. Gunter, and Scott Nettles. PLAN: A Packet Language for Active Networks. In *Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming Languages*, pages 86–93, London, UK, January 1998. ACM Press.

[HL73] Hurwicz and Leonid. The Design of Mechanisms for Resource Allocation. *American Economic Review*, 63(2):1–30, May 1973.

[HM02] J. Hodges and R. Morgan. Lightweight Directory Access Protocol (v3): Technical Specification. RFC 3377, Network Working Group, September 2002.

[HP03] Infrastructure and Management Solutions for the Adaptive Enterprise. White Paper, Hewlett Packard, May 2003. Available from `http://whitepapers.zdnet.co.uk/`.

[HSL+03] Mike Hibler, Leigh Stoller, Jay Lepreau, Robert Ricci, and Chad Barb. Fast, Scalable Disk Imaging with Frisbee. In *Proceedings of the 2003 USENIX Annual Technical Conference*, pages 283–296, San Antonio, TX, June 2003. USENIX Association.

[IBM98] Using IPSEC to Construct Secure Virtual Private Networks. White Paper, IBM, 1998. Available from `http://whitepapers.zdnet.co.uk`.

[IBM02] IBM. Living in an On Demand World. White Paper, October 2002. Available from `http://www-306.ibm.com/e-business/doc/content/literature/literature_ebu%siness.html`.

[Ide04] Best Practices for Resolving Business-Critical Application Problems. White paper, Identify Software, October 2004. Available from `http://itresearch.forbes.com/`.

[ITI04] P2P and Music Statistics for September 2004. Press Release, IT Innovations and Concepts (ITIC), October 2004. Available from `http://www.itic.ca/DIC/News/`.

[Jac04] Robert Jacques. Application Downtime Costs More Than $100K An Hour, April 2004. Article in vnunet.com. Available from `http://www.vnunet.com/News/1154519`.

[JD96] D. Jonscher and K. R. Dittrich. Argos – A Configurable Access Control System for Interoperable Environments. In *Database Security, IX: Status and Prospects*, pages 43–60. Chapman & Hall, 1996.

[JXT01] Project JXTA: An Open, Innovative Collaboration. White Paper, Sun Microsystems, Inc., April 2001. Availble from `http://www.jxta.org`.

[KA98] S. Kent and R. Atkinson. Security Architecture for the Internet Protocol. RFC 2401, Internet Engineering Task Force (IETF), 1998. Available from `http://www.ietf.org/rfc/rfc2401.txt`.

[Kay03] John Kay. *The Truth About Markets: Their Genius, Their Limits, Their Follies*. Allen Lane, May 2003.

[Ken81] S. Kent. Protecting Externally Supplied Software in Small Computers. Tehnical Report MIT-LCS-TR-255, Massachusetts Institute of Technology, January 1981. Available from `http://www.ncstrl.org/`.

[KH03] Evangelos Kotsovinos and Tim Harris. Role-Based Resource Management. In *Proceedings of the 8th CaberNet Radicals Workshop*, Corsica, France, October 2003.

[KHG00] Jeffrey O. Kephart, James E. Hanson, and Amy R. Greenwald. Dynamic Pricing by Software Agents. *Computer Networks*, 32(6):731–752, 2000.

[KMP+04] Evangelos Kotsovinos, Tim Moreton, Ian Pratt, Russ Ross, Keir Fraser, Steven Hand, and Tim Harris. Global-Scale Service Deployment in the XenoServer Platform. In *Proceedings of the First Workshop on Real, Large Distributed Systems (WORLDS '04)*, San Francisco, CA, December 2004.

[KRRS04] Brad Karp, Sylvia Ratnasamy, Sean Rhea, and Scott Shenker. Spurring Adoption of DHTs with OpenHash, a Public DHT Service. In *Proceedings of the 3nd International Workshop on Peer-to-Peer Systems (IPTPS '04)*, San Diego, CA, February 2004.

[KS91] J. J. Kistler and M. Satyanarayanan. Disconnected Operation in the Coda File System. In *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles (SOSP13)*, volume 25, pages 213–225, Pacific Grove, CA, 1991. ACM Press.

[KS02] Michael Kozuch and M. Satyanarayanan. Internet Suspend/Resume. In *Proceedings of the Fourth IEEE Workshop on Mobile Computing Systems and Applications*, page 40, Callicoon, New York, June 2002. IEEE Computer Society.

[KS03] E. Kotsovinos and D. Spence. The XenoServer Open Platform: Deploying Global-Scale Services for Fun and Profit. In *Proceedings of ACM SIGCOMM '03 (poster session)*, Karlsruhe, Germany, August 2003.

[KWL+03] K. Keahey, V. Welch, S. Lang, B. Liu, and S. Meder. Fine-Grain Authorization Policies in the GRID: Design and Implementation. In *1st International Workshop on Middleware for Grid Computing*, Rio de Janeiro, Brazil, June 2003.

[Lam86] Butler Lampson. Personal Distributed Computing: The Alto and Ethernet Software. In *Proceedings of the ACM Conference on The history of personal workstations*, pages 101–131, Palo Alto, CA, 1986. ACM Press.

[LBRT97] Miron Livny, Jim Basney, Rajesh Raman, and Todd Tannenbaum. Mechanisms for High Throughput Computing. *SPEEDUP Journal*, 11(1), June 1997.

[LGT+01] Jason Lee, Dan Gunter, Brian Tierney, Bill Allcock, Joe Bester, John Bresnahan, and Steve Tuecke. Applied Techniques for High Bandwidth Data Transfers Across Wide Area Networks. In *Proceedings of International Conference on Computing in High Energy and Nuclear Physics*, Beijing, China, September 2001.

[Lin03] LinkPro Technologies. Web Site Mirroring From A Staging Server To Multiple Web Servers. White Paper, 2003. Available from `http://whitepapers.zdnet.co.uk`.

[LJ03] R. Lepro and S. Jackson. Usage Record - XML Format. GGF Working Draft, Global Grid Forum, Usage Record (UR) Working Group, December 2003. Available from `http://www.psc.edu/~lfm/Grid/UR-WG/`.

[LLM88] Michael Litzkow, Miron Livny, and Matthew Mutka. Condor – A Hunter of Idle Workstations. In *Proceedings of the 8th International Conference of Distributed Computing Systems (ICDCS)*, San Jose, CA, June 1988.

[LLS02] Leonidas Lymberopoulos, Emil Lupu, and Morris Sloman. An adaptive policy based management framework for differentiated services networks. In *3rd International Workshop on Policies for Distributed Systems and Networks (POLICY '02)*, pages 147–158. IEEE Computer Society, June 2002.

[LMSY96] Emil C. Lupu, Damian A. Marriott, Morris S. Sloman, and Nicholas Yialelis. A Policy Based Role Framework for Access Control. In *Proceedings of the first ACM Workshop on Role-Based Access Control*, page 11, Gaithersburg, Maryland, USA, 1996. ACM Press.

[LMW02] Ninghui Li, John C. Mitchell, and William H. Winsborough. Design of a Role-Based Trust Management Framework. In *Proceedings of the IEEE Symposium on Security and Privacy*, Oakland, CA, May 2002.

[LPL+03] Markus Lorch, Seth Proctor, Rebekah Lepro, Dennis Kafura, and Sumit Shah. First experiences using xacml for access control in distributed systems. In *XMLSEC '03: Proceedings of the 2003 ACM workshop on XML security*, pages 25–37. ACM Press, 2003.

[LS79] Butler Lampson and Howard Sturgis. Crash Recovery in a Distributed Data Storage System. Unpublished Tehnical Report, Xerox Palo Alto Research Center, June 1979. Available from `http://research.microsoft.com/Lampson/`.

[LS99] Emil C. Lupu and Morris Sloman. Conflicts in Policy-Based Distributed Systems Management. *IEEE Transactions on Software Engineering*, 25(6):852–869, 1999.

[LSP03] Stefan M. Larson, Christopher D. Snow, and Vijay S. Pande. Folding@Home and Genome@Home: Using Distributed Computing to Tackle Previously Intractable Problems in Computational Biology. *Modern Methods in Computational Biology*, 2003.

[Mac04] Rodney Mach. Accounting Interchange Natural Language Description. GGF Working Draft, Global Grid Forum, Usage Record (UR) Working Group, April 2004. Available from `http://www.psc.edu/~lfm/Grid/UR-WG/`.

[MBHJ98] D. Marinescu, L. Blni, R. Hao, and K. Jun. An Alternative Model for Scheduling on a Computational Grid. In *Proceedings of the Thir-*

teenth International Symposium on Computer and Information Sciences, Antalya, Turkey, October 1998.

[MC04] Johan Muskens and Michel R. V. Chaudron. Prediction of Run-Time Resource Consumption in Multi-task Component-Based Software Systems. In *Proceedings of the 7th International Symposium on Component-Based Software Engineering (CBSE7)*, Edinburgh, Scotland, May 2004.

[Mic03] Privacy-Enabling Enhancements in the Next-Generation Secure Computing Base. White paper, Microsoft Corporation, November 2003. Available at `http://www.microsoft.com/resources/ngscb/productinfo.mspx`.

[MJK94] G. Mansfield, T. Johannsen, and M. Knopper. Charting Networks in the X.500 Directory. RFC 1609, Network Working Group, March 1994. Available from `ftp://ftp.internic.net/rfc/rfc1609.txt`.

[MKKW99] David Mazieres, Michael Kaminsky, M. Frans Kaashoek, and Emmett Witchel. Separating Key Management from File System Security. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles (SOSP17)*, pages 124–139, Kiawah Island, South Carolina, USA, December 1999.

[MMS03] A. Maedche, B. Motik, and L. Stojanovic. Managing Multiple and Distributed Ontologies on the Semantic Web. *The VLDB Journal*, 12(4):286–302, 2003.

[MPH02] T. Moreton, I. Pratt, and T. Harris. Storage, Mutability and Naming in *Pasta*. In *Proceedings of the International Workshop on Peer-to-Peer Computing at Networking 2002*, Pisa, Italy, May 2002.

[MW77] J. M. McQuillan and D. C. Walden. The ARPANET Design Decisions. *Computer Networks*, 1(5), August 1977.

[MZE02] C. Mascolo, L. Zanolin, and W. Emmerich. XMILE: an XML based Approach for Incremental Code Mobility and Update. Automated Software Engineering. 9(2):151–165, 2002.

[Nec97] George C. Necula. Proof-Carrying Code. In *Conference Record of POPL '97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL97)*, pages 106–119, Paris, France, January 1997.

[New03] S. Newhouse. Grid Economic Services. GGF Working Draft, Global Grid Forum, Grid Economic Services Architecture (GESA) Working Group, June 2003. Available from `http://www.doc.ic.ac.uk/~sjn5/GGF/gesa-wg.html`.

[New04] S. Newhouse. Resource Usage Service. GGF Working Draft draft-ggf-rus-service-1, Global Grid Forum, Resource Usage Service (RUS) Working Group, February 2004. Available from `http://www.doc.ic.ac.uk/~sjn5/GGF/rus-wg.html`.

[NFS89] NFS: Network File System Protocol Specification. RFC 1094, Network Working Group, March 1989. Available from `http://www.faqs.org/rfcs/rfc1094.html`.

[NH82] R.M. Needham and A.J Herbert. *The Cambridge Distributed Computing System*. Addison-Wesley, November 1982.

[NO95] Matunda Nyanchama and Sylvia Osborn. Access Rights Administration in Role-Based Security Systems. In *Proceedings of the 8th IFIP WG 11.3 Working Conference on Database Security*, volume A-60 of *IFIP Transactions*, Bad Salzdetfurth, Germany, August 1995. North-Holland (Elsevier).

[NSN+97] B. D. Noble, M. Satyanarayanan, D. Narayanan, J. E. Tilton, J. Flinn, and K. R. Walker. Agile Application-Aware Adaptation for Mobility. In *Symposium on Operating Systems Principles (SOSP '97)*, pages 276–287, Kiawah Island, South Carolina, USA, 1997. ACM Press.

[NT94] B. Clifford Neuman and Theodore Ts'o. Kerberos: An Authentication Service for Computer Networks. *IEEE Communications Magazine*, 32(9):33–38, 1994.

[OAPV04] David Oppenheimer, Jeannie Albrecht, David Patterson, and Amin Vahdat. Distributed Resource Discovery on PlanetLab with SWORD. In *Proceedings of the First Workshop on Real, Large Distributed Systems (WORLDS '04)*, San Francisco, CA, December 2004.

[Obj91] Object Management Group and X/Open. The Common Object Request Broker: Architecture and Specification (CORBA). Technical Report 91.12.1, Object Management Group (OMG), 1991. Available from `http://www.omg.org`.

[OR04] Angela D. Orebaugh and Gilbert Ramirez. *Ethereal Packet Sniffing*. Syngress Publishing, February 2004.

[OW99] P. C. Van Oorschot and M. J. Wiener. Parallel Collision Search with Cryptanalytic Applications. *Journal of Cryptology*, 12(1):1–28, 1999.

[Par00] Sanjay Parthasarathy. The Simplest Way to Define .NET. White Paper, Microsoft Corporation, December 2000. Available from `http://www.microsoft.com/malaysia/net/whitepapers.htm`.

[Pat02] Dave Patterson. A New Focus for a New Century: Availability and Maintainability → Performance, January 2002. Keynote talk in the Conference on File and Storage Technologies (FAST 2002). Available from `http://www.usenix.org/publications/library/proceedings/fast02/`.

[PB02] Peter R. Pietzuch and Jean Bacon. Hermes: A Distributed Event-Based Middleware Architecture. In *Proceedings of the 22nd International Conference on Distributed Computing Systems (ICDCS)*, pages 611–618, Vienna, Austria, July 2002. IEEE Computer Society.

[PCAR02] Larry Peterson, David Culler, Tom Anderson, and Timothy Roscoe. A Blueprint for Introducing Disruptive Technology into the Internet. In *Proceedings of the 1st Workshop on Hot Topics in Networks (HotNets-I)*, Princeton, New Jersey, USA, October 2002.

[PP04] Calicrates Policroniades and Ian Pratt. Alternatives for Detecting Redundancy in Storage Systems Data. In *USENIX 2004 Annual Technical Conference*, pages 73–86, Boston, MA, June 2004.

[PPD+95] R. Pike, D. Presotto, S. Dorward, B. Flandrena, K. Thompson, H. Trickey, and P. Winterbottom. Plan 9 from Bell Labs. *Computing Systems*, 8(3):221–254, 1995.

[PPT+92] R. Pike, D. Presotto, K. Thompson, H. Trickey, and P. Winterbottom. The Use of Name Spaces in Plan 9. In *Proceedings of the 5th ACM SIGOPS European Workshop*, pages 72–76, Mont Saint-Michel, France, September 1992.

[PPTH72] R. Parmelee, T. Peterson, C. Tillman, and D. Hatfield. Virtual Storage and Virtual Machine Concepts. *IBM Systems Journal*, 11(2):99–130, 1972.

[PS01] Deval Parikh and Mohanbir Sawhney. Where Value Lives in a Networked World. *Harvard Business Review*, pages 79–86, January 2001.

[Pul03] Darren W. Pulsipher. Policy Use Cases for Grid Systems. GGF Informational Memo, Global Grid Forum, Grid Policy (POLICY) Research Group, August 2003. Available from `http://forge.gridforum.org/projects/policy-rg/`.

[PV02] Larry Peterson and Amin Vahdat. Dynamic Slice Creation. Technical Report PDN–02–005, PlanetLab Consortium, October 2002.

[RBC+04] Hrabri Rajic, Roger Brobst, Waiman Chan, Jeff Gardiner, John P. Robarts, Andreas Haasand, Bill Nitzberg, Hrabri Rajic, and John Tollefsrud. Distributed Resource Management Application API Specification 1.0. GGF Recommendation, Global Grid Forum, Distributed Resource Management Application API (DRMAA) Working Group, June 2004. Available from `http://www.drmaa.org/`.

[RBTD99] Mike Rizzo, Bob Briscoe, J. Tassel, and K. Damianakis. A Dynamic Pricing Framework to Support a Scalable, Usage-Based Charging Model for Packet-Switched Networks. In *IWAN '99: Proceedings of the First International Working Conference on Active Networks*, pages 48–59. Springer-Verlag, 1999.

[REG+03] Sean Rhea, Patrick Eaton, Dennis Geels, Hakim Weatherspoon, Ben Zhao, and John Kubiatowicz. Pond: The OceanStore Prototype. In *Proceedings of the 2nd USENIX Conference on File and Storage Technologies (FAST'03)*, San Francisco, CA, March 2003.

[Rit01] Jordan Ritter. Why Gnutella Can't Scale. No, Really, February 2001. Unpublished. Available from `http://www.darkridge.com/~jpr5/doc/gnutella.html`.

[RLS98] Rajesh Raman, Miron Livny, and Marvin Solomon. Matchmaking: Distributed Resource Management for High Throughput Computing. In *Proceedings of the Seventh IEEE International Symposium on High Performance Distributed Computing (HPDC7)*, Chicago, IL, July 1998.

[RLS00] Rajesh Raman, Miron Livny, and Marvin Solomon. Resource Management through Multilateral Matchmaking. In *Proceedings of the Ninth IEEE Symposium on High Performance Distributed Computing (HPDC9)*, pages 290–291, Pittsburgh, PA, August 2000.

[RLS03] Rajesh Raman, Miron Livny, and Marvin Solomon. Policy Driven Heterogeneous Resource Co-Allocation with Gangmatching. In *Proceedings of the Twelfth IEEE International Symposium on High Performance Distributed Computing (HPDC12)*, Seattle, WA, June 2003.

[Rob95] B. Robertson. Toy Story: A Triumph of Animation. *Computer Graphics World*, 18(8):28–38, August 1995.

[Rog98] Ron Rogerson. Circular 3/98: JANET Network Charges. White Paper, Joint Information Systems Committee, March 1998. Available from `http://www.jisc.ac.uk/index.cfm?name=news_circular_3_98`.

[RPM+99] Dickon Reed, Ian Pratt, Paul Menage, Stephen Early, and Neil Stratford. Xenoservers: Accounted Execution of Untrusted Code. In *Proceedings of the 7th Workshop on Hot Topics in Operating Systems (HotOS-VII)*, Rio Rico, AZ, 1999. IEEE Computer Society Press.

[RRPK01] David Ratner, Peter Reiher, Gerald J. Popek, and Geoffrey H. Kuenning. Replication Requirements in Mobile Environments. *Mobile Networks and Applications*, 6(6):525–533, 2001.

[RT04] Stephan Reiff-Marganiec and Kenneth J. Turner. Feature Interaction in Policies. *Computer Networks*, 45(5):569–584, 2004.

[RTBS01] Paul Rutten, Mickey Tauman, Hagai Bar-Lev, and Avner Sonnino. Is Moore's Law Infinite? The Economics Of Moore s Law. White Paper, Kellogg School of Management, Northwestern University, 2001. In Kellogg TechVenture 2001 Anthology. Available from `http://www.ranjaygulati.com/new/research/ISMOORES.pdf`.

[Rym01] Arthur Ryman. Simple Object Access Protocol (SOAP) and Web Services. In *Proceedings of the 23rd International Conference on Software Engineering*, page 689, Toronto, Ontario, Canada, May 2001. IEEE Computer Society.

[Sat92] Mahadev Satyanarayanan. The Influence of Scale on Distributed File System Design. *IEEE Transactions on Software Engineering*, 18(1):1–8, 1992.

[Sat04] Tetsuya Sato. Annual Report of the Earth Simulator Center. Tehnical Report, The Earth Simulator Center, Japan Agency for Marine-Earth Science and Technology, February 2004. Available from `http://www.es.jamstec.go.jp/`.

[SB78] Harold S. Stone and S. H. Bokhari. Control of Distributed Processes. *IEEE Computer*, 11(7):97–106, July 1978.

[SC92] Larry Smarr and Charles E. Catlett. Metacomputing. *Communications of the ACM*, 35(6):44–52, 1992.

[SH82] John F. Shoch and Jon A. Hupp. The "Worm" Programs — Early Experience with a Distributed Computation. *Communications of the ACM*, 25(3):172–180, 1982.

[SH03] David Spence and Tim Harris. XenoSearch: Distributed Resource Discovery in the XenoServer Open Platform. In *Proceedings of the Twelfth IEEE International Symposium on High Performance Distributed Computing (HPDC12)*, Seattle, WA, June 2003.

[She99] S. Shepler. NFS Version 4 Design Considerations. RFC 2624, Network Working Group, June 1999.

[SHH04] David Spence, Steven Hand, and Tim Harris. XenoSearch II: Distributed Location-Based Server Co-Selection, 2004. Submitted to a refereed international conference, currently under review.

[SLDI01] James Edward Stada, Steven Logghe, Griet De Ceuster, and Lambertus H. Immers. Time-Of-Day Modeling Using a Quasi-Dynamic Equilibrium Assignment Approach. In *Proceedings of TRISTAN IV*, São Miguel, Azores Islands, Portugal, June 2001.

[SM99] Neil Stratford and Richard Mortier. An Economic Approach to Adaptive Resource Management. In *Proceedings of the fifth Workshop on Hot Topics in Operating Systems (HotOS-VII)*, page 142, Rio Rico, AZ, 1999. IEEE Computer Society Press.

[Son99] Sony Online. Everquest, 1999. On-line Multiplayer Computer Game. Web Site at `http://www.everquest.com`.

[SPM04] Jeffrey Shneidman, David C. Parkes, and Laurent Massoulié. Faithfulness in Internet Algorithms. In *Proceedings of the ACM SIGCOMM Workshop on Practice and Theory of Incentives in Networked Systems*, pages 220–227, Portland, Oregon, USA, September 2004.

[Sri01] Kunwadee Sripanidkulchai. The Popularity of Gnutella Queries and Its Implications on Scalability. White Paper, Carnegie Mellon University, February 2001. Featured on O'Reilly's `http://www.openp2p.com` website. Available from `http://www-2.cs.cmu.edu/~kunwadee/research/p2p/gnutella.html`.

[SS75] Jerome H. Saltzer and Michael D. Schroeder. The Protection of Information in Computer Systems. *Proceedings of the IEEE*, 63(9):1278–1308, September 1975.

[SS96] M. Spasojevic and M. Satyanarayanan. An Empirical Study of a Wide-Area Distributed File System. *ACM Transactions on Computer Systems*, 14(2):200–222, 1996.

[Sun90] V. S. Sunderam. PVM: a Framework for Parallel Distributed Computing. *Concurrency: Practice and Experience*, 2(4):315–339, 1990.

[Sun99] Sun Microsystems, Inc., Palo Alto, California. *Java Remote Method Invocation Specification*, Revision 1.7, Java 2 SDK edition, December 1999.

[Sun02] N1 Grid - Introducing Just In Time Computing. White Paper, Sun microsystems, 2002. Available from `http://wwws.sun.com/software/solutions/n1/docs.html`.

[TCF+03] S. Tuecke, K. Czajkowski, I. Foster, J. Frey, S. Graham, C. Kesselman, T.Maquire, T. Sandholm, D. Snelling, and P. Vanderbilt. Open Grid Services Infrastructure (OGSI) - Version 1.0. Technical Report draft-ggf-ogsi-gridservice-33, Global Grid Forum, June 2003. Available from `http://www.ggf.org/ogsi-wg/`.

[Tch04] Dmitri Tcherevik. Managing Web Services With Unicenter Web Services Distributed Management. White Paper, Computer Associates, May 2004.

[TL03] Douglas Thain and Miron Livny. Building Reliable Clients and Servers. In Ian Foster and Carl Kesselman, editors, *The Grid:*

Blueprint for a New Computing Infrastructure. Morgan Kaufmann, 2003.

[TTL04] Douglas Thain, Todd Tannenbaum, and Miron Livny. Distributed Computing In Practice: The Condor Experience. *Concurrency and Computation: Practice and Experience*, To appear, 2004.

[TW96] David L. Tennenhouse and David J. Wetherall. Towards an Active Network Architecture. *SIGCOMM Computer Communication Review*, 26(2):5–17, 1996.

[TXKN03] P. Triantafillou, C. Xiruhaki, M. Koubarakis, and N. Ntarmos. Towards High Performance Peer-to-Peer Content and Resource Sharing Systems. In *Proceedings of the 1st Biennial Conference on Innovative Data Systems Research (CIDR 2003)*, Pacific Grove, CA, January 2003.

[vD00] Leendert van Doorn. A Secure Java Virtual Machine. In *Proceedings of the 9th USENIX Security Symposium*, pages 19–34, Denver, Colorado, August 2000.

[vLF98] G. von Laszewski and I. Foster. Usage of LDAP in Globus. Short Note, The Globus Alliance, April 1998. Available from `http://www.globus.org`.

[VMW99] VMware Virtual Platform. White Paper, VMware Inc, 1999.

[WCL+01] Dan Werthimer, Jeff Cobb, Matt Lebofsky, David Anderson, and Eric Korpela. SETI@HOME - Massively Distributed Computing for SETI. *Computing in Science and Engineering*, 3(1):78–83, 2001.

[WFLY04] Xiaoyun Wang, Dengguo Feng, Xuejia Lai, and Hongbo Yu. Collisions for Hash Functions MD4, MD5, HAVAL-128 and RIPEMD, August 2004. In Cryptology ePrint Archive. Available from `http://eprint.iacr.org/`.

[WGT98] D. Wetherall, J. Guttag, and D. Tennenhouse. ANTS: A Toolkit for Building and Dynamically Deploying Network Protocols. In *Proceedings of the First IEEE Conference on Open Architectures and Network Programming (IEEE OPENARCH)*, San Francisco, CA, April 1998.

[WGT99] David Wetherall, John Guttag, and David Tennenhouse. ANTS: Network Services Without the Red Tape. *IEEE Computer*, 32(4):42–48, April 1999.

[WJOP01] Ian Wakeman, Alan Jeffrey, Tim Owen, and Damyan Pepper. SafetyNet: A Language-Based Approach to Programmable Networks. *Computer Networks*, 36(1):101–114, 2001.

[WPP02] Limin Wang, Vivek Pai, and Larry Peterson. The Effectiveness of Request Redirecion on CDN Robustness. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI '02)*, Boston, MA USA, December 2002.

[WRW96] Ann Wollrath, Roger Riggs, and Jim Waldo. A Distributed Object Model for the Java System. In *Proceedings of the Conference on Object-Oriented Technologies*, pages 219–231, Toronto, Canada, June 1996. Sun Microsystems Laboratories, USENIX. Available online at `http://www.usenix.org/publications/library/proceedings/coots96/wollrath%.html`.

[WS03] A. Westerinen and R. Strechay. Grid Policy Framework Mapping. GGF Working Draft, Global Grid Forum, Grid Policy (POLICY) Research Group, September 2003. Available from `http://forge.gridforum.org/projects/policy-rg/`.

[WSG02] Andrew Whitaker, Marianne Shaw, and Steven D. Gribble. Scale and Performance in the Denali Isolation Kernel. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI 2002)*, ACM Operating Systems Review, Winter 2002 Special Issue, pages 195–210, Boston, MA, USA, December 2002.

[WSS+01] A. Westerinen, J. Schnizlein, J. Strassner, M.Scherling, B. Quinn, S. Herzog, A. Huynh, M. Carlson, J. Perry, and S. Waldbusser. Terminology for Policy-Based Management. RFC 3198, November 2001. Available from `http://www.faqs.org/rfcs/rfc3198.html`.

[Ylo96] T. Ylonen. SSH – Secure Login Connections over the Internet. In *Proceedings of the 6th USENIX Security Symposium*, page 37, San Jose, CA, July 1996.