STRATEGY-PROOF RESOURCE PRICING IN FEDERATED SYSTEMS

MARIAN MIHAILESCU

B. Eng., "POLITEHNICA" UNIVERSITY OF BUCHAREST, ROMANIA

A THESIS SUBMITTED FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

DEPARTMENT OF COMPUTER SCIENCE
NATIONAL UNIVERSITY OF SINGAPORE

2012

DECLARATION

No portion of the work referred to in this thesis has been submitted in support of an application for another degree of qualification at this or any other university or institution of learning.

Abstract

Resource sharing on the Internet is becoming increasingly pervasive. Recently, there is growing interest in federated systems where globally distributed and commoditized resources are traded, such as in peer-to-peer computing, grids, and cloud computing. An important issue in the allocation of shared resources in the context of large federated systems, such as federated clouds, is that users are rational and attempt to maximize their self-interest.

In this thesis, we investigate the use of resource pricing with financial incentives to allocate shared resources when users are *rational*. Using mechanism design, we propose a strategy-proof, VCG-based resource pricing scheme, with provable economic properties that include *individual rationality*, *incentive compatibility* and *budget balance*. We show that *multiple resource types per consumer request* can be allocated to multiple resource providers with polynomial time complexity. Although we trade-off Pareto efficiency and obtain 80% optimal economic efficiency, we achieve strategy-proof, budget balance and *computational efficiency*, and at the same time increase the number of allocations by 40% under different market conditions.

To improve scalability, we propose a distributed auction scheme that leverages on a peer-to-peer DHT overlay network for resource lookups and distribute pricing by resource type. Simulations and experiments on PlanetLab show that the number of resource types in a consumer request does not affect the scalability of our distributed scheme, and the average allocation time increases logarithmically with the number of users. Lastly, in a federated cloud composed of four Amazon EC2 regions, we show using traces from the Amazon cloud that rational users with dynamic pricing increased user welfare by 10% over EC2 spot pricing. As an application of our pricing scheme, we have prototyped SkyBoxz, a federated cloud platform designed for cloud consumers to utilize and cloud providers to manage virtual machines across multiple public and private clouds.

The end point of rationality is to demonstrate the limits of rationality.

— Blaise Pascal

List of Publications

Conference Papers

1. M. Mihailescu and Y.M. Teo, *The Impact of User Rationality in Federated Clouds*, 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, Ottawa, Canada, May 13–16, 2012.

2. M. Mihailescu and Y.M. Teo, *A Distributed Market Framework for Large-scale Resource Sharing*, Proceedings of Euro-Par, pages 418–430, Ischia, Italy, Aug 31–Sept 3, 2010.

3. M. Mihailescu and Y.M. Teo, *Strategic-Proof Dynamic Resource Pricing of Multiple Resource Types on Federated Clouds*, Proceedings of 10th International Conference on Algorithms and Architectures for Parallel Processing, pages 337–350, Busan, Korea, May 21–23, 2010 (**Best paper award**).

4. M. Mihailescu and Y.M. Teo, *Dynamic Resource Pricing on Federated Clouds*, Proceedings of 10th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, pages 513–517, Melbourne, Australia, May 17–20, 2010.

5. M. Mihailescu and Y.M. Teo, *On Economic and Computational-Efficient Resource Pricing in Large Distributed Systems*, Proceedings of 10th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, pages 838–843, Melbourne, Australia, May 17–20, 2010 (*IEEE TCSC Doctoral Symposium Paper*).

6. Y.M. Teo and M. Mihailescu, *A Strategy-proof Pricing Scheme for Multiple Resource Type Allocations*, Proceedings of 38th International Conference on Parallel Processing, pages 172–179, Vienna, Austria, Sept 22–25, 2009.

Journal Papers

1. M. Mihailescu and Y.M. Teo, *Strategy-proof Dynamic Resource Pricing on Federated Clouds*, Future Generation Computer Systems, Elsevier (*under review*).

2. M. Mihailescu and Y.M. Teo, *A Distributed Framework for Pricing and Allocation of Multiple Resource Types*, Models and Algorithms for High-Performance Data Management and Mining on Computational Grids, Special Issue of Journal of Grid Computing, Springer (*invited submission*).

Acknowledgements

As I write these lines, I find myself overwhelmed by the prospect of finishing such a defining chapter of my life. There are many people to whom I would like to thank for their advices, support, and friendship shown over these past years. I owe immense gratitude to my supervisor, Professor Yong Meng Teo, for his support, starting with my internship in NUS, and continuing during my PhD. Professor Teo was the one who taught me what research is, how to look at a problem, what questions to ask, and how to present my thoughts clearly. Most importantly, he gave me the room to follow my ideas and enjoy the work I was doing. Thank you, Professor Teo, for all your help!

I have to thank my thesis committee, Professors Ben Leong and Tulika Mitra, for their comments and feedback during various stages of my PhD. They have given me their precious time, and influenced my work greatly. I would like to further thank the people I have worked with over the years in NUS: Pham Hai Nam, Aman Bansal, Andrei Deftu, and Zhao Cong. My days in the Computer Systems Laboratory would not have been as enjoyable without the talented colleagues and friends: Dr Verdi March, Seth Hetu, Bogdan Marius Tudor, and Cristina Carbunaru. Thank you all for the memorable discussions about research, life and everything else in between.

To my family and friends, I am so grateful for your support and understanding. I know it was not easy for you. I owe my life-long gratitude to my parents, for raising and supporting me, and for allowing me to find my way so far away from home. To my beloved wife, Claudia, which has been besides me from the beginning, thank you for your love, kindness, patience, and for pushing me to give my best. To Mihai and Mariuca, thank you for your friendship and company in my first years of my PhD. I love you all!

Lastly, to my friends from the “animal farm” in Romania, and to my new-found friends in Singapore, thank you for making the time seem to pass so much faster!

Table of Contents

List of Publications ii

Acknowledgements iv

Table of Contents v

List of Definitions viii

List of Figures ix

List of Tables xi

List of Equations xii

List of Algorithms xiv

1 Introduction 1

1.1 Motivation 3
1.2 Objectives 6
1.3 Contributions 8
1.4 Thesis Organization 9

2 Related Work 12

2.1 Market-based Resource Allocation 13
2.2 Issues in Resource Pricing 16
2.3 Resource Pricing Models 18
2.3.1 Fixed Pricing 19
2.3.2 Equilibrium Price 21
2.3.3 Proportional Share 23
2.3.4 Bargaining 26

2.3.5 Auctions 27
2.4 Summary 33

3 Strategy-proof Resource Pricing **35**
3.1 Preliminaries 36
3.2 Design Considerations 39
3.2.1 Multiple Resource Types per Request 40
3.2.2 Economic and Computational Properties 41
3.2.3 Trade-offs 45
3.3 Mechanism Design Framework 46
3.4 Proposed Scheme 53
3.4.1 Winner Determination 56
3.4.2 Payment Computation 56
3.4.3 Generalized Reverse Auction Algorithm 58
3.4.4 Proofs of Economic Properties 63
3.5 Theoretical Analysis and Limitations 67
3.6 Summary 72

4 Distributed Resource Pricing **74**
4.1 Preliminaries 75
4.2 Issues in Distributed Auctions 77
4.3 Proposed Scheme 79
4.3.1 User Roles 82
4.3.2 Achieved Properties 85
4.3.3 Generalized Distributed Auction Algorithm 86
4.4 Theoretical Analysis and Limitations 91
4.5 Summary 95

5 Federated Cloud Prototype **97**
5.1 Issues and Objectives 98
5.2 Proposed Federated Cloud Architecture 100
5.3 Prototype Implementation 103
5.3.1 SkyBoxz Modules 103
5.3.2 Service Platform 109
5.4 Summary 111

6 Experimental Evaluation 112
6.1 Experimental Setup . . . 113
6.2 Strategy-proof Pricing . . . 116
6.2.1 Trade-off Analysis . . . 117
6.2.2 Economic Efficiency . . . 120
6.2.3 Multiple Resource Types per Request . . . 125
6.3 Distributed Pricing . . . 130
6.3.1 Vertical Scalability . . . 131
6.3.2 Horizontal Scalability . . . 133
6.3.3 Impact of Network Delay . . . 134
6.3.4 Impact of Arrival Rate . . . 135
6.4 User Rationality in Federated Clouds . . . 136
6.4.1 Methodology . . . 137
6.4.2 Results . . . 143
6.5 Summary . . . 148

7 Conclusions 150
7.1 Thesis Summary . . . 151
7.2 Further Directions . . . 157

References 160

A Impact of Rationality in English Auctions 171

B Prototyping Using FreePastry 173

C SkyBoxz Organization 177

List of Definitions

1 Resource Market 13
2 Pricing 15
3 Pricing Mechanism 15
4 Auction 27
5 Social Choice Function 48
6 Mechanism 48
7 Individual Rationality 48
8 Incentive Compatibility 49
9 Strategy-proof Mechanism 49
10 Pareto Efficiency 50
11 Pareto Improvement 50
12 Budget Balance 51
13 Vickrey-Clarke-Groves Mechanism 51
14 Market-based Resource Allocation Problem 54

List of Figures

2.1 Market-based Resource Allocation Models 14
2.2 Pricing Models for Resource Allocation 18
2.3 A Taxonomy of Auction Types 28

3.1 Steps in the Allocation of Shared Resources 37
3.2 Provider Welfare with Fixed and Dynamic (Market) Pricing 39
3.3 Resource Description 40
3.4 Design Considerations 42
3.5 Trade-offs in Resource Pricing 45
3.6 Mechanism Design Framework 47
3.7 Proposed Pricing Mechanism 55
3.8 Market with Two Consumers and Three Providers 61
3.9 Market with Monopolistic Provider 71

4.1 Distributed Resource Pricing Scheme 80
4.2 Two-Phase Commit Protocol 85
4.3 Deadlock in Distributed Auctions 87
4.4 Deadlock-free Protocol 88
4.5 Distributed Allocation Diagram 92

5.1 SkyBoxz Architecture 101
5.2 SkyBoxz Implementation 104
5.3 Example EC2 API Request and Response 106
5.4 SkyBoxz Database 107
5.5 SNAP Architecture 110

6.1 Economic Efficiency Loss 118
6.2 Computational Efficiency Gain 119
6.3 Economic Efficiency vs Computational Efficiency 120

6.4 Comparison with One-sided Auctions 122
6.5 Successful Consumer Requests in Different Market Conditions 125
6.6 Varying the Number of Resource Types 126
6.7 Successful Consumer Requests with Multiple Resource Types 128
6.8 Average Consumer Welfare with Multiple Resource Types 128
6.9 Allocated Provider Resources with Multiple Resource Types 129
6.10 FreePastry Simulator Bootstrap Time 130
6.11 Increasing the Number of Resource Types in a Request 132
6.12 Increasing the Number of Users 133
6.13 Impact of Network Delay 134
6.14 Impact of Arrival Rate 136
6.15 Determining User Welfare on EC2 138
6.16 Amazon EC2 Spot Prices 139
6.17 Spot Price Validation 142
6.18 `c1.medium` Results 144
6.19 `c1.xlarge` Results 146

A.1 Varying the Number of Untruthful Users 172

B.1 Initializing a FreePastry Overlay 174
B.2 Input File for the FreePastry Prototype 176

C.1 Resources on SkyBoxz 179
C.2 Monitoring on SkyBoxz 179
C.3 Cloud Providers on SkyBoxz 180
C.4 Acquire Resources on SkyBoxz 181
C.5 Project Information on SkyBoxz 182
C.6 View Experiments Results on SNAP 183

List of Tables

2.1 Summary of Related Works 33

3.1 Example of Payment Computation 62
3.2 Total Welfare and Selected Winners 62
3.3 VCG Payments for All Winners 63

6.1 Experiments and Tools 113
6.2 Comparison of Proposed Scheme with Combinatorial Auctions 117
6.3 Price Variation under Different Market Scenarios 121
6.4 Comparison with Fixed Pricing 123
6.5 Economic Efficiency with Multiple Resource Types 127

List of Equations

3.1 Social Choice Function . . . 48
3.2 User Welfare . . . 48
3.3 User Welfare with Declared Information . . . 48
3.4 Individual Rationality . . . 48
3.5 Incentive Compatibility . . . 49
3.6 Maximize Global Social Welfare . . . 49
3.7 Maximize Minimal Welfare . . . 49
3.8 Minimize Differences of Welfare . . . 50
3.9 Maximize the Number of Users with Positive Welfare . . . 50
3.10 Budget Balance . . . 51
3.11 VCG Winner Determination . . . 51
3.12 VCG Payment Computation . . . 51
3.13 Proposed Provider Payment Function . . . 57
3.14 Proposed Consumer Payment Function . . . 57
3.15 Provider Payment . . . 64
3.16 Provider Utility . . . 64
3.17 Consumer Utility . . . 64
3.18 Achieving Budget Balance . . . 64
3.19 Optimal Mechanism Output . . . 65
3.20 Provider Valuation . . . 65
3.21 Provider Payment . . . 65
3.22 Achieving Provider Payments Incentive Compatibility . . . 66
3.23 Consumer Payment . . . 66
3.24 Consumer Valuation . . . 67
3.25 Achieving Consumer Payments Incentive Compatibility . . . 67
3.26 Achieving Incentive Compatibility . . . 67
3.27 Allocation Time . . . 68

3.28 Winner Determination Complexity 68
3.29 Allocation Time Complexity 68
3.30 Provider Payment in Monopoly Situation 70
3.31 Consumer Payment in Monopoly Situation 70
4.1 Distributed Allocation Time 93
4.2 Communication Time 93
4.3 Queue Time and Computational Time 94
4.4 Distributed Allocation Time Complexity 94

List of Algorithms

1 MarketMaker 59
2 DetermineWinners 60

3 SendPublish Example 83
4 SendRequest Example 83
5 Request Broker: ReceiveRequest 89
6 Resource Broker: ReceiveLookup 89
7 Request Broker: ReceiveTicket 89
8 Resource Broker: ReceivePosition 90
9 Request Broker: ReceivePayment 90
10 Resource Broker: ReceiveCommit 91

Chapter 1

Introduction

Recent advances in computer technology and the Internet have led to the expansion of distributed systems. Currently, distributed systems such as in peer-to-peer computing, grid computing, and more recently cloud computing, are converging towards federated sharing of computing resources [39, 43, 119]. The common vision in resource sharing pictures computing resources as commodity utilities, where user applications connect to a global pool of resources and use the amount needed. Users of these distributed systems can both consume and provide shared resources. Accordingly, there is growing research interest in large-scale resource sharing, and new, emerging, platforms that enable it [19, 100, 126, 129].

A fundamental problem in the architecture of distributed systems is the allocation of shared resources [7, 42, 95, 124, 125]. Classical resource allocation mechanisms, used in local or cluster environments, assume that users do not deviate from the protocols imposed by the system designer, and thus are not suitable in federated distributed systems, where resources from multiple providers are shared and traded [26, 81, 119]. Recent work in distributed systems acknowledges that users consuming and providing shared resources are self-interested parties with their own goals and objectives [26, 82, 84]. Classified as *rational users*, they can exercise their partial or complete autonomy in or-

der to achieve their individual objectives and maximize their own benefit [109]. They are able to devise strategies and manipulate the system to their advantage, even if by doing so they break the rules of the system. For example, performance in file sharing peer-to-peer systems is affected by free-riders, users that consume more than their fair share [82]; in a computational grid, users compete for the same resources, which results in increased waiting times [126]; some users of SETI@home, a popular distributed computing project, modified the software client to report false-negatives in order to achieve higher rankings [82].

Although rational users are not trusted to follow the deployed algorithms and protocols, it is assumed that they respond to incentives to maximize their personal gain [82]. Accordingly, authors have proposed to study the allocation of shared resources using mechanism design [3], a framework that helps structure incentives such that users behave according to the designed protocols. Mechanism design is a branch of game theory, a mathematical framework commonly used in economics to study strategic situations where the outcome of one participant depends on the behavior of other participants [84]. Generally, game theory attempts to find a set of strategies in which rational users are unlikely to change their behavior. A rational user may represent either an individual user, a group, or an organization, depending on the application context. In computer science, game theory has been studied mostly in multi-agent systems and decision-making in artificial intelligence [49]. However, recent work in peer-to-peer networking [81, 82], grid or cluster computing [125, 129], Internet routing [44, 119], general graph algorithms [39, 43], and resource allocation [68, 129] exploit of mechanism design to create different types of incentives for rational users. When trading and allocating shared resources, mechanism design enables the use of resource pricing to include financial incentives in user payments. Consequently, payments for users that share resources are based on both the cost of sharing declared by the rational user, and the amount of financial incentive necessary for the user to declare the cost truthfully.

When the price of the shared resource declared by the rational user is the same as the truthful cost, the pricing mechanism is said to be *strategy-proof*.

1.1 Motivation

Generally, computer scientists consider users voluntary and obedient in following the designed protocols [3]. Users that do not behave according to the system rules are considered faulty, and thus significant research effort is directed towards effective fault tolerance mechanisms [23, 24]. When allocating resources, the main concern of the system designers is computational complexity, which classifies the allocation problem according to its inherent difficulty. The allocation problem is considered difficult when finding the solution requires significant computational resources, such as computation time, memory, storage or communication. In this thesis, we consider that an allocation is not computationally efficient when the allocation algorithm runs in non-polynomial time. Thus, computer scientists focus mainly on *computational efficiency*, while trying to maximize resource utilization and achieve scalability [36]. For example, in solving the task assignment problem [131], solutions such as the Hungarian algorithm [67] assume that users declare their resource costs truthfully, and focus on finding the task-to-resource assignment that maximizes a global optimization function in polynomial time. However, the allocation of shared resources in a federated system must take into account that users, consumers and providers of resources, are rational in maximizing their own interest, and may declare false information to improve their own optimization function [82, 109].

In contrast to resource sharing systems used in research and academic communities, emerging platforms such as cloud computing have been put to commercial use. According to a 2011 Forrester report [98], cloud computing market will increase from $20 billion to more than $240 billion by 2020. Currently, in addition to voluntary and al-

truistic sharing, computer resources and services are traded over the Internet by rational users, with consumer payments made online using a credit card [1, 2]. In social systems, market-oriented economies have proved their advantages over alternative means to control and manage resource allocation [15]. Accordingly, some of the successful ideas of these economic models can be used for resource pricing in large, federated computer systems, and for the study of market-oriented resource allocation mechanisms [69]. In contrast to computer scientists, economists use mechanism design to look at the impact of competition on rational users, and to design allocation mechanisms that use incentives to maximize the *economic efficiency*, a global optimization function. Besides economic efficiency, other desired economic properties are strategy-proof, budget balance, and Pareto efficiency [29, 57, 120].

Having these two different approaches motivates us to examine the trade-offs involved and investigate how resource pricing and financial incentives can be used for the allocation of shared resources in the context of large federated systems with rational users in a computationally efficient way. To the best of our knowledge, this thesis is the first study of allocating consumer requests with multiple resource types per request to multiple providers considering *economic* properties, such as rational individuality, incentive compatibility, budget balance, Pareto efficiency, and *computational* measures, such as computation time and communication time. To achieve both the economic and computational efficiencies, the two key issues are:

1. **Economic Issue**

   According to the Myerson-Satterthwaite impossibility theorem [77], an economic system that incentivizes rational users cannot achieve the best economic efficiency while maintaining budget balance. In a budget-balanced system, the sum of all user payments is zero, i.e. payments made by the consumers are equal to the payments received by the providers. Thus, budget balance is a desired property in self-sustaining federated systems, with no external source of currency or a central authority to allo-

cate resources and collect payments.

2. **Computational Issue**

   Achieving the optimal allocation is not computationally efficient when allocating consumer requests with multiple resource types per request. Finding the providers and consumers that maximizes the global optimization function was shown to be an instance of the set packing problem [50], known to have an exponential complexity.

By considering both the computer science and the economic perspectives, we propose in this thesis a novel approach for pricing and allocating computer resources in federated systems with rational users. Our primary motivation originates in the existing trade-offs among strategy-proof and economic efficiency, on one side, and the economic efficiency and computational efficiency, on the other side. In the context of federated systems where rational users consume and provide resources, our approach is to trade-off Pareto efficiency for strategy-proof, budget balance and computational efficiency. In contrast, existing work in economic-based resource allocation focuses mainly on achieving Pareto efficiency at the expense of one economic property and computational efficiency. Losing the strategy-proof property results in users being untruthful, and exploiting the allocation mechanism to improve their self-interest [70, 71, 79]. Without budget balance, the allocation system needs a third party to supply the budget deficit or surplus, and thus is not commercially viable [27, 48]. Lastly, without computational efficiency, a practical implementation of the allocation algorithm is not feasible [88, 103]. In our work, we study the economic properties of the novel pricing scheme we propose using the mechanism design framework, and the computational properties using theoretical analysis and simulations. As proof of concept, we implemented the proposed scheme in the context of a federated cloud.

1.2 Objectives

Resource markets have been previously proposed in distributed systems such as clusters [28, 46, 108, 129] and grids [4, 40, 52, 110, 125, 126]. More recently, an active research topic in cloud computing is its economic aspect, including the problem of pricing cloud resources and services [14, 63, 100, 130]. An important challenge in current market-based approaches is the pricing and allocation of consumer requests with multiple resource types per request. For example, many of the existing cloud providers specify a fixed price for each resource type or service they offer, with consumers having to create separate requests for each resource type, and aggregate them manually after allocation. In this approach, another issue is that resources in a consumer request are allocated from a single resource provider. In this context, a crucial research question is:

> *In a federated computer system, where providers and consumers of resources are rational in maximizing their self-interest, how can pricing be used to incentivize user behavior when allocating a consumer requests with multiple resource types to multiple resource providers?*

The main objective of our work is to design a *strategy-proof* resource pricing mechanism for allocating consumer requests with multiple resource types per request to multiple resource providers. *Strategy-proof*, also known as "incentive compatibility dominant strategy", is a property achieved when rational users have no incentives to be untruthful, since their interest is maximized by the pricing scheme. One of the challenges in achieving strategy-proof is the trade-off with the other economic properties, namely budget balance and Pareto efficiency. To study strategy-proof, we use the theoretical mechanism design framework to formally define the valuation and the welfare, or "interest", of a rational user for the allocation result.

Simple pricing schemes that allow the allocation of requests for only one resource type do not deliver the highest valuations to consumers, since they need to aggregate different resource types manually, at the cost of welfare and economic efficiency [97, 121]. However, there are several challenges in allocating requests for multiple resource types, such as increased computational complexity and scalability [89]. We refer to vertical scalability, when the number of resource types in a request increases, and horizontal scalability, when the number of users in the system increases. The computational complexity is determined by the winner determination algorithm, which decides the providers and consumers that are allocated, and by the payment computation, which determines the total payments for the winners.

With cloud computing, the long-envisioned dream of computing as utility is achieved [20]. Many of the current standalone providers offer resources and services using pay-per-use fixed pricing [1, 2]. With an increasing number of cloud users, it is expected that more providers will offer similar services. Furthermore, with interoperability among providers, cloud consumers have a rational choice of the same service across clouds, to improve reliability and availability [16]. In this context, the aim of federated clouds[1], a topic of recent interest, is to integrate resources from different providers such that access is transparent for the users. Our ensuing goal is to design a federated cloud platform that increases reliability and elasticity by trading cloud resources and services across multiple clouds using a market-based pricing mechanism.

[1]The US National Institute of Standards and Technology (NIST) uses the designation *hybrid cloud* for the "composition of two or more distinct cloud infrastructures (private, community, or public) that remain unique entities, but are bound together by standardized or proprietary technology that enables data and application portability". In this thesis, we will be using the designation *federated cloud* for a similar system.

1.3 Contributions

Towards achieving our objectives, the key contributions of this thesis are:

1. **Strategy-proof Pricing for Market-based Resource Allocation**

   We use pricing and financial incentives to allocate requests from rational users. We express the problem of allocating shared resources as a mechanism design optimization problem, and formally prove the properties achieved by our market-based resource pricing mechanism. In addition to individual rationality and incentive compatibility, which account for strategy-proof, we also study other economic properties of the proposed mechanism that determine our trade-off, such as budget balance and Pareto efficiency.

2. **Dynamic Pricing for Multiple Resource Types per Request**

   Optimal allocation algorithms are often not feasible to implement due to the computational complexity, and resource providers use either fixed pricing, or create separate markets for different resource types. We show that consumer requests with multiple resource types per request can be allocated using a computational efficient scheme that prices resources from multiple providers dynamically, based on demand and supply, and study the benefits of dynamic pricing in large federated systems.

3. **A Scalable Distributed Auction Scheme**

   Centralized mechanisms have proved impractical in large systems [56]. We study how to efficiently divide resource information in large federated systems, and show, using experiments validated on PlanetLab, that our scheme achieves vertical scalability, i.e. when increasing the number of resource types in a consumer request, and horizontal scalability, i.e. when increasing the number of users in the system. Our approach leverages peer-to-peer overlay networks, where a distributed hash table is used to maintain resource information.

4. **A Platform for Managing Resources in Federated Clouds**

   In cloud computing, globally distributed resources can be shared and traded over the Internet, as a service, without the users having any knowledge of the underlying architecture. We propose a platform where providers can add their public or private clouds to a federated, or hybrid, cloud. By leveraging the proposed pricing mechanism and open-source software, our federated cloud prototype is able to allocate consumer requests with multiple resource types on private clouds, based on the Eucalyptus and OpenNebula middlewares, and public clouds, such as Amazon EC2.

1.4 Thesis Organization

The structure of the thesis is the following:

Chapter 2. Related Work

We present current market-based resource allocation models. We classify the key challenges in allocating resources using pricing, and evaluate the current work, focusing on the pricing mechanisms they employ. Our findings are that most of the existing solutions trade incentive compatibility or are not feasible to implement.

Chapter 3. Strategy-proof Resource Pricing

We present an overview of the process of pricing and allocation of shared resources. We divide allocation into three steps, and focus in this chapter on the pricing mechanism, which is responsible for determining the winners of the allocation, and for computing the user payments. We identify the major design considerations and trade-offs. Our approach is to aim for strategy-proof and budget balance, which allows us to use a computationally efficient pricing algorithm. We introduce our proposed mechanism and payment functions in the context of

the mechanism design framework, and formally prove individual rationality, incentive compatibility, and budget balance. Using theoretical analysis, we study the proposed pricing algorithm and evaluate its computational complexity and limitations.

Chapter 4. Distributed Resource Pricing

Scalability becomes an issue when increasing the number of resource types in a consumer request or the number of providers. Moreover, most of the pricing schemes are centralized, with a single market-maker or auctioneer to determine the allocation results. To address these issues, we propose the use of distributed auctions. We look at the challenges of current distributed auctions schemes, and propose a novel architecture that leverages the distributed hash table in a peer-to-peer overlay to maintain resource information and perform pricing by resource type. The role of the market-maker is divided between a resource broker and a request broker, which are selected, for each allocation, from all the existing users in the system. We present a distributed synchronization protocol, required for maintaining the economic properties of our pricing scheme while avoiding concurrency issues. We perform a theoretical analysis of our distributed algorithm to study its scalability and identify its limitations.

Chapter 5. Federated Cloud Prototype

In this chapter, we present the implementation of our proposed pricing and allocation mechanism in the context of federated clouds. We prototype a platform for managing multiple public and private clouds, where interoperability between providers is achieved using web services and virtual private networks. Our federated cloud prototype is designed to manage infrastructure services and to support the development of cloud platforms that enable the use of services on top of the infrastructure provided by our platform.

Chapter 6. Experimental Evaluation

Our scheme sacrifices economic efficiency for strategy-proof, budget balance and computational efficiency. In this chapter, we evaluate the economic efficiency loss using simulations, and compare our proposed scheme with other pricing mechanisms in different market scenarios. We take into consideration the economic and computational efficiencies by measuring total welfare, individual user welfare, successful consumer requests, allocated provider resources, and average allocation time. Scalability is studied by varying the number of resource types in a consumer request, and the total number of users in the system. We use both a centralized and distributed implementation of our scheme in our evaluation. Lastly, we look at the rationality of existing cloud computing users by estimating the consumer welfare obtained in a study of spot price traces from Amazon EC2.

Chapter 7. Conclusions

We present a summary of the thesis and the key contributions of our work. We discuss possible directions for future work, including the extension of our scheme to provide incentives for brokers, and the evaluation of economic efficiency when the strategy-proof constraint is relaxed.

Chapter 2

Related Work

The idea of using economic-based models in the allocation of shared resources is not new [97, 121]. Although many authors have studied the use of resource markets for sharing computer resources [45, 61, 68, 84], there are many challenges that prevent practical developments. One of these challenges is that users participating in resource allocation are rational. A *rational user* maximizes his interest, or welfare, given his current knowledge of the system. In contrast to the user model generally considered in computer science, where the user is either obedient or faulty, economic-based models take into account user rationality in the design of the resource allocation mechanism. However, there are several issues to consider in economic market-based resource allocation models, classified as microeconomic and macroeconomic [61].

- *Microeconomics* analyzes the individual participants in resource allocation, i.e. providers and consumers, and their interactions in the resource market. This includes, among others, identifying the rational participants, the study of their behavior, the alternatives that maximize their interest, and the constraints imposed by other participants. For example, some of the factors that affect the behavior of a participant, such as resource prices, are influenced by the behavior of other participants [84]. Rational users constitute a major issue in the development of distributed systems and are cur-

rently an important research topic [36, 82, 109]. The microeconomic aspects in the allocation of shared resources are an important part in the scope of this thesis.

- *Macroeconomics* studies the economic model as a whole, including aggregate resource demand and supply, collusions between participants, and the currency system. Usually, macroeconomic studies are used for the evaluation and development of large-scale economic policies and strategies, with less scope in computer science. For example, the lack of widely-deployed banking and secure currency transfer technologies have led to the adoption in computer systems of sharing models such as voluntary participation, network of favors, and tit-for-tat, which do not employ pricing and consequently do not allow users to request for multiple resource types [9, 25, 122]. Accordingly, users have to aggregate different resource types manually, resulting in lower economic efficiency. In contrast, in this thesis we focus on resource pricing and the use of a common currency that allows user requests with multiple resource types per request.

In this chapter, we introduce important concepts in market-based resource allocation, and present several pricing models, namely fixed pricing, equilibrium price, bid-based proportional share, bargaining, and auctions. We focus on several key economic issues in allocating shared resources when users are rational: *strategy-proof*, *budget balance*, *multiple resource types per request*, and *Pareto efficiency*. Since finding a Pareto efficient allocation is a NP-complete problem [50], we also consider the *computational complexity* of the pricing algorithm, in addition to the above.

2.1 Market-based Resource Allocation

Definition 1 (Resource Market)**.** *A **resource market** refers to the environment, expressed in terms of rules and mechanisms, where resources within an economy are exchanged.*

Different resource markets employ distinct mechanisms for trading or exchanging resources. In Figure 2.1, we classify existing economic models used in the allocation of shared resources by the exchange method used, into two broad categories: *bartering* and *pricing*.

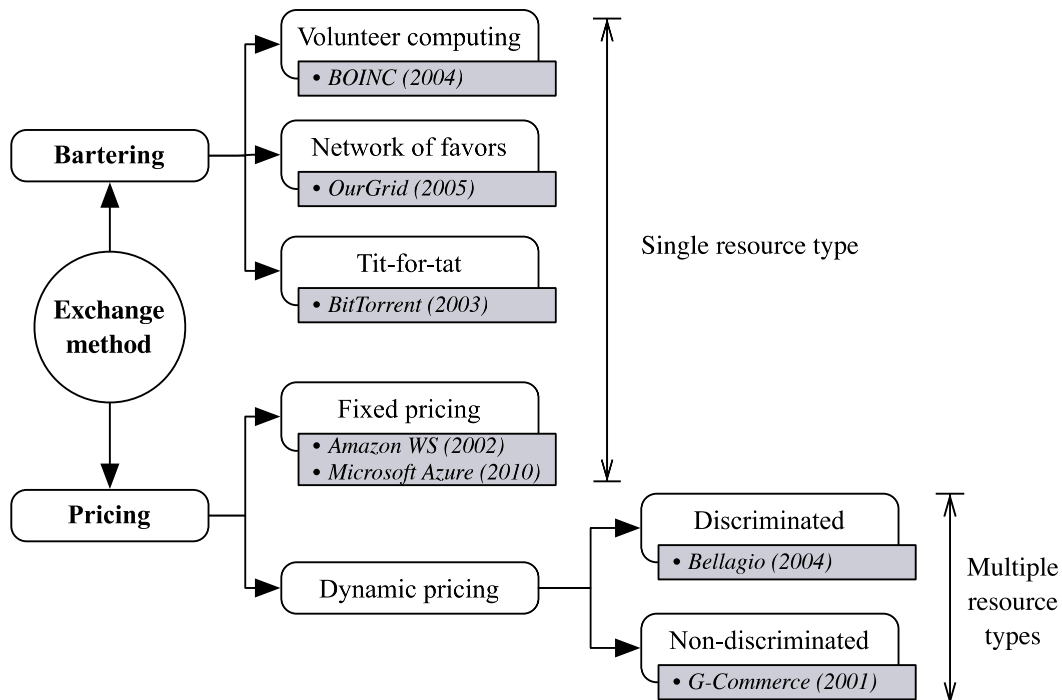


Figure 2.1: Market-based Resource Allocation Models

Volunteer computing [8], network of favors [9], and tit-for-tat [32] are examples of economic models used for allocation of shared resources that do not use pricing. In bartering, resources and services are traded directly, based on the exchange value, without a monetary system. Users in *volunteer computing* contribute their resources, such as CPU cycles or disk storage, by running a software client that polls a server for new jobs or for reporting the results of completed jobs. To deal with incorrect results due to malfunctions or to rational users, the server performs each job using at least two users, and accepts results only if the results are consistent. In the *network of favors* model, each user keeps track of other users that provide resources for their jobs, and rewards them

by prioritizing their requests when their own resources are idle. In contrast, when using the *tit-for-tat* strategy, users that behave selfishly and do not cooperate in sharing are penalized by the other participants. Since bartering does not employ a common form of currency, these models support the trading of only one resource type. For example, BitTorrent [32] exchanges blocks from the same file, and OurGrid [9] is used for CPU cycles. To trade different types of resources, allocation mechanisms need to use pricing and a common type of currency in which to express the value of each resource type.

Considering a common form of currency, the *price* of a resource is the monetary expression of the exchange value for that resource. Based on the concept of price, we define *pricing* as follows:

Definition 2 (Pricing)**.** ***Pricing*** *represents the process of computing the economic exchange value of resources relative to a common currency.*

Economic models for the allocation of shared resources may use fixed or dynamic pricing. When using *fixed pricing* [5, 6, 115], the currency payment for each resource type is the predefined fixed price, set by the provider, for all allocations. In contrast, when using *dynamic pricing*, the payment for resources of the same type is computed for each allocation according to the demand and supply, by the pricing mechanism used.

Definition 3 (Pricing Mechanism)**.** *In a resource market, the process of matching market participants, e.g. providers and consumers, to engage in an exchange using pricing is called* ***pricing mechanism****.*

Specifically, a resource allocation system uses a *discriminated pricing mechanism* when resources from different providers have a different price in an allocation and a *non-discriminated pricing mechanism* when resources with the same type have the same price for all the providers that participate in the allocation.

Recent work in distributed systems found that users sharing computing resources are self-interested parties, with their own goals and objectives [109, 82, 126]. Usually,

these parties can exercise their partial or complete autonomy to achieve their objectives and to maximize their interest. They can devise strategies and manipulate the system to their advantage, even if by doing so they break the rules of the system. Accordingly, achieving *strategy-proof* has become a major issue in the context of distributed systems where resources are shared and traded over the Internet [36, 72]. To deal with rational users, strategy-proof models employ incentives to motivate users to behave in certain ways. For example, volunteer computing uses *moral incentives* to engage new users into sharing, while the network of favors and tit-for-tat models use *coercive incentives* to make certain users behave according to the protocol. Having a common currency to measure the welfare of market participants enables pricing mechanisms to use *financial incentives*. When using financial incentives, user payments are computed considering both the published prices, and the financial incentives given to the respective market participants. However, achieving strategy-proof while maintaining consumer payments equal to provider payments results in losing Pareto efficiency, and thus also the maximum economic efficiency [50]. Accordingly, *Pareto efficiency* is a key challenge for economic models that allocate shared computing resources.

2.2 Issues in Resource Pricing

Allocating resources using pricing poses several challenges, which we have grouped as *pricing issues* and *allocation issues*. This section introduces briefly these issues, while the next section details the issues in the context of existing pricing models proposed for the allocation of shared computing resources. The pricing issues we consider are :

1. *Resource description* refers to the public information used when publishing provider resources and consumer requests. This may include only resource information and price, or may contain additional constraints, such as in combinatorial auctions, where a bidding language is used to express resource bundles and boolean constrains be-

tween bundle resources [34, 83]. Resource description can simplify or complicate the pricing process.

2. *Resource location* refers to the method in which provider resources and consumer requests are published in the resource market [62]. This can be either a simple message to a centralized entity, containing the resource description, or more complex, when having a distributed resource location system [22, 58, 59]. Resource location establishes the list of participants in the market before pricing can take place.

3. *Winner determination* selects the users that take part in an allocation. The winner determination algorithm influence many other aspects as resource allocation, such as the economic properties and the computational efficiency [34, 103].

4. *Payment computation* determines the payments for the users selected by the winner determination. To achieve strategy proof for the pricing mechanism, the payment computation method needs to take into account financial incentives for rational users [39, 84].

The main allocation issues are:

1. *Multiple resource types per request* refers to consumer requests for more than one resource type, which are more difficult to allocate. Many of the existing pricing schemes are designed to allocate requests for only one resource type, with the users having to aggregate resources manually [26, 68].

2. *Strategy-proof* is achieved when rational users are incentivized to declare accurate information about resources when they publish or make requests. In contrast, with pricing schemes that are not incentive compatible and thus not strategy-proof, untruthful users stand to gain more by declaring false information [36, 37].

3. *Pareto efficiency* is achieved by maximizing the economic efficiency, which is the sum of both providers and consumers welfare. Usually, achieving Pareto efficiency can be done only by using an algorithm with exponential complexity [68, 103].

4. *Computational efficiency* takes into consideration the computation time and communication cost of the pricing and allocation algorithms, including the winner determination and payment computation algorithms [45, 84].

2.3 Resource Pricing Models

Pricing models were proposed to allocate shared resources in distributed systems such as clusters, grids, and clouds [7, 18, 46, 52, 74, 108]. In Figure 2.2, we classify existing work, based on the pricing mechanism used, into five categories: *fixed price*, *equilibrium price*, *proportional share*, *bargaining*, and *auctions*. In the following, we present more details about each pricing scheme, and discuss the trade-offs in each approach with respect to the key challenges identified in the previous section.

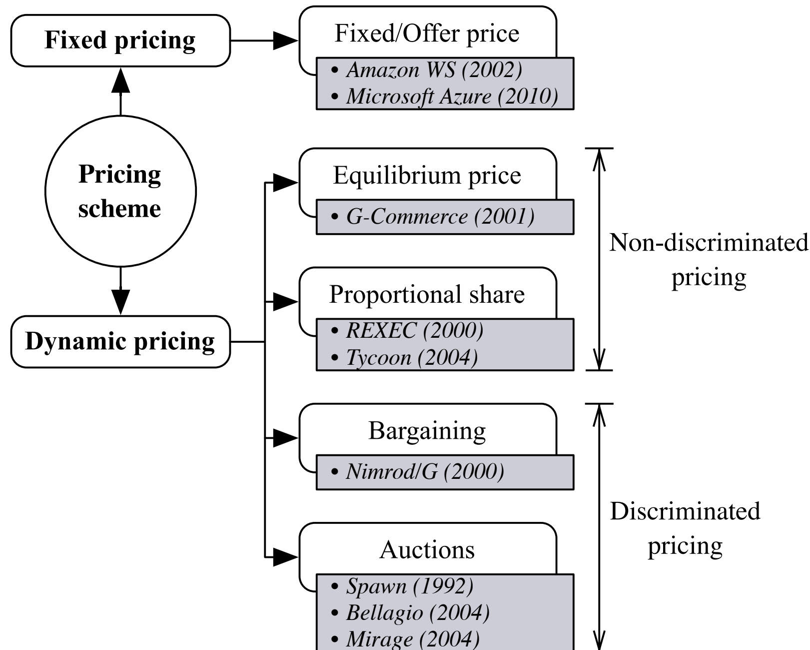


Figure 2.2: Pricing Models for Resource Allocation

2.3.1 Fixed Pricing

In *fixed pricing*, resource payments are directly determined by the offer price. The *offer price* is set by the resource provider and represents the amount of currency that the provider is willing to accept for the allocation to take place. Resources shared by the same provider are traded for the same price. This pricing mechanism is currently used by providers in cloud computing, where resources are provided over the Internet on-demand, as a service, with the underlying infrastructure transparent to the user [10]. At present, each resource provider maintains its own public cloud, with no interoperability among providers. From an economic point of view, each provider creates a closed market, e.g. the Amazon EC2 market [5], where the provider has monopoly on resources. This allows providers to set a fixed price on resources, and the consumers, i.e. the public cloud users, pay the offer price for resources. However, an issue in the closed market model is that only the monopolistic provider can add new resource types in the system, while adjusting the offer price based on its own preferences. Accordingly, early cloud services, such as Sun Grid Compute Utility [115], are restricted to trading only one resource type, such as CPU time.

While simple to implement, fixed pricing is not economic efficient in a system with multiple providers, or where users are both providers and consumers of resources [74]. With a large number of providers (sellers) and users (buyers), fixed pricing cannot adapt to the changes in demand and supply. To address this issue, *federated (hybrid) clouds*, a topic of recent interest, aim to integrate cloud resources from different providers, to increase elasticity and reliability [10, 100]. Federated clouds use *dynamic pricing* to set resource payments according to the forces of demand and supply. In addition, dynamic pricing schemes can also facilitate consumer to request for multiple resource types [115, 74].

The *Sun Grid Compute Utility*, introduced by Sun Microsystems [115] (acquired by Oracle), was one of the first utility computing services that provided cloud users access to computing resources, for the fixed price of US$ 1 per CPU-hour. It used open source technologies such as Solaris, Sun Grid Engine, and Java. Applications running in the SUN Grid were self-contained, compatible with the Solaris OS, and had no interactive access. Cloud users could run batch jobs, with the application and data size limited to 10 GB. In addition, there was available a list of popular applications that could be installed, where cloud users could also publish their own software [115]. After Amazon started to offer a similar service for a smaller fixed price, the Grid Compute Utility was discontinued, mainly due to the decreasing demand. As discussed previously, one of the issues with fixed pricing is that it cannot adapt to changes in demand and supply. To address this issue, Oracle currently promotes an open cloud initiative [21], with interoperability between private and public clouds, where a dynamic mechanism can be used to price resources.

Amazon Elastic Compute Cloud (EC2) [5] offers a web-based service for cloud users to create virtual machines, on which any application can be executed on various operating systems, including Linux, OpenSolaris, and Windows Server 2003. It uses XEN virtualization [128] to create different types of VM *instances*, which approximate different hardware configurations. For example, a *large instance* is the equivalent of "a system with 7.5 GB of memory, 4 EC2 Compute Units (2 virtual cores with 2 EC2 Compute Units each), 850 GB of instance storage, 64-bit platform". Cloud users pay a *fixed* hourly price for each virtual machine, based on the type of instance. By reserving instances for a large period of time, the hourly fixed price can be decreased.

Amazon S3 is an online storage facility that provides cloud users with seemingly unlimited storage in the form of a web-service. Cloud users may employ the S3 service for web hosting, image hosting, and as a backup system. Objects stored by cloud users can be retrieved using either the web-services API, as HTTP downloads, or the

BitTorrent protocol [32]. To use S3, cloud users pay a fixed price for each GB of data, plus additional costs for bandwidth usage. Currently, Amazon expanded its offer to ten different EC2 virtual machine instance configurations with different prices, and practices tiered pricing for both storage and bandwidth. We see this as the first step towards dynamic pricing, where users can request for custom configurations with multiple resource types. In addition to fixed pricing, Amazon started in December 2009 to use spot pricing, where cloud users may bid for resources, and allocation is valid while the cloud user bid exceeds an hourly computed "spot price". If the cloud user maximum price becomes smaller than the hourly spot price, the cloud user instances are terminated. While this pricing scheme is good for applications that have flexible start times and no constraints on application termination, it does not offer the cloud user any guarantee on the duration of the service. Thus, applications that require high availability are not suitable for this pricing model. As a standalone provider, Amazon cannot provide cloud users the advantages of a marketplace where multiple providers compete for similar services, including higher availability, elasticity and reliability [74].

The *Azure Services Platform* [2] is a platform to build and host Windows applications in the Microsoft cloud. It provides three services to users: the operating system for scalable computing and storage, a cloud-based database server, and a middleware that allows running applications in the cloud. Using Hyper-V virtualization [64], Microsoft offers several *instance* types that approximate different hardware configurations, similar to Amazon EC2, for a *fixed* hourly price. For database services, storage, and bandwidth, Microsoft uses tiered pricing.

2.3.2 Equilibrium Price

Equilibrium price is used in market clearing price mechanisms, where resources with the same type form a pool of equivalent choices and consumers have no advanced knowledge about which resource provider is used for allocation. For each resource type,

the payment is determined according to the respective supply and demand curves, thus allocated resources of the same type have the same price, independent of the provider. Market clearance mechanisms generally use a third party, called *market-maker*, to set the price for each resource type. This payment scheme is similar to the ideal *supply and demand equilibrium* [49], the concept behind the competitive free market where the price is not influenced by either providers or consumers.

In order to reach equilibrium, both providers and consumers are queried for their willingness to sell or to buy at different price points. However, this results in high communication overhead. Furthermore, determining the supply and demand curves at a specific moment in time is a computationally hard problem. Generally, strategy-proof is not an issue with this mechanism because the price is set by the market-maker according to the supply and demand. However, using this mechanism does not maximize user utility. Instead, the goal of these mechanisms is to clear the market of resources, i.e. maximize resource utilization. Thus, systems such as G-commerce [126] use different models for each resource type, and users use specific individual functions to approximate supply and demand.

G-Commerce [124, 125, 126] defines a set of economic policies and mechanisms for controlling resource allocation in Grid systems. To model the relations between supply, demand, and value, G-commerce is based on the following assumptions:

1. The relative worth of a resource must be determined by supply and demand.
2. The price of a given resource is its worth relative to the value of a unit resource called *currency*.
3. Relative worth is accurately measured only when market equilibrium is reached.

There are two possible types of providers in G-Commerce: CPU providers, which sell CPU "slots" (based on CPU speed), and disk providers, which sell disk space. Each provider type uses a different function to determine supply.

For example, a CPU provider computes:

$$mean_CPU_price = \frac{revenue}{now} \times \frac{1}{slots}$$

where $mean_CPU_price$ is the average price per time unit per slot, *revenue* is the total amount of currency required by the provider, *now* is the current clock time, and *slots* is the number of slots that the provider supports. Similarly, a disk provider calculates:

$$mean_DISK_price = \frac{revenue}{now} \times \frac{1}{capacity}$$

where *capacity* represents the total disk space that the provider sells. To determine the demand at a given price point, each consumer first computes the average rate at which it would have spent its currency with the current price. Next, it computes the maximum amount of currency that it is able to spend until the next budget refresh.

A comparative study [126] between G-Commerce and auctions shows the superiority of the former over a second-price sealed-bid auction implementation. However, G-Commerce has a high computational overhead for each provider and consumer, and a different model for each provider, based on the type of resources it sells. More importantly, finding the equilibrium price is NP-complete, and approximation algorithms need to be used, leading to the loss of incentive compatibility [125, 129].

2.3.3 Proportional Share

Used more for process scheduling, Proportional Share [113] gives each user a weighted share of the available resources in a multi-user system. Proportional Share (PS) maximizes utilization, as it always provides resources to needy processes. However, PS does not provide incentives for users to reveal their truthful values. As a result, rational users can weight their processes unfairly, resulting in a loaded system, with low value pro-

cesses consuming more resources than high value processes that are not making useful progress [71]. Systems that use PS for pricing, such as Tycoon [70], use a mix between auctions and a proportional share-based function to compute allocations, at the cost of high communication overhead and high user interaction.

The *Millennium Allocator* project [28] is a system implemented at the University of California at Berkeley (UCB) Millennium Cluster, where each node shares four basic computational resources: CPU time, physical memory, I/O bandwidth, and network bandwidth. The cluster functions as a computational economy that optimizes user utility. Each cluster node acts as an independent provider, while the consumers are computer applications, purchasing resources based on the utility delivered to the users. The architecture of the system comprises of five layers: resources, resource managers, economic front-end, access modules, and user applications. Users assign values to their applications and execute them on the cluster using the appropriate access module. The economic front-end translates values, expressed by the users in a common currency, into the appropriate resource shares. For each resource type, the appropriate resource manager enforces the allocation of resources. The Millennium Allocator has three functional requirements: means for users to express utility (currency), policies to translate utility into resource allocation, and mechanisms to enforce allocations. In order for the cluster users to obtain the resources they paid for, the Millennium Allocator uses a modified OS kernel to enforce allocations. The policy used by the economic front-end is based on the opportunity-cost. Thus, user payments are based on the value of the opportunity to use a resource, which is being denied to other competing users. Under high contention, users pay more to use the system. When multiple jobs are competing for a resource r, proportional share takes place and each user i receives a weighted share of:

$$share = r \times \frac{b_i}{\sum_{j=1}^{n} b_j}$$

The size of the share and the price are computed dynamically as jobs enter and leave the system.

Having a modified kernel in order to enforce allocation is only possible in a cluster pertaining to one administrative authority. Accordingly, this approach is not practical in a distributed system that spans across multiple authorities, where providers cannot be forced into using specific OS kernels. Furthermore, rational users are able to modify open source kernels in order to "free-ride" [28]. Another disadvantage of the Millennium economic front-end policy is that cluster users are not aware of the amount of currency they are charged for running a job. Lastly, providers are not able to share other resource types or limit resource usage.

Tycoon [70] is a market-based distributed resource allocation system that allows consumers to differentiate the utility of their jobs using bid-based Proportional Share. Furthermore, it does not impose the manual bidding overhead on consumers. The main components of Tycoon are: the service location service (SLS), bank, auctioneer, and agent. The *auctioneers* register the availability of resources to the SLS. The location service is also queried by the *client agents* in order to find resources. A client selected by an auctioneer requests the *bank* for a transfer of currency from his account to the auctioneer's account and receives a transfer receipt. The receipt is then verified by the auctioneer and the client agent is able to execute the application. One obvious drawback of this design is the service location service and the bank, which are centralized entities and may become a bottleneck and a central point of failure. The auctioneer collects bids from consumers, allocates resources and accounts for resource usage. Allocation is done using a Proportional Share-based function that considers a bidding interval specified by the consumer. The *bidding interval* defines the number of available seconds for spending the budget. Thus, the PS formula employed by Tycoon is:

$$share = r \times \frac{\frac{b_i}{t_i}}{\sum_{j=0}^{n-1} \frac{b_j}{t_j}}$$

To capture the consumer preference, the agent uses a linear utility function, where the consumer specifies a positive weight w_i for each machine. If the consumer is allocated a fraction r_i from machine i, then the total utility is computed as:

$$U = \sum_{i=1}^{n} w_i r_i$$

The agent's goal is to maximize this utility under a given budget.

The design of Tycoon raises several issues. Firstly, the system is not incentive-compatible, as auctioneers are able to exploit the system by charging a higher price for resources. Furthermore, the consumer preference is expressed manually for each machine in the system. Lastly, the communication cost in Tycoon is high, as all the auctioneers constantly update the SLS, and the user agents query the SLS for information about auctioneers.

2.3.4 Bargaining

The bargaining or negotiation model does not rely on a third party when mediating an allocation, such as a *market-maker* or an auctioneer. The provider attempts to maximize the price, while the consumer strives to minimize it. Overall utility is maximized, as the consumer selects the provider with the lowest price, while a provider selects the consumer with the highest payment, from multiple consumer offers.

One of the issues in bargaining is market dynamics, as resources are constantly being added to or removed from the market by providers, while consumers enter or withdraw requests. Accordingly, there are difficulties in implementing a bargaining mechanism that adopts a negotiation strategy that optimizes the user utility while considering market dynamics [110]. Another issue with bargaining is the high communication cost, as each consumer must contact all providers and manage several rounds of negotiation in the absence of the market-maker or auctioneer.

Nimrod/G [17] is a system that uses the Globus middleware [47] for dynamic resource discovery and for dispatching jobs in a computational grid. In Nimrod/G, a parametric engine acts as a job-control agent, which is responsible for the creation of jobs, maintenance, user interaction, scheduling, and dispatching. Various parameters can be used by the engine to reach an optimal scheduling policy, such as user preferences, reliability of resources, user priorities, resource states, and capabilities or application requirements. Nimrod/G works on the consumer's behalf and attempts to complete a job within a given deadline and cost. The deadline represents the time when the consumer requires the job result.

The parametric engine adapts the list of provider machines running a job in order to find sufficient resources for meeting the deadline. However, in doing so, the price changes with the arrival of competing consumers. One of the drawbacks of Nimrod/G is the lack of a contract between the provider and the consumer, where the consumer negotiates the price of resources and discovers if a job can be performed within a given budget. Moreover, Nimrod/G is not suitable in a distributed setting since users from different organizations are not able to add resources to the global pool.

2.3.5 Auctions

In auctions, a third party called the *auctioneer* takes bids from one participant (usually the consumer) and selects the winning bidder for the allocation [66].

Definition 4 (Auction)**.** *An* ***auction*** *is the exchange process where a market participant, e.g. consumer or provider, submits a bid, and the most advantageous bid is selected by another participant for trade.*

The minimum price a provider is willing to accept and the maximum price a consumer is willing to offer in an auction are known as *reserved prices*. There are many types of auctions, with different rules for determining the winning bidders and allocation prices.

Auction variations include time limits, price limits, knowledge of the identity or actions of other participants, etc. A taxonomy of auction types is presented in Figure 2.3.

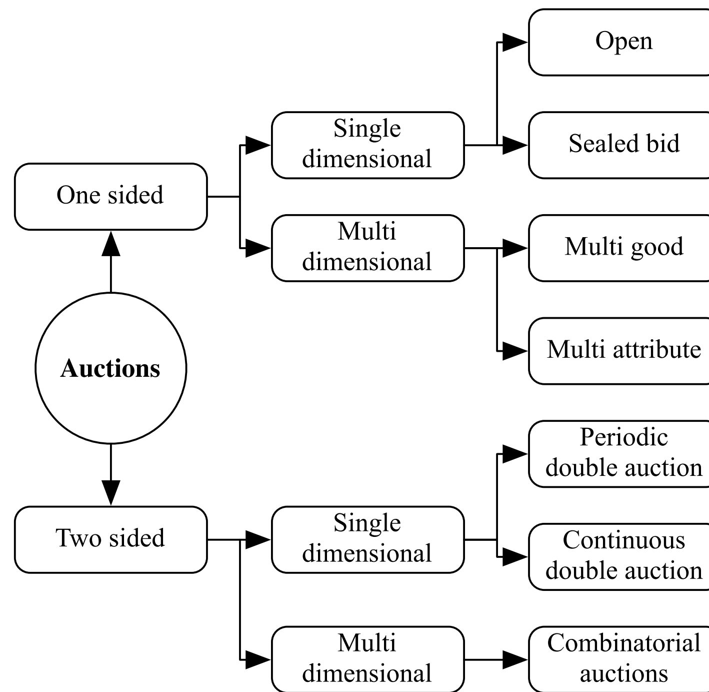


Figure 2.3: A Taxonomy of Auction Types

In *one-sided* auctions, providers offer one resource type for sale, and consumers submit their bids to the auctioneer. The auctioneer then selects the winning consumer and computes the payment according to some mechanism. If the bids are public, the auction is *open*, otherwise it is *sealed bid*. In contrast, *two-sided* auctions allow the auctioneer to take bids from both providers and consumers simultaneously, and compute the market-clearing price. Also called double auctions, they may take place *periodically*, at certain intervals of time, or *continuously*, while there are still resources and requests in the market. The winners of a double auction are the consumers with prices higher than, and the providers with prices lower than the clearing price, respectively. However, most one-sided auctions and double auctions are *single-dimensional*, which means only one resource type is allocated in an auction. A *multi-dimensional* type of

auction where different types of resources can be sold as part of a bundle, i.e. multi-good, is *combinatorial auctions* [34]. A user can express his preference for resources inside a bundle using boolean operators, within the rules of a bidding language. Combinatorial auctions are double-sided, where the auctioneer collects the bids for different resource bundles, and selects the winners. Due to the complexity introduced by auctioning multiple resource types, computing an efficient outcome for a combinatorial auction is a NP-complete problem [79, 103]. Thus, many resource allocation systems prefer to use single-dimensional auctions such that allocations can be computed in polynomial time.

The English auction is one of the most common auction types, where the provider starts with the reserved price and the consumers submit ascending bids until no consumer is willing to bid further. The consumer with the highest bid wins and pays the highest price. This auction has many disadvantages, such as the *winner's curse* [66], maximized revenues for providers, and the need for all participants to be in communication during the auction [49]. Some of these disadvantages are overcome by the second-price sealed-bid (Vickrey) auction [73], where consumers submit their bids to the auctioneer without knowledge of the other bids. The consumer with the highest bid wins, and the payment is the second-highest price offered by a consumer, which has been shown to represent the expected utility of the resource. The Vickrey auction is strategy-proof, as participants are incentivized to bid truthfully [73, 120].

From an algorithmic point of view, single-dimensional auctions are the easiest price-setting mechanism to implement. Early systems such as Spawn [121] and Popcorn [97] use the Vickrey auction to allocate computational resources. However, single-dimensional auctions are not suitable when a user needs multiple resources of different types, e.g. in a computational grid an application needs CPU, memory and bandwidth. In this case, the consumer must participate in multiple auctions, and it has to win all of them. Otherwise, delays are introduced and the allocation is not feasible.

Spawn [121] is a market-based computational system for utilizing the idle CPU time from a computer network. It uses sealed-bid, second-price (Vickrey) auctions where the consumers are users that wish to purchase computational time, while the providers represent users that offer for sale unused, otherwise wasted, processing time of their systems. Each auction is associated with a *resource manager*, responsible for initiating and monitoring the application that uses the purchased CPU time. The resource manager maintains a list of *neighboring auctions* which supply price and availability information. Each application has a *manager* module and a *worker* module. The manager module is responsible for coordination, message-passing, and the creation of application sub-tasks. Process managers fund their child processes, dynamically controlling the fraction of funds allocated to each sub-task. The *root* manager controls the total amount of funding in the hierarchy. At the highest level, the allocation of funding is negotiated by human system administrators.

Reservations in Spawn are made using fixed-time CPU slices. This leads to poor utilization when applications do not use their entire slot. Furthermore, a consumer reservation for a resource prevents other consumers from acquiring it, even if they are willing to pay more for it. Using a single-dimensional auction, Spawn is able to make reservations for only one resource type. However, this does not provide flexibility in a large system where users share many resource types. Another drawback of Spawn is that human negotiation is not practical in a large distributed system.

Combinatorial Auctions

In combinatorial auctions, multiple goods can be auctioned simultaneously as part of a *bundle*. Furthermore, a consumer may express a preference for goods inside a bundle using boolean operators, within the rules of the *bidding language* used. The importance of combinatorial auctions is best reflected by real-world applications, such as allocation of airport landing slots [92], auctions for bandwidth spectrum licenses [60], or procuring of transporting services [107].

After the providers and consumers express their valuations using the bidding language, the auctioneer collects the bids and computes an allocation. This step is called the *winner determination problem*, as the auctioneer selects the auction winners such that the allocation maximizes the auction social choice function. Unlike single-dimensional auctions, where only one resource is auctioned at the time with one consumer winning the auction, a combinatorial auction may result in more than one winner. In the last step, the auctioneer computes the payments for the auction winners, providers and consumers. In many cases, combinatorial auctions may use a Vickrey-Clarke-Groves [73] pricing mechanism for winner payments, such that the auction is strategy-proof [83].

Despite many useful properties, computing an allocation in a combinatorial auction requires a NP-complete algorithm. This is because the winner determination problem is an instance of the *set packing problem* [50], which determines if k subsets from a list of subsets of set S are disjoint. Furthermore, because there is an exponential number of possible subset of goods (bundles), the bidding language in combinatorial auctions must be very concise and expressive. Lastly, as shown by Myerson and Satterthwaite [77], an efficient VCG payment scheme that uses combinatorial auctions is not budget balanced, as the provider and consumer incentives are not equal.

Combinatorial auctions are used in *Bellagio* [12], which provides an interface for consumers to perform resource discovery queries and construct bids. User authentication and the account balance with the user's virtual currency are implemented using a PostgresSQL database. Resource allocation is performed using SHARE [46], a market-clearance combinatorial exchange system. The first step in the bidding process involves resource discovery, which is followed by translating the discovered resources (candidate nodes) into a *resource set*. The biding language allows: multi-unit allocations (nR), complementary resources ($R1 \wedge R2$), substitutes ($R1 \vee R2$), range constraints and set of sets allocations. The resource exchange component in Bellagio functions as follows:

i) periodically receive new offers and bids from providers and consumers, respectively; ii) perform resource discovery and construct node-sets in order to validate bids; iii) clear the exchange (winner determination problem); iv) determine payments; and v) report allocations and payments.

The winner determination problem is addressed in SHARE using a greedy approximation, which is fast but not optimal. Another drawback is that the pricing mechanism is based on the *Threshold scheme* [88], a variation of the VCG mechanism that allows budget-balance and low computational complexity at the cost of incentive compatibility. Losing incentive compatibility leads to performance loss in the presence of rational users. Lastly, Bellagio uses a fixed budget scheme for the consumer, which does not allow consumers to share their resources in order to gain additional currency.

Mirage [27] is a sensor network resource allocation system for sharing a sensor network testbed among users. Each testbed user is allowed to run an application after participating in a first-price repeated combinatorial auction. In each round of auctions, there are multiple consumers (the testbed users), and a single provider (the testbed), which considers all the bids submitted before the auction round started. The submission of bids takes place in a two-phase process: resource discovery finds candidate nodes in the sensor network, and the actual submission of bids takes place on the nodes selected during resource discovery.

Mirage relies on a central *bank* to store each project's account. Being a closed economic system, the testbed users have no means for earning currency, which is distributed by the system as priorities. Priorities are assigned for each project individually by a system administrator. The winner determination problem is resolved, like Bellagio, using a greedy approximation. As the algorithm is not strategy-proof, testbed users reported several strategic exploits [12].

2.4 Summary

In this chapter, we have analyzed current approaches that use pricing to allocate shared resources in distributed systems. In concordance with the Myerson-Satterthwaite impossibility theorem [77], none of the above systems manages to achieve strategy-proof, budget balance, and maximum economic efficiency, when allocating consumer requests with multiple resource types per request in a dynamic system, where users can both provide and consume resources. Most solutions trade incentive compatibility, assuming users are obedient or faulty, and thus do not consider rational users that may increase their welfare by exploiting the system, at the cost of performance degradation.

Table 2.1 shows a comparison of the pricing schemes presented in this chapter, considering the *economic* properties, i.e. strategy-proof, budget balance, and maximum economic efficiency; and *computational efficiency*, which takes into account the complexity of winner determination, payment computation, and communication cost.

Property	**Fixed Pricing**	**Equilibrium Price**	**Proportional Share**	**Bargain**	**One-sided Auctions**	**Combinatorial Auctions**
Strategy-proof	×	✓	×	×	✓	✓
Budget Balance	✓	✓	✓	✓	✓	×
Economic Efficiency	×	×	×	×	×	✓
Multiple Resource Types per Request	×	✓	✓	✓	×	✓
Computational Eff.	✓	×	✓	×	✓	×

Table 2.1: Summary of Related Works

Fixed pricing is currently used by many cloud computing providers because it is simple to implement, computationally efficient, and does not require strategy-proof due to the lack of interoperability between providers. However, fixed pricing is not suitable in a federated system, with multiple providers. *Equilibrium price* is the ideal model

that achieves incentive compatibility, because prices are set by the forces of supply and demand; budget balance, because seller payments equal buyer payments; and is also able to allocate consumer requests with multiple resource types per request. However, computing the equilibrium price is not feasible since the algorithm is NP-complete and thus the allocation time is high. Moreover, during the computation of the equilibrium price, the curves of supply and demand may change. *Proportional share* has low algorithm complexity and low communication costs, however it is not incentive compatible and does not achieve Pareto efficiency. Similarly, *bargaining* is not incentive compatible, and has high communication costs. One-sided *auctions* cannot allocate multiple resource types, while *combinatorial auctions* require a NP-complete algorithm, and thus it is not feasible to implement. Moreover, combinatorial auctions are usually not budget-balanced, and require the auctioneer to manage the budget deficit or surplus.

Chapter 3

Strategy-proof Resource Pricing

In this chapter, we introduce a novel pricing mechanism that can allocate consumer requests with multiple resource types per request in a resource market where providers and consumers are rational users. Generally, computer scientists address resource allocation by assuming that users are either obedient or Byzantine adversaries, and focus on computational efficiency, i.e., *how do we find an allocation?* In contrast, economists design market mechanisms considering incentives for rational users, and focus on economic efficiency, i.e., *what makes a good allocation?* In our work, we consider two important aspects: i) the economic impact of competition among rational users, and ii) the computational cost of incentivizing the behavior of rational users in order to preserve the desired properties in the presence of competition. Accordingly, the key properties that our mechanism focuses on are: *strategy-proof*, *budget balance*, and *computational efficiency*.

To identify the major design considerations of a pricing mechanism that allocates resources when users are rational, we look at the resource allocation process and identify its main components. We discuss the desired properties from two different perspectives, namely economic and computational. From an economic perspective, a pricing mechanism that allocates a consumer request with multiple resource types cannot achieve

incentive compatibility, budget balance, and Pareto efficiency at the same time [77]. Consequently, many related works discussed in the previous chapter have traded incentive compatibility in order to achieve Pareto efficiency and budget balance. However, in a large distributed system with rational users, spanning more than one administrative domain, Pareto efficiency is inherently lost when users are untruthful [37, 79, 103]. Thus, losing the incentive compatible property leads also to losing the Pareto efficiency property. This finding motivates us to look for a different trade-off between the economic and computational efficiencies.

From a computational perspective, a pricing mechanism should compute the allocation of requests with multiple resource types in polynomial time, while maximizing the number of allocated resources and requests. However, an optimal allocation mechanism for consumers requests with multiple resource types requires a NP-complete algorithm [50]. Accordingly, many existing systems support only requests for one resource type. From a consumer perspective, this is not efficient, since the user has to manually aggregate resources of different types. In contrast, our approach is to trade Pareto efficiency in order to achieve incentive compatibility, budget balance, and computational efficiency, while being able to allocate consumer requests with multiple resource types in polynomial time. To formally prove the achieved economic properties, we leverage on the mechanism design framework by formulating the market-based resource allocation problem as a general mechanism design problem.

3.1 Preliminaries

The allocation of shared resources is a complex process, which can be divided into several steps. In our approach, we divide resource allocation into three steps, shown in Figure 3.1. *Resource location* provides users with ways to publish and lookup resources. Next, the *pricing mechanism* is used to match consumers and providers, and to

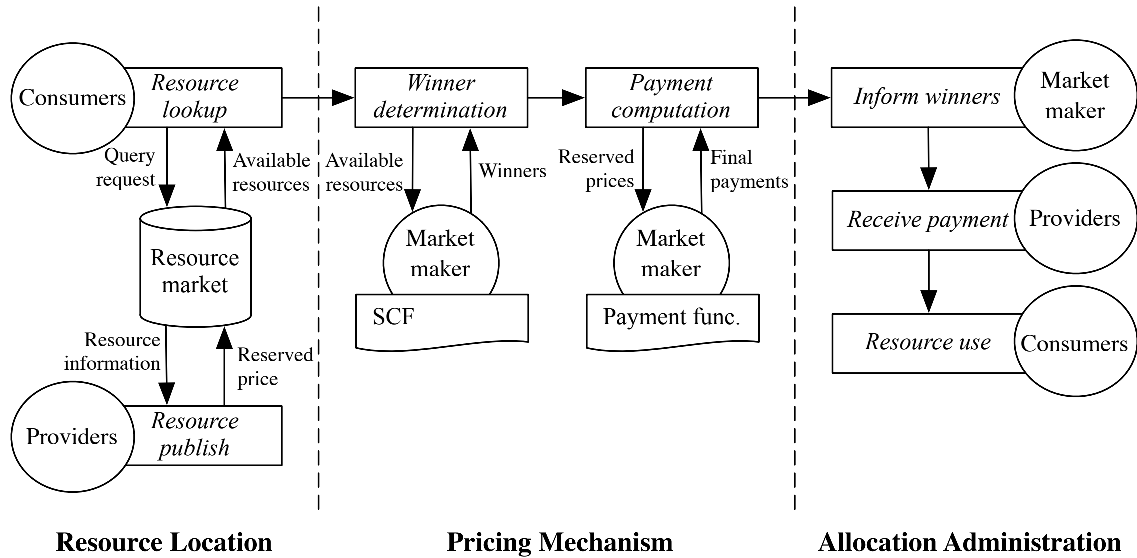


Figure 3.1: Steps in the Allocation of Shared Resources

compute payments. Lastly, *allocation administration* ensures allocation and payments take place, such that a consumer may start using resources.

1. **Resource Location**

   A key problem when dealing with large collections of shared resources is the performance of resource lookup [58, 59, 62]. As shown in Figure 3.1, the resource market receives *resource information* from providers in a *publish* message, and *query requests* from consumers in a *lookup* message. In resource location, the resource market groups the resource information such that it can respond efficiently to query requests. Searching for available resources to allocate is especially difficult when the number of resources in the market is large or dynamic. Moreover, resource information and availability may vary over time as resources join, leave, or fail in the system. The time required to perform resource lookup is a significant part in the total allocation time. To perform efficiently in these conditions, the resource location service requires scalability and support for dynamic resources. In addition, resource lookup requires support for the discovery of multiple resource types, such that con-

sumers do not have to manually aggregate resources. Lastly, resource location needs to be strategy-proof, such that truthful users achieve the best possible allocation.

2. **Pricing Mechanism**

   The pricing mechanism is the key part in market-based resource allocation. Considering the *social choice function* (*SCF*), consumer requests, and provider available resources, the pricing mechanism determines the users that get allocated in *Winner determination*. Next, using the published prices and the specific payment functions, the pricing mechanism computes the user payments, for both consumers and providers, in *Payment computation*. The performance of the pricing mechanism depends on several factors, such as the economic properties, computational efficiency, scalability, and the number of successful consumer requests and allocated provider resources.

3. **Allocation Administration**

   In the last step of resource allocation, the winning users are informed about the allocation results, and the consumer may start using the resources. Payments take place using a scalable and secure payment system. Additional features, such as management or monitoring of resources, can be included in this step.

In this thesis, we focus on resource location and the pricing mechanism, as the key factors in the pricing and allocation of shared resources. In this chapter, we consider a simple resource location scheme, with a centralized market-maker where providers and consumers submit their requests. Our objective is to design a pricing mechanism to address the required economic and computational properties. We discuss a more complex resource location mechanism in the next chapter, where we consider the scalability of our mechanism by distributing the role of the market-maker.

3.2 Design Considerations

In a resource market with a large number of providers (sellers) and consumers (buyers), the welfare obtained by both providers and consumers depends on the pricing scheme used. For example, fixed pricing cannot reflect the current market resource price due to changing resource demand and supply, which leads to lower provider and consumer welfare, and to imbalanced markets, such as under-demand. Figure 3.2 shows the provider welfare for fixed pricing (a), and dynamic pricing (b). The two pricing schemes achieve similar provider welfare when the demand is in equilibrium with the fixed price (shaded A). However, in the case of under-demand, the fixed price tends to be higher than the market price and buyers may look for alternative resources. In contrast, the dynamic scheme is able to allocate resources when the dynamic price is higher than the underlying resource costs (shaded A + B). In the case of over-demand, the fixed price limits the provider welfare, which is increased with a dynamic scheme by using a higher resource price (shaded A + C).

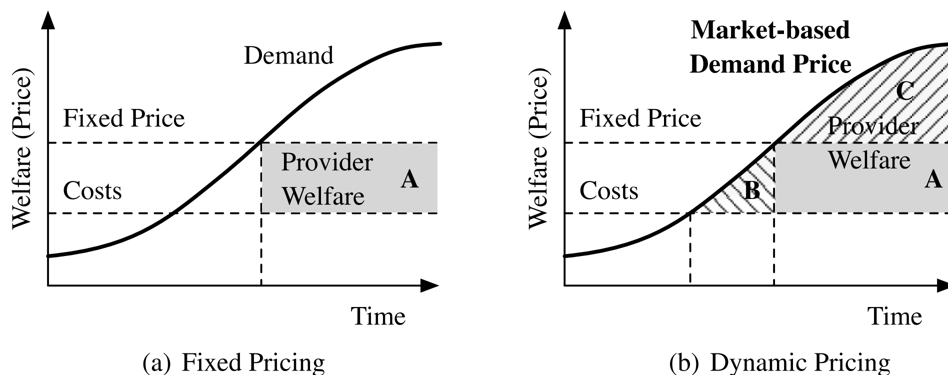


Figure 3.2: Provider Welfare with Fixed and Dynamic (Market) Pricing

Another issue in fixed pricing is that it does not incentivize consumers for increase demand or providers to offer multiple resource types. For example, early cloud computing markets such as Sun Grid Compute Utility [115] were restricted to one resource type, CPU time. Recent markets, such as Amazon S3 and EC2, introduced other re-

source types, including storage and bandwidth. Currently, Amazon has expanded its offer to ten resource types, called instances, with different fixed price for each instance type, and practice tiered pricing for storage and bandwidth [1]. We see this as an early step towards dynamic pricing and multiple resource types allocations.

3.2.1 Multiple Resource Types per Request

Allocating a consumer request for multiple resource types may lead to an increase in consumer welfare, when the consumer does not aggregate resources manually. To model multiple resource types requests, we use in this thesis a simplified resource model[1], shown in Figure 3.3. A resource type is loosely defined, and can be a hardware resource, a service, or a combination. The provider publishes a resource type by specifying the number of available items and the cost per item. A consumer request may consist of more than one resource types (multiple resource types), and contains the number of requested items for each type, and the bid that represents the total price the consumer is willing to pay for all the resources request to be allocated.

```
Resource_Type = Description
Publish = Provider_Id, Resource_Type, Items, Cost
Request = Consumer_Id, (Resource_Type, Items)+, Bid
```

Figure 3.3: Resource Description

To illustrate this simple model, consider a New York Times employee that uses 100 EC2 small instances to convert 4TB of TIFF files into PDF format [54]. The cloud user requires multiple resource types (computational power and storage), and multiple items (100 instances and 4TB space) to complete the job. However, since Amazon does not support consumer requests with multiple resource types, the user had to acquire each

[1] in a federated system with multiple providers, lack of interoperability may result in price incompatibilities, determined by different measures for the same resource type. Such issues, however, are not within the scope of this thesis, where we assume a uniform format for pricing.

resource type, using several requests. In contrast, using the above proposed request model, the user can submit only one request for all resource types. For computational power, Amazon EC2 provides eight different resource types, called instances. A *small instance* consists of 1 EC2 compute unit (approx. 1 GHz CPU from 2007) and 1.7 GB memory, priced at $0.085/hour, while a *quadruple extra-large instance* consists of 26 EC2 compute units and 64 GB memory, priced at $2/hour. Assuming the user uses only small instances, the first resource type in the request is `ec2.m1.small`, and the number of items is `100`. Amazon S3 provides storage for a monthly fee of $0.125 per GB[2]. Thus, the second resource type in the request is `s3.storage`, and the number of items is `4,096`GB. Lastly, the bid contains the maximum price the user is willing to pay, e.g., `$1000`:

```
Request = [NYT, (ec2.m1.small, 100), (s3.storage, 4096), $1000]
```

3.2.2 Economic and Computational Properties

As discussed in the previous section, we structure the allocation of shared resources into three main steps, namely resource location, pricing mechanism, and allocation administration. In Figure 3.4 we present their important properties, grouped in two categories: economic and computational. Properties such as *multiple resource types* and *scalability* are an issue for all steps of resource allocation, whereas properties such as *budget balance* and *Pareto efficiency* are considered only by the pricing mechanism. The economic properties are:

1. *Strategy-proof*

   Recent results from distributed systems show that users sharing resources are rational and try to maximize their own interest, even if by doing so they degrade the overall performance of the system [82, 109]. In our preliminary experiments, we compare

[2]Displayed prices are for the Amazon *us-east* region. Storage pricing is for the 50TB tier.

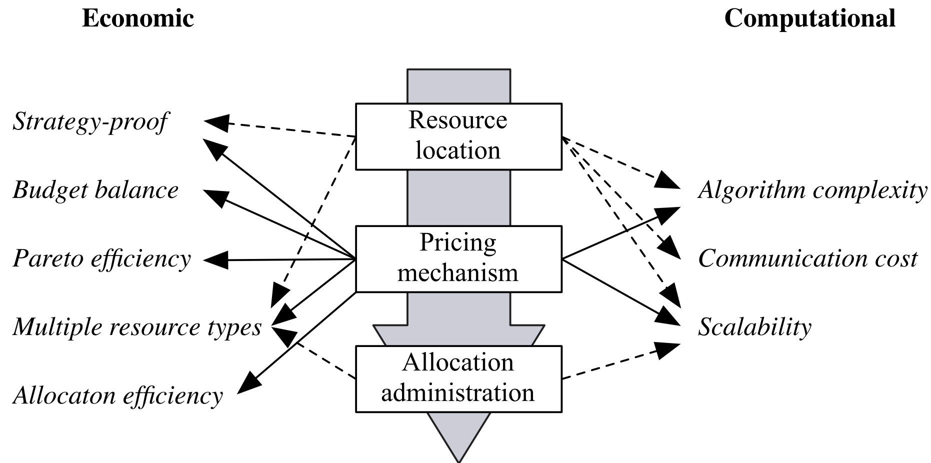


Figure 3.4: Design Considerations

the number of successful requests when all users are truthful to the case of having untruthful users, in the case of English auction. Our results included in Appendix A show that, in the presence of untruthful users, the number of successful requests is decreased. A strategy-proof resource allocation system is designed to incentivize rational users to be truthful, such that the allocation efficiency and the number of successful consumer requests are increased. Achieving the strategy-proof property is thus essential in a federated system, where users are rational, and is one of the main objectives of our work. One of the issues in achieving strategy-proof is the incentive mechanism used in the payment scheme, which may lead to different payments for consumers and providers, and budget imbalances. This is an issue because a third party is required to balance the budget.

2. *Budget Balance*

   In a budget-balanced system, the payments made by buyers are equal to the payments received by sellers. Without budget-balance, currency may be lost when allocating resources. This is an important issue in closed economic systems, which are systems with no external source of currency. Achieving budget balance when providing financial incentives constraints the payment functions that can be used, and lowers the economic efficiency of an allocation.

3. *Pareto Efficiency*

   Maximizing economic efficiency leads to a Pareto-efficient allocation. Accordingly, most of the economic-based resource allocation system discussed in the previous chapter focus on achieving economic efficiency, at the expense of other economic properties. The main issue in allocating a consumer request with multiple resource types is that maximizing efficiency requires a NP-complete pricing algorithm. Moreover, according to the Myerson-Satterthwaite theorem, maximum economic efficiency cannot be achieved at the same time with other economic properties, such as strategy-proof and budget balance.

4. *Multiple Resource Types per Request*

   Allocating consumer requests with multiple resource types per request requires more complex resource location mechanisms and payment functions, which is why most distributed systems currently consider only one resource type, such as CPU [115, 9] or disk space [6, 124]. This issue determines users that require more than one resource types to manually aggregate resources, resulting in increased waiting times and lower economic efficiency.

5. *Allocation Efficiency*

   In contrast to economic efficiency, which is a system-centric measure, the allocation efficiency provides user-centric measures for the performance of the allocation. *Consumer efficiency* represents the percentage of successful consumer requests, and *provider efficiency* represents the percentage of allocated provider resources. We consider that an allocation is more efficient when at least one of the consumer and provider efficiencies are increased. However, increasing consumer or provider efficiencies may lead to a decrease in the overall economic efficiency, or to a more elaborate pricing and allocation algorithm, with greater algorithm complexity. The design of the pricing scheme needs to find a balance or trade-off between different performance characteristics, system-centric and user-centric.

From a computational perspective, an efficient allocation is achieved by minimizing the time between the user request and the resource allocation, i.e., the *allocation time*. The computational properties we consider are:

1. *Algorithm Complexity*

   Algorithm complexity is a crucial property when considering the implementation of the pricing mechanism. Optimal allocation of consumer requests with multiple resource types requires exponential algorithms and is thus not feasible to implement. To achieve computational efficiency, existing allocation mechanisms use either simple mechanisms, such as fixed pricing and auctions, or approximate more complex mechanisms, such as the NP-complete combinatorial auction, at the expense of economic efficiency. Our approach to design a mechanism that achieves strategy-proof at the expense of economic efficiency allows us to develop a polynomial algorithm to implement our proposed payment functions.

2. *Communication Cost*

   In a system with distributed users, communication can greatly influence the computational complexity of pricing and allocation. The communication cost represents the overhead introduced by all the messages required during allocation. For example, in resource location, communication costs vary if the resource lookup mechanism is centralized or distributed. Communication can also have an impact on allocation time if the pricing mechanism is distributed.

3. *Scalability*

   Resource allocation is scalable when overall efficiency (economic and computational) is maintained in the presence of large workloads. We consider two dimensions for scalability. *Vertical scalability* considers the increase in the number of resource types in a consumer request, and *horizontal scalability* considers the increase in the number of users in the system. The issue to consider is that both resource location and the pricing mechanism must scale vertically and horizontally.

3.2.3 Trade-offs

Considering the above properties, we summarize two important trade-offs in allocating shared resources using pricing: the *economic trade-off* and the *computational trade-off*. Figure 3.5 shows the relation between economic and computational properties and their trade-offs.

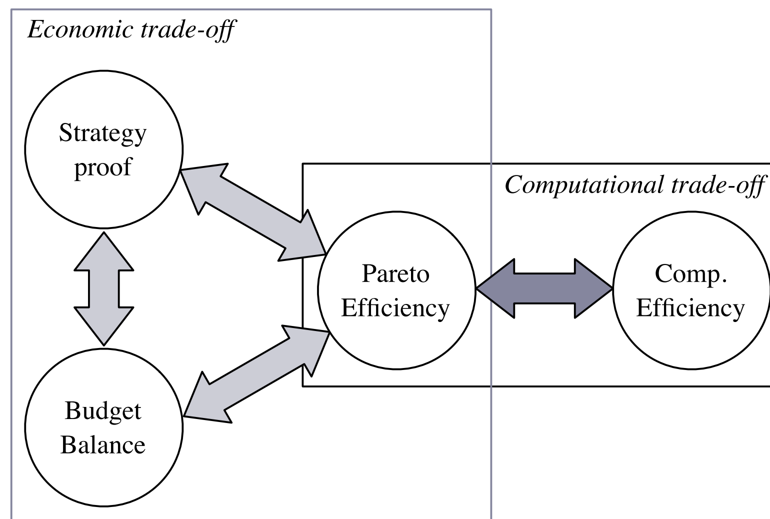


Figure 3.5: Trade-offs in Resource Pricing

1. **Economic Trade-off**

   According to the Myerson-Satterthwaite impossibility theorem [77], no pricing mechanism is strategy-proof, budget-balanced, and Pareto efficient at the same time. Most of the related works have focused on achieving maximum economic efficiency, at the loss of other economic properties. For example, combinatorial auctions results in an outcome that is Pareto efficient and strategy-proof, but it is not budget balanced, and the third-party auctioneer or market-maker has to supply the budget imbalance. This is not desirable in a competitive market with rational users, and thus most of the existing pricing mechanisms proposed for the allocation of shared computing resources have traded strategy-proof [3]. In contrast, we focus on achieving strategy-proof and budget-balance, at the expense of Pareto efficiency. The motivation is that, in the

presence of rational users, losing incentive compatibility will also lead to losing Pareto efficiency when users are not truthful.

2. **Computational Trade-off**

   Allocating consumer requests with multiple resource types per request has been shown to be an instance of the set-packing problem [50], which determines if k subsets from a list of subsets of set S are disjoint. This is a well-known NP-problem, and it results that the optimal solution, which is Pareto efficient, cannot be achieved using a computationally-efficient algorithm. This result defines the computational trade-off between Pareto efficiency and computational efficiency.

We propose to allocate shared resources using a strategy-proof pricing scheme. Our mechanism is designed to achieve budget balance, at the expense of Pareto efficiency. This trade-off allows us to implement our mechanism using a computationally-efficient algorithm that determines the allocation winners and their payments. We formally define our pricing mechanism using the mechanism design framework, and study its economic properties.

3.3 Mechanism Design Framework

Mechanism design [84], an area of game theory, studies methods to structure incentives for rational agents to behave in an intended manner. An *agent* represents a decision-maker entity in an economic model. For example, in a resource market, providers and consumers are two common types of agents. In this section, we provide an overview of the mechanism design framework, which we use in this thesis to formalize the desired economic properties in pricing, and to study the proposed scheme. We refer to rational agents as users, where an user may represent an individual user, a group, or an organization. Given a set of n users, each has private information (also called *type*) $t_i \in T_i$, and

a set $a_i(t_i) \in A$ of alternative strategies to choose from when participating in resource allocation. Informally, each user selects one strategy a_i, and the mechanism processes all the users selected strategies to produce an outcome $o \in O$, and a set of payments to and from users, p_i. This behavior is shown in Figure 3.6.

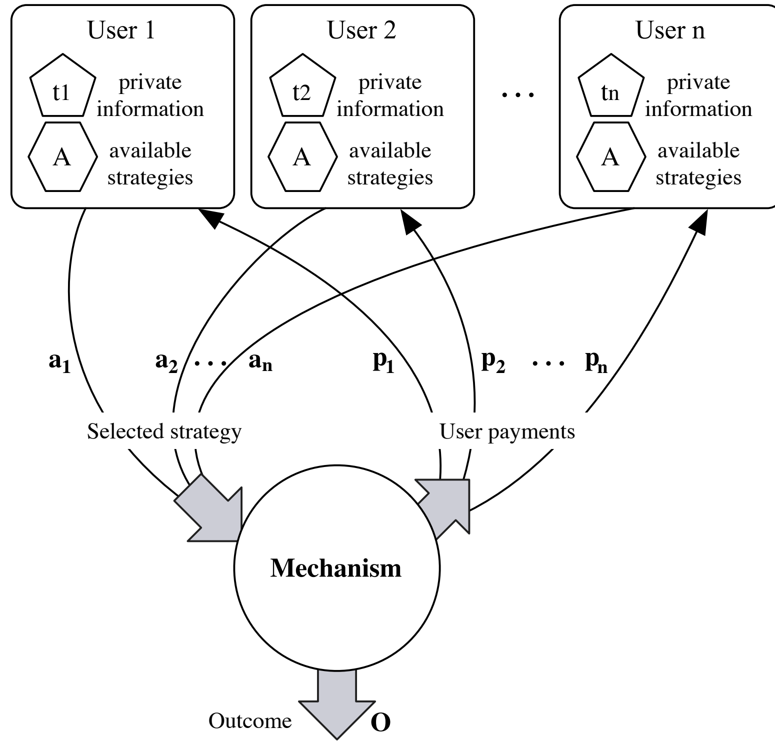


Figure 3.6: Mechanism Design Framework

A *mechanism design problem* consists of an outcome specification and a set of user utilities. The output specification maps each vector a_i of private information t_i, i.e. the strategies, to the set of allowed outputs, O. In a mechanism design problem, the output function can be optimized in different ways, according to the chosen *social choice function*.

Definition 5 (Social Choice Function)**.** *A* ***social choice function*** *takes the preferences of* n *users and chooses a single alternative output.*

$$f : T_1 \times T_2 \times \cdots \times T_n \to O \tag{3.1}$$

The solution to a mechanism design problem is formally defined as follows.

Definition 6 (Mechanism). *A* ***mechanism*** *M is a pair (o, p_i), where $o = o(a_1, a_2, \ldots, a_n)$ is the selected outcome, and $p_i = p_i(a_1, a_2, \ldots, a_n)$ is the payment for each user.*

The *valuation* for a particular outcome o of user i is denoted by $v_i(t_i, o)$. The strategy chosen by the user, a_i, affects both the final outcome, o, and its payment p_i. Thus, we can derive the user overall *welfare* or *utility* as:

$$u_i = v_i(t_i, o) + p_i(a_i) \tag{3.2}$$

When choosing a strategy a_i, a user may decide not to reveal its private information, t_i. We use $a_i = t_i^d$ to denote the information declared by the user i, and $a_{-i} = t_{-i}^d$ to denote the information declared by all the other users except i. Thus, we can express the user welfare from Equation 3.2 as:

$$u_i(t_i^d, t_{-i}^d) = v_i(t_i, o(t_i^d, t_{-i}^d)) + p_i(t_i^d, t_{-i}^d) \tag{3.3}$$

The *rational* behavior of a user is defined in a mechanism by *individual rationality*.

Definition 7 (Individual Rationality). *In an* ***individual-rational*** *mechanism, rational users gain higher utility from actively participating in the mechanism than from avoiding it. This is expressed formally as:*

$$u_i \geq 0 \tag{3.4}$$

A mechanism in which the users strategies are to declare their private information is called *truthful* or *incentive compatible*. The *revelation principle* [49] states that any mechanism design problem solution with dominant strategies, i.e. the best-response strategy no matter what strategy the other users adopt, can be converted into a truthful mechanism.

Theorem 1 (Revelation Principle)**.** *If there exists a mechanism that implements a given problem with dominant strategies then there exists an incentive compatible implementation as well.*

The revelation principle was introduced and proved by Gibbard [51], and later extended by Vickrey for the second-price sealed bid auction [120].

Definition 8 (Incentive Compatibility)**.** *A mechanism M is* ***incentive compatible*** *or* ***truthful****, if $A = T$ with the following property:*

$$\begin{aligned} \forall i, \forall a_i \in T, \forall a_{-i} \in T_{n-1}, \quad u_i(t_i, a_{-i}) &= v_i(t_i, o(t_i, a_{-i})) + p_i(t_i, a_{-i}) \\ \geq \quad u_i(a_i, a_{-i}) &= v_i(t_i, o(a_i, a_{-i})) + p_i(a_i, a_{-i}) \end{aligned} \tag{3.5}$$

Informally, in an incentive compatible mechanism the dominant strategy for a user is to declare its private information, such that $a_i = t_i$. A mechanism is *strongly truthful* when truth-telling is the only dominant strategy.

Definition 9 (Strategy-proof Mechanism)**.** *A mechanism that is both incentive-compatible and individual rational is said to be* ***strategy-proof****.*

As discussed previously, the output of the mechanism is optimized according to the social choice function used. Several well-known social choice functions that characterize known economic paradigms are:

i) Maximize the global social welfare (capitalism):

$$f(t_1, \ldots, t_2) = max_o \sum_i u_i(t_i, o) \tag{3.6}$$

ii) Maximize the minimal welfare (socialism):

$$f(t_1, \ldots, t_2) = max_o min_i(u_i(t_i, o)) \tag{3.7}$$

iii) Minimize the differences of welfare (communism):

$$f(t_1,\ldots,t_2) = min_o max_{i,j}(u_i(t_i,o) - u_j(t_j,o)) \tag{3.8}$$

iv) Maximize the number of users with positive welfare:

$$f(t_1,\ldots,t_2) = max_o \#(i; u_i(t_i,o) > 0) \tag{3.9}$$

In the model we propose[3], economic efficiency is achieved by maximizing the sum of user utilities, such as in Equation 3.6. The concept of *economic efficiency* is different from the engineering approach commonly used in computer science, where allocating resources to competing applications does not deliver the greatest value to the users given a limited amount of resources. In contrast, economic systems measure efficiency with respect to the user valuation for resources.

Definition 10 (Pareto Efficiency)**.** ***Pareto efficiency*** *is achieved when, given an allocation, no Pareto improvement can be performed.*

Definition 11 (Pareto Improvement)**.** *Given an initial allocation, a* ***Pareto improvement*** *is a shift to a new allocation that can make at least one user better off, without making any other user worse off.*

For example, providing a *20GB* disk quota in a system with a *60GB* disk to each of three users, which initially have no quota, is a Pareto efficient solution. Moreover, allocating *40GB* to one user, *20GB* to the second, and none to the third is also a Pareto efficient solution, as the only way to allocate space for the third user is to decrease the quota of at least one of the other users. In contrast, allocating a *10GB* share to each user, such that half of the disk space remains unallocated, is not Pareto efficient.

[3] In our work, we consider $u_i = v_i + p_i$, as in Equation 3.2, a model that is known as the *quasilinear utility model*.

Definition 12 (Budget Balance)**.** *In a **budget-balanced** mechanism, the sum of all user payments is zero. Formally, it is expressed as:*

$$\sum_i p_i = 0 \tag{3.10}$$

Budget balance ensures that allocations do not result in budget deficit or surplus.

In our work, we focus mainly on strategy-proof and budget balance in order to allocate resources when providers and consumers are rational. Strategy-proof ensures that payments include financial incentives for truthful users. Budget balance ensures that, for each request, consumer payments equal provider payments. This is important because budget balance eliminates the need of a third party, such as a market-maker or auctioneer, to supply the incentives for the users. Our proposed mechanism uses a social choice function that maximizes economic efficiency, given the strategy-proof and budget-balance constrains. While this may not lead to Pareto efficiency because of the above constraints, our trade-off ensures that the achieved economic efficiency is determined only by truthful user valuations.

Vickrey-Clarke-Groves Mechanisms

A prevalent result of mechanism design, Vickrey-Clarke-Groves (VCG) mechanisms [29, 57, 73, 120] are a general technique for constructing truthful mechanisms.

Definition 13 (VCG Mechanism)**.** *A mechanism M belongs to the **VCG** class if*

$$o = max_o \sum_{i=1}^{n} u_i(t_i, o) \tag{3.11}$$

$$p_i = \sum_{j \neq i} u_j(t_j, o) + h_i \tag{3.12}$$

where h_i is an arbitrary function of private information of the other agents.

VCG mechanisms have been shown to be both truthful and Pareto-efficient when pricing one [120] or more resource types [29]. Informally, they are incentive compatible

because user payments are determined independently of their declared private information; hence there are no incentives for rational users to declare false information. Furthermore, VCG mechanisms are efficient because the payment is a function of all other users valuations, and individual rational because all users welfare is positive, with the welfare of the users involved in an exchange greater than 0.

Formally, the truthful property can be expressed by the *equilibrium* $\forall i,\ a_i = t_i^d = t_i$, meaning that truth-telling is the dominant strategy for all agents. The proof [29, 120] is derived directly from the mechanism definition:

$$u_i = v_i(t_i, o) + p_i(a_i) = v_i(t_i, o) + \sum_{i \neq j} v_j(t_j, o) + h_i$$

Agent i has no influence in determining the output of function h_i. Furthermore, using this strategy choice that affects the output o, the agent is only able to maximize the sum:

$$v_i(t_i, o) + \sum_{i \neq j} v_j(t_j, o) = \sum_{i=1}^{n} v_i(t_i, o)$$

However, this sum is also maximized by the VCG mechanism when using the social choice function in Equation 3.11, thus the rational agent has no incentives to declare false valuations.

The main issue of using VCG mechanisms in resource allocation is that no pricing mechanism achieves maximum economic efficiency, truthfulness, and budget-balance at the same time, a result known as the *Myerson-Satterthwaite impossibility theorem* [77]. Indeed, VCG mechanisms are usually not budget-balanced [49] and require a third-party, called a *market-maker*, to mediate between consumers and providers, and to provide the surplus or deficit budget. Parkes et al. [88] argue that budget-balance is possible in a Vickrey-Clarke-Groves mechanism if payments are implemented on one side of the exchange, and that side has no aggregation. In resource allocation, an aggregation involves a bundle containing more than one resource type. In addition, a

practical issue in implementing a VCG mechanism for the allocation of consumer requests for multiple resource types is that Pareto efficient solutions require an algorithm with exponential computational complexity. In this context, *algorithmic mechanism design* [45, 84] studies the computational aspect of mechanism design. Using a model based on polynomial-time centralized computation, Nisan and Ronen [84] found that both the output and payment functions must be polynomial-time computable for the mechanism to be tractable. However, a centralized computational model is not scalable in systems where both users and resources are distributed. Thus, Feigenbaum et al. [45] introduce *distributed algorithmic mechanism design* and develop a theory for network complexity in regard to a distributed mechanism, which considers the number of messages sent in the network, the size of the message, and the local computational done on each distributed user. Considering these results, we analyze the proposed scheme both for computational complexity and communication requirements.

3.4 Proposed Scheme

As discussed previously, a mechanism design problem consists of an outcome specification and a set of user utilities. The output specification maps each vector of private information $t_1, \ldots t_n$ to the set of allowed outputs, $o \in O$. We formulate the resource allocation problem in a resource sharing system where rational users can both provide and consume resources as a general mechanism design problem as follows.

Definition 14 (Market-based Resource Allocation Problem)**.** *Given a market with consumer requests and provider resources, each market participant is modeled as a rational user i with private information t_i. The provider private information t_s^r denotes the underlying costs of the offered resource r, including fixed and operational costs. The consumer private information t_b^R denotes the maximum price the consumer is willing to pay such that resources are allocated to satisfy its request R. User's i valuation is*

t_i^r for providers and t_i^R for consumers if the resource r (request R) is allocated, and 0 *otherwise. For a request R, the goal of the mechanism is to allocate resources such that the underlying resource costs are minimized.*

Accordingly:

- *The possible outputs of the mechanism are all partitions $x = x_1 \dots x_n$ of resources that satisfy the request R, where x_i is the set of resources that user i contributes to the allocation.*
- *The objective function is $g(x,t) = \sum_{|x_i|>0} \sum_{j \in x_i} t_i^j$.*
- *User's i valuation is $v_i(x,t_i) = \sum_{j \in x_i} t_i^j$.*

This is an optimization problem, since the output specification is given by a positive real valued objective function, $g(x,t)$, and the output o minimizes g. Moreover, it is utilitarian[4] since the objective function satisfies the relation $g(o,t) = \sum_i u_i(t_i,o)$. An utilitarian optimization problem allows us to apply a VCG-based payment function to the design problem, which we describe in the following section.

To allocate shared resources in a minimum amount of time, we propose an *auction-based mechanism*, which takes into consideration all existing resources in the market when a consumer request is received. Since it is based on auctions, our proposed pricing mechanism has low computational complexity, at the expense of Pareto efficiency. Specifically, to allocate consumer requests for multiple resource types, we propose the use of *reverse auctions*, where consumer requests are auctioned and provider resources are selected for allocation. Auctions are generally modeled as bidding games of incomplete information. With incomplete information, the utility of each user is private information. Thus, each user estimates the other users private information and predict

[4]In this thesis, we use the currency unit to measure utility (welfare) and the $ symbol to specify the virtual currency in our system. The relationship between virtual and real currencies is outside the scope of this thesis.

their welfare according to the utility function. The users strategies are functions that convert their private information into a currency amount that constitutes the bid. Accordingly, our proposed pricing mechanism uses sealed bids to model user utilities, both for the consumer reserved price, and for the provider resource information.

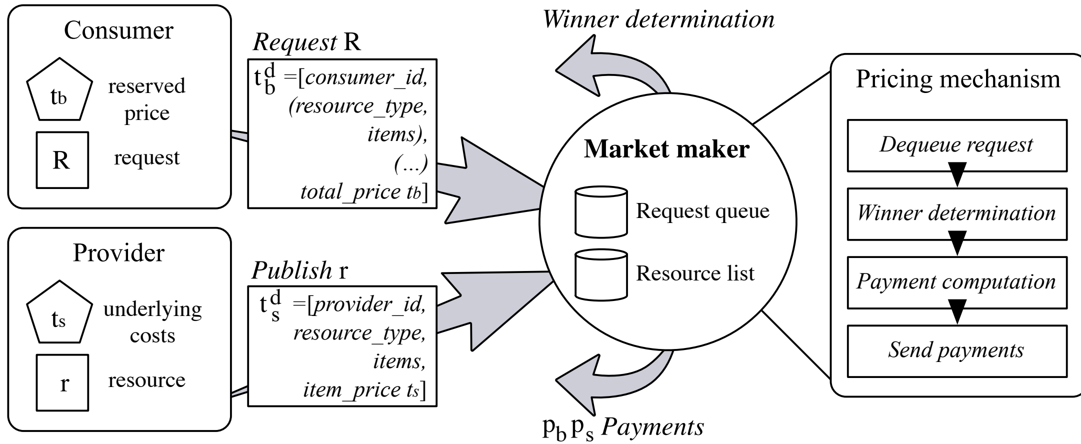


Figure 3.7: Proposed Pricing Mechanism

As shown in Figure 3.7, we propose to employ a third party, namely the *market-maker* or *auctioneer*, which: i) collects the sealed bids, ii) selects the winners, and iii) computes the payments. *Winner determination* decides the providers selected for allocation, such that the underlying resource costs are minimized. Next, *Payment computation* determines the payments for the winning providers, according to the market supply and not as a function of the provider published prices.

The pricing mechanism we propose has several advantages. Firstly, since it is an online mechanism, pricing is computed on-demand, on consumer request arrivals. This results in faster allocations, since it eliminates the waiting time of an offline mechanism. Secondly, the allocation time is further reduced due to the low algorithm complexity of the auction-based scheme. Lastly, using a third-party market-maker allows the computation of incentives independent of the user valuations, such that the strategy-proof property can be achieved by the pricing mechanism.

3.4.1 Winner Determination

Given a set of provider items of different resource types, and a set of consumer requests with multiple resource types per request, the winner determination computes a feasible allocation of items to the consumers. More formally, the winner determination problem finds a partition $x_R = x_1 \ldots x_n$ of resources that satisfy the consumer request R, where x_i is the set of resources that the provider i contributes to the allocation.

In the proposed mechanism, a user, provider or consumer, sends requests to the market-maker, containing the declared information, which represents the input for our pricing functions. Firstly, the market-marker selects a consumer request and determines the set of successful providers. Consumer requests are selected on a *First Come First Serve* basis to minimize waiting time. We show in Section 3.4.4 that, while the First Come First Serve policy may lead to a loss in economic efficiency, it is nevertheless crucial in achieving the strategy-proof property. The winning providers are determined such that all items of all resource types in the consumer request are allocated and the underlying resource costs are minimized. By minimizing the resource costs, the total welfare of the providers is maximized.

3.4.2 Payment Computation

After the winner determination step selects the consumer request and the provider resources that will be allocated, the mechanism computes, in the second step, the payments for providers and consumers, according to the following payment functions.

Provider Payment Function

The payment received by a provider, p_s, is determined using the following VCG-based [39] provider payment function we propose:

$$p_s = \begin{cases} 0, & \text{if provider } s \text{ does not contribute} \\ & \text{resources to satisfy the request} \\ c_{M|s=\infty} - c_{M|s=0} & \\ & \text{if provider } s \text{ contributes} \\ & \text{resources to satisfy the request} \end{cases} \tag{3.13}$$

where:

$c_{M|s=\infty}$ is the lowest cost to satisfy the consumer request without the resources from provider s;

$c_{M|s=0}$ is the lowest cost to satisfy the consumer request when the cost of resources from provider s resources is 0.

This is a VCG-class payment function, where $c_{M|s=\infty}$ corresponds to $h_i(t_{-i})$, the arbitrary function of resource prices of the other participants, and $c_{M|s=0}$ corresponds to $v_s(t_s, o)$, the valuation of provider s (see Equation 3.12). VCG payments can be applied because our market-based resource allocation problem is an utilitarian optimization problem.

Consumer Payment Function

Given the set S of providers with resources to satisfy a request R, the price paid by the consumer for resources, p_b, is determined using the consumer payment function:

$$p_b = -\sum_{s \in S} p_s \tag{3.14}$$

The above function defines the consumer payment function as the total sum of the provider payments, and determines our trade-off in economic efficiency. Although the

above function does not maximize economic efficiency such as a VCG function, it achieves the budget balance property of our mechanism.

3.4.3 Generalized Reverse Auction Algorithm

The proposed pricing algorithm is presented in Algorithm 1 and Algorithm 2. Assume a market consisting of consumer requests (request_queue) and provider available resources (resource_list) published to a market-maker, together with their reserved prices, t_i (private information). Consumer requests join a request queue where each request is scheduled for allocation based on a policy that is independent of the buyer's valuation, such as FCFS (line 4). First, we determine the winning providers, using the function DetermineWinners, as shown in Algorithm 2. For each resource type (line 31), the market-maker sorts the available resource list based on the provider reserved price, t_s (line 33). Next, it determines the winning providers by selecting the providers from the head of the sorted list such that all items in the request may be allocated (lines 39–45). Subsequently, we determine the payments for each winning provider p_s by computing the two associated costs, $c_{M|s=\infty}$ and $c_{M|s=0}$ (lines 8–19). Lastly, we compute the consumer payment p_b (lines 20–21) and inform the winners of the allocation if their welfare is greater than 0 (lines 24–29). If the request cannot be allocated, it will be placed back in the request queue, and will be considered for allocation when more resources are available (line 25). The monopoly situation, which occurs when $c_{M|s=\infty}$ cannot be determined, is handled using different pricing functions, as explained in Section 3.5.

Example

Consider the market for *computational resource* (CPU) and *storage* (DISK) presented in Figure 3.8, with three providers and two consumers. Provider S_1 sells one item (instance) of computational resource for \$1/hour, S_2 provides one instance of compu-

Algorithm 1: MarketMaker

```
   Data  : request_queue, resource_list
   Result: winners, payments
1  while request_queue ≠ ∅ do
2      winners ← ∅
3      payments ← ∅
4      request ← request_queue.Dequeue ()
5      // determine winners
6      <winners, c_{M|s} > ← DetermineWinners (request, resource_list)
7      foreach seller in winners do
8          // determine c_{M|s=∞}
9          new_resource_list ← resource_list − seller.resources
10         <nil, c_{M|s=∞} > ← DetermineWinners (request, new_resource_list)
11         if error then
12             // monopoly situation detected
13             // ...
14         end
15         // determine c_{M|s=0}
16         c_{M|s=0} ← c_{M|s}
17         foreach resource_type in request do
18             c_{M|s=0} ←
               c_{M|s=0} − seller.resource_type.items ∗ seller.resource_type.price
19         end
20         payment ← c_{M|s=∞} − c_{M|s=0}
21         payments.Add (seller, payment)
22     end
23     // if the request cannot be allocated, put it back in the
           queue
24     if request.price > payments.total then
25         request_queue.Enqueue (request)
26     else
27         // allocate winners with payments
28         // ...
29     end
30 end
```

Algorithm 2: DetermineWinners

```
   Data  : request, resource_list
   Result: winners, payments
31 foreach resource_type in request do
32     resources ← Filter (resource_list, resource_type)
33     PriceSort (resources)
34     while resource_type.items > 0 do
35         if resources.Empty() then
36             return error
37         end
38         // determine sellers for each resource type
39         resource ← resources.Head()
40         seller ← resource.owner
41         if seller.resource_type.items ≥ resource_type.items then
42             items ← resource_type.items
43         else
44             items ← seller.resource_type.items
45         end
46         seller.resource_type.items ← seller.resource_type.items − items
47         resource_type.items ← resource_type.items − items
48         winners.Add (seller, items)
49         payments.Add (seller, items ∗ seller.resource_type.price)
50     end
51 end
```

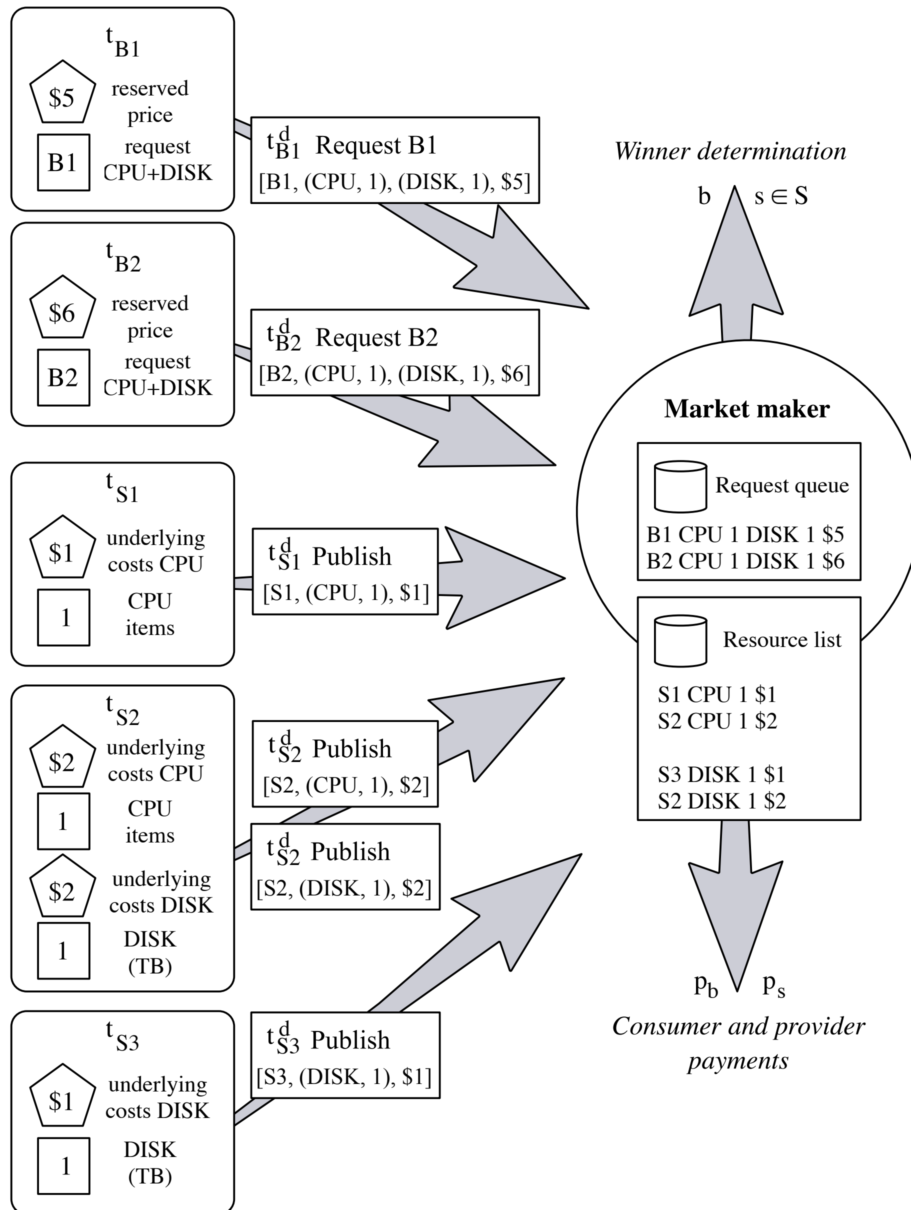


Figure 3.8: Market with Two Consumers and Three Providers

tational resource at \$2/hour and 1TB of storage at \$2/hour, and S_3 provides 1TB of storage for \$1/hour. Consumers B_1 and B_2 require each one instance of computational resource and 1TB of storage for one hour at a total of \$5 and \$6, respectively.

```
S1_Publish = [S1, (CPU, 1), $1]
S2_Publish = [S2, (CPU, 1), $2]
S2_Publish = [S2, (DISK, 1), $2]
S3_Publish = [S3, (DISK, 1), $1]
B1_Request = [B1, (CPU, 1), (DISK, 1), $5]
B2_Request = [B2, (CPU, 1), (DISK, 1), $6]
```

Assume request B_1 arrives before request B_2 and the resource list is sorted based on the resource price. Using the FCFS policy B_1 is selected by the market-maker and the winning providers are S_1 for computational resource and S_3 for storage. Table 3.1 shows the payment computation for the providers S_1 and S_3, and the consumer B_1. From this example, we observe both the strategy-proof and budget-balance properties achieved by our mechanism, and the trade-off in terms of economic efficiency.

User	$\mathbf{c_{M\|s=\infty}}$	$\mathbf{c_{M\|s=0}}$	**Payment (\$)**
S_1	2 + 1 = 3	0 + 1 = 1	-3 + 1 = -2
S_3	1 + 2 = 3	1 + 0 = 1	-3 + 1 = -2
B_1	N/A	N/A	- (-2 - 2) = 4

Table 3.1: Example of Payment Computation

To illustrate the trade-off, we present next a solution that uses VCG payments for both providers and consumers to achieve Pareto efficiency at the expense of budget-balance. We compute the total welfare, a measure of economic efficiency, for all possi-

Total Welfare (\$)		**Allocation**
w/o S_1	**6 - 2 - 1 = 3**	**B_2 buys from S_2, S_3**
w/o S_2	6 - 1 - 1 = 4	B_2 buys from S_1, S_3
w/o S_3	**6 - 1 - 2 = 3**	**B_2 buys from S_1, S_2**
w/o B_1	6 - 1 - 1 = 4	B_2 buys from S_1, S_3
w/o B_2	**5 - 1 - 1 = 3**	**B_1 buys from S_1, S_3**
maximum	6 - 1 - 1 = 4	B_2 buys from S_1, S_3

Table 3.2: Total Welfare and Selected Winners

ble exchanges in Table 3.2, where winning users are shown in bold font. To maximize the sum of user utilities according to our social choice function, the winners selected in our example are S_1, S_3 and B_2, with the payments shown in Table 3.3.

User	Payment ($)
S_1	-1 - (4 - 3) = -2
S_3	-1 - (4 - 3) = -2
B_2	6 - (4 - 3) = 5

Table 3.3: VCG Payments for All Winners

From Table 3.2, VCG payments (B_2 buys from S_1, S_3) have higher total welfare than our proposed algorithm (B_1 buys from S_1, S_3). However, VCG payments achieve Pareto efficiency with a budget deficit of $1 and require an algorithm with exponential complexity, making the trade-off in terms of both budget-balance and computational efficiency.

3.4.4 Proofs of Economic Properties

In this section, we look at the economic properties of the proposed pricing scheme, and prove that it achieves individual rationality, budget balance, and incentive compatibility.

Theorem 2. *The proposed mechanism is individual rational.*

Proof. Consider the allocation of a request R from a consumer b. The output of our mechanism is $x = x_1 \dots x_n$, where x_i is the set of resources that provider i contributes to the allocation. Formally, the IR property is achieved when $u_i \geq 0$ for any winner participant i. The users that are not winners and do not participate in the allocation have $p_i = 0$, thus $u_i = 0$. The winning participants in the allocation are: b, the consumer, and S, a set of providers such that $s \in S \iff |x_s| > 0$.

When determining the winners, our mechanism chooses the providers with the lowest underlying costs. Let σ be the winning provider with the highest costs, and ς the provider with the lowest costs that is not a winner. Consequently:

$$v_\sigma = \max_{s \in S} v_s, v_\varsigma = \min_{j \notin S} v_j; v_\sigma \geq v_\varsigma$$

The payment received by the provider σ is:

$$p_\sigma = -c_{M|\sigma=\infty} + c_{M|\sigma=0} = -(v_{S-\sigma} + v_\varsigma) + v_{S-\sigma} = -v_\varsigma \tag{3.15}$$

Thus, the utility of the provider σ is:

$$u_\sigma = v_\sigma - v_\varsigma \geq 0 \tag{3.16}$$

Given the consumer payment, $p_b = -\sum_{s \in S} p_s$, consumer utility is:

$$u_b = v_b - \sum_{s \in S} p_s \tag{3.17}$$

The market-maker implements the mechanism by solving the winner determination problem and computing user payments. From Equation 3.16, we see that provider utility is always positive. To ensure that consumer utility is also positive, the market-maker verifies that the result in Equation 3.17 is greater than zero. Since all participants having positive welfare, the proposed mechanism is individual rational. □

Theorem 3. *The proposed mechanism is budget-balanced.*

Proof. It is trivial from Equation 3.14 that the proposed mechanism uses the consumer payment function to achieve budget balance. Indeed, given a consumer request R, and the set of winning providers S, the sum of all user payments is:

$$\sum_i p_i = p_S + p_b = \sum_{s \in S} p_s - \sum_{s \in S} p_s = 0 \tag{3.18}$$

□

Theorem 4. *The proposed mechanism is incentive compatible.*

Proof. We show incentive compatibility by proving that both the provider payment function and the consumer payment function are incentive compatible.

Lemma 1. *Provider payment function is incentive compatible.*

Proof. From Equation 3.13, we can see that the provider payment function is based on VCG, and thus achieves Pareto efficiency for providers. Accordingly, given an allocation output o, no Pareto improvement can be performed to increase the utility of that provider without decreasing the utility of some other rational user. Thus, no alternative output o' obtained by declaring a different value t_s^d can improve the utility of provider s:

$$o(t_s, t_{-i}) \geq o'(t_s^d, t_{-i}) \tag{3.19}$$

The utility of a provider is the sum of its valuation and its payment. The valuation is a function of its private information and the output of the mechanism, and based on the previous equation we derive:

$$v_s(t_s, o(t_s, t_{-i})) \geq v_s(t_s, o'(t_s^d, t_{-i})) \tag{3.20}$$

From Equation 3.12, the provider payment is a function of the valuation of the other users and an arbitrary function of private information of the other users. Accordingly, given the result in Equation 3.19, the provider payments for the truthful and for the declared private information are:

$$\sum_{j \neq i} v_j(t_j, o(t_s, t_{-i})) + h_i(t_{-i}) \geq \sum_{j \neq i} v_j(t_j, o'(t_s^d, t_{-i})) + h_i(t_{-i}) \tag{3.21}$$

By adding the terms in Equation 3.20 and Equation 3.21, we obtain:

$$\begin{gathered} v_s(t_s, o(t_s, t_{-i})) + \sum_{j \neq i} v_j(t_j, o(t_s, t_{-i})) + h_i(t_{-i}) \geq \\ \geq v_s(t_s, o'(t_s^d, t_{-i})) + \sum_{j \neq i} v_j(t_j, o'(t_s^d, t_{-i})) + h_i(t_{-i}) \end{gathered}$$

where the sum of factors represent the provider utility. Accordingly, the provider payment function is incentive compatible:

$$u_s(t_s, t_s^d, t_{-i}) \leq u_s(t_s, t_s, t_{-i}) \tag{3.22}$$

□

Lemma 2. *Consumer payment function is incentive compatible.*

Proof. From Equation 3.12, the consumer payment function depends on the provider payments p_s, and is independent of the consumer valuation, v_b. Provider payments depend on the private information of other providers, t_{-s}, and the provider valuation for an outcome, $v_s(t_s, o)$. Thus:

$$p_b(t_b^d, t_{-i}) = p_b(t_b, t_s) = p_b(t_b, t_{-i}) \tag{3.23}$$

Given this result, to achieve incentive compatibility we only require the consumers to have at least the same valuation independent of the outcome selected by our pricing mechanism, $v_b(t_b, o)$. This is achieved by selecting consumer requests using a strategy that is independent of the consumer valuation, such as first-come-first-serve (FCFS). Thus, requests are considered by a market-maker based on the time of their arrival, and not the intrinsic value for the consumer valuation, such that:

$$v_b(t_b, o'(t_b^d, t_{-b})) = v_b(t_b, o(t_b, t_{-b}))$$

Furthermore, the market-maker does not take into consideration the valuation of providers when computing the payment for a consumer, as long as the condition for IR is satisfied. This allows us to rewrite the above equation as:

$$v_b(t_b, o'(t_b^d, t_{-b})) = v_b(t_b, o(t_b, t_{-b}^d)) = v_b(t_b, o(t_b, t_{-i})) \tag{3.24}$$

where t_{-b} represents the private information of all other consumers, and t_{-i} the private information of all other users (providers and consumers). Adding Equation 3.24 and Equation 3.23, we obtain the equality:

$$v_b(t_b, o'(t_b^d, t_{-b})) + p_b(t_b^d, t_{-i}) = v_b(t_b, o(t_b, t_{-i})) + p_b(t_b, t_{-i})$$

that is reduced to:

$$u_b(t_b, t_b^d, t_{-i}) = u_b(t_b, t_b, t_{-i}) \tag{3.25}$$

This relation expresses the IC property of the consumer payment function. □

From Lemma 1 and Lemma 2 results, the equality

$$t_i^d = t_i \tag{3.26}$$

shows the incentive-compatible property of the proposed mechanism. □

3.5 Theoretical Analysis and Limitations

Computational efficiency is a major design criteria in the allocation of shared resources. Optimal mechanisms such as combinatorial auctions are not feasible to implement since the winner determination algorithm is NP-complete. The computational properties we consider in our design include the algorithm complexity and scalability. In this section, we study these properties by analyzing the run-time complexity of the winner determination algorithm and of the consumer and provider payment functions. Without considering queuing time, we define the total allocation time as:

$$T = T_w + T_p \tag{3.27}$$

where T_w is the time taken to determine the winners and T_p is the time taken to compute the payment. We consider the following inputs: RT, the number of resource types in a consumer request; I_{RT_k}, the number of items from the resource type RT_k in a request; S_{RT_k}, the number of providers with resource type RT_k. We use $\sum_k S_{RT_k}$ to denote the total number of published resources.

The winner determination algorithm in Algorithm 2 contains two loops (lines 27 and 30) for the number of resource types in the request and for the number of items of each resource type, while the inner code (lines 31–42) takes a constant amount of time. Finding all resources with the same type (line 28) depends on $\sum_k S_{RT_k}$, while sorting resources according to their price takes $O(S_{RT_k} \log S_{RT_k})$. Thus, in the worst case, the complexity of the winner determination algorithm is:

$$T_w = O(RT \times (\sum_k S_{RT_k} + S_{RT_k} \log S_{RT_k} + I_{RT_k})) \tag{3.28}$$

Similarly, we compute the complexity of the payment functions (Algorithm 1, lines 7–18), which, in the worst case scenario, is $T_p = O(I_{RT_k} \times T_w)$, when each winning provider is allocated one item. Thus, the allocation time when using the proposed scheme is:

$$\begin{aligned} T &= T_w + O(I_{RT_k} \times T_w) \\ &= O(I_{RT_k} \times T_w) \\ &= O(I_{RT_k} \times RT \times (\sum_k S_{RT_k} + S_{RT_k} \log S_{RT_k} + I_{RT_k})) \end{aligned} \tag{3.29}$$

In summary, the complexity of the proposed algorithm is a polynomial function of the number of resource types in a consumer request (RT), the number of items requested for each resource type (I_{RT_k}), the total number of published resources ($\sum_k S_{RT_k}$), and the number of providers with resource type k (S_{RT_k}). While the complexity of our algorithm is not an issue, the scalability becomes a problem when the number of resource types in consumer requests or the number of providers increases. In addition, scalability is also

an issue with a centralized market-maker, which may become a bottleneck. To address scalability, we propose in the next chapter a distributed auction scheme, which eliminates the centralized market-maker, and allocates different resource types in a consumer request in parallel.

The Monopoly Situation

One of the limitations of a VCG payment scheme is the monopoly situation [91]. In a monopoly situation, a consumer request may not get allocated even though resources are available. This issue occurs when one or several providers gain exclusive price control with their resources by providing a sufficiently large portion of items in the market. The authors found that the situation occurs frequently in transient states, i.e. before the size of the system reaches steady-state. In system where the numbers of providers fluctuate, the monopoly situation may occur anytime.

In the proposed reverse auction-based mechanism, two main cases lead to a monopoly situation:

- without the monopolistic provider, there are not sufficient resources to satisfy a consumer request;
- without the monopolistic provider, the consumer payment is higher than the consumer reserved price.

The occurrence rate of the monopoly situation depends on the number of users in the market. For example, in a market with more than 80 users, the occurrence rate is less than 3% [91]. In a monopoly situation, winner determination matches a consumer request with provider resources, but payments cannot be computed by the payment functions. To allocate resources when a monopoly situation occurs, we use a modified payment scheme, which is budget-balanced but partial strategy-proof, as the provider payment function is not incentive-compatible.

The proposed provider payment function for monopoly situation is:

$$p_s = f \cdot V_s \tag{3.30}$$

Similarly, the consumer payment function is:

$$p_b = f/E \cdot V_b \tag{3.31}$$

where V_s, V_b represent the declared price of the allocated resources, based on n_{rt}, the number of items from resource type rt, and p_{rt}, the published item price of resource type rt:

$$\forall i, V_i = \sum_{rt} n_{rt} \cdot p_{rt}$$

In the consumer payment function, E represents the ratio between the total consumer and provider published prices:

$$E = \frac{V_b}{\sum_{s \in S} V_s}$$

The partial incentive compatibility is achieved using any arbitrary function f, such that $1 \leq f \leq E$.

Example

Considering the same example as above, after the first allocation, a VCG mechanism cannot allocate the second consumer request because payments cannot be computed. Figure 3.9 shows the market after the allocation of B1, with provider S2 and consumer B2. With a VCG-based payment scheme to allocate the request B2, winner determination finds the provider $S2$, however $c_{M|s=\infty}$ and $c_{M|s=0}$ cannot be determined and allocation cannot take place. In this market, S2 is a monopolistic provider since there are not sufficient resources in the market to allocate the request of consumer B2 without the participation of S2. However, an allocation is in fact possible, as the reserved price of S2 is smaller than the reserved price of B2.

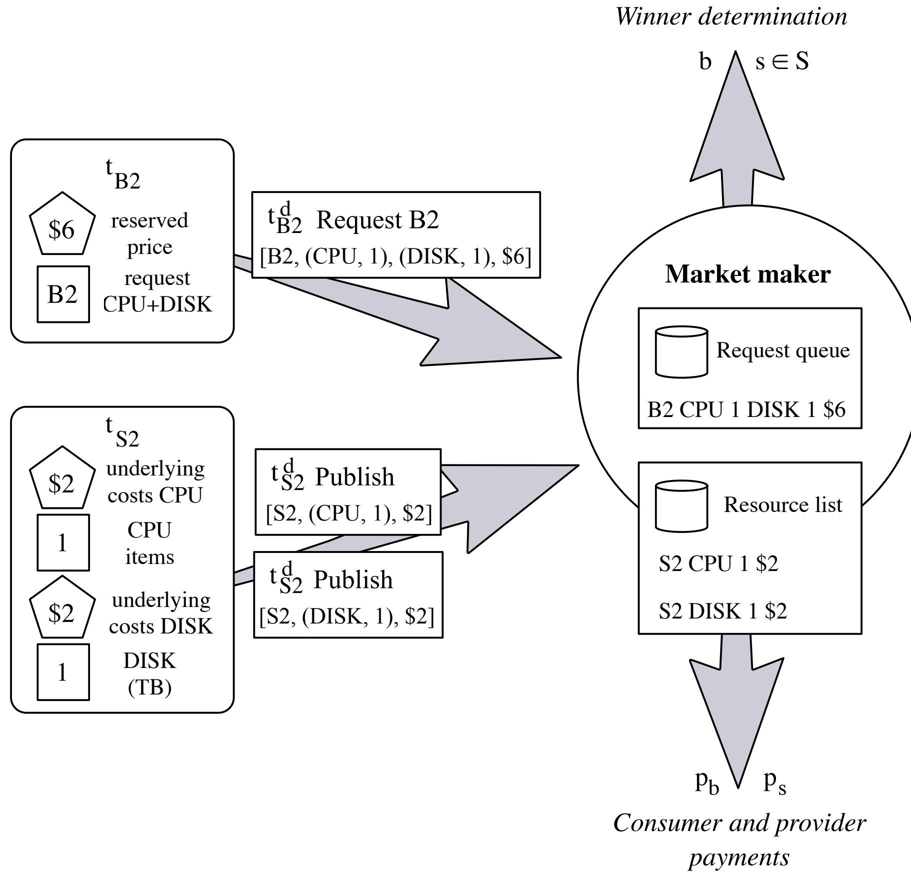


Figure 3.9: Market with Monopolistic Provider

Using the above pricing function and the arbitrary function $f = (E+1)/2$, we calculate the following user payments:

$V_{B2} = 6$

$V_{S2} = 1 \cdot 2 + 1 \cdot 2 = 4$

$E = \frac{V_{B2}}{V_{S2}} = 1.5$

$f = \frac{E+1}{2} = 1.25$

$p_{S2} = f \cdot V_{S2} = 5$

$p_{B2} = f/E \cdot V_{B2} = 5$

Although this scheme allows the computation of user payments, it does not guarantee incentive compatibility, and monopolistic providers can increase their welfare simply by publishing a higher resource price.

3.6 Summary

In this chapter, we propose a strategy-proof pricing mechanism than can be used to allocate consumer requests with multiple resource types per request. The design of our scheme takes into consideration economic and computational properties, such as strategy-proof, budget-balance, Pareto efficiency, multiple resource type allocations, computational efficiency, allocation efficiency, and scalability, both vertical and horizontal. We assume a system with consumers and providers, and model a closed virtual economy, where users may earn currency by sharing resources. We propose to achieve strategy-proof, budget balance, computational efficiency, and trade-off Pareto efficiency. The main contributions of the proposed pricing and allocation mechanism are the following: i) formulating the market-based resource allocation problem as a general utilitarian mechanism design optimization problem, which allows the use of a strategy-proof VCG mechanism; ii) the design of provider and consumer payment functions, with support for consumer requests with multiple resource types per request; and iii) the analysis of the proposed pricing mechanism, with formal proofs for the achieved economic properties.

We introduce a simple resource type model, such that sellers may provide and consumers may request multiple items and multiple resource types. Resource allocation is divided into three steps: i) resource location, which provide users with ways to publish and search for resources; ii) the pricing mechanism, which matches providers and consumers, and computes payments; and iii) allocation administration, where payments take place and the consumer can start using the allocated resources. Our theoretical analysis of the proposed pricing mechanism found scalability to be an issue. Specifically, the allocation time of the consumer request depends on the number of resource types in the request and the number of providers. Thus, with a centralized market-maker, under high load generated by consumer requests with many resource types and

with a large number of providers, the allocation time and, consequently, the consumer waiting time, is expected to increase. Lastly, we have presented an alternative allocation scheme for the monopoly situation, when there are not sufficient resources to satisfy a consumer request, or when the consumer payment is higher than its reserved price. Although the alternative scheme presented allows allocations in the monopoly situation, it does not achieves incentive compatibility, and providers may publish higher resource prices to increase their welfare.

Chapter 4

Distributed Resource Pricing

The previous chapter introduced our proposed strategy-proof pricing mechanism that can be used to allocate consumer requests with multiple resource types per request in systems where rational users provide and consume resources. Using the mechanism design framework, we have shown that our proposed scheme achieves strategy-proof and budget balance, while it trades Pareto efficiency to achieve computational efficiency. However, we have found that scalability is a limitation of the proposed scheme: *horizontal scalability*, when increasing the number of resource types, and *vertical scalability*, when increasing the number of users. This might become an issue for systems such as clouds, which are currently expanding both in breadth, i.e. the number of resource types offered, and in depth, i.e. the number of providers. As cloud computing is becoming mainstream, Software as a Service (SaaS) in addition to Infrastructure as a Service (IaaS) offerings can be treated as resource types. Our theoretical analysis has shown that horizontal scalability is an issue because the allocation time for a consumer request increases proportionally with the number of resource types in the request. Vertical scalability is an issue when the number of users increases, because our scheme is based on a centralized auction model, while the users are distributed over the network. In this chapter, we introduce a distributed pricing scheme that addresses both vertical and horizontal scalability in large resource markets. The key idea is to use an overlay network

to distribute provider resource information and consumer requests by resource type in a distributed hash table. To address concurrency issues, we propose a three-phase commit algorithm that uses Lamport logical clocks to synchronize pricing for different resource types belonging to the same consumer request.

4.1 Preliminaries

Traditionally, auctions are carried out by a third party, called market-maker or auctioneer, which collects the bids, selects the winners, and computes the payments [66, 89]. In the previous chapter, we have considered for simplicity a centralized market-maker, to which providers publish resources, and consumers send requests. However, having a centralized entity reduces the scalability of the system and degrades the performance when the number of users is large and when their behavior is dynamic [56]. To address this issue, we propose to distribute the task of the market-maker to multiple users, providers or consumers. Accordingly, there is no dedicated market-maker, since any existing user can be selected to play the role of the market-maker for an allocation.

The economic properties of the pricing mechanism, such as strategy-proof and budget balance, need to be maintained in the distributed scheme. To perform winner determination and compute user payments according to the proposed mechanism, users that play the role of the market-maker need accurate up-to-date information about the resources in the system, provided by resource discovery. We propose the use of a peer-to-peer overlay that provides efficient routing and object location within a network where a peer node represents a provider or a consumer. In our proposed scheme, a globally consistent peer-to-peer protocol maintains the structural properties of the overlay while users dynamically join, leave or fail in the network. Resource information is shared in the overlay network between users. Resource discovery is performed by leveraging the distributed hash table [117] used by the overlay network protocol [116].

Distributed hash table (DHT) is a decentralized lookup scheme that provides scalable object location with high result guarantee in large distributed systems [102, 114]. Similar to the traditional hash-table data structure, a DHT provides an interface, *lookup*, to retrieve a key-value pair. The *key* is an identifier assigned to a resource, such as a hash value associated with a resource shared by a provider. The *value* is an object that is stored in the DHT, which can be the resource itself, an index to the resources, or resource metadata. There are several important DHT concepts, responsible for its properties:

- Structured overlay network – users are organized as a *structured* overlay network. The overlay balances between the routing performance and the overhead of the protocol that maintains the routing information.
- Key-to-node mapping – DHT maps a key k to an overlay node n, where n is the node *closest* to k in a shared identifier space. The relation defined by *closest* has different meanings in DHT implementations, e.g. the first node in clockwise direction from key k [114], or the node with the closest identifier to k [102]. Node n is referred to as the node *responsible* for k.
- Data-item distribution – all key-value pairs with the key k are stored at the same node n, regardless of the owner of the pair. This is done by using the *store* interface provided by the DHT.

Finding a key k implies routing a request to the node that stores k. Routing in a DHT is usually performed in $O(\log N)$ steps[1], where N denotes the total number of nodes [114]. This is because each node n maintains a *finger table* of m entries, where each entry i points to the node that stores the key $n+2^{i1}$. When $N < 2^m$, the finger table consists of only $O(logN)$ unique entries [114]. Accordingly, by utilizing the finger-tables, each key can be located in $O(\log N)$ hops. Using a DHT as the underlying data structure in our

[1] with caching, lookups can be completed in several hops, instead of $O(\log N)$.

scheme allows us to leverage the overlay network and sets the premise for a scalable, distributed pricing mechanism.

4.2 Issues in Distributed Auctions

The allocation of resources using distributed auctions has been studied in the past [55, 89], and overlay networks have been proposed for resource discovery in federated clouds [96]. Two common applications for distributed auctions are the allocation of network bandwidth [13, 127], and the task assignment problem [131]. The typical approach for distributed auctions is for several providers to auction for the same resource type at the same time in the distributed system. With no centralized market-maker, each consumer sends bids to different market-makers at the same time, in rounds. However, there are several economic and computational challenges, which we discuss below.

1. *Multiple Resource Types per Request*

   Most of the related works consider only one resource type (for example bandwidth [13], or tasks [131], respectively), and thus the distributed auction schemes they propose are not relevant in allocating consumer requests with multiple resource types.

2. *Social Choice Function*

   Since consumers bid directly to multiple providers, the auction process takes place in rounds with open bids, similar to an English auction. This may be desired when the objective is to maximize the provider welfare [35, 118]. However, the social choice function in our pricing mechanism is to maximize the total system welfare, which include both the provider and consumer.

3. *Strategy-proof*

   Auctioning with open bids result in losing incentive-compatibility and strategy-proof. This is because providers do not bid their truthful valuation since they must take into account effects such as the winner's curse [66], common in public auctions.

4. *Economic Efficiency*

   Eliminating the market-maker leads to auctions that are budget-balanced by sacrificing Pareto efficiency, such that maximum economic efficiency is not achieved [34].

5. *Communication Time*

   In a distributed scheme, an overhead of additional messages is required in the winner determination process [55]. For example, each consumer sends bids to multiple providers several times, depending on the number of auction rounds. In addition, the influence of network delay in the auction process is another issue in distributed auctions due to the increased communication overhead [84].

6. *Concurrency*

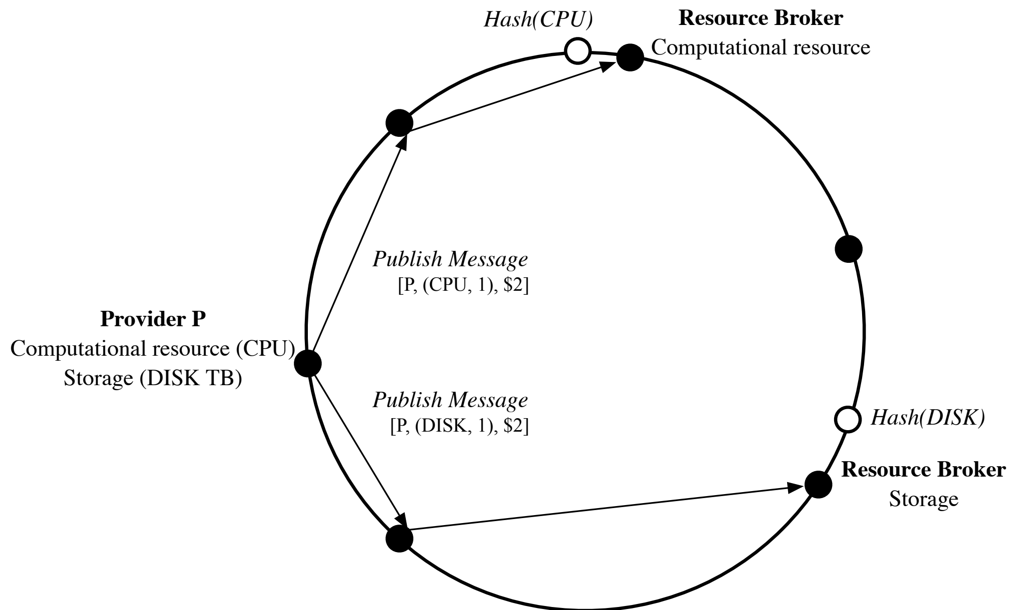
   In distributed auctions with no market-maker, concurrency issues arise from bidding to multiple providers for the same resource type. If the role of the market-maker is distributed among multiple participants, concurrency issues can severely decrease the performance of pricing and allocation [84].

The pricing mechanism proposed in the previous chapter addresses the economic issues outlined above. Our reverse auction-based scheme can allocate consumer requests with multiple resource types, the social choice function maximizes the total welfare, and achieves strategy-proof. Thus, the key issue is distributing the role of the market-maker, while preserving the economic properties, with low overhead from communication and concurrency. We propose a distributed scheme where users are part of a peer-to-peer overlay network, such that any user can be assigned the role of the market-maker in an allocation. To maintain the economic properties of our pricing scheme, the main challenge is to distribute the resource information, such that it is available when determining the winners and computing payments. Our approach is to leverage the DHT for resource lookup and distributed pricing by resource type. The next section presents in detail our scheme, while the subsequent section analyses its communication and concurrency overhead.

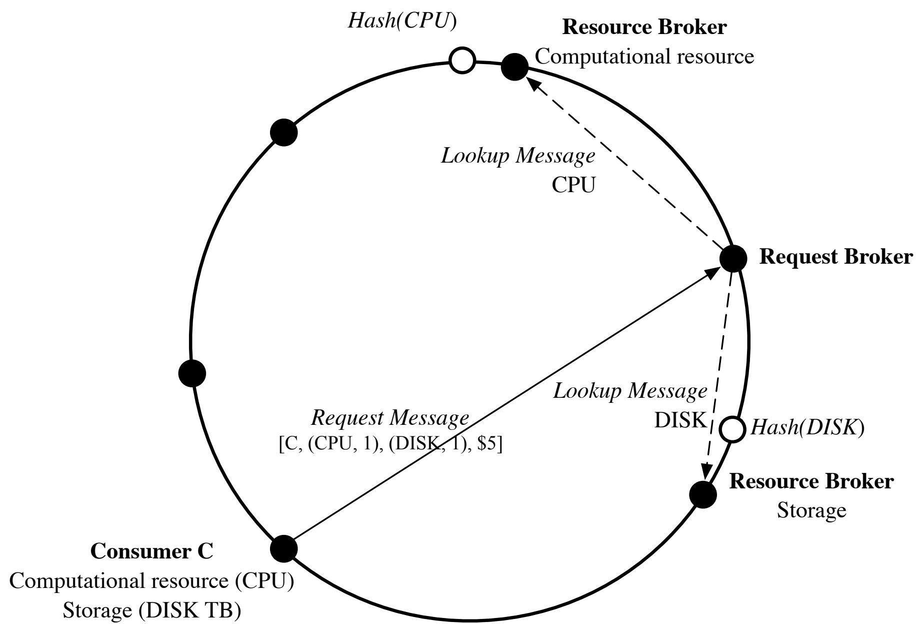
4.3 Proposed Scheme

We consider the problem of allocating shared resources in a distributed market context. Rational users, providers and consumers, are organized as nodes in a peer-to-peer overlay network, where each user has limited information about the other users. Providers publish resource information using the DHT *store* interface, where the *key* is the hash of the resource type, and the *value* is the resource metadata, such as provider information, number of items, and price per item. The node responsible for a resource type in the overlay is the *resource broker*. We use the hash of the resource type as the key for resource data distribution such that all information about that resource type is stored at the same resource broker. Thus, each resource broker will be able to compute payments for consumer requests containing the respective resource type, for which the broker is the node responsible.

A consumer request is sent by the user using the DHT *store* interface. We use an arbitrary key, such that the request is routed in the overlay to a random peer, which we refer to as the *request broker*. By routing the request to a random peer, we achieve the incentive compatibility property for consumers similar to the way we select requests using the first-come-first-serve strategy in the centralized scheme. The role of the request broker is to use the *lookup* interface provided by the DHT to find resource information about all the resource types in the request. Accordingly, lookup requests for each resource type in the request are routed by the overlay, from the request broker to the resource brokers that are responsible for the respective resource type keys. Having all information about a resource type allows resource brokers to compute payments using the proposed scheme and to send them to the request broker. After receiving all payments from the resource brokers, the request broker decides if the allocation is possible, i.e. the sum of all provider payments is less or equal to the consumer price.



(a) Provider and Resource Brokers



(b) Consumer and Request Broker

Figure 4.1: Distributed Resource Pricing Scheme

Figure 4.1 shows the provider *publish* operation and the consumer *request* operation where the hash of the resource type shared is used as the key, and the resource metadata, i.e. owner information, number of items, and the price per item, is used as the value in the DHT store and lookup operations. There are two resource types, *computational resource* and *storage*, published at two resource brokers, the peers in the overlay responsible for the keys representing the hash of each resource type, respectively.

The *provider* in Figure 4.1(a) publishes two resource types, namely *computational resource* and *storage*, to the respective resource brokers using the DHT interface:

```
store(hash(resource type), resource metadata)
```

The resource metadata stored by the resource broker contains the provider identifier, the number of items for that resource type, and the price per item. This is similar to the information stored by the centralized market-maker in the scheme proposed in Chapter 3. We use dotted lines to represent messages routed by the overlay network, with a routing complexity of $O(\log N)$, where N is the number of peers in the overlay.

The *consumer* in Figure 4.1(b) sends a request for two resource types, *computational resource* and *storage*, using the DHT interface:

```
store(hash(random key), request metadata)
```

where the request metadata contains all the resource types, the number of items for each resource type, and the total price the buyer is willing to pay. The request is routed by the overlay to a random user that becomes the request broker. The request broker enqueues the request and, for each resource type, uses the DHT interface:

```
lookup(resource type)
```

to find the resource brokers and request the number of items for the respective resource types. When receiving a request, the resource brokers use the pricing mechanism described in Chapter 3 to determine the winning providers and to compute their payments.

The payments are then sent back to the request broker. In Figure 4.1, we use a continuous line to represent a message exchanged directly between peers, without being routed by the overlay. After the payments for each resource type are received, the request broker performs the allocation, informs the consumer, and continues with the next request from the queue.

4.3.1 User Roles

In contrast to the centralized scheme, where a user could be either a provider or a consumer, in the proposed distributed scheme a user additionally can be a request broker or a resource broker. In this section, we present in more detail the user roles in the peer-to-peer overlay.

Provider

Providers join the overlay network to share resources in exchange for currency in the system. The published resources are available as long as the provider is part of the overlay; when the provider leaves the overlay, its published resources become unavailable. A publish message is sent for each resource type and is received by the user with the node identifier "closest" to the hash of the respective resource type, i.e. the resource broker. A provider may update the published information for a resource type, such as the number of available items and price per item, by sending another publish message. Providers are not aware of the node identifiers of the resource brokers, and thus use the hash of the resource type to send messages to the resource brokers, which are routed by the overlay network. An example of creating a publish message is shown in Algorithm 3.

When the allocation is performed, the resource broker informs a provider about the payments and the consumer node identifier, such that direct communication can be established between provider and consumer.

Algorithm 3: SendPublish Example

```
resource_type ← "Computational resource"
items ← 2
itemPrice ← 2
resource ← createResource (resource_type, items, itemPrice)
key ← hash (resource_type)
metadata ← resource
store (key, metadata) // store operation provided by the DHT
```

Consumer

Consumers join the overlay network to find resources that match their requests. In the proposed scheme, a consumer request may contain multiple types of resources. The request message sent by a consumer contains all the resource types, the number of items for each resource type, and the total price the consumer is willing to pay. An example of creating and sending a consumer request is shown in Algorithm 4.

Algorithm 4: SendRequest Example

```
resource_type_A ← "Computational resource"
items_A ← 1
resource_type_B ← "Storage"
items_B ← 1
request ← CreateRequest (resource_type_A, items_A)
request.Add (resource_type_B, items_B)
request.SetReservedPrice (5)
key ← hash (random (resource_type_A + resource_type_B))
metadata ← request
store (key, metadata) // store operation provided by the DHT
```

The message is received by the user with the node identifier "closest" to a random key generated by the consumer, which becomes the request broker. If the request is allocated, the request broker responds with a list of payments and provider node identifiers, such that direct communication can be established between provider and consumer.

Resource Broker

A peer in the overlay becomes a resource broker when it is responsible for a resource type, i.e. the user node identifier is "closest" to the hash of a resource type. The relation "closest" is defined by the specific overlay implementation, e.g. can be numerically closest in the case of Pastry [102], the first node in clockwise direction in the case of Chord [114], etc. Thus, a user becomes a resource broker when: i) receives a publish message from a provider, and ii) receives a lookup request from a request broker.

Resource brokers maintain a list of published resources for each resource type they are responsible for. When receiving a lookup request, the resource broker determines the winners and computes the provider payments according to the provider payment function in Equation 3.13. The result is returned to the request broker directly, without being routed in the overlay network. After a successful allocation, the resource broker removes the allocated resources from its list and informs the winning providers about the allocation and payments.

Request Broker

A peer in the overlay is a request broker when it becomes responsible for the random key generated by a consumer making a request, i.e. the user node identifier is "closest" to the request key. The request broker sends a lookup request for each resource type in the consumer request and waits for payments from resource brokers. Initially, since the request broker does not know the node identifier of the resource brokers, lookups are routed by the overlay using the resource type identifier. If allocation is possible, i.e. the total provider payments are less than the consumer price, the request broker sends a *commit* message directly to the resource brokers, using their node identifiers. In addition, the request broker informs the consumer about the payments and allocation.

4.3.2 Achieved Properties

The distributed auction mechanism outlined above is, in fact, a two-phase commit protocol. As shown in Figure 4.2, the request broker lookup corresponds to the *commit-request* phase, and the resource broker response containing payments is the *agreement* message. If the allocation is possible, the request broker sends the *commit* message to the resource brokers and informs the consumer. By using a blocking protocol, such as two-phase commit, the economic properties of the pricing mechanism are preserved in the distributed setting, because lookup requests are serialized at the resource brokers. Informally, incentive compatibility for providers is maintained because resource brokers do not compute payments for other requests while waiting for a commit message. Thus, the consumer requests are serialized and performed according to the first-come-first-serve strategy by the request broker. Incentive compatibility for consumers is maintained by using a random key for the consumer request. Allocation of requests for multiple resource types is achieved by sending multiple lookup requests to resource brokers. Lastly, individual rationality and budget balance is achieved by the request broker by checking that the providers payments are less than the consumer reserved price.

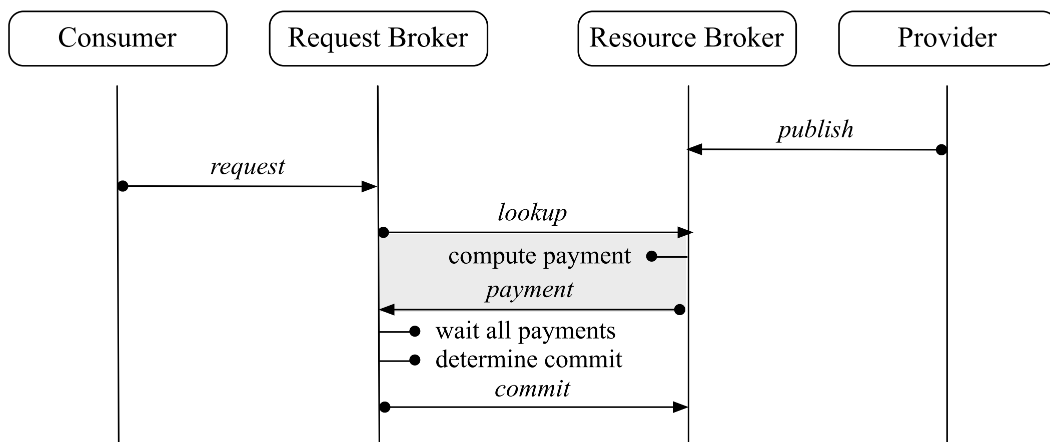


Figure 4.2: Two-Phase Commit Protocol

There are two key advantages in the proposed distributed auction mechanism. Firstly, the payment computation for a consumer request is parallelized. Thus, a request broker divides the consumer request into several lookup messages, one for each resource type. Each lookup is received by different resource brokers, which compute provider payments at the same time, such that the computation time for the entire allocation is reduced to the computation time for only one resource type. Accordingly, we have addressed the issue of horizontal scalability, since the allocation time of a consumer is not proportional to the number of resource types requested. Secondly, requests for different resource types are also processed in parallel. In contrast to the centralized model, where all consumer requests are processed by the market-maker sequentially, having a distributed scheme allows different resource brokers to make concurrent allocations. This solves the issue of vertical scalability, since in the distributed scheme the allocation times will not increase linearly with the number of users in the system. An experimental analysis of the horizontal and vertical scalability is presented in Chapter 6. Using the peer-to-peer model, where any user can be a provider, consumer, resource broker and request broker, not only adds scalability to the our market-based resource allocation solution, but also eliminates the need for a dedicated centralized market-maker.

4.3.3 Generalized Distributed Auction Algorithm

When serializing consumer requests, concurrency is an issue when simultaneous requests with multiple resource types arrive at the resource brokers in different order. Figure 4.3 shows an example where deadlock occurs. Two consumer requests with common resource types, i.e. *computational resource* and *storage*, are routed by the DHT overlay to different request brokers, i.e. *Request Broker 1* and *Request Broker 2*. Each request broker searches for resources using the hash of the respective resource type as lookup *key*, and finds the resource brokers responsible for the respective resource types: *Computational Resource Broker* and *Storage Broker*, respectively. However, due

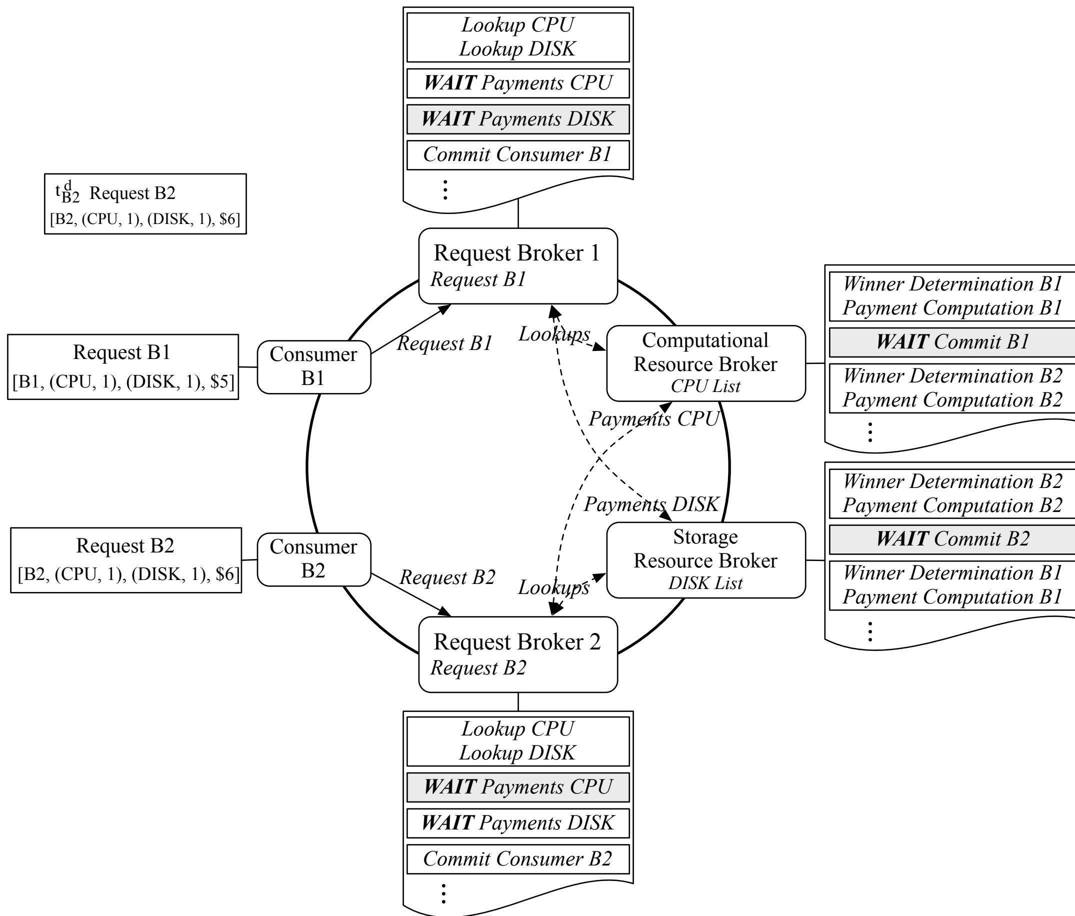


Figure 4.3: Deadlock in Distributed Auctions

to network delays, the lookups arrive out-of-order, i.e. *Computational Resource Broker* receives the lookup request from *Request Broker 1* first, then the request from *Request Broker 2*, while *Storage Broker* receives the request from *Request Broker 2* first, and from *Request Broker 1* second. *Computational Resource Broker* sends the provider payments for *Request B1* computational resource to *Request Broker 1* and blocks waiting for the *commit* message. Similarly, *Storage Broker* sends the provider payments for *Request B2* storage to *Request Broker 2* and blocks waiting for the *commit* message. However, no resource broker receives the *commit* message since the request brokers are waiting for the provider payments for all resource types before committing.

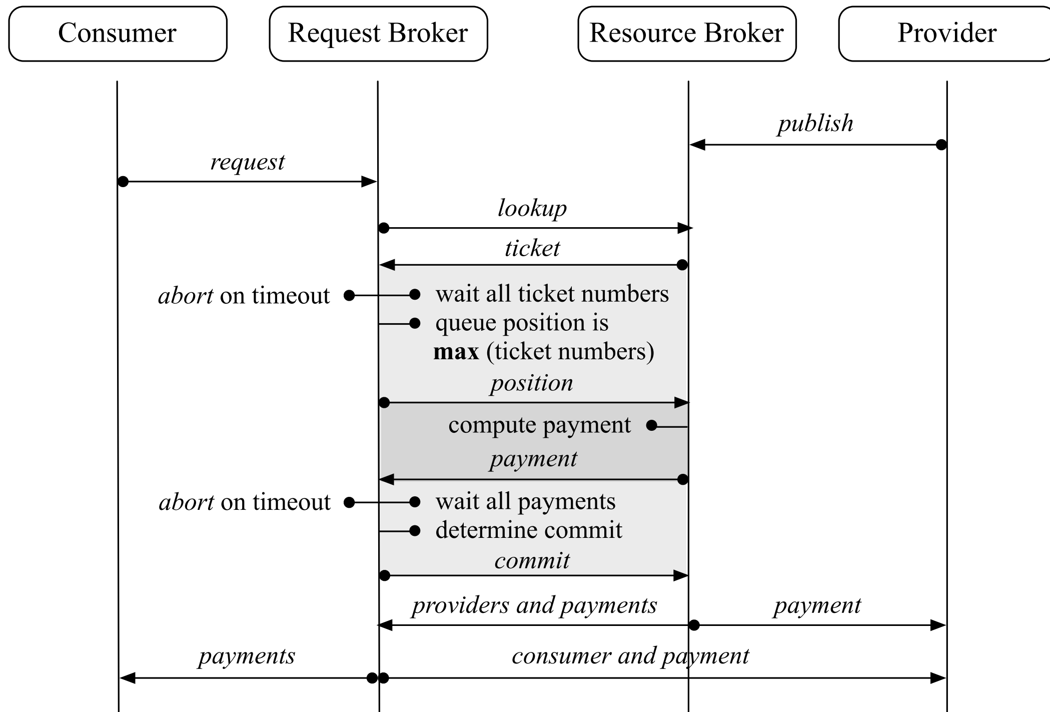


Figure 4.4: Deadlock-free Protocol

In order to prevent deadlock, we propose an algorithm which makes use of a three-phase commit protocol in conjunction with Lamport's logical clocks, as shown in Figure 4.4. The algorithm uses the Lamport timestamps, referred to as *tickets*, to determine the partial order of requests and subsequently sort the pricing of each resource type at the resource brokers. The algorithm contains three steps, detailed below.

Step 1. In the first phase, the request broker receives a request from the consumer, and makes lookup requests for each resource type in the consumer request (see Algorithm 5).

The resource broker responds to the lookup message with a *ticket* number, which is a Lamport logical timestamp with the number of the next request in the resource broker queue, as in Algorithm 6.

Algorithm 5: Request Broker: ReceiveRequest

```
    Input: request
    Data : request_queue
1   request_queue.Enqueue (request)
2   foreach resource_type in request do
3   |   key ← hash (resource_type)
4   |   value ← request.resource_type
5   |   SendLookup (key, value)
6   end
```

Algorithm 6: Resource Broker: ReceiveLookup

```
    Input: resource_type
    Data : lookup_priority_queue, next_ticket
1   lookup_priority_queue.Enqueue (resource_type, next_ticket)
2   SendTicket (resource_type.source, next_ticket)
3   next_ticket = next_ticket + 1
```

Step 2. The request broker waits for all *ticket* numbers, corresponding to each resource type in the consumer request. In phase two, shown in Algorithm 7, after all ticket numbers are collected, the request broker sends a message with the *maximum* ticket number to all resource brokers, which rearrange the resource type requests in the queue accordingly. If one of the resource brokers fails, the request broker will send an *abort* message to the other resource brokers.

Algorithm 7: Request Broker: ReceiveTicket

```
    Input: ticket
    Data : request_queue, request, ticket_list
1   ticket_list.Add (ticket)
2   if ticket_list.size = request.size then
3   |   position ← max (ticket_list)
4   |   foreach ticket in ticket_list do
5   |   |   SendPosition (ticket.source, position)
6   |   end
7   end
```

Next, the resource brokers compute the provider payments using our pricing function in Algorithm 8, and send back the total payment for the consumer to the request broker from the top of the lookup requests queue.

Algorithm 8: Resource Broker: ReceivePosition

```
    Input: position
    Data : lookup_priority_queue, ticket, resource_type, resource_list
1   lookup_priority_queue.Remove (resource_type)
2   lookup_priority_queue.Add (resource_type, position)
3   if lookup_priority_queue.head = position then
4   |   winner_list ← DetermineWinners (resource_type, resource_list)
5   end
6   SendPayment (position.source, TotalPayments (winner_list))
7   ticket = position + 1
```

Step 3. In phase three, after all payments are received by the request broker, a *commit* message is sent as in Algorithm 9 to the resource brokers if the allocation can be performed. If the allocation is not possible or a timeout occurs, it will send an *abort* message to the other resource brokers.

Algorithm 9: Request Broker: ReceivePayment

```
    Input: payment
    Data : request_queue, request, ticket_list, payment_list
1   payment_list.Add (payment)
2   if payment_list.size = request.size then
3   |   if payment_list.total ≤ request.price then
4   |   |   commit ← YES
5   |   else
6   |   |   commit ← NO
7   |   end
8   |   foreach payment in payment_list do
9   |   |   SendCommit (payment.source, commit)
10  |   end
11  |   SendAllocation (request.source, payment_list.total)
12  |   request_queue.Remove (request)
13  end
```

The resource broker responds with the provider node identifiers, and the individual payment for each provider. Finally, the resource broker informs the providers, and the request broker the consumer about the allocation (see Algorithm 10). The providers and consumer communicate directly to allocate and use the resources.

Algorithm 10: Resource Broker: ReceiveCommit

```
    Input: commit
    Data : lookup_priority_queue, resource_type, resource_list, winner_list
1 if commit then
2     foreach provider in winner_list do
3         resource_list.Remove (provider, resource_type.items)
4         SendAllocation (provider, resource_type.items)
5     end
6 end
7 lookup_priority_queue.Remove (resource_type)
```

4.4 Theoretical Analysis and Limitations

Our distributed scheme addresses scalability issues in federated systems with a large number of users and resource types, by distributing resource information and payment computation among users organized in a peer-to-peer overlay network. However, our distributed scheme introduces a communication and synchronization overhead. In this section, we analyze the factors that influence the average allocation time and identify the limitations of the proposed distributed auctions scheme.

Figure 4.5 shows a diagram of the proposed three-phase commit distributed auction protocol. We define the average allocation time, T_{alloc}, as the total time taken since a consumer submits a request, *sendRequest*, until it receives the allocation results, *receiveAllocation*, averaged for all successful consumer requests. We identify three components that determine the total allocation time: i) communication time, T_n, which represents the time taken to transmit messages in the network; ii) queue time, T_q, which

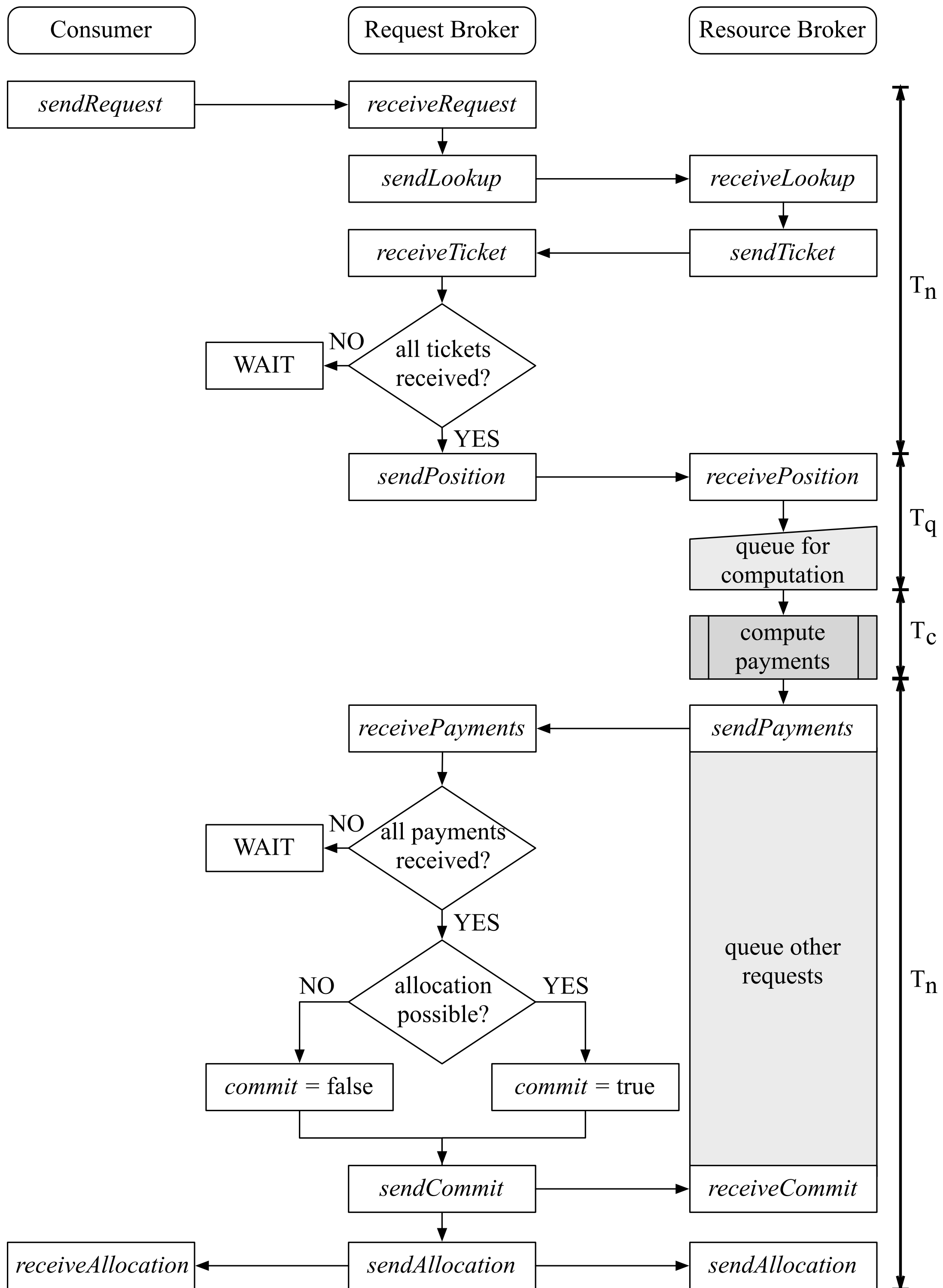


Figure 4.5: Distributed Allocation Diagram

is the delay since a position message is received by the resource broker until payments can be computed; and iii) computational time, T_c, which is the time taken by the pricing algorithm to determine the payments.

$$T_{alloc} = T_n + T_q + T_c \tag{4.1}$$

Communication Time

The communication cost incurred to allocate a consumer request is determined by the number of messages exchanged between the consumer, request broker, and resource broker. These messages are the request message, lookup message, ticket message, position message, payments message, and commit message. We consider a stable overlay network, where routing takes at most $\log N$ steps, where N is the total number of users in the overlay. Thus, the request message is routed from the consumer to the request broker in at most $\log N$ hops. Similarly, the lookup message is routed from the request broker to the resource broker in at most $\log N$ steps. After the resource broker receives the lookup message, it can use the sender address to reply with the ticket number in one hop. Similarly, the position, payments and allocation messages are forwarded in one hop. Considering an average network delay time, d, the total communication time is thus:

$$T_n = d(\log N + \log N + 1 + 1 + 1 + 1) = 2d(\log N + 2) \tag{4.2}$$

Queue Time and Computational Time

To determine the queue time and the computational time, we assume Poisson arrivals for the consumer requests and use queuing theory by considering the resource broker a M/M/1 system [80]. The service time, which consists of the queue time and the computational time, is determined in a M/M/1 system as:

$$T_s = T_q + T_c = \frac{1}{\mu - \lambda}$$

where μ is the average service rate of the resource broker, and λ is the average lookup arrival rate for a resource type. Since the resource broker serializes the requests, the service rate is given by the time since the computation starts, until the commit message is received, when a new lookup can be processed. Accordingly, the service time includes the computation time (T_c), sending the payment message (d), and receiving the commit message (d). Thus, the service rate is:

$$\mu = \frac{1}{T_c + 2d}$$

Accordingly, the total service time is:

$$T_s = \frac{T_c + 2d}{1 - (T_c + 2d)\lambda} \tag{4.3}$$

In the previous chapter, we have determined the runtime complexity of our pricing function is, for a request for one resource type, a polynomial function of the number of items requested and the number of providers. For simplicity, we consider that the computation time T_c is negligible, compared to the average network delay time d, which in the Internet is ranging from several milliseconds to several hundreds of milliseconds. Thus, ignoring the computation time, we can express the total allocation time for a buyer request as:

$$T_{alloc} = 2d(\log N + 2) + \frac{2d}{1 - 2d\lambda} - 2d = 2d(\log N + 1 + \frac{1}{1 - 2d\lambda}) \tag{4.4}$$

From Equation 4.4, the factors that affect the allocation time in the proposed distributed scheme are: i) $\log N$, where N is the size of the overlay network; ii) d, the network delay; and iii) λ, the arrival rate of requests for a resource type. Since the allocation time is not linear with the number of users, the distributed scheme achieves vertical scalability. Similarly, the allocation time does not depend on the number of resource types

in the request. This means that the distributed scheme achieves horizontal scalability. However, the network delay is a factor that influences the allocation time because of the synchronization used by the proposed three-phase commit protocol. This overhead can degrade the performance of the allocation system especially when the arrival rate of requests for a resource type is very high. We study the limitations due to this overhead using an experimental analysis of the performance degradation with large network delays and under high arrival rates in Chapter 6.

In summary, the main limitation of the distributed scheme is due to network delay. In our distributed pricing scheme, all consumer requests are routed to the request broker, which then routes requests for resource types to the resource brokers. For consumer requests containing multiple resource types, this allows the winner determination and payment computation for each resource type in parallel, resulting in lower allocation time. However, for requests containing a small number of resource types, the allocation time may increase when compared to a centralized market-maker. This is because of the time needed to route the request in the overlay network. Moreover, the synchronization mechanism used for concurrency and to ensure the strategy-proof property of our scheme adds an overhead to pricing. In our experimental evaluation, presented in Chapter 6, we look more closely at the size of the overhead and the impact on the allocation time for consumer requests for a small number of resource types.

4.5 Summary

In this chapter we have introduced a distributed auctions mechanism that eliminates the centralized market-maker or auctioneer by distributing resource information and pricing computation among users, organized in a peer-to-peer network. Having a distributed resource market allows for greater scalability, both when having a large number of users, i.e. horizontal scalability, and when having consumer requests with multiple resource

types, i.e. vertical scalability. To solve concurrency issues that may lead to deadlocks, we propose a three-phase commit protocol that makes use of Lamport's logical clocks for synchronization. The main limitation of this approach is the overhead added to the allocation by the synchronization mechanism used to preserve the economic properties of the pricing scheme, which introduces delays due to the network speed.

Our main contribution is an auction mechanism that leverages the distributed hash table in a peer-to-peer overlay network to store resource information, which is discovered using the peer-to-peer lookup mechanism, when computing payments for an allocation. Providers publish resource information in the DHT using the hash of the resource type as the key. Similarly, consumers send requests to a user in the overlay using a random key. In addition to being a provider and consumer, any user in the overlay network can play two additional roles to help perform allocations in the system. Resource brokers are users responsible for a resource type, i.e. the user node identifier is "closest" to the hash of a resource type identifier. They receive publish messages and maintain the list of available resources for the respective resource type. Request brokers are the users with the node identifier "closest" to a request key. They send parallel lookup messages for each resource type and receive provider payments from resource brokers. Request brokers inform the consumers and resource brokers inform the providers about payments and allocation.

Chapter 5

Federated Cloud Prototype

Cloud computing is an emerging platform that promises to achieve the long-envisioned dream of computing as utility. Currently, cloud consumers can use resources and services from public clouds over the Internet at pay-per-use price rates. On the other hand, private clouds are built to offer similar resources and services as public clouds, in an enterprise. Having the means for a private cloud to temporarily use resource from a public cloud as a part of an elastic resource capacity strategy can be critical in scaling enterprise cloud resources [10, 11, 53]. This is achieved by *federated clouds*, a recent paradigm that aims to integrate private and public clouds to increase elasticity and reliability [74, 85, 100]. In a federated cloud market, dynamic pricing sets resource payments according to the forces of demand and supply. However, cloud providers and consumers are rational, self-interested parties, trying to maximize their own benefit [14]. Accordingly, both providers and consumers should not be trusted to reveal their truthful valuations for the allocation of cloud resources [123]. Current federated clouds, similar to standalone cloud providers, use fixed pricing, which does not take into account user rationality [74]. In this chapter we present SkyBoxz, an integrated platform for the management of multiple public and private clouds that uses our proposed dynamic pricing mechanism to allocate cloud consumer requests with multiple resource types across different clouds.

5.1 Issues and Objectives

Cloud computing is a platform that provides users with elastic capacity without upfront costs. Federated clouds are formed by combining private and public clouds to provide users with resizable and elastic resource capacities [10]. *Public clouds* are available to all users, while *private clouds* use a similar infrastructure to provide services for users within an organization. Cloud infrastructure services, also known as *Infrastructure as a Service* (IaaS), deliver computer infrastructure using a virtualized environment to cloud consumers over the Internet. In IaaS, providers typically use fixed pricing, such that strategic user behavior is limited. Examples of major IaaS providers include Amazon and Microsoft, which provide several types of resource types through their services, Elastic Compute Cloud (EC2) [1], and Windows Azure Compute [2], respectively. A resource type, called an instance type, is a virtualized environment that approximates a certain hardware configuration. Currently, cloud computing usage is increasing both in *breadth*, i.e. the number of resource types offered, and in *depth*, i.e. the number of providers [20]. With an increasing number of cloud users, it is expected that more providers will offer similar services. Important issues when providing resources and services across clouds are:

1. *Interoperability*. With interoperability, a federated cloud give users a choice of the same service across clouds to improve reliability and elasticity [65]. Federated clouds need to integrate resources from different cloud providers such that access is transparent to the users.
2. *User rationality*. In a federated cloud with many consumers and providers, rationality is an essential issue as both consumers and providers can strategize to improve their welfare [14, 94, 31]. A federated cloud needs to provide incentives such that both providers and consumers declare their truthful valuations for the resources.

3. *Multiple instance types per request*. A federated cloud needs to provide resources from different clouds without the consumer having to manually integrate the resources. Although there are efforts towards interoperability of different cloud computing middleware [76, 75, 106], existing federated clouds such as SpotCloud [112], ComputeNext [33], and ScaleUp [104] allow consumer requests for only one resource type.
4. *Dynamic pricing*. Fixed pricing, currently used by many cloud providers, is not efficient as it does not adapt to the changes in demand and supply [74]. For example, when demand becomes lower than supply, providers can decrease the price to incentivize consumers to use more resources. This issue becomes more critical in a federated cloud, where demand and supply are more unpredictable due to a larger number of providers and consumers.

In contrast to resource sharing systems used in research and academic communities where users are voluntary, cloud computing has been put into commercial use and its economic model is based on pricing. Previous unsuccessful cloud computing attempts such as Intel Computing Services required users to negotiate written contracts and pricing. Currently, online banking and currency transfer technologies allow cloud providers to use fixed pricing, with consumer payments made online using a credit-card. To address dynamic pricing, Amazon introduced spot pricing, an allocation scheme that sets the resource price according to consumer demand. Spot pricing is similar to the uniform price auction, where consumers bid the maximum price they are willing to pay for the resource, also called the reserved price, but pay the spot price, usually determined as the lowest winning bid. If the supply can be adjusted, such as the case of a single provider, it was shown that this type of auction is truthful and can prevent consumer collusion [35]. However, in a federated cloud, supply is distributed among multiple resource providers, and spot pricing is not incentive-compatible. Thus, a federated cloud where users are rational needs a strategy-proof pricing scheme to allocate resources.

We address these issues in SkyBoxz, a federated cloud platform that provides interoperability between different public and private clouds, and allocates consumer requests with multiple resource types using our proposed strategy-proof, dynamic pricing mechanism. Our objective is to provide cloud infrastructure resources to consumers in a federated cloud where users are rational. A user can be an individual, a group, or an organization, depending on the application context. The next section presents the architecture of SkyBoxz, while the subsequent section details our prototype implementation.

5.2 Proposed Federated Cloud Architecture

SkyBoxz [111] is a platform for managing resources in a federated cloud with multiple use cases. For cloud providers and consumers, SkyBoxz provides a holistic approach to managing and monitoring instances on multiple clouds (Infrastructure as a Service). For consumers, SkyBoxz provides a service platform to deliver a consumer request with multiple resource types - virtual machine instances or services - in a federated or hybrid cloud (Platform as a Service). SkyBoxz uses open-source software components and can aggregate cloud instances managed by several private cloud middleware, such as Eucalyptus [76] and OpenNebula [75], and public cloud providers, such as Amazon EC2 [5].

An overview of the SkyBoxz architecture is shown in Figure 5.1. The main components of the proposed platform are: the management interface, the pricing mechanism, the persistent database, and the cloud library. Virtual private networking (VPN) and shared storage offers interoperability, while consumer application platforms can run on top of SkyBoxz using the infrastructure provided. SkyBoxz uses web services to connect to public clouds, and to existing cloud middleware. Both cloud resource providers and consumers interact with SkyBoxz using a web-based *management interface*, which provide two advantages: i) is platform independent, and ii) it allows consumers ac-

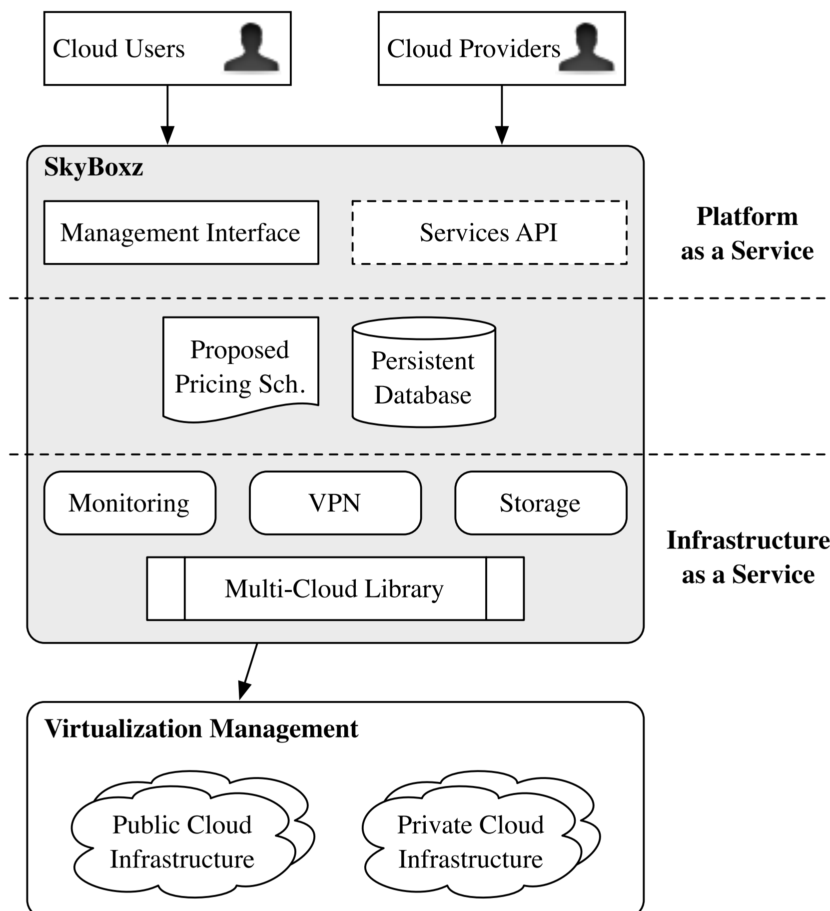


Figure 5.1: SkyBoxz Architecture

cess to multiple clouds without having to install additional software components on their system. Using the management interface, resource providers can add new clouds to SkyBoxz, and configure, for each cloud, the resource types, number of items provided, and the price per item. Resource types in SkyBoxz represent different virtual machine instance configurations, where the cloud provider can specify the number of cores available, the memory size, and the disk space for the instance. For consumers, the web interface allows the creation, monitoring, and termination of a request with multiple resource types. A consumer specifies the number of resource types in the request, the total price and, for each resource type, the virtual machine configuration with the number of cores, the amount of memory, and disk space required.

SkyBoxz uses the *persistent database* to store its information, such as user accounts, cloud credentials, and resource type details. Both consumer and provider information is stored in the database, including the total amount of virtual currency available. Provider information contains the cloud accounts and credentials, such as connection strings and certificates, the available resource types, and the virtual images available to consumers. Using permissions set by the providers, the database stores a map between the SkyBoxz consumer accounts and the cloud resources available for each cloud. Moreover, SkyBoxz allows providers to configure the access to their public or private clouds using permissions that can be shared by a group of SkyBoxz consumers. By using different account security certificates, SkyBoxz consumers are uniquely identified to each cloud provider, even when using shared cloud accounts. The persistent database also stores *monitoring* information from the running instances, which can be provided to consumers or providers.

The information in the database is periodically updated using the *multi-cloud library* component, which is responsible for the communication protocols with different cloud middleware from private and public clouds. Currently, SkyBoxz supports Eucalyptus [41] and OpenNebula [86], two widely-used cloud middleware, and the Amazon EC2 public cloud. However, the cloud library is designed to be modular, such that other types of cloud middleware or public clouds can be added in the future with ease.

Pricing and allocation is done in SkyBoxz using our *proposed* strategy-proof *pricing scheme*. Using the declared information about each resource type submitted by providers, SkyBoxz allocates each consumer request using a first-come-first-serve strategy, and computes the payments using the provider and consumer payment functions introduced in Chapter 3. Since the pricing scheme is strategy-proof, both providers and consumers are incentivized to declare reserved prices according to their valuation. Currently, SkyBoxz runs a centralized implementation of the pricing algorithm, which is more efficient when the number of resource types in a consumer request is small.

SkyBoxz is used internally in School of Computing, and by collaborators from the Faculty of Computer Science and Engineering, Ho Chi Minh University of Technology, and the Shanghai Advanced Research Institute. As the number of users and resource types provided is increased, we will consider a distributed implementation of the pricing scheme.

One of the issues in a federated cloud infrastructure is interoperability. With interoperability, our dynamic pricing scheme can allocate a consumer request on any cloud, based on demand and supply. SkyBoxz achieves interoperability by using compatible virtual machine images with the same base software stack across all clouds. The user persistent *storage* module is decoupled from the virtual machine storage, such that instances from all providers offer the same capabilities when acquired with the same virtual machine image. In addition, SkyBoxz allows instances from the same request to be allocated to different clouds. To handle communication between clouds with different network configurations, SkyBoxz creates a virtual private network (*VPN*) between the instances acquired by the same user.

5.3 Prototype Implementation

SkyBoxz is a platform that offers infrastructure services by allowing consumers to acquire and use cloud virtual machine instances directly, and platform services by allowing the development of custom applications or frameworks that access SkyBoxz to acquire resources without the user interaction. In this section, we present the implementation of the prototype platform and an example application built on top of SkyBoxz.

5.3.1 SkyBoxz Modules

The SkyBoxz management interface is a web application built on top of Drupal v.6.16 [38], an open-source content management system and application framework. Drupal

allows an extensible back-end and has built-in user registration and management, web layout customization, forums, statistics, logging, and administration. Information about the federated cloud is stored in a MySQL v.5.1 database, which is updated using shell scripts executed periodically by a time-based job scheduler. The cloud library and pricing algorithm are implemented in PHP, a scripting language that allows execution both from the job scheduler shell scripts, and from Drupal scripts. The cloud library consists of framework components that implement different API calls for the middleware and public clouds accessible from SkyBoxz. Currently, communication with clouds using the Eucalyptus middleware is done using the restricted EC2 API set published by the

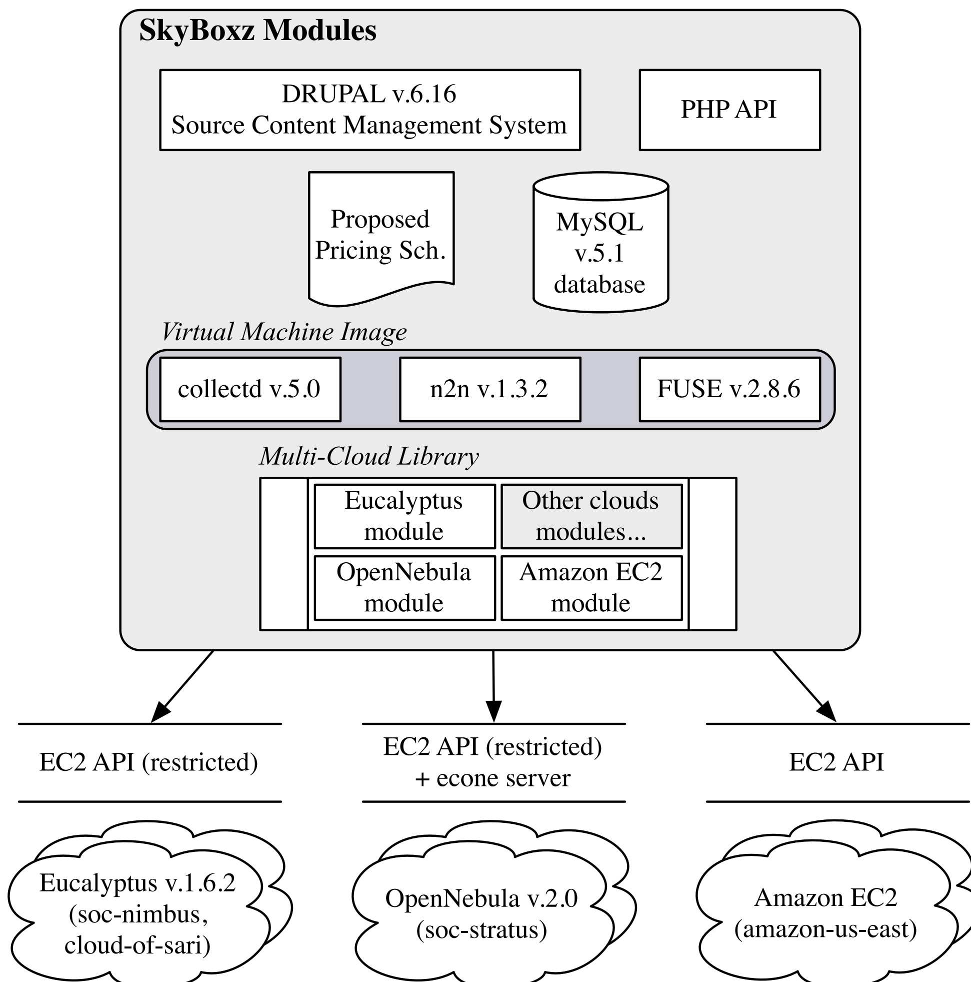


Figure 5.2: SkyBoxz Implementation

Eucalyptus controller node. Similarly, clouds using the OpenNebula middleware are accessed using the EC2 restricted EC2 API, which is interpreted by an additional server which runs in the OpenNebula cloud. The Amazon EC2 public cloud is accessed using the full set of EC2 API calls. A diagram containing the implementation modules is presented in Figure 5.2.

There are currently seven cloud providers available in SkyBoxz, both public and private. The SkyBoxz public clouds are four Amazon EC2 regions: *us-east* in Virginia, USA; *us-west* in California, USA; *eu-west* in Ireland; and *ap-southeast* in Singapore. Private clouds are *soc-nimbus*, from School of Computing, NUS, using the Eucalyptus v.1.6.2 middleware; *soc-stratus* from School of Computing, NUS, using OpenNebula v.2.0.b1; and *cloud-of-sari* from Shanghai Advanced Research Institute, China, using Eucalyptus v.2.0. Details about the configuration of Eucalyptus and OpenNebula middleware are available in Appendix C.

The communication between SkyBoxz and existing providers is using HTTPS requests according to a set of APIs, provided by Amazon EC2. Eucalyptus implements a subset of the EC2 APIs, while interaction with OpenNebula is done using an access point provided by an addition server, `econe`, which translates a subset of the EC2 API to OpenNebula. The EC2 API actions are similar to remote procedure calls, with request and response message pairs. Requests are signed by the cloud consumer using a secret authentication key. A sample request for a small instance type and the response is presented in Figure 5.3. Using the EC2 restricted API, SkyBoxz can create and terminate instances, and query the available images. Additional functionality, available in Eucalyptus clouds, allows the providers to configure five instance types, by setting the number of cores, memory and disk space.

The information in the SkyBoxz database is described by the schema in Figure 5.4. The `providers` table stores the connection strings and administrator credentials, if available, for the different cloud providers.

Example SkyBoxz POST Request:

```
request=1&user=marianmi&ntypes=1&type=ec2.m1.small&items=1&image=linux&
price=10&pricing=dynamic
```

EC2 API Request:

```
https://ec2.amazonaws.com/?Action=RunInstances&ImageId=ami-60a54009&
InstanceType=m1.small&MaxCount=1&MinCount=1&
Placement.AvailabilityZone=us-east-1b&Monitoring.Enabled=true&AuthParams
```

EC2 Response:


```
<RunInstancesResponse xmlns="http://ec2.amazonaws.com/doc/2009-08-15/">
  <reservationId>r-47a5402e</reservationId>
  <ownerId>AIDADH4IGTRXXKCD</ownerId>
  <groupSet>
    <item>
      <groupId>default</groupId>
    </item>
  </groupSet>
  <instancesSet>
    <item>
      <instanceId>i-2ba64342</instanceId>
      <imageId>ami-60a54009</imageId>
      <instanceState>
        <code>0</code>
        <name>pending</name>
      </instanceState>
      <privateDnsName></privateDnsName>
      <dnsName></dnsName>
      <keyName>example-key-name</keyName>
      <amiLaunchIndex>0</amiLaunchIndex>
      <instanceType>m1.small</instanceType>
      <launchTime>2011-08-07T11:51:50.000Z</launchTime>
      <placement>
        <availabilityZone>us-east-1b</availabilityZone>
      </placement>
      <monitoring>
        <enabled>true</enabled>
      </monitoring>
    </item>
  </instancesSet>
</RunInstancesResponse>
```


Figure 5.3: Example EC2 API Request and Response

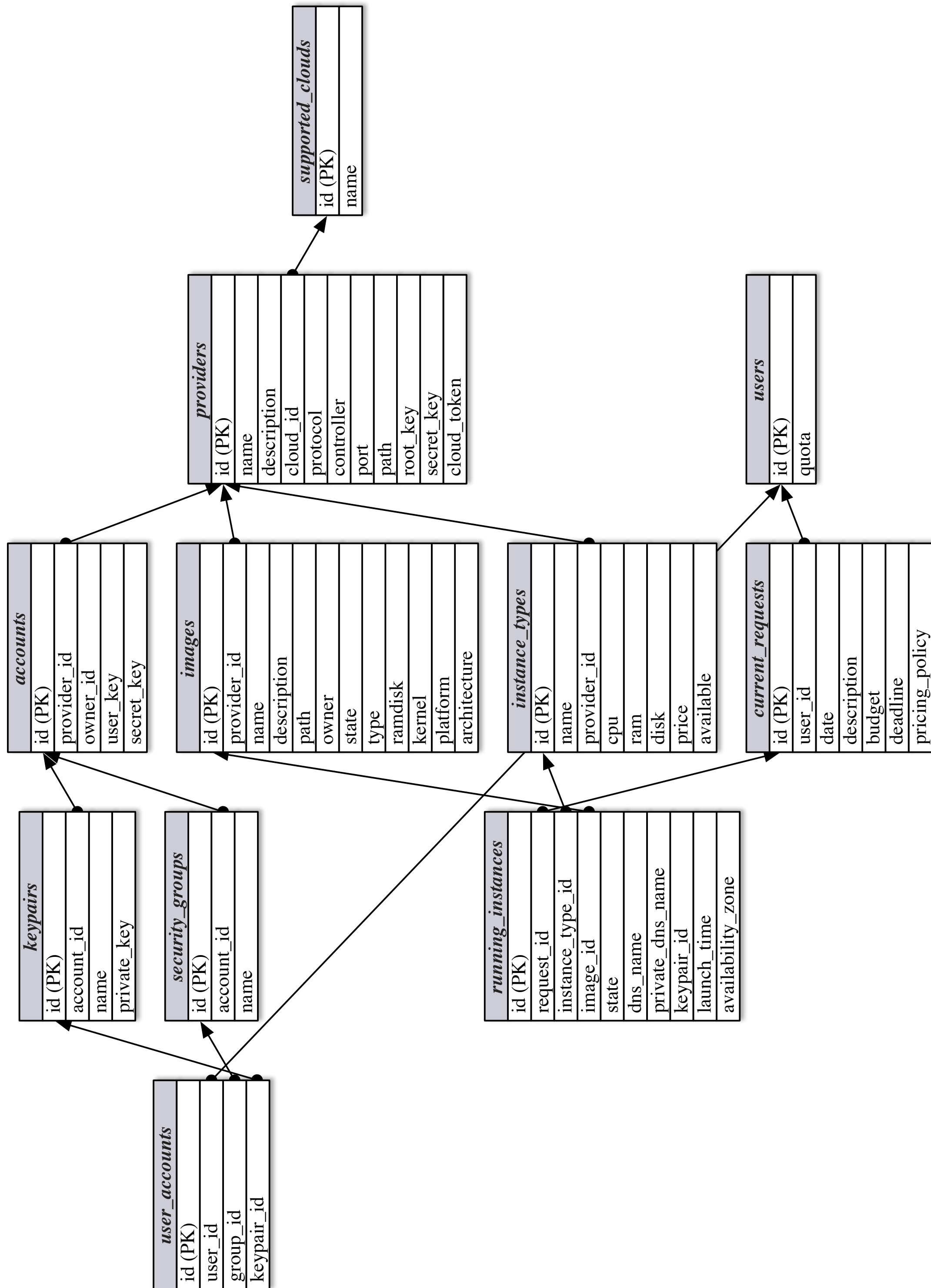


Figure 5.4: SkyBoxz Database

Each cloud added to SkyBoxz is described by the following information: name, description, middleware (where the list of supported middleware is available in the table `supported_clouds`), query protocol (which can be `http` or `https`), IP address of the cloud controller or access point, access port (usually the `http` or `https` port), path to the middleware scripts. Optionally, when using the Eucalyptus middleware, cloud providers can specify the access id and encryption key for administrator access, which gives the cloud providers the possibility of remotely configuring resource types though SkyBoxz. Cloud accounts are added to SkyBoxz to the table `accounts`, by specifying the user access id and the encryption key. The cloud accounts can be shared between multiple SkyBoxz users. Each cloud has different resource types, which are stored in the table `instance_types`. This table contains information such as number of cores, size of memory, disk space, etc. Lastly, images available for each cloud provider are stored in the table `images`. Each image is described by the platform it runs, e.g. `Linux` or `Windows`, the architecture, `32 bit` or `64 bit`, path to the image, kernel and ramdisk used, etc. The current requests submitted to SkyBoxz are stored in the table `current_requests`, which is linked to the table `running_instances` that contains the information about the instance type, image, state and the cloud on which each instance in the request is allocated to. The relationship between the cloud accounts, configured by the cloud providers, and the SkyBoxz accounts, is determined by several tables in the database, `user_accounts`, `keypairs`, and `security_groups`, as follows. Each SkyBoxz account is added to a security group defined by the cloud provider, and has a unique private key, linked together using the table `user_accounts`. This table is linked to the table `users`, which stores the amount of currency for a SkyBoxz user. The user id is the same as the Drupal id, to ease the user management process by integrating an existing Drupal module.

Virtual machines images available through SkyBoxz have, for each platform and architecture, a similar software stack installed to allow interoperability across different providers. To facilitate monitoring, the `collectd` v.5.0.1 daemon runs on each instance and transfers information periodically about the instance load to Drupal, where it can be accessed by the users. In addition, all instances allocated to a consumer request are added to a `n2n` v.1.3.2 virtual private network (VPN) which is set up for each consumer. `n2n` [78] is an open-source VPN application that uses a peer-to-peer architecture for VPN membership and routing. SkyBoxz manages the `n2n` supernode, while the `n2n` client is installed in all images and available when instances are run. The role of the supernode is to register clients and route packages for instances that are not able to communicate directly. Each SkyBoxz user has his own VPN network, such that all their running instances can access each-other. Lastly, the `FUSE` v.2.8.6 daemon is installed in all virtual machine images, to allow users to attach several types of network storage. `FUSE` (Filesystem in User Space) is a kernel module that allows virtual file systems, and currently supports, among others, Amazon S3, WebDAV and SFTP.

5.3.2 Service Platform

SkyBoxz is built as an extensible framework that can be used by other platforms and applications. Currently, other Drupal modules and web applications can be configured to run on top of SkyBoxz, and access resources from the federated cloud. As an example application that uses SkyBoxz, we present in this section SNAP (Snapshots of Program Execution on Different Computation Platforms), a project jointly developed by Massachusetts Institute of Technology and National University of Singapore, funded by the Singapore-MIT Alliance for Research and Technology (SMART) Innovation Grant.

SNAP is a program testing platform for numerical instability, where developers can automate the testing and verification of numerical applications across a wide range of hardware and software platforms. Application developers can use SNAP to acceler-

ate the manual task of program testing when they develop, migrate, or evaluate new or existing applications. Tests can be created to run on different hardware and software platforms. The applications that have been tested with SNAP include sequential and parallel programs, executed in virtual machines. Currently, SNAP offers several software platforms, such as `gcc` and `Scilab` for sequential programs, and `OpenMP` and `MPI` for parallel programs. Program testing is performed by uploading the application code, selecting the hardware and software configuration, and submitting the test from SNAP. Comparison of the program results on different configurations can be performed to determine whether the tested program suffers from numerical instability.

Figure 5.5 shows the architecture of SNAP. Using the PHP API exported by our implementation, SNAP is able to send requests for IaaS resources to SkyBoxz. Hardware platforms selected in SNAP are requests to SkyBoxz, which maps them to different cloud providers. When starting a test, the application developer may choose from different sequential or parallel configurations. These configurations are matched by SkyBoxz

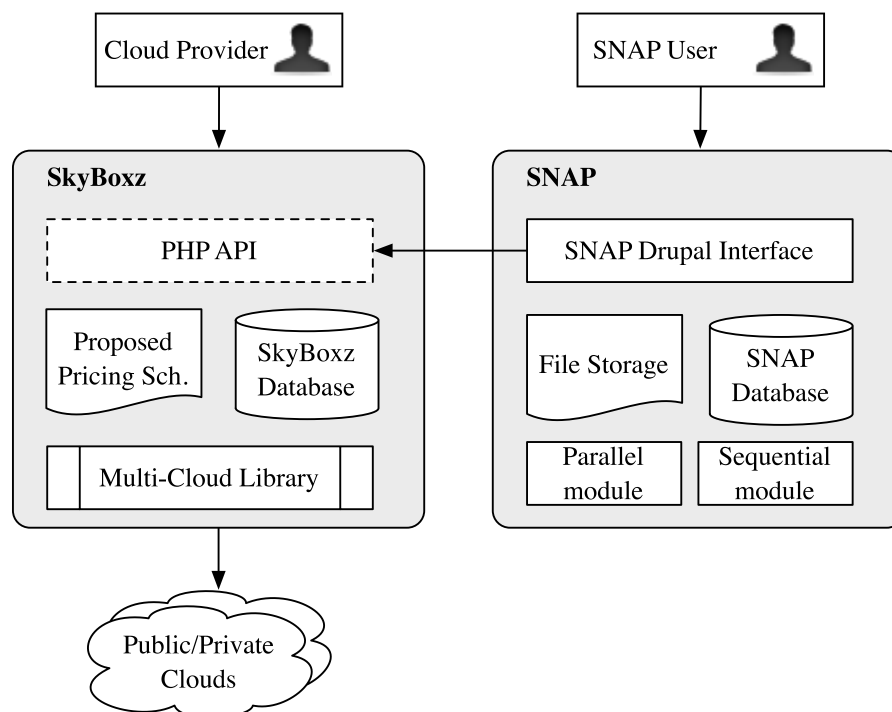


Figure 5.5: SNAP Architecture

to the corresponding instance types and virtual machine images from the database. The files required are uploaded to SNAP, which is responsible to distribute them to the machine instances. SNAP maintains its own database for user, results, and logging, using a relationship table to map SNAP accounts with existing SkyBoxz accounts. Pricing and allocation is handled by SkyBoxz, which allocates SNAP requests using our dynamic pricing scheme. Payments for resources used when testing programs in SNAP are managed by SkyBoxz using the account associated with the SNAP user.

5.4 Summary

Cloud computing is an emerging platform that provides resources on-demand, as a service, over the Internet. The aim of federated clouds is to integrate resources from different cloud providers such that access is transparent to the users. Towards this, we have implemented SkyBoxz, a platform for managing resources in federated clouds where users are rational. In SkyBoxz, cloud providers can add their private or public cloud to a hybrid cloud, and configure the resource types they share. Consumers can submit a request with multiple resource types, with allocations determined using our proposed strategy-proof pricing scheme. Interoperability between different cloud providers is achieved by using virtual machine images with the same software stack, together with virtual private networking and shared storage. SkyBoxz is built as an extensible framework that can be used by other applications. To this extent, we have presented SNAP, a program testing platform for numerical instability that can run on different hardware and software platforms provided by SkyBoxz. Currently, SNAP can access several cloud providers, both public and private, and is used by consumers from Singapore, Vietnam, and China.

Chapter 6

Experimental Evaluation

We have formally proven using mechanism design the economic properties of the proposed pricing mechanism: individual rationality, incentive compatibility, and budget balance. Furthermore, we have theoretically analyzed the computational complexity of the proposed algorithm. In this chapter, we further the study of our pricing mechanism using experimental evaluation. We analyze the trade-off between the economic and computational efficiency, and the impact of allocating multiple resource types in different market conditions, such as under-demand and over-demand. Our theoretical analysis found that the allocation time when using the proposed pricing scheme is not scalable with the number of resource types in a consumer request, and with the number of users. The distributed scheme we proposed tries to address this issue, however the synchronization mechanism and network delays add an additional overhead. In this chapter, we analyze the scalability and determine the overhead cost of the distributed scheme, using an implementation on top of the FreePastry overlay network. Lastly, we study the rationality of the users in a federated cloud composed of several Amazon EC2 regions.

6.1 Experimental Setup

Traditionally, efficiency in computer science is measured using system-centric performance metrics such as the number of completed jobs, and average system utilization. All user applications are equally important and optimizations ignore the user's valuation for resources. Thus, resource allocation is unlikely to deliver the greatest value to the users, especially when resources are limited. In contrast, economic systems measure efficiency with respect to user's valuations for resources (utility). For example, in a Pareto efficient system where economic efficiency is maximized, a user's utility cannot improve without decreasing the utility of another user. In our work, we evaluate the performance of our scheme using several performance metrics, both system-centric and user-centric. Table 6.1 shows an overview of our evaluation and the performance metrics and tools used in our study.

We first study the trade-off between the economic and computational efficiencies by measuring the total user welfare, and the allocation time, respectively. The total

Experiment	Measures	Tools
trade-off analysis	• total welfare • allocation time	• jCase v.0.2
economic efficiency	• succ. consumer requests • user welfare	• auctions simulator • FreePastry v.2.1 simulator
multiple resource types per request	• succ. consumer requests • alloc. provider resources • user welfare	• FreePastry v.2.1 simulator
scalability	• allocation time	• FreePastry v.2.1 simulator • PlanetLab prototype
user rationality	• resource price • user welfare	• MATLAB v.7.12

Table 6.1: Experiments and Tools

user welfare is the sum of the provider and consumer welfare, where the welfare is determined as the difference between the user valuation for that resource and the payment received. Pareto efficiency is achieved by maximizing the total user welfare. The allocation time is measured by the wallclock time taken to complete a request. To determine the trade-off, we compare the economic and computational efficiencies with combinatorial auctions, a pricing mechanism that is known to be Pareto efficient. We consider in our evaluation two types of combinatorial auctions. The first combinatorial auction we consider uses a VCG payment scheme to achieve strategy-proof at the cost of budget-balance. The second combinatorial auction is based on the Threshold scheme proposed by Parkes et al. [88], which tries to achieve budget-balance at the cost of incentive compatibility.

We further the study of the economic efficiency by analyzing the consumer and provider efficiencies. The consumer efficiency is given by the successful consumer requests, while the provider efficiency is given by the allocated provider resources. To better understand our findings, we compare our results with other pricing schemes, namely one-sided auctions and fixed pricing. We selected for comparison traditional one-sided auctions and fixed pricing because they are widely used due to their simplicity and relatively good outcomes. Fixed pricing, for example, is widely used currently by cloud providers such as Amazon [5] and Microsoft [2]. In addition to provider and consumer efficiencies, we also determine the individual user welfare. The user welfare allows us to understand how the economic trade-off is achieved, since it measures individual rationality and the impact of financial incentives over the payments. We are considering three market conditions: balanced market, over-demand, and under-demand, where demand equals supply, demand is higher than supply, and demand is lower than supply, respectively. Lastly, we consider the case of consumer requests with multiple resource types per request under different market conditions.

To evaluate the scalability of our proposed distributed scheme, we study the computational efficiency and determine the average allocation time. As our theoretical evaluation has shown, scalability in a centralized pricing scheme is limited when increasing the number of resource types (vertical scalability) and when increasing the number of users (horizontal scalability). Thus, in our evaluation we compare the vertical and the horizontal scalability of the centralized and the distributed scheme. In addition, we look at the impact of network delay and request arrival rate, which our theoretical analysis found to add an overhead to the allocation. Lastly, we investigate the rationality of existing cloud computing users by comparing the user welfare obtained by the Amazon spot pricing scheme with our scheme. We consider that different region in Amazon EC2 constitute a federated cloud.

Our experimental analysis makes use of several software tools, both open-source and our own implementation. To analyze the trade-off between economic and computational efficiencies, we use the open-source combinatorial auctions simulator jCase v.0.2 [105]. jCase is an open-source combinatorial auctions simulator developed in Java that uses an XML-based input to define simulation scenarios. jCase already has implemented several pricing schemes, such as a VCG-based combinatorial auction, and the Threshold [88] algorithm. To compare with the proposed scheme, we extended jCase to implement the centralized algorithm of our scheme.

To compare the economic efficiency and evaluate the impact of multiple resource types in traditional auctions, we developed a simple discrete-event simulator, which can allocate requests given in an XML file using English auctions and the proposed scheme. This simple auctions simulator was also used to measure the impact of user rationality in English auctions, results which are included in Appendix A.

The majority of our experiments have been performed using an application built on top of FreePastry v.2.1 [90], an open-source DHT overlay network environment. FreePastry offers a discrete-event simulator that is able to run FreePastry applications.

Thus, we were able to both simulate large systems, and validate the results in a prototype deployment on the research network PlanetLab [93]. Our FreePastry application can operate in two different modes. In the centralized mode, the node that starts the FreePastry overlay network has the role of a centralized market-maker, by maintaining the list of provider resources and the queue of consumer requests. All the other nodes in the overlay act as provider or consumers, and exchange messages only with the market-maker. In the distributed mode, any node can become a resource broker or request broker. The workload for the FreePastry application is defined in an XML file. We implemented two pricing schemes for comparison: fixed pricing, and our proposed scheme, both centralized and distributed.

Our study on user rationality uses spot price traces for different instance types in four EC2 regions, over a period of one month. We use MATLAB v.7.12 scripts to analyze the traces and determine the consumer requests and consumer welfare for spot pricing and our proposed scheme.

6.2 Strategy-proof Pricing

In this section, we study the performance of our proposed pricing mechanism. We analyze the computational end economic trade-off by comparing our scheme with combinatorial auctions. Next, evaluate the economic efficiency, namely the number of successful requests of our scheme, and compare to traditional one-sided auctions and fixed pricing. Lastly, we evaluate the economic efficiency with consumer requests with multiple resource types per request in different market conditions. For simplicity, we used a centralized implementation of our pricing mechanism.

6.2.1 Trade-off Analysis

The main objective of our trade-off analysis is to study the economic efficiency lost by achieving strategy-proof and budget balance. To this extent, we measure the total welfare (*Welfare*) obtained by our scheme, and compare to combinatorial auctions, a pricing scheme known to be Pareto-efficient, at the cost of computational efficiency. Thus, in addition to the loss in economic efficiency, we also measure the allocation time (*Alloc. Time*) obtained by the two pricing schemes. We use jCase [105] to compare our scheme with two combinatorial auctions implementations: a pure VCG scheme, and the Threshold algorithm proposed by Parkes et al. [88]. We measure the economic efficiency as the sum of user welfare, consumers and providers. The classical combinatorial auction that uses VCG-based pricing functions for both providers and consumers is not budget-balanced. For completeness, we also measure budget-balance (*BB*), given by the sum of provider and consumer payments. The second combinatorial auctions scheme, based on the Threshold scheme [88], achieves budget balance, but loses the strategy-proof property.

Pricing Mechanism	**Users**	**IC**	**BB ($)**	**Welfare ($)**	**Alloc. Time**
Combinatorial/VCG	20	✓	-1,402	2,470	9.6m
	40	✓	-1,544	6,321	2.5h
	80	✓	-1,557	14,567	67.4h
Combinatorial/Threshold	20	–	5	2,491	9.8m
	40	–	9	6,223	2.5h
	80	–	6	14,384	49.5h
Proposed Scheme	20	✓	0	1,871	1s
	40	✓	0	5,483	3s
	80	✓	0	11,561	5s
	100	✓	0	14,369	7s
	200	✓	0	28,564	20s
	500	✓	0	65,948	1.9m
	1,000	✓	0	132,729	7.7m

Table 6.2: Comparison of Proposed Scheme with Combinatorial Auctions

In our experiments, we assume a balanced market where demand equals supply, modeled using an equal number of providers and consumers *Users* [1]. A user is both a provider and a consumer. Each provider publishes multiple resource types, sampled from a uniform distribution between 1 and 10. Similarly, the number of resource types in a consumer request was sampled from a similar distribution. In each simulation run, we perform 50 allocation rounds and generate a total of $50 * Users$ consumer requests. We vary the number of users from 20 to 80 for the combinatorial auctions, and up to 1,000 for our proposed scheme. Our results are shown in Table 6.2, where *IC* column is used to show the schemes that are incentive compatible. *Alloc. Time* shows the average allocation time for a request in hours (*h*), minutes (*m*), and seconds (*s*). We can see from the results that our pricing scheme achieves lower economic efficiency. For example, for 80 users, the total welfare is $11,561 for our scheme, while combinatorial

[1] although it is not expected for a balanced market to have an equal number of providers and consumers, this model is sufficient for the measurement of total welfare and allocation time.

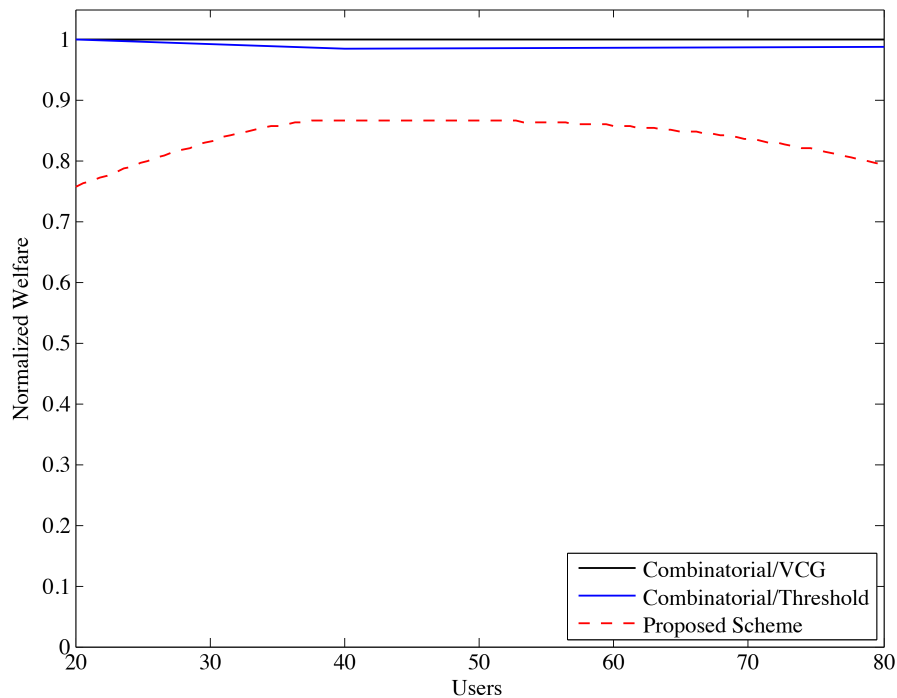


Figure 6.1: Economic Efficiency Loss

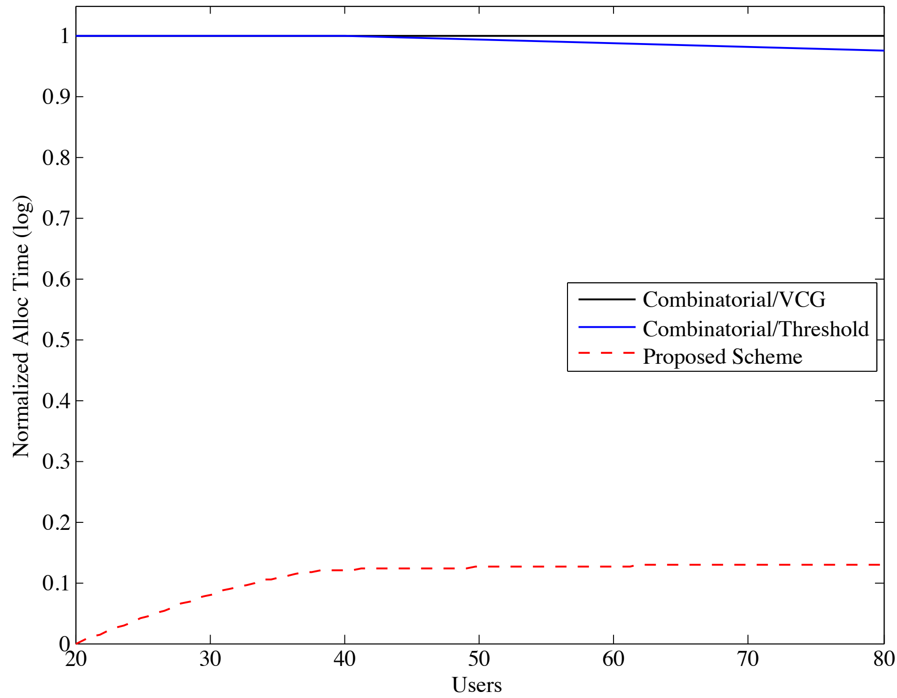


Figure 6.2: Computational Efficiency Gain

auctions achieve $14,567 for the VCG scheme, and $14,384 for the Threshold scheme. We show the economic efficiency loss in Figure 6.1. According to our results, the proposed scheme achieves around 80% from the maximum economic efficiency. However, since winner determination in combinatorial auctions is a NP-complete problem, each experiment with 80 users takes more than 60 hours to execute on a 8-core Intel Xeon, 1.86 GHz machine with 4GB RAM. In contrast, our scheme takes just 5 seconds. Figure 6.2 shows the normalized gain in allocation time of our scheme in a logarithmic scale when having up to 80 users in the simulator. The complete picture of our trade-off can be seen in Figure 6.3. Our scheme sacrifices 20% economic efficiency, however it achieves substantially lower allocation times. In addition, our scheme is strategy-proof and budget-balance. In contrast, the VCG-based combinatorial auctions scheme has a negative budget-balance, which means the market-maker has to supply to additional currency required for payments. On the other hand, the Threshold scheme achieves budget-balance, however it loses the strategy-proof property. Nevertheless, both com-

binatorial auctions schemes are not feasible to implement due to the large allocation times required.

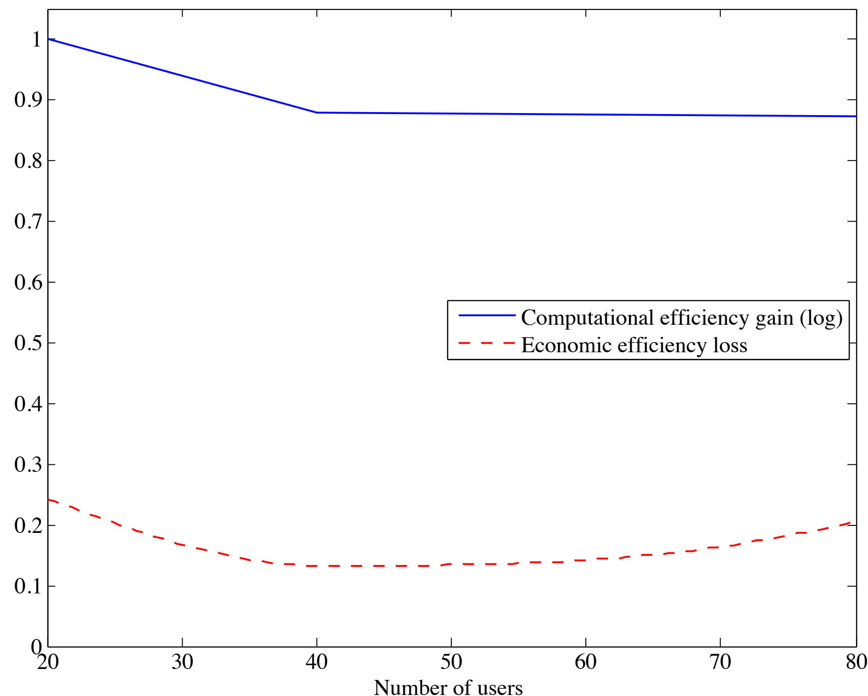


Figure 6.3: Economic Efficiency vs Computational Efficiency

6.2.2 Economic Efficiency

In the previous section, we studied the economic efficiency achieved by our scheme as the total provider and consumer welfare. In this section, our objective is to have a more detailed analysis of the economic efficiency by looking at each element that determines the total welfare, namely the consumer and provider welfare, the number of successful consumer requests (consumer efficiency), and the number of allocated provider resources (provider efficiency). Informally, the total welfare is increased when any of its elements is increased. We use for comparison two pricing schemes that are widely used when allocating consumer requests for one resource type, namely one-sided auctions and fixed pricing.

1. **Comparison with One-sided Auctions**

   Comparison with traditional one-sided (English) auctions is performed using the discrete-event auctions simulator we developed. The simulator has two queues ordered using the First-Come-First-Serve strategy: one for consumer requests and the second for provider published resources. One-sided auctions take place by matching a provider resource with all available requests in the consumer queue. In contrast, our scheme matches a consumer request with all available resources in the provider resource queue. As in a real market, we model three market conditions: *balanced* where resource demand matches supply, *under-demand* where resource demand is less than supply, and *over-demand* where resource demand exceeds supply. Market conditions are modeled by varying the arrival rates of consumer requests and provider published resources. In a *balanced market*, both rates are equal and are sampled from an exponential distribution with a mean of one arrival/minute. For *under-demand* and *over-demand* markets, we set the mean consumer request arrival rate at 0.5 and 2 arrivals per minute, respectively. Table 6.3 compares the results of 10,000 consumer requests under different market scenarios and with different

Price Variation(%)	Succ. Consumer Requests		
	Auctions	**Proposed**	**Increase(%)**
Balanced Market			
10	5,475	6,918	26.4
20	5,454	6,933	27.1
40	5,452	6,903	26.6
Under-Demand			
10	6,637	7,894	18.9
20	6,645	7,904	19.0
40	6,637	7,909	19.2
Over-Demand			
10	3,348	3,906	16.7
20	3,380	3,906	15.6
40	3,364	3,912	16.3

Table 6.3: Price Variation under Different Market Scenarios

price variation. For simplicity, each consumer request consists of one resource type and one item. The standard deviation between the providers and consumers private information is captured in the *Price Variation* column. Accordingly, the private information is generated in the simulator using a uniform distribution between $(100 - PriceVariation)$ and $(100 + PriceVariation)$. A greater price variation means that the difference between consumer reserved prices and provider costs is higher on the average, which in turn leads to a smaller number of possible allocations due to individual rationality.

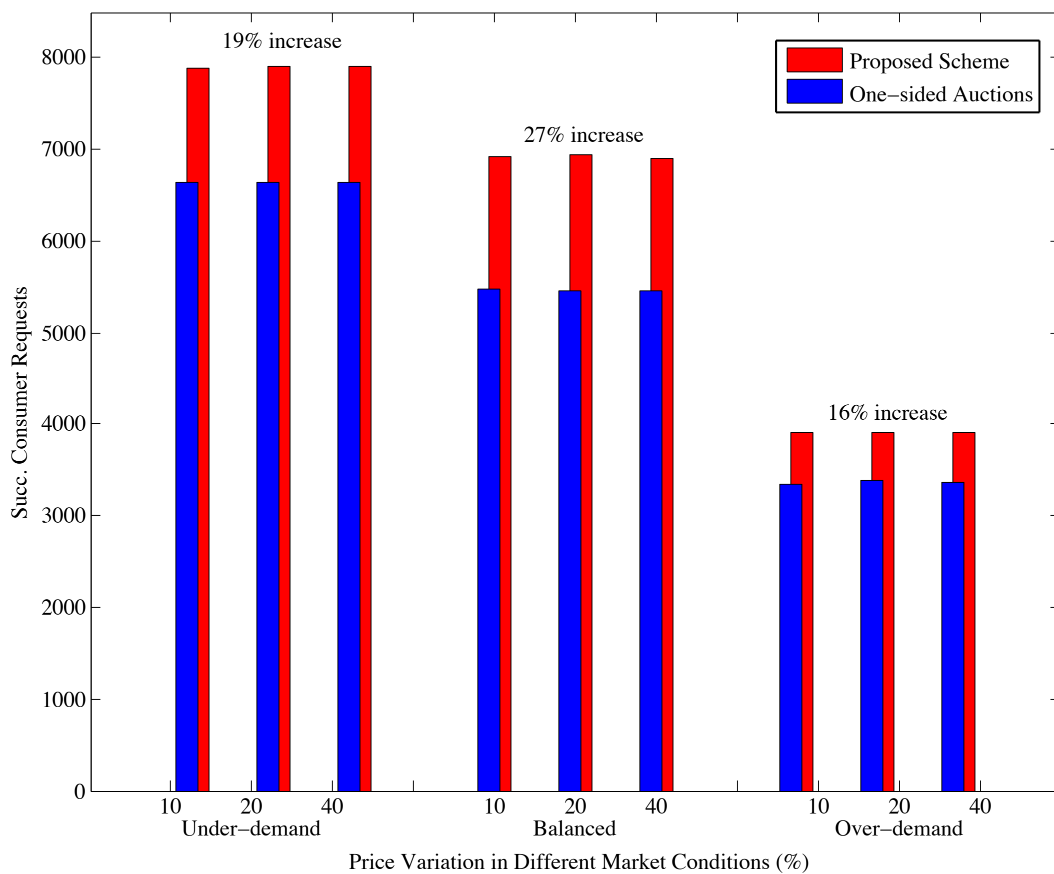


Figure 6.4: Comparison with One-sided Auctions

Figure 6.4 shows that our scheme achieves an overall improvement of 16% in terms of the total number of successful consumer requests over traditional auctions when the demand exceeds supply, and over 25% in a balanced market.

2. **Comparison with Fixed Pricing**

In this experiment, we compare the individual user welfare achieved by our scheme with the welfare achieved by the fixed pricing scheme, currently used by many cloud providers. The *user welfare* is determined by the difference between the user utility and payment. In the proposed scheme, the user utility is the same as the published price, since both providers and consumers are shown to be truthful, and thus $t_i = t_i^d$. In the case of fixed pricing, we similarly consider a truthful consumer, i.e. the published request price represents the consumer utility. However, we do not make the same assumption about providers, which have a fixed resource price that may differ from the provider utility. Thus, in our evaluation, we compare only the average consumer welfare when using fixed and dynamic pricing, respectively.

We consider a balanced market, where demand and supply are equal. In the FreePastry simulator, the centralized market-maker receives consumer requests and provider resource publish events with equal probability, and with an inter-arrival time of one second. These events are uniformly distributed among 10,000 FreePastry nodes, and contain a number of resource types uniformly distributed between 1 and 3, chosen randomly from a total of 5 resource types. The number of items for each resource type is generated according to an exponential distribution with a mean of 10. Providers have the fixed price of 100 per item. When using our pricing scheme,

Metric	Price Variation (%)	Pricing Scheme	
		Fixed	*Proposed*
Consumer Welfare ($)	10	3.5	4.6
	20	7.4	9.3
	50	18.8	23.3
Succ. Consumer Requests (%)	10	47.7	62.5
	20	48.8	62.2
	50	49.5	62.1

Table 6.4: Comparison with Fixed Pricing

we vary the provider price from 100 by a maximum of 10%, 20% and 50%, i.e. the price is generated according to a uniform distribution between 90 and 110, 80 and 120, and 50 and 150, respectively. Consumer reserved price is varied according to the same percentage, shown in Table 6.4 in the column *Price Variation*. The simulation runs for 600,000 events, which, for an arrival rate of one second, give a total simulation time of approximately seven days. To reduce sampling error, we run our experiments three times and compute the average. Our results show that our proposed dynamic scheme increases both the consumer welfare and the percentage of successful consumer requests, shown in the *Succ. Consumer Requests* column. Given that the mean consumer utility is 100, and a *theoretical*[2] maximum welfare for an item is achieved with the minimum payment, i.e. $(100 - PriceVariation)$, we consider that the maximum welfare equals the price variation. Accordingly, using the proposed dynamic pricing scheme in a balanced market, the consumer welfare is increased by approximately 10%, while at the same time the number of successful consumer requests is also increased by more than 25%.

We summarize the results for both one-sided auctions and fixed pricing in Figure 6.5, where we include the case of under-demand and over-demand for fixed-pricing. Our scheme performs consistently better than one-sided auctions, and achieves a better consumer efficiency than fixed pricing in the case of under-demand and balanced market. When demand is higher than supply, our dynamic scheme will compute higher provider payments according to the market price, and thus the number of successful requests will be lower than when using fixed pricing.

[2]The actual maximum welfare can be computed using a NP-complete algorithm, and is smaller than the *theoretical* welfare.

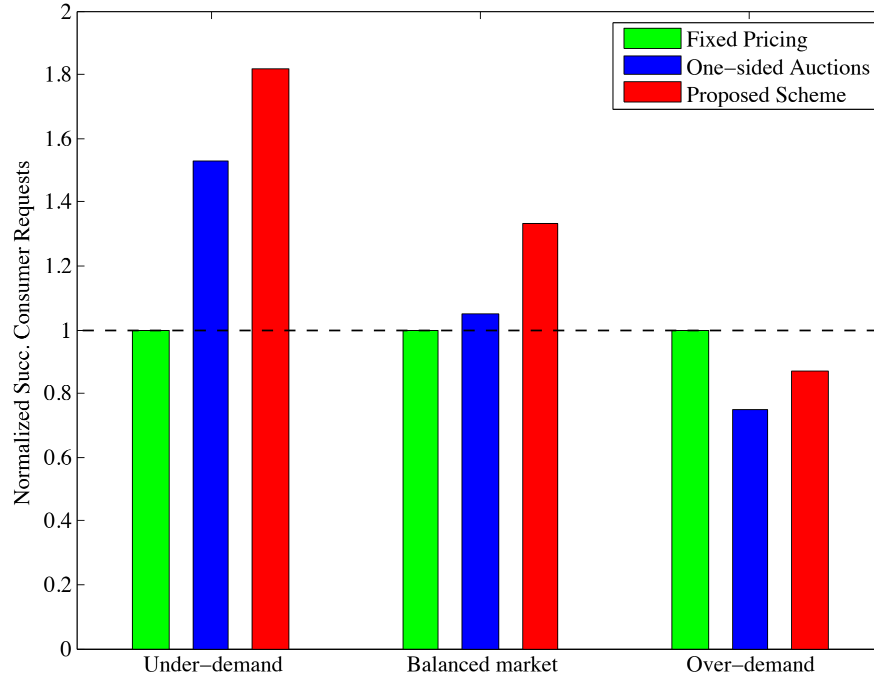


Figure 6.5: Successful Consumer Requests in Different Market Conditions

6.2.3 Multiple Resource Types per Request

In contrast to one-sided auctions and fixed pricing, where users have to manually aggregate resources, our proposed pricing scheme can allocate a consumer request with multiple resource types. In the next experiments, our objective is to study the economic efficiency when having requests with multiple resource types under different market conditions, and compare our results with one-sided auctions and fixed pricing.

In the first experiment, we study the impact of the multiple resource types on the number of successful consumer requests. For this, we compare our scheme with one-sided (English) auctions, where we consider a request successfully allocated only when the consumer wins all the one-sided auctions for all of the resource types in the request at the same time. We are using a similar setup and the same discrete-event auctions simulator as in the previous section, where we vary the number of resource types in both consumer requests and provider publish between 1 and 16. Figure 6.6 shows that when

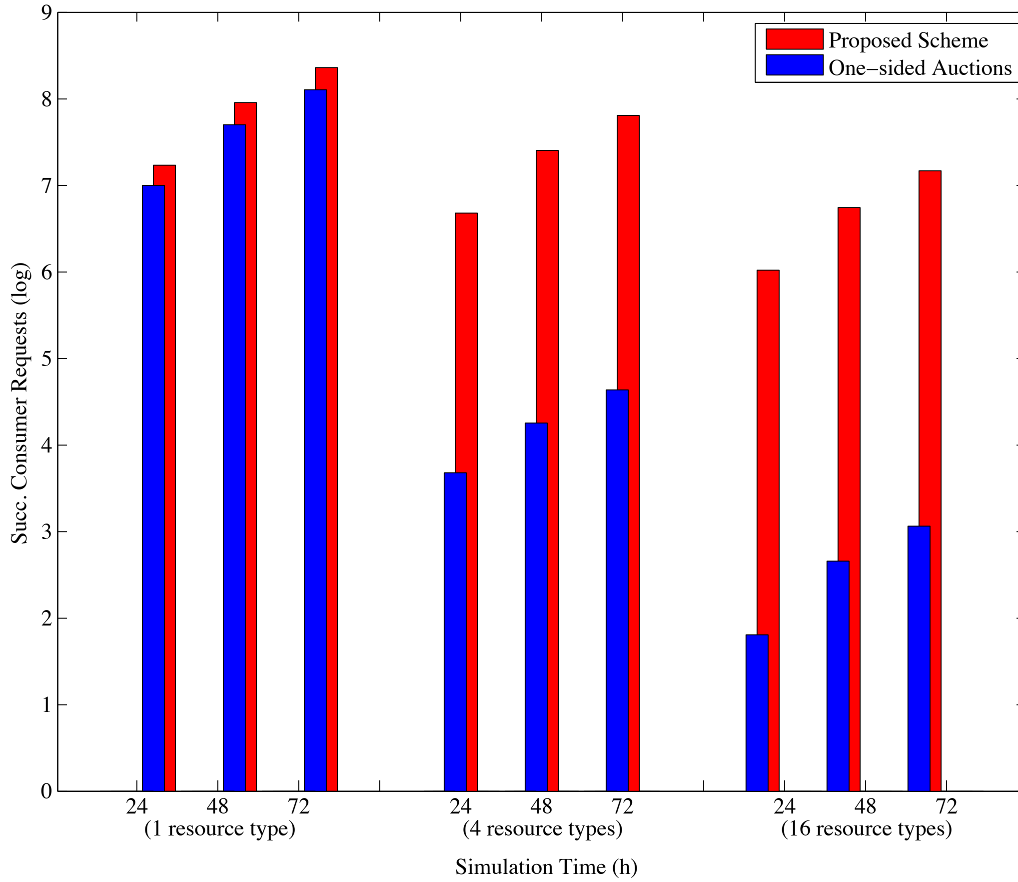


Figure 6.6: Varying the Number of Resource Types

performing allocations for requests with multiple resource types, our scheme performs much better than traditional auctions as the number of resource types increases.

In the next experiment, we compare our scheme with fixed pricing. We vary the number of resource types in a request from 5 to 20 with a price variation of 20%. We also consider different market conditions. In the case of a balanced market, the probability for a consumer request is set to 50%, while for under-demand is 33%, and for over-demand is 66%, respectively. Event arrival rate is set to $1s$, where an event can be a consumer request or provider publish. We measure the average consumer welfare, the percent of successful consumer requests (*Succ. Consumer*) and the percent of allocated provider resources (*Alloc. Provider*). Table 6.5 presents our results. In the case of fixed pricing, the percentage of successful consumer requests is close to 50% for

Resource Types	**Succ. Consumer (%)**		**Alloc. Provider (%)**		**Consumer Welfare ($)**	
	Fixed	*Proposed*	*Fixed*	*Proposed*	*Fixed*	*Proposed*
Under-Demand						
5	48.4	82.3	24.1	41.8	6.2	10.9
10	47.4	86.3	23.4	44.1	4.7	9.4
20	46.5	89.7	22.1	46.4	3.3	8.1
Balanced Market						
5	48.2	62.4	47.5	61.0	6.2	7.9
10	47.1	62.9	46.5	62.5	4.7	6.3
20	46.2	63.3	46.0	64.0	3.4	4.9
Over-Demand						
5	48.2	42.1	95.4	75.5	6.2	6.1
10	47.4	41.4	93.1	74.2	4.7	4.8
20	46.2	40.4	91.7	73.0	3.4	3.7

Table 6.5: Economic Efficiency with Multiple Resource Types

all market conditions, since the consumer item price is uniformly distributed with the mean equal to the provider item price. However, the percentage of successful consumer requests decreases when the number of resource types increases, since the number of providers that are allocated to satisfy a request also increases.

In contrast, in the case of our proposed dynamic scheme, the percentage of successful consumer requests varies under different market conditions, according to the forces of demand and supply. As in Figure 6.7, when demand is lower than supply, the percentage of successful consumer requests is higher than in the case of a balanced market, while for over-demand the percentage decreases further. Using the proposed auction scheme achieves a higher percentage of successful consumer requests and provider allocated resources, in the case of under-demand and balanced market. When demand is higher than supply and the number of resource types in a request increases, there is premise for monopolistic providers [91] when there are not enough providers in the market to compute payments using the VCG-based payment function. In this case, the proposed scheme cannot allocate provider resources while maintaining the strategy-proof property, and an alternative pricing function for the monopoly situation needs to

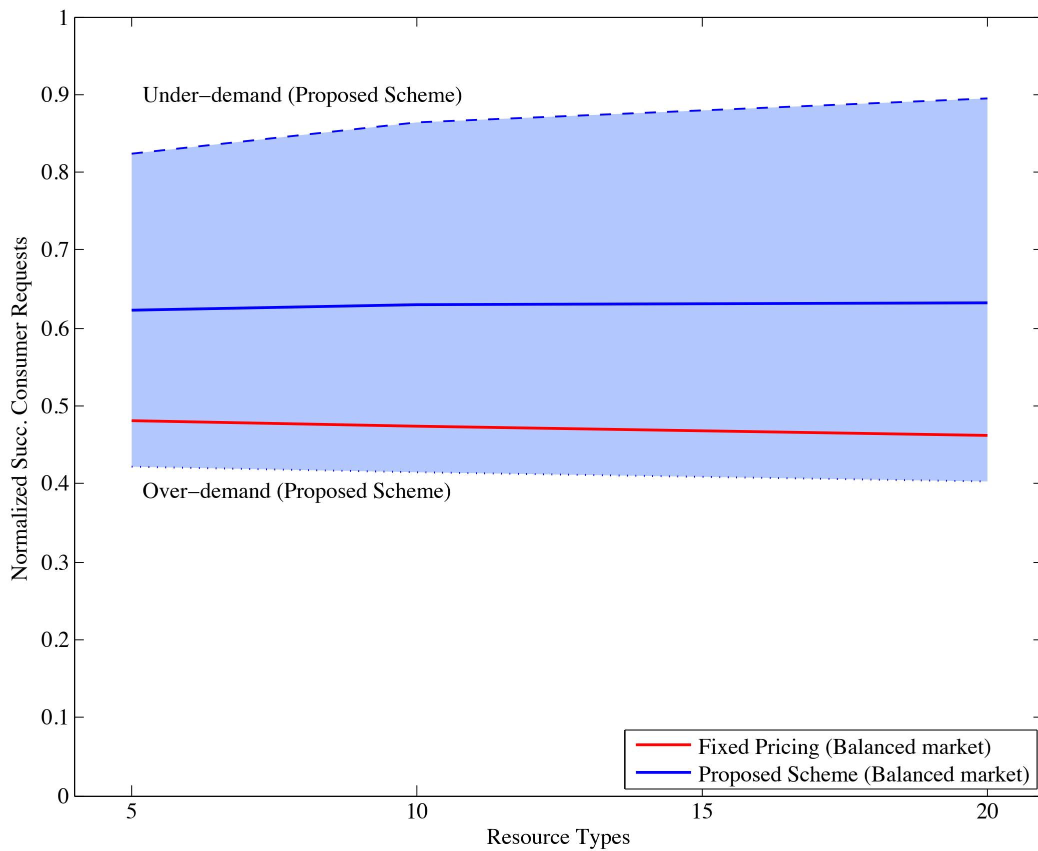


Figure 6.7: Successful Consumer Requests with Multiple Resource Types

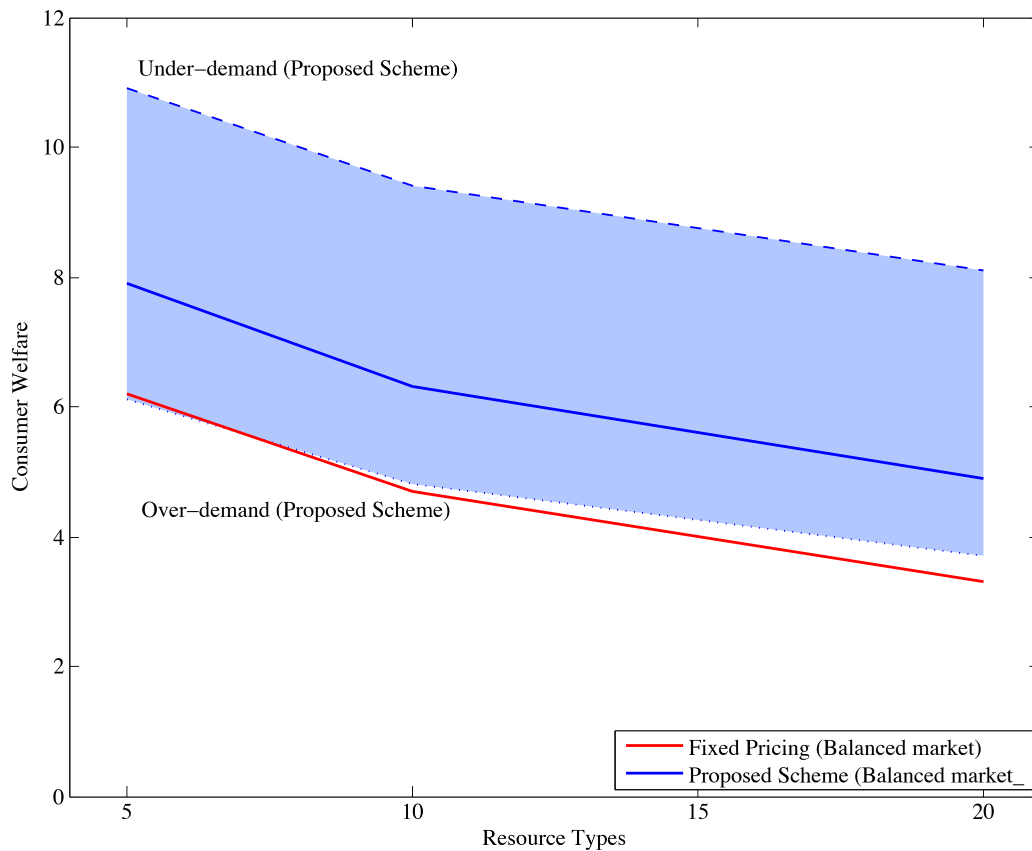


Figure 6.8: Average Consumer Welfare with Multiple Resource Types

be used, which is not strategy-proof.

Similarly to the above, using fixed pricing results in the same mean consumer welfare when varying market conditions, and welfare decreases when increasing the number of resource types. Figure 6.8 shows that, for our proposed pricing scheme, the consumer welfare is higher when compared to fixed pricing, and varies according to demand and supply: consumer welfare increases when demand is lower than supply, and decreases when demand is higher.

Figure 6.9 shows the number of allocated provider resources normalized, for fixed pricing (shaded red area) and the proposed scheme (shaded blue area), under different market conditions when the number of resource types varies from 5 to 20. In the case of under demand and balanced market, our scheme is able to allocate a larger number of provider resources, and performs worse only in the case of over-demand. This is be-

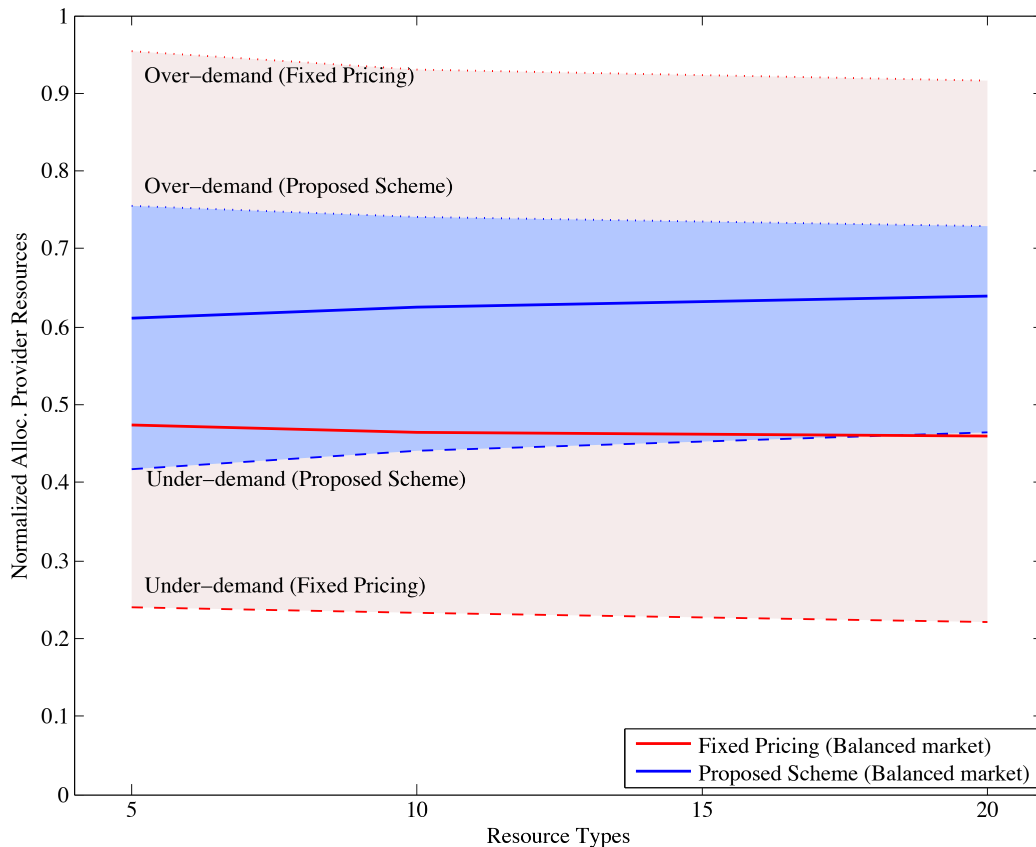


Figure 6.9: Allocated Provider Resources with Multiple Resource Types

cause when is over-demand, our pricing mechanism will compute higher provider payments, according the current the market price. Similarly, in the case of under-demand, the dynamic scheme is able to adapt by using a lower resource price and it achieves a larger number of allocated resources. In the case of a balanced market, the economic efficiency is increased by approximately 15%.

6.3 Distributed Pricing

In this section, our main objective is to analyze the scalability of our distributed pricing scheme and determine the overhead introduced by the synchronization protocol. Our study uses the FreePastry overlay network in two different environments: the FreePastry simulator, where we can simulate a large distributed market, and PlanetLab [93], where we deployed a prototype of our scheme and use it to validate the simulator results on a smaller number of distributed nodes. As shown in Figure 6.10, FreePastry simulator

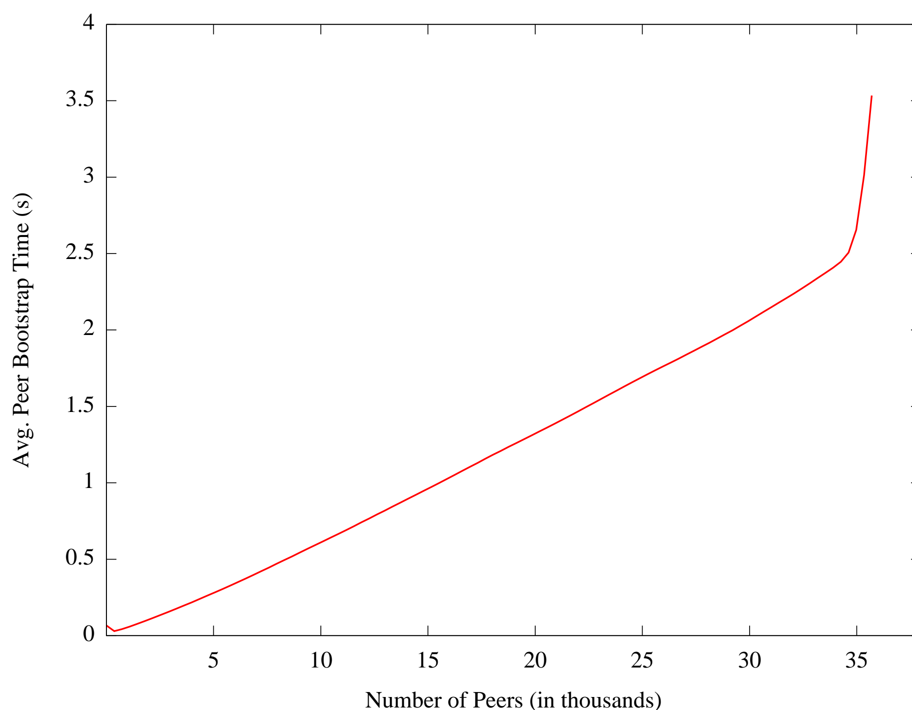


Figure 6.10: FreePastry Simulator Bootstrap Time

supports close to 35,000 peers before running into scalability issues of its own. The *bootstrap time* in the simulator represents the time taken by a node to join the overlay. When increasing the number of users in the simulator, the boot time is also increased because of the stabilization protocol employed by FreePastry. In order for the overlay to reach steady-state, we feed the workload to the simulator in two steps. In the first step, the workload consists only of user join requests, with inter-arrival times of one second. In the second step, after the network stabilizes, we input the second workload that contains the consumer requests and provider resource publish.

6.3.1 Vertical Scalability

To study vertical scalability, with respect to the number of resource types in a consumer request, we compare the average allocation time obtained by the distributed scheme with the centralized scheme. Both the distributed and the centralized scheme are implemented in FreePastry. In the latter, the first peer in the overlay is delegated to be the market-maker, while the other peers are either consumers or providers.

In our experiments, we use 5, 10, 20 and 40 resource types in a consumer request for both the centralized and distributed implementation. We run the distributed application on both the simulator, and on the PlanetLab prototype. We create an overlay network containing 50 peers, where each peer generates 100 events. The total inter-arrival rate of the events is exponentially distributed with the mean of one second. We consider a balanced market, where demand equals supply. Accordingly, each user event has an equal probability for a publish or a request message. Additionally, the number of items for each resource type is generated from an exponential distribution with mean 10, and the price for a resource item is uniformly distributed between 80 and 120 (price variation is set to 20%).

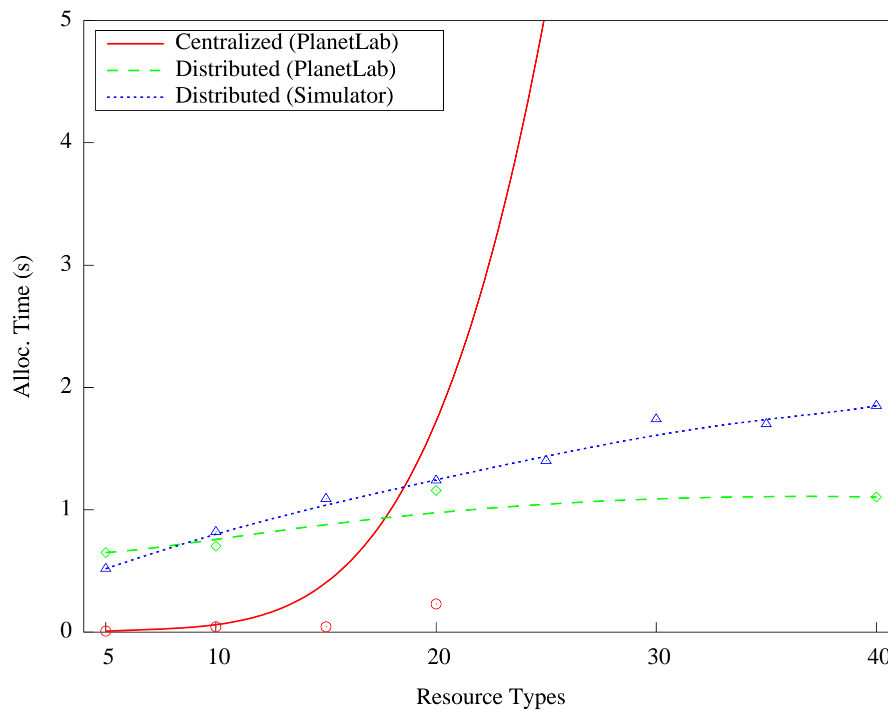


Figure 6.11: Increasing the Number of Resource Types in a Request

Our results, shown in Figure 6.11, show that in the distributed market, the auction protocol imposes a greater overhead and, when having a small number of resource types in a request, the average allocation time is higher than the centralized implementation. For example, when having requests for 10 resource types, the allocation time is $0.1s$ for the centralized scheme, while for the distributed scheme is $0.7s$. However, as the number of resource types increases, the distributed scheme proves to be scalable and maintains a consistent allocation time of less than one second. For example, when having consumer requests for 25 resource types, the average allocation time for the centralized scheme is $5s$, while for the distributed scheme on the PlanetLab prototype it is $1.1s$. It can be seen that the average allocation time obtained in simulations is higher than the measured time on PlanetLab. This is because in the simulator we used a fixed network delay between peers of $100ms$, which is higher than the actual delay between PlanetLab nodes.

6.3.2 Horizontal Scalability

To study the horizontal scalability, with respect to the number of users, we use only the FreePastry simulator because PlanetLab is not able to provide the necessary number of nodes for this scalability study. In this experiment, we vary the network size from 1,000 to 30,000 users. We use similar settings as in the previous study: one second event inter-arrival rate, balanced market, number of items from an exponential distribution with mean 10, price uniformly distributed between 80 and 120, and 100*ms* fixed network delay. The number of resource types in a consumer request is sampled from a uniform distribution between 1 and 10. We run the simulations for 600,000 events, which represent one simulation week.

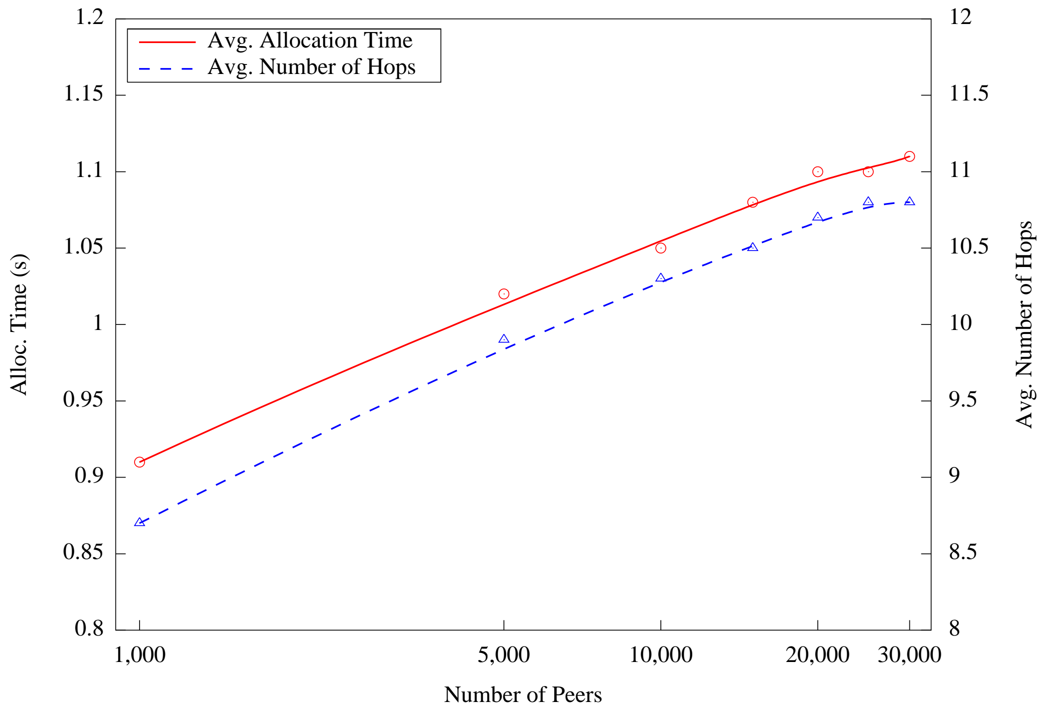


Figure 6.12: Increasing the Number of Users

Figure 6.12 presents our results. As can be seen, the proposed distributed pricing scheme is horizontally scalable, as the average allocation time increases logarithmic with the number of users. For example, the average allocation time when having 1,000

users is 0.91s, and for 30,000 users is 1.1s. Additionally, we measured the average number of hops for all messages required by the proposed distributed protocol and found that both curves have the same gradient. Thus, we conclude that our scheme adopts the horizontal scalability of the underlying overlay network.

6.3.3 Impact of Network Delay

Computational efficiency is a major design criterion in the allocation of shared resources. In addition to vertical and horizontal scalability, our theoretical analysis in Chapter 4 shows that network delay is one of the factors that influences the allocation time in the proposed distributed scheme. To evaluate the impact of network delay, we simulated an overlay network of 10,000 peers, with an event inter-arrival rate of one second in a balanced market. We simulated 600,000 events for different values of network delay, ranging from 100ms to 900ms. An event is a consumer request or a

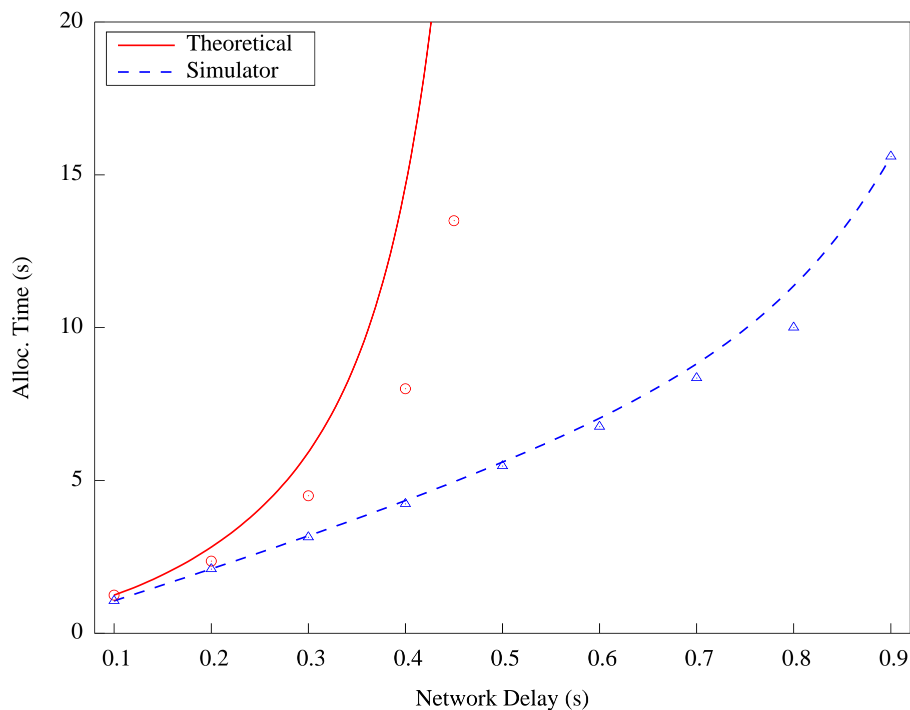


Figure 6.13: Impact of Network Delay

provider resource publish. For comparison, we plotted average allocation time for the simulations together with the theoretical worst-case scenario in Figure 6.13. Although the network delay from the PlanetLab nodes was much lower ($\sim 0.04s$) than the values used in our simulation, we found that a large network delay can result in increased user waiting times. For example, when the network delay is $0.9s$, the average allocation time increases to more than $15s$.

6.3.4 Impact of Arrival Rate

In addition to network delay, out theoretical evaluation of the distributed scheme has found that the consumer request arrival rate influences the allocation time. In our distributed scheme, each consumer request is routed by the overlay network to a random peer that becomes the request broker. Thus, in a large network, the consumer request rate is not an issue for scalability. However, each request broker sends lookup requests to the resource brokers for each resource type in the consumer request. Thus, the issue is the arrival rate of resource type lookups at the resource broker, which synchronizes with the request broker using the protocol described in Chapter 4.

To study the impact of arrival rate, we vary the request inter-arrival rate between $0.15s$ and $1s$ in the FreePastry simulator. We use consumer requests for one resource type and a fixed network delay of $100ms$, to achieve a similar inter-arrival lookup rate at the resource broker. The FreePastry overlay has 10,000 peers and we generate consumer request events and provider publish events with equal probability, for a balanced market. We run the simulator for 600,000 events. As can be seen in Figure 6.14, having the inter-arrival time lower than $0.25s$ leads to an exponential increase in allocation time. For example, the allocation time for an inter-arrival time of $0.15s$ is more than $2s$, almost double than the allocation time obtained for an inter-arrival time of $0.2s$. The knee value depends on the network delay because of the synchronization mechanism, with the curve in Figure 6.14 specific for the $100ms$ delay.

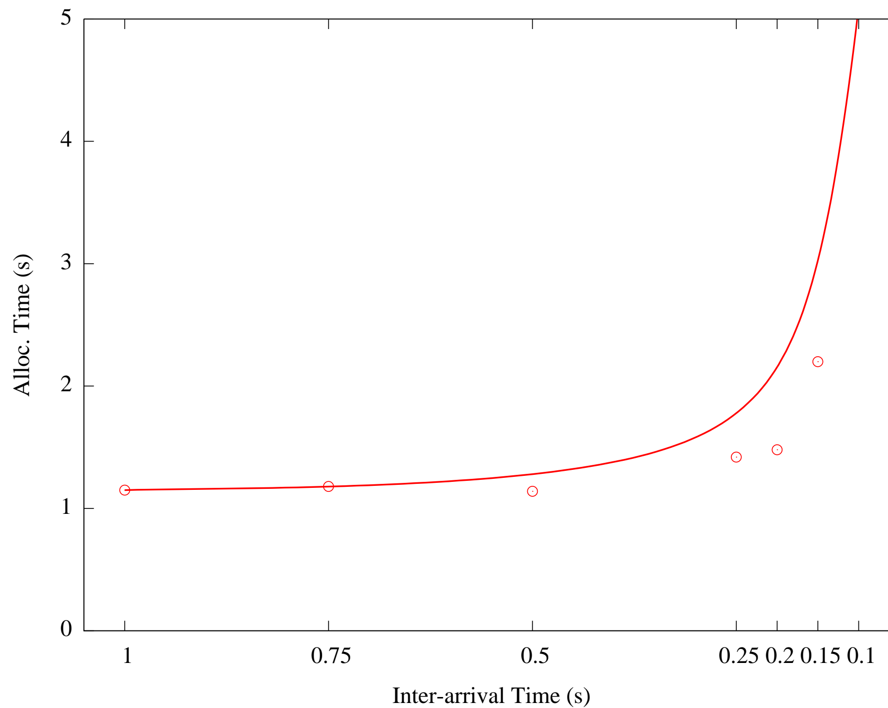


Figure 6.14: Impact of Arrival Rate

6.4 User Rationality in Federated Clouds

Cloud infrastructure services, also known as *Infrastructure as a Service* (IaaS), deliver computer infrastructure using a virtualized environment to cloud consumers over the Internet. In IaaS, providers typically use fixed pricing, such that strategic user behavior is limited. Lately, Amazon EC2 has started to offer resources using a dynamic pricing scheme, spot pricing. In contrast to our scheme, in spot pricing the resource price is set only by the demand. Cloud consumers specify bids that represent their reserved price, i.e. the maximum hourly price they are willing to pay for the resource type (where the resource type is the required instance type). A spot request may contain several instances, all of the same instance type. In contrast to our scheme, consumers cannot request for multiple resource types. The spot price is determined by Amazon dynamically, based on the current queue of spot requests. When the current spot price is

lower than the consumer bid, the respective spot request is dequeued and allocated, and the instances are started. After each hour, the spot price is updated, and, if the current spot price becomes higher than the consumer bid price, the instances allocated for the consumer spot request are terminated. The cloud consumer can specify a persistent flag for the spot request, such that after termination the request will be put back in the queue and restarted when the spot price becomes lower than the consumer bid price.

In this section, our objective is to study the user rationality in a federated cloud by comparing the consumer welfare of spot pricing and our proposed strategy-proof scheme in different Amazon EC2 regions. We use the EC2 spot price traces available at CloudExchange [30] in four regions: *us-east-1* (Northern Virginia), *us-west-1* (Northern California), *eu-west-1* (Ireland), and *ap-southeast-1* (Singapore). When using the spot pricing scheme, requests are allocated to the region geographically closest to the consumer, whereas in the case of our strategy-proof pricing scheme, requests are allocated dynamically to any EC2 region. We focus in our study on computational-intensive instance types (*c1.medium*, *c1.xlarge*), with the assumption that data availability and the latency of instances allocated in other regions are not an issue for the consumers.

6.4.1 Methodology

According to Amazon, the spot price is determined periodically by the load of EC2 [5]. In the absence of any other public information provided by Amazon about the infrastructure or workload in EC2, we propose a bottom-up approach to predict the consume welfare. Our approach, shown in Figure 6.15, contains three steps. In the first step, we use a *pricing model* to predict the hourly load of an EC2 region, using the spot price history. In the second step, we use a *workload model* to generate consumer spot requests that produce the load predicted in the first step. In the last step, we create a request queue for all the EC2 regions, and measure the consumer welfare obtained when allocating using the spot pricing scheme, and the strategy-proof scheme.

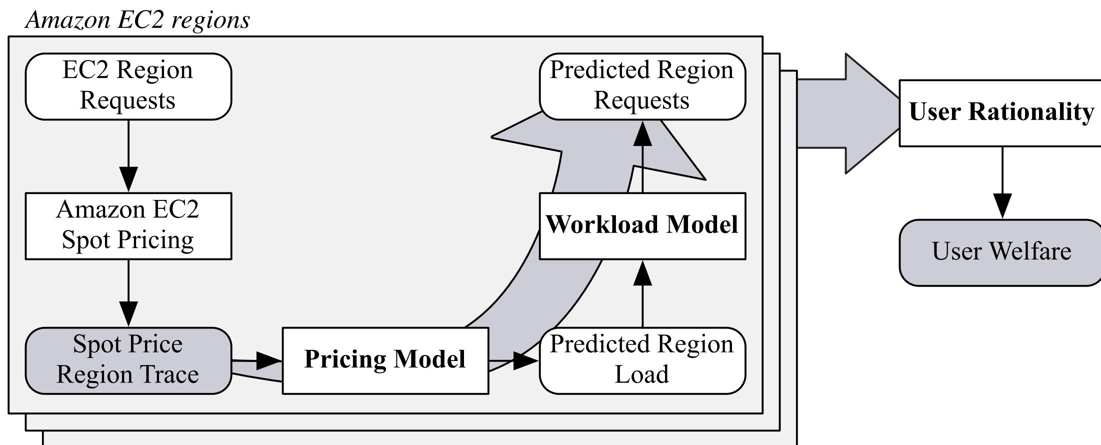


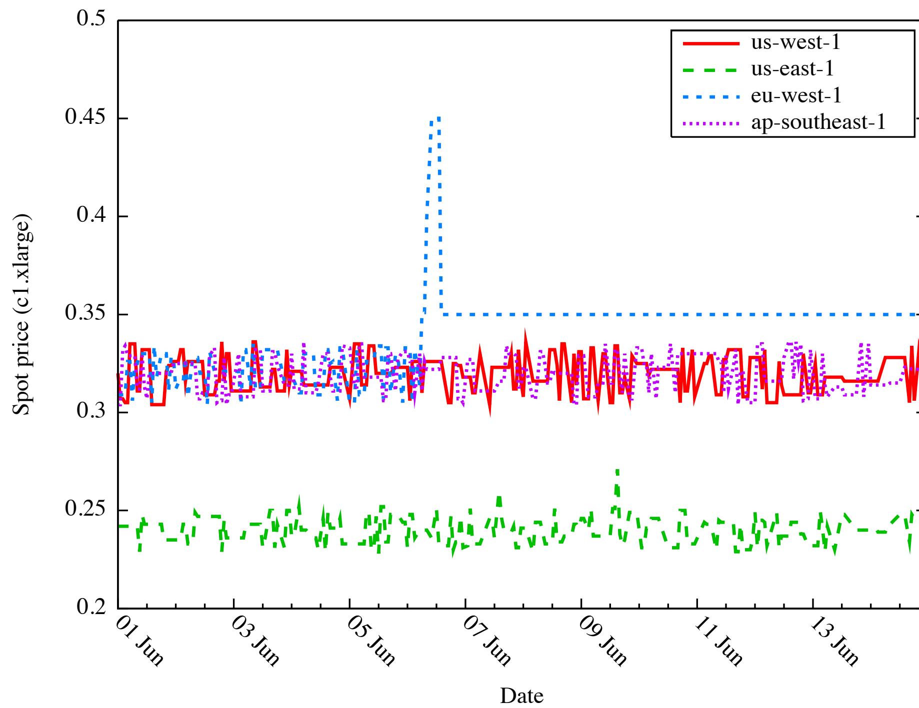
Figure 6.15: Determining User Welfare on EC2

Pricing Model

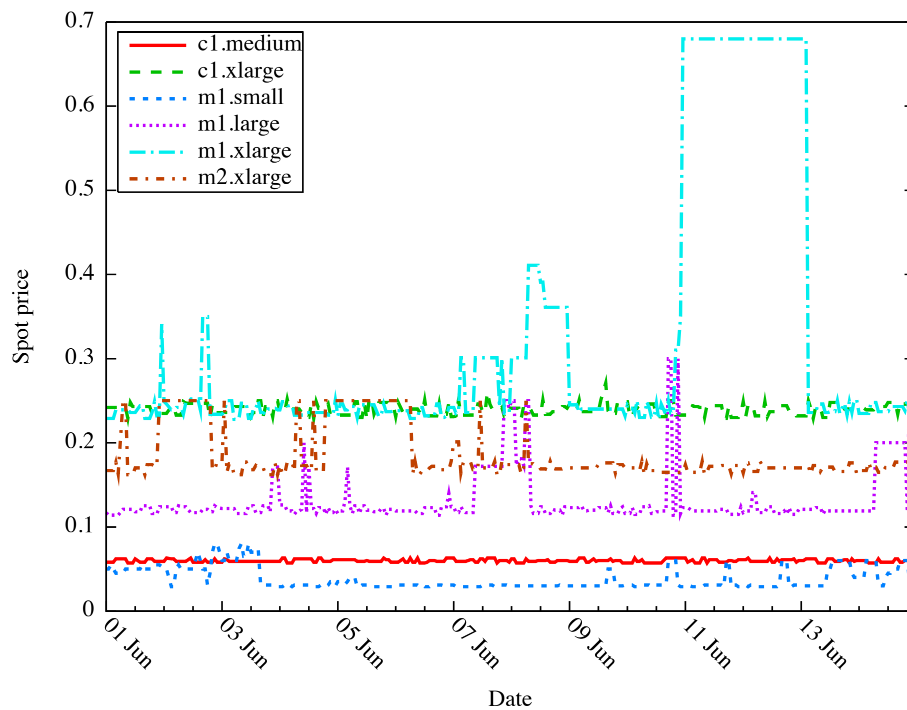
The pricing model is based on three observations we determined after the examination of the EC2 spot prices.

1. Figure 6.16(a) shows the spot prices of the same instance type, *c1.xlarge*, across different EC2 regions in the first weeks of June 2010. As can be seen from the figure, spot prices have spikes when the demand increases. However, these spikes are not correlated between different regions. Our statistical tests found 10%–20% correlations in a one year time period. For example, on June 6, there is a steep increase in the spot price in the *eu-west-1* region, but a decrease in the region *us-east-1*. Accordingly, our first observation is that *each region computes spot prices independently*, based on the respective region load. Thus, we can consider in our evaluation that each EC2 region is acting as a different resource provider in a federated cloud.

2. In Figure 6.16(b), we plot the spot prices for different resource types within the same EC2 region, *us-east-1*, for the same period of time as above. An examination of the data shows that, within the same region, there are spikes in the spot price for different instance types, according to the demand for the respective resource type.



(a) in different EC2 regions for the same instance type



(b) in the same EC2 region for different instance types

Figure 6.16: Amazon EC2 Spot Prices

However, the spikes are not correlated among different resource types, with a 10%–20% correlation found in one year time of traces. For example, on June 4, there is a steep increase in the spot price of the *m1.large* instance type, and at the same time a steep decrease for the *m1.small* instance type. Accordingly, our second observation is that *each region has a different pool of resources for each instance type*. Based on this observation, we can analyze different instance types independently.

3. The plots in Figure 6.16 shows us that spot prices are usually within a small range, with spikes when the demand increases. Our last observation is that *each instance type in each region has a minimum reserved price (r), and a maximum spot price (p_f)*. Although spot prices can increase for short periods of time to values higher than the fixed instance price, we use, for simplicity, the fixed price as the maximum spot price.

Based on the above observations and assumptions, we propose to use a power function to estimate the load of each EC2 region from the trace data. Accordingly, our model uses the following function for spot pricing:

$$spot_price = r + (p_f - r) * load^k \tag{6.1}$$

where k is a parameter that determines the gradient of the increase in spot price relative to load.

Workload Model

Using the pricing model outlined above, we can use the spot prices to predict the load of the EC2 regions, for each instance type. In this section, we present our workload model, which determines the consumer requests that generate the respective load. We need to address two issues. The first issue is determining the size of the EC2 region instance pool, for each instance type. The size of the instance pool is used to compute

the total number of instances running in an EC2 region. The second issue is, given the total number of instances running, how to determine the number of consumer requests and the number of instances in a request.

Amazon does not provide any historical information about the number of instances running in EC2. However, it has been found that the unique identifier assigned to an EC2 instance at the time it starts running can be used to determine the total number of virtual machines started in the Amazon cloud until that point in time [101]. Accordingly, using the identifiers of instances started at different dates is possible to estimate the number of instances started in that interval of time. In our experiments, we look at the spot prices from June 2010, for which there are available the traces with the spot prices [30], the number of instances started daily, and the distribution of instance types in the *us-east* region [99]. We consider the distribution of instance types in the other EC2 regions similar to *us-east*, and the size of the regions proportional to the number of public IP addresses from each region, published by Amazon.

Using the number of instances started daily from each instance type, our workload model determines the number of consumer spot requests, the number of instances for each request, and the request arrival time, as follows. Firstly, we determine the number of instances in a request, which we assume it follows an exponential distribution. The distribution mean is computed by fitting the number of CPUs requested in parallel workload traces from production systems. Specifically, in our experiments, we use the traces of jobs that run on HPC clusters, available at the Parallel Workload Archive [87]. Secondly, we start to generate requests for each instance type in an EC2 region, containing a number of instances according to the exponential distribution determined above. We consider all the consumer requests with the total number of instances similar to the number of instances started hourly in the Amazon region from the EC2 trace. Lastly, we generate the request inter-arrival time, using a Poisson distribution with the mean of $(3600 - N)$ seconds, where N is the number of hourly requests determined in

the previous step.

Validation

Spot prices are expressed by discrete values, in cents (¢), with maximum one decimal point. Accordingly, each discrete spot price corresponds to a range load values, with the same spot price for each range. For example, for the *m1.small* instance type in the *us-east-1* EC2 region, the minimum price is $r = 2.5¢$, and the fixed price is $p_f = 8.5¢$. Using the parameter $k = 4$, we determine the spot price $p = 2.5¢$ for the load range $0-36\%$, $p = 2.8¢$ for the range $47-51\%$, etc. As the load increases, the ranges become smaller and there is a steeper increase of the spot price, similar to the spikes observed in the trace data. The value for the parameter k was determined experimentally, using trial and error. We validated our model by comparing the spot prices from the EC2 trace with the spot prices generated using Equation 6.1. Figure 6.17 shows the spot price for the

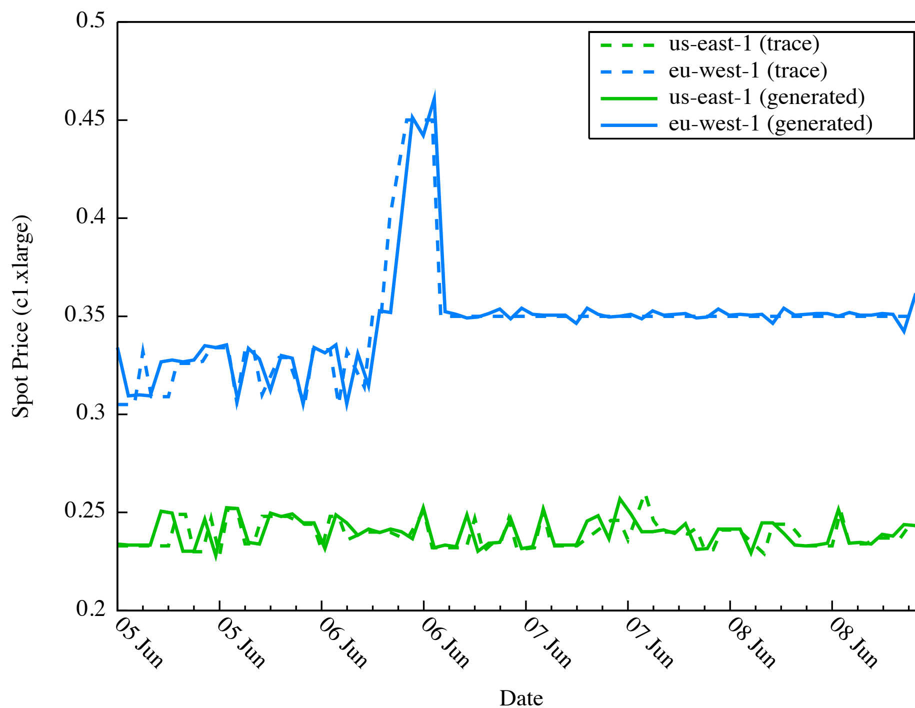


Figure 6.17: Spot Price Validation

c1.xlarge instance type from the trace using a dotted line, and the spot price generated using our model using a continuous line. For the figure to be clear, we plotted only the *us-east-1* and *eu-west-1* regions, between June 5–8.

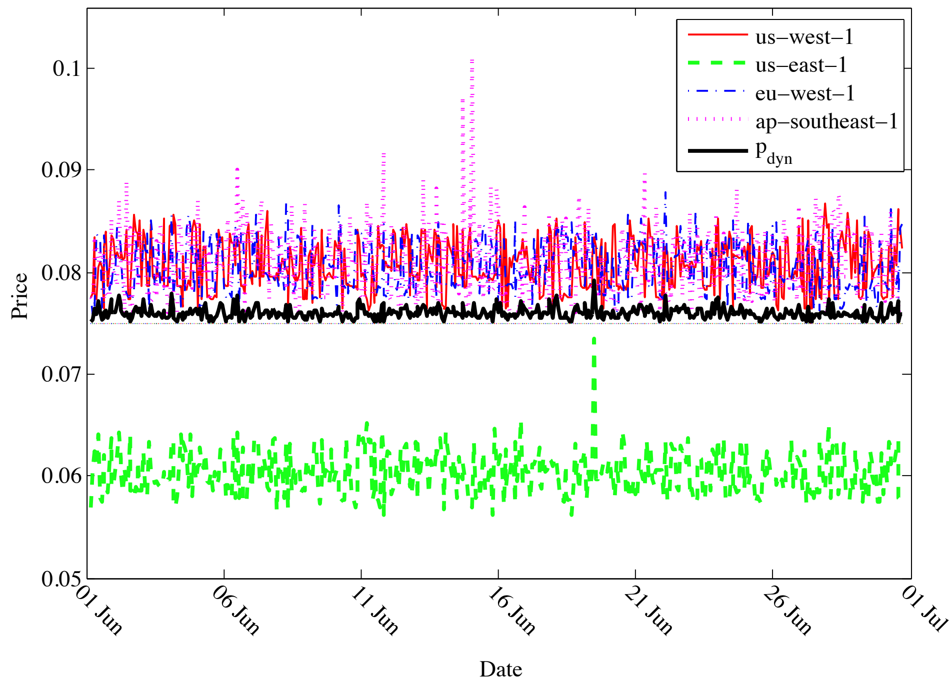
6.4.2 Results

From the trace data, we can observe that the *us-west*, *eu-west* and *ap-southeast* regions are similar with respect to both the reserved price, r, and the fixed price, p_f. For example, for the instance type *c1.xlarge*, the reserved price is 30.4¢ and the fixed price is 76¢ in all three regions. However, for the *us-east* region, both the reserved price and the fixed price are different than in the other regions, for all instance types. For example, the reserved price for the *c1.xlarge* instance type is 22.8¢ and the fixed price is 60¢. Using r, p_f, and the spot prices from the traces, $spot_price$, we determine the hourly load ranges using the function in Equation 6.1, as follows:

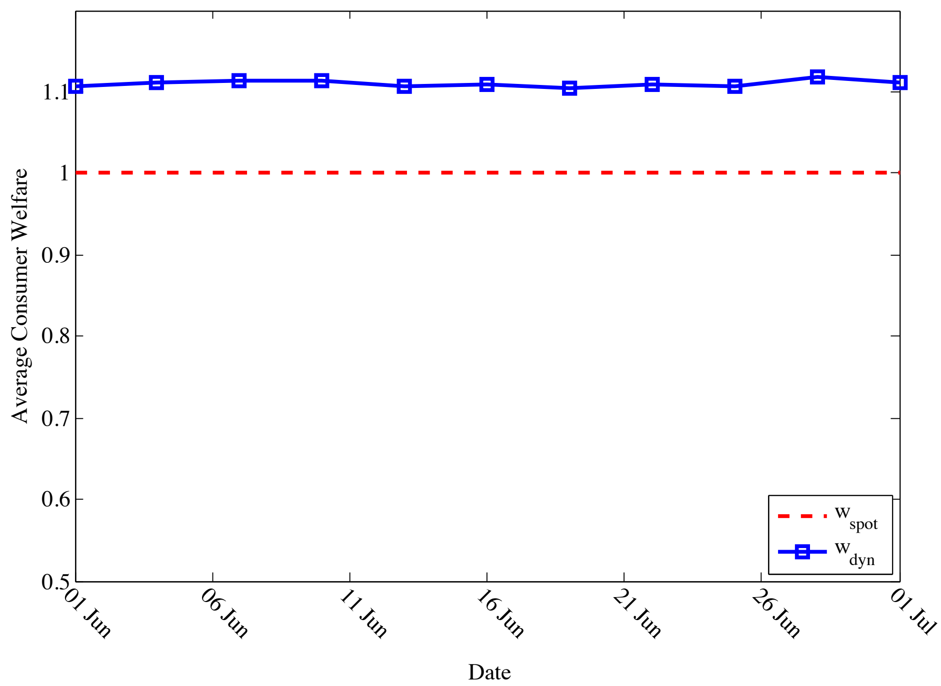
$$load_{min} = \sqrt[k]{\frac{spot_price - r}{p_f - r}} * 100 \tag{6.2}$$

$$load_{max} = \sqrt[k]{\frac{spot_price + 0.1 - p_0}{p_f - r}} * 100 \tag{6.3}$$

Next, we estimate the number of instances of each instance type running each hour, based on the total number of hourly running instances, the load range, ($load_{min} - load_{max}$), and the distribution between the instance types. According to [99], in June 2010, there were 9% *c1.medium* instances and 28% *c1.xlarge* instances from the total number of virtual machines started on EC2. Lastly, we generate consumer requests for a number of instances according to an exponential distribution with the mean 42.2, determined by the fitting function from the parallel workload trace. We used the *LLNL-Thunder* log, which contains several months of records from the #2 machine in the 2004 Top500 list, installed at Lawrence Livermore National Lab [87].



(a) Spot and Strategy-proof Prices



(b) Consumer Welfare

Figure 6.18: `c1.medium` Results

Our implementation uses MATLAB scripts to predict the consumer requests as detailed above, and to order the requests in all the EC2 regions into one queue, sorted by the request arrival time. We allocate the consumer requests using the two pricing schemes using the First-Come-First-Serve policy. Consumer welfare when using spot pricing, w_{spot}, and the strategy-proof pricing scheme, w_{dyn}, is determined using the following functions:

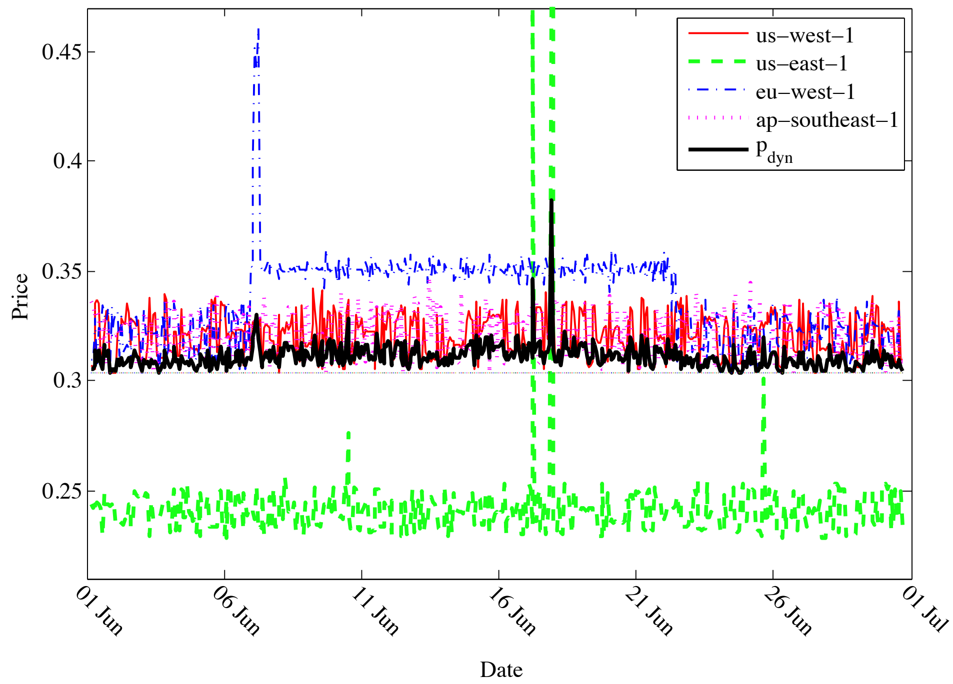
$$w_{spot} = \sum (p_f - p_{spot}) \quad (6.4)$$

$$w_{dyn} = \sum (p_f - p_{dyn}) \quad (6.5)$$

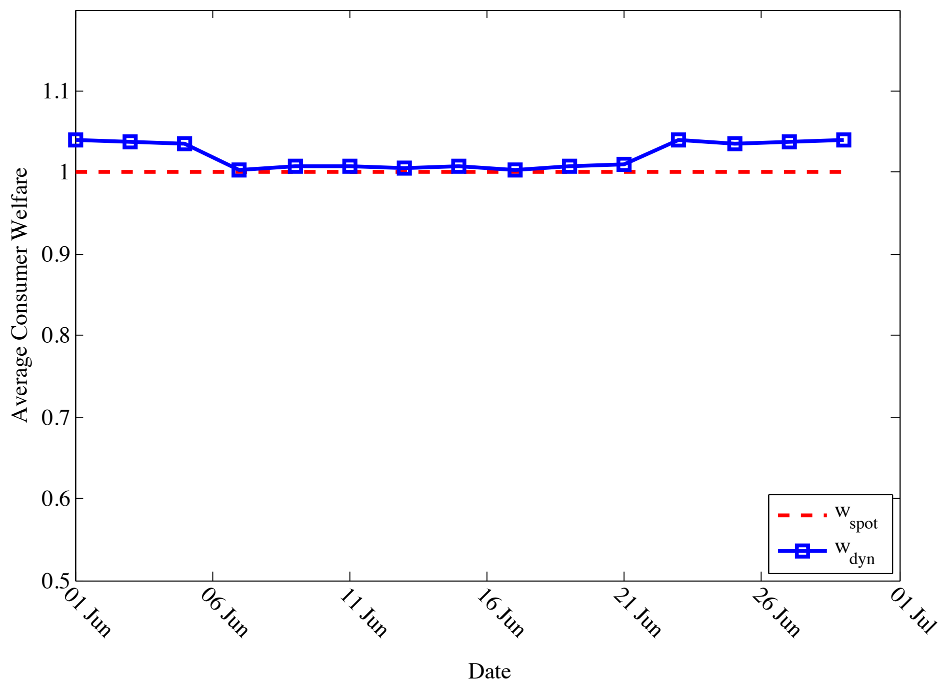
where p_{spot} is the instance price determined in a region when using spot pricing, and p_{dyn} is the price determined across regions by the strategy-proof scheme.

Figure 6.18 shows our results for the *c1.medium* instance type. In Figure 6.18(a), the black line represents the dynamic price determined using the strategy-proof scheme, with requests being allocated to any EC2 region. The thin lines represent the spot prices determined using the proposed spot pricing function, from Equation 6.1, in all of the EC2 regions, with a request being allocated only in the region where it was generated. It can be seen from the figure that the instance prices computed using the strategy-proof scheme are not lower than the reserved price of the three similar EC2 regions, due to the financial incentives required to achieve strategy-proof. When allocating consumer requests for one instance type, the strategy-proof scheme is similar to a Vickrey auction [120], where the payment is equal to the second provider price. In the case of Amazon EC2 regions, the lowest second price that can be found is the same reserved price for the three regions, *us-west-1*, *eu-west-1*, and *ap-southeast-1*.

A comparison of the normalized consumer welfare achieved by the two pricing schemes is shown in Figure 6.18(b). It can be seen that using the strategy-proof scheme results in higher consumer welfare by more than 10% during the entire period. This is



(a) Spot and Strategy-proof Prices



(b) Consumer Welfare

Figure 6.19: `c1.xlarge` Results

mostly because the strategy-proof scheme can allocate instances to different providers, according to the market price set by demand and supply. Thus, requests that generated in a region with high demand can be allocated by our scheme in a different region, such that consumer welfare is increased.

Similar to the above, we evaluated the *c1.xlarge* instance type. Figure 6.19(a) shows the different prices obtained by the spot pricing scheme and the strategy-proof scheme. For the *c1.xlarge* instance type, the price determined by our strategy-proof scheme is not less than 0.075¢, the second-best reserved price. We plotted the consumer welfare in Figure 6.19(b). Our results show that, for the *c1.xlarge* instance type, the average consumer welfare obtained using the strategy-proof scheme is higher by 4% in the first days and in the last week of the observed period. However, for the rest of the period, the average consumer welfare obtained using the strategy-proof scheme is only marginally higher, by 1%, to the welfare obtained using the spot pricing scheme. By correlating the welfare with the demand and prices from Figure 6.19(a), we can observe that the performance of the strategy-proof scheme starts to degrade beginning with day 6, when there is a high demand in the *eu-west-1* region, until day 23, when the demand decreases. Moreover, in day 18, our strategy-proof scheme performs similar to spot pricing, due to a sudden spike in demand in the *us-east-1* region. Similar to our previous findings [74], the consumer welfare achieved by the strategy-proof scheme varies according to demand and supply. Thus, the consumer welfare increases when demand is lower than supply and decreases when demand is higher.

By analyzing the EC2 traces and computing the consumer welfare, we try to determine the rationality of cloud computing users. While users have found methods to decrease their costs when running instances on EC2, no strategy uses the pricing schemes available in EC2. In this context, rational behavior may include, for example, reselling reserved instances for profit, or bidding on instances in regions with lower demand and prices. Our evaluation shows that consumer welfare can be increased when users be-

have rational by more than 10%, even when having payments larger than the minimum spot price. We regard this as a sign that rationality is currently not a significant issue in cloud computing, with standalone providers that can adjust their supply to influence the resource market. For example, constraining supply to be lower than demand leads to a lower consumer welfare, and a better outcome for the provider. However, cloud computing usage is currently increasing both in *breadth*, i.e. the number of instance types and services offered, and in *depth*, i.e. the number of providers. With an increasing number of cloud consumers, it is expected that more providers will offer similar services. With interoperability among providers, consumers will have a choice of the same service across different clouds to improve reliability and elasticity, and rationality can become an important issue as consumers and providers strategize to improve their welfare.

6.5 Summary

In this chapter, we evaluated the performance of the proposed pricing mechanism in different conditions. Our experimental evaluation complements the theoretical analysis of the economic properties and the scalability of our scheme. We evaluate the trade-off between the economic and computational efficiencies by measuring the total welfare and the request allocation time. Compared to combinatorial auctions, which maximizes total welfare, our scheme sacrifices 20% economic efficiency. However, the running time of our algorithm is greatly reduces, from hours in combinatorial auctions, to seconds. Our in-depth study of the economic efficiency showed that our scheme can allocate 25% more consumer requests than one-sided auctions, and 35% more requests than fixed pricing, while increasing the consumer welfare by 10%. In the case of requests with multiple resource types, the number of allocated consumer requests can be increased by up to 40% when there is low resource demand.

We evaluated the scalability of our distributed auctions scheme using simulations validated by a prototype implementation on PlanetLab. Our results showed that our scheme is scalable when increasing the number of resource types in a consumer request, and the number of users in the system. However, due to the overhead of the synchronization mechanism, the allocation time for consumer requests with less than 25 resource types is higher than a centralized implementation, and increases exponentially with the network delay.

Lastly, we studied the rationality of consumers in a federated cloud composed of four Amazon EC2 regions, using real traces of EC2 spot prices. Our evaluation shows that, in the case of spot pricing, rational users can increase their welfare by more than 10% by bidding on resources in regions with lower demand. Thus, strategy-proof pricing is required in federated clouds to incentivize rational users to bid truthfully.

Chapter 7

Conclusions

Currently, there is growing interest in federated sharing of computing resources, with emerging architectures such as cloud computing, where commoditized resources are shared and traded over the network. However, several challenges remain when allocating shared resources, such as incentivizing rational users, which can be untruthful in order to increase their welfare, and supporting multiple resource types per request, such that consumers do not integrate resources manually. In this thesis, we have designed a strategy-proof resource pricing mechanism that can be used to allocate consumer requests with multiple resource types, and prototype a federated cloud platform that uses our mechanism to integrate resources from multiple cloud providers. We have considered the allocation of shared resources from two perspectives. From the economic perspective, we achieve strategy-proof by providing incentives for truthful users, and budget balance by having equal consumer and provider payments. From a computational perspective, our pricing and allocation algorithm computes user payments in linear time, and our distributed auctions mechanism is scalable when increasing the number of users and the number of resource types in a consumer request.

7.1 Thesis Summary

The major contributions of this thesis are: i) strategy-proof pricing for market-based resource allocation, ii) a dynamic pricing for multiple resource types per request, iii) a scalable distributed auctions scheme, and iv) a platform for managing resources in federated clouds. These contributions are detailed below.

1. **Strategy-proof Pricing for Market-based Resource Allocation**

   While there is a consensus on the benefits of market-based approaches for the allocation of shared resources, there is no common view on the appropriate economic mechanism that should be used. A study of related works has shown several approaches that have been considered, such as bartering, fixed pricing, market-clearing equilibrium price, proportional share, bargaining, and single or multi-dimensional auctions. However, most of the authors have a single view when allocating shared resources: either purely economic, with their primary goal economic efficiency, or purely computational, with their main objective computational efficiency. In contrast, our work considers both the economic and computational aspects, using the mechanism design framework, which combines utility maximization from economics, rationality and Nash equilibrium from game theory, and algorithm design and complexity from computer science. From an economic point of view, there is a trade-off among individual rationality, which defines the rational users, incentive compatibility, which rewards truthful behavior, budget balance, which means that provider and consumer payments are equal, and maximum economic efficiency. From a computational point of view, there is a trade-off between Pareto efficiency and computational efficiency. We have shown how a computationally efficient pricing scheme can be used to allocate resources in a federated system with rational users by using financial incentives.

(a) *Formal Approach to Market-based Resource Allocation*

We formulate the market-based resource allocation problem as an utilitarian optimization mechanism design problem, with the output specification given by a positive real valued objective function, such that our problem is compatible with the classic economic mechanism, the Vickrey-Clarke-Groves auction. We propose to trade-off Pareto efficiency, which allows us to achieve the other economic properties, namely *individual rationality*, *incentive compatibility*, *and budget balance*, while using a *computational efficient* algorithm. We define the economic properties in the mechanism design framework, and express our solution using a VCG-based function for providers, and a budget balance function for consumers. We formally prove that our solution achieves the economic properties, and show in our theoretical analysis of the pricing algorithm its linear complexity.

(b) *Economic and Computational Efficiency Trade-off*

We have evaluated our trade-off both theoretical and experimental. Our experimental study compares our scheme with two different implementations of combinatorial auctions, a scheme that is Pareto efficient. The VCG combinatorial auction scheme is not budget balanced, while the Treshhold combinatorial auction scheme is not strategy-proof. We measure the total welfare for providers and consumers, as an indicator of economic efficiency, and the average allocation time, as an indicator of computational efficiency. Although our mechanism has close to 20% loss in terms of total welfare, the allocation time has been reduces by several orders of magnitude, from hours to seconds. Our theoretical analysis showed that our pricing scheme allocates resources in linear time. This makes our mechanism feasible to implement in several systems where pricing is used, such as cloud computing, online auctions and exchanges, or online advertising.

(c) *Pricing in the Monopoly Situation*

One of the limitations of our scheme, similar to other VCG mechanisms, is the monopoly situation, when consumer requests may not be allocated even if resources are available, because one or several providers gain control of the market by providing a sufficiently large portion from the total resources. For the monopoly situation, we propose alternative pricing functions, which are partial strategy-proof. This is because the monopolistic provider can adjust the supply to influence the market price.

2. **Dynamic Pricing for Multiple Resource Types per Request**

The pricing mechanism involves matching market participants, namely consumers and providers, to engage in an exchange using a common type of currency. Often, consumers require multiple resource types, which may be allocated from different providers. In such cases, optimal allocation algorithms are often infeasible to implement, and resource providers either use fixed tiered pricing, or create separate markets for each resource type. The users have to request resources from each provider, and integrate them manually, leading to a loss in economic efficiency, among others. The pricing and allocation mechanism we propose addresses this issue by leveraging reverse auctions.

(a) *Reverse Auction Scheme*

In traditional auctions, a third party, designated as the market-maker or auctioneer, receives bids from the consumers and determines the allocation winners. Accordingly, this results in consumers having to bid manually for each resource type. In contrast, in our mechanism the providers submit the reserved price for each resource type as a bid to the market-maker, which auctions consumers requests sequentially, using the First-Come-First-Serve policy. We compute the payments for the winning providers for each resource type in the consumer request, and perform the allocation when the request reserved price is higher than

the total seller payment. Our proposed reverse auction mechanism is very effective in allocating consumer requests. Our experimental evaluation has shown that, even in the case of requests for one resource type, our scheme is able to allocate over 25% more requests than traditional auctions. However, our mechanism greatly out-performs traditional auctions in the case of consumer requests with multiple resource types, when it achieves a number of successful requests several orders of magnitude higher than the consumer bidding individually on each resource type and winning all auctions at the same time.

(b) *Dynamic Pricing in Different Market Conditions*

Our pricing scheme computes payments dynamically, based on demand and supply. In contrast to fixed pricing, our scheme can incentivize consumers when demand is low with lower payments and providers when demand is high with higher payments. We have studied the cases of under-demand and over-demand, and found that our scheme can allocate 15-20% more requests for requests with one resource type. We have achieved good results with multiple resource types per request. Compared to fixed pricing, currently used by many cloud providers, our scheme allocated up to 30% more consumer requests in a balanced market. However, the advantage of our scheme over fixed pricing can be seen when demand and supply fluctuates. For example, in the case of under-demand, our scheme allocated up to 80% more consumer requests and 65% more provider resources, with increasingly better results as the number of resource types in a consumer requests is higher. The economic efficiency is also improved, as the user welfare is also increased by our scheme by close to 30% when demand is low. However, in contrast to the previous results, the user welfare decreases as the number of resource types in a consumer request increases. Our results show that both the allocation efficiency, and the economic efficiency is increased by our scheme over simple pricing mechanisms that are currently in use.

3. **A Scalable Distributed Auctions Scheme**

   The scalability of the pricing and allocation mechanism is of great importance in large federated systems. Our theoretical analysis has shown that our pricing algorithm is a function of the number of resource types in a consumer request, and the number of providers in the market. Moreover, auctions mechanisms are inherently centralized due to the market-maker, which is responsible for receiving resource information from providers, and requests from consumers. To address these issues, we proposed a distributed auctions framework, which leverages the distributed hash table in a peer-to-peer overlay to maintain resource information based on the resource type.

   (a) *Distribute Resources by Resource Type*

   In our distributed scheme, the role of the centralized market-maker is divided between multiple users in the system. Thus, the proposed distributed auction mechanism creates a peer-to-peer overlay containing both providers and consumers, where the node closest to the identifier of a resource type, called a resource broker, is responsible for maintaining the information and computing payments for the providers of the respective resource type. Similarly, a consumer request with multiple resource types is sent to a peer in the overlay, known as the request broker, responsible of allocating each resource type, and computing the consumer payment. Providers publish resources using the P2P *store* operation, while consumers submit requests using the P2P *lookup* operation.

   (b) *Three-phase Commit Pricing Protocol*

   We proposed a three-phase distributed protocol that makes use of Lamport's logical clocks to synchronize pricing of a request between request broker and resource brokers, such that the economic properties of our scheme are main-

tained. Our experiments using an implementation on top of FreePastry, an open-source peer-to-peer overlay, have shown that our distributed auction mechanism achieves vertical scalability when the number of resource types is increase, and horizontal scalability, when the number of users is increased. Our simulations with up to 30,000 peers have shown a logarithmic increase in the allocation time for a consumer request with up to 40 resource types. The main limitation of our approach is the overhead added to the allocation by the synchronization mechanism, which results in larger allocation times in the case of consumer requests for a small number of resource types. Our theoretical and experimental analysis focused on the network delay and the request arrival rates as the main limiting factors.

4. **A Platform for Managing Resources in Federated Clouds**

   There are several practical applications of our pricing and allocation mechanism, such as in cloud computing, online auctions, online advertising or search page ranking systems. In this thesis, we have proposed SkyBoxz, a federated cloud prototype that integrates cloud infrastructure services from multiple providers, transparent to the users.

   (a) *SkyBoxz Federated Cloud*

   Cloud computing provides users with seemingly infinite capacity without up-front costs. Federated clouds integrate private and public clouds to increase elasticity and reliability. SkyBoxz, our proposed federated cloud platform, allows consumers to request different types of virtual machine instances, which can be allocated to any provider in SkyBoxz. Interoperability among providers is achieved using similar virtual machine images, which the same software stack, in each cloud. All instances created by the same user are added to a virtual private network. In addition to Infrastructure-as-a-Service, SkyBoxz can

be used to create service platforms that leverage the federated cloud. Our prototype currently provides resources from seven providers to consumers from Singapore, Vietnam, and China.

(b) *User Rationality in Federated Clouds*

Cloud providers and consumers are rational, self-interested parties that maximize their own interest. Accordingly, both providers and consumers are not be trusted to reveal their truthful valuations in the allocation of cloud resources. Current federated clouds, similar to standalone cloud providers, use fixed pricing, which does not take into account user rationality. Spot pricing, recently introduced by Amazon, takes into account the market demand. In contrast, our scheme prices resources based on both market demand and supply. We study the user rationality using spot price traces in a federated cloud composed of four Amazon EC2 regions. Our results show that, with spot pricing, currently used in EC2, rational users can improve their welfare by more than 10% when being untruthful. In contrast, our strategy-proof pricing scheme incentivizes users to be truthful.

7.2 Further Directions

The strategy-proof pricing mechanism presented in this thesis offers several future research directions, including: i) pricing with incomplete information, ii) partial strategy-proof allocations, iii) pricing mechanism with SLA extensions, and iv) incentive mechanism for resource brokers.

1. *Pricing with Incomplete Information*

   When pricing a consumer request, our mechanism requires complete information about all the providers that offer each resource type in the respective request. Without complete information, strategy-proof is no longer achieved, and economic ef-

ficiency decreases as users become untruthful. Maintaining complete information is not difficult with a centralized resource location system, where both providers and consumers submit their publish and resource requests to a centralized market-maker. However, in a distributed scheme, resource location becomes an issue due to the synchronization mechanisms required to maintain up-to-date information about all providers with the same resource types. In our distributed auction scheme, we use the lookup operation in the peer-to-peer overlay to discover providers, and the three-phase-commit protocol to ensure all resource brokers have up-to-date resource information. However, this approach leads to increased allocation times when the network delay or the request arrival rates are high. A pricing scheme that achieves a reasonable trade-off between the amount of provider resource information required for pricing and the economic properties will facilitate a reduction of the request allocation time. This scheme can be employed, for example, when there is high demand for a resource type, which can influence the allocation times for all consumer requests for that resource type. Moreover, having the possibility of using different pricing schemes for each resource types in a request allows the flexibility of achieving different trade-offs between scalability and the economic properties.

2. *Partial Strategy-proof Allocations*

   There are several cases when achieving strategy-proof may produce the same, or worse, allocation results as more efficient, but only partial strategy-proof, pricing schemes. For example, in the monopoly situation, a strategy-proof pricing scheme cannot allocate resources, since the state of the system is not incentive-compatible. In this case, only consumers can be incentivized to bid truthfully. Identifying instances where a strategy-proof pricing scheme can be substituted with other schemes can improve the allocation results and reduce the overall allocation time.

3. *Pricing with SLA Extensions*

   Our work focused on the trade-offs between the economic and computational efficiencies that influence pricing. However, there may be other requirements, in addition to the economic and computational properties, that may decide the best resource allocation. One of the user requirements when using shared resources is achieving a specific Service Level Agreement (SLA), that is, an agreement between two or more parties to share and use resources under specified limits and penalties. Thus, provisioning the allocation mechanism with methods to perform pricing based on different service levels is desirable for both providers and consumers.

4. *Incentive Mechanism for Resource Brokers*

   The key property in our pricing scheme is strategy-proof. This is because federated systems involve distributed users that are rational in maximizing their own interest. We have considered that users are either providers or consumers of shared resources, and designed our provider and consumer payment schemes to include financial incentives. We have moved further in this direction with an application that has large potential for commercial exploitation: a federated cloud platform that can be used to manage resources from multiple clouds. However, having a platform that can administer resources belonging to other users defines a new type of market participant, namely a resource broker. Similar to the provider and consumer, the resource broker is a rational user interested in maximizing his own benefit. In this context, the allocation of cloud resources requires a payment scheme that is designed to include incentives not only for providers and consumers, but also for resource brokers.

References

[1] Amazon Web Services. `http://aws.amazon.com`, Dec 2011.

[2] Microsoft Cloud Services. Windows Azure Platform. `http://www.microsoft.com/azure`, Dec 2011.

[3] I. Abraham, D. Dolev, R. Gonen, and J. Halpern. Distributed Computing Meets Game Theory: Robust Mechanisms for Rational Secret Sharing and Multiparty Computation. *Proc. of the 25th Annual ACM Symp. on Principles of Distributed Computing*, pages 53–62, Denver, USA, Jul 2006.

[4] J. Altmann, C. Courcoubetis, G. D. Stamoulis, M. Dramitinos, T. Rayna, M. Risch, and C. Bannink. GridEcon: A Market Place for Computing Resources. *Proc. of the 5th Intl. Workshop on Grid Economics and Business Models*, pages 185–196, Las Palmas de Gran Canaria, Spain, Aug 2008. Springer.

[5] Amazon Elastic Compute Cloud (Amazon EC2). `http://aws.amazon.com/ec2`, Dec 2011.

[6] Amazon Simple Storage Service (Amazon S3). `http://aws.amazon.com/s3`, Dec 2011.

[7] B. An, C. Miao, and Z. Shen. Market Based Resource Allocation with Incomplete Information. *Proc. of the 20th Intl. Joint Conf. on Artificial Intelligence*, pages 1193–1198, Hyderabad, India, Jan 2007.

[8] D. P. Anderson and G. Fedak. The Computational and Storage Potential of Volunteer Computing. *Proc. of the 6th IEEE Intl. Symp. on Cluster Computing and the Grid*, pages 73–80, Singapore, May 2006.

[9] N. Andrade, W. Cirne, F. V. Brasileiro, and P. Roisenberg. OurGrid: An Approach to Easily Assemble Grids with Equitable Resource Sharing. *Proc. of the 9th Intl. Workshop on Job Scheduling Strategies for Parallel Processing*, pages 61–86, Seattle, USA, Jun 2003.

[10] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. H. Katz, A. Konwinski, G. Lee, D. A. Patterson, A. Rabkin, I. Stoica, and M. Zaharia. Above the Clouds: A Berkeley View of Cloud Computing. Technical Report UCB/EECS-2009-28, EECS Department, University of California, Berkeley, Feb 2009.

[11] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. H. Katz, A. Konwinski, G. Lee, D. A. Patterson, A. Rabkin, I. Stoica, and M. Zaharia. A View of Cloud Computing. *Communications of the ACM*, 53(4):50–58, Apr 2010.

[12] A. Auyoung, B. Chun, A. Snoeren, and A. Vahdat. Resource Allocation in Federated Distributed Computing Infrastructures. *Proc. of the First Workshop on Operating System and Architectural Support for the On-demand IT InfraStructure*, Sep 2004.

[13] P. Belzarena, A. Ferragut, and F. Paganini. Network Bandwidth Allocation via Distributed Auctions with Time Reservations. *Proc. of the 28th IEEE Intl. Conf. on Computer Communications*, pages 2816–2820, Rio de Janeiro, Brazil, Apr 2009.

[14] O. A. Ben-Yehuda, M. Ben-Yehuda, A. Schuster, and D. Tsafrir. Deconstructing Amazon EC2 Spot Instance Pricing. *Proc. of the 3rd Intl. Conf. on Cloud Computing*, page (to appear), Athens, Greece, Nov 2011.

[15] M. Blaugh. *Economic Theory in Retrospect*. Cambridge University Press, 1997.

[16] R. B. Bohn, J. Messina, F. Liu, J. Tong, and J. Mao. NIST cloud computing reference architecture. *World Congress on Services*, pages 594–596, Washington, DC, USA, Jul 2011.

[17] R. Buyya, D. Abramson, and J. Giddy. Nimrod/G: An Architecture of a Resource Management and Scheduling System in a Global Computational Grid. *Proc. of the 4th Intl. Conf. on High Performance Computing in Asia-Pacific Region*, pages 283–289, Beijing, China, May 2000.

[18] R. Buyya, D. Abramson, J. Giddy, and H. Stockinger. Economic Models for Resource Management and Scheduling in Grid Computing. *Concurrency and Computation: Practice and Experience*, 14(13–15):1507–1542, 2002.

[19] R. Buyya, R. Ranjan, and R. N. Calheiros. InterCloud: Utility-Oriented Federation of Cloud Computing Environments for Scaling of Application Services. *Proc. of the 10th Intl. Conf. on Algorithms and Architectures for Parallel Processing*, pages 13–31, Busan, Korea, May 2010.

[20] R. Buyya, C. S. Yeo, and S. Venugopal. Market-Oriented Cloud Computing: Vision, Hype, and Reality for Delivering IT Services as Computing Utilities. *Proc. of the 10th IEEE Intl. Conf. on High Performance Computing and Communications*, pages 5–13, Dalian, China, Sep 2008.

[21] J. Carolan, S. Gaede, J. Baty, G. Brunette, A. Licht, J. Remmell, L. Tucker, and J. Weise. Introduction to Cloud Computing Architecture - White Paper. *Sun Microsystems*, pages 1–40, Jun 2009.

[22] M. Castro, P. Druschel, A.-M. Kermarrec, J. J. T. Close, and A. Rowstron. One Ring to Rule them All: Service Discovery and Binding in Structured Peer-to-Peer Overlay Networks. *Proc. of the 10th ACM SIGOPS European Workshop*, pages 140–145, Saint-Emilion, France, Jul 2002.

[23] M. Castro and B. Liskov. Proactive recovery in a Byzantine-fault-tolerant system. *Proc. of the 4th Symp. on Operating Systems Design and Implementation*, San Diego, USA, Oct 2000.

[24] M. Castro and B. Liskov. Practical Byzantine Fault Tolerance and Proactive Recovery. *TOCS*, 20(4):398–461, Nov. 2002.

[25] B. Chun, Y. Fu, and A. Vahdat. Bootstrapping a Distributed Computational Economy with Peer-to-Peer Bartering. *Workshop on Economics of Peer-to-Peer Systems*, Berkley, USA, Jun 2003. `http://www2.sims.berkeley.edu/research/conferences/p2pecon/papers/s7-chun.pdf`.

[26] B.-G. Chun, R. Fonseca, I. Stoica, and J. Kubiatowicz. Characterizing Selfishly Constructed Overlay Routing Networks. *Proc. of the 23th IEEE Intl. Conf. on Computer Communications*, pages 1329–1339, Hong Kong, China, Mar 2004.

[27] B. N. Chun, P. Buonadonna, A. AuYoung, C. Ng, D. Parkes, J. Shneidman, A. Snoeren, and A. Vahdat. Mirage: a Microeconomic Resource Allocation System for Sensornet Testbeds. *Proc. of the Second IEEE Workshop on Embedded Networked Sensors*, pages 19–28, Syndey, Australia, May 2005.

[28] B. N. Chun and D. E. Culler. Market-based Proportional Resource Sharing for Clusters. Technical Report UCB/CSD-00-1092, EECS Department, University of California, Berkeley, Sep 1999.

[29] E. H. Clarke. Multipart Pricing of Public Goods. *Public Choice*, 11(1):17–33, Sep 1971.

[30] Cloud Exchange. `http://cloudexchange.org`, Dec 2011.

[31] CloudTweaks. 3 Quick Ways To Reduce Your Amazon EC2 Cloud Charges. `http://www.cloudtweaks.com/2011/01/3-quick-ways-to-reduce-your-amazon-ec2-cloud-charges`, Dec 2011.

[32] B. Cohen. Incentives Build Robustness in BitTorrent. *Workshop on Economics of Peer-to-Peer Systems*, Berkley, USA, Jun 2003. `http://www2.sims.berkeley.edu/research/conferences/p2pecon/papers/s4-cohen.pdf`.

[33] ComputeNext. Buy and Sell Cloud Resources. `http://computenext.com`, Dec 2011.

[34] P. Cramton, Y. Shoham, and R. Steinberg, editors. *Combinatorial Auctions*. The MIT Press, 2006.

[35] A. Danak and S. Mannor. Resource Allocation with Supply Adjustment in Distributed Computing Systems. *Proc. of the 30th IEEE Intl. Conf. on Distributed Computing Systems (ICDCS'10)*, pages 498–506, Genova, Italy, Jun 2010.

[36] R. K. Dash, N. R. Jennings, and D. C. Parkes. Computational Mechanism Design: A Call to Arms. *IEEE Intelligent Systems*, 18(6):40–47, Nov 2003.

[37] S. de Vries and R. V. Vohra. Combinatorial Auctions: A Survey. *INFORMS Journal on Computing*, 15(3):284–309, Feb 2003.

[38] Drupal. Open Source CMS. `http://drupal.org`, Dec 2011.

[39] E. Elkind. True Costs of Cheap Labor Are Hard to Measure: Edge Deletion and VCG Payments in Graphs. *Proc. of the 6th ACM Conf. on Electronic Commerce*, pages 108–116, Vancouver, Canada, Jun 2005.

[40] C. Ernemann, V. Hamscher, and R. Yahyapour. Economic Scheduling in Grid Computing. *Proc. of the 8th Intl. Workshop on Job Scheduling Strategies for Parallel Processing*, pages 128–152, Edinburgh, UK, Jul 2002.

[41] Eucalyptus. The Open Source Cloud Platform. `http://open.eucalyptus.com`, Dec 2011.

[42] T. Eymann, M. Reinicke, O. Ardaiz, P. Artigas, L. D. de Cerio, F. Freitag, R. Messeguer, L. Navarro, D. Royo, and K. Sanjeevan. Decentralized vs. Centralized Economic Coordination of Resource Allocation in Grids. *First European Across Grids Conference*, pages 9–16, Santiago de Compostela, Spain, Feb 2003.

[43] J. Feigenbaum, C. H. Papadimitriou, and S. Shenker. Sharing the Cost of Multicast Transmissions. *Journal of Computer and System Sciences*, 63(1):21–41, Aug 2001.

[44] J. Feigenbaum, R. Sami, and S. Shenker. Mechanism Design for Policy Routing. *Distributed Computing*, 18(4):293–305, Mar 2006.

[45] J. Feigenbaum and S. Shenker. Distributed Algorithmic Mechanism Design: Recent Results and Future Directions. *Proc. of the 6th Intl. Workshop on Discrete Algorithms and Methods for Mobile Computing and Communications*, pages 1–13, Atlanta, USA, Sep 2002.

[46] M. Feldman, K. Lai, and L. Zhang. A Price-Anticipating Resource Allocation Mechanism for Distributed Shared Clusters. *Proc. of the 6th ACM Conference on Electronic Commerce*, pages 127–136, Vancouver, Canada, Jun 2005.

[47] I. T. Foster. Globus Toolkit Version 4: Software for Service-Oriented Systems. *Journal of Computer Science and Technology*, 21(4):513–520, Jul 2006.

[48] Y. Fu, J. S. Chase, B. N. Chun, S. Schwab, and A. Vahdat. SHARP: An architecture for secure resource peering. *Proc. of the 19th ACM Symp. on Operating Systems Principles*, pages 133–148, Bolton Landing, USA, Oct 2003.

[49] D. Fudenberg and J. Tirole. *Game Theory*. MIT Press, 1991.

[50] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, USA, 1990.

[51] A. F. Gibbard. Manipulation of Voting Schemes: A General Result. *Econometrica*, 41(4):587–601, 1973.

[52] J. Gomoluch and M. Schroeder. Market-Based Resource Allocation for Grid Computing: A Model and Simulation. *Proc. of the Intl. Middleware Conf.*, pages 211–218, Rio de Janeiro, Brazil, Jun 2003.

[53] L. M. V. Gonzalez, L. Rodero-Merino, J. Caceres, and M. A. Lindner. A Break in the Clouds: Towards a Cloud Definition. *Computer Communication Review*, 39(1):50–55, Jan 2009.

[54] D. Gottfrid. Self-service, Prorated Super Computing Fun! `http://open.blogs.nytimes.com/2007/11/01/self-service-prorated-super-computing-fun`, Dec 2011.

[55] P. Gradwell, M. A. Oey, R. J. Timmer, F. M. T. Brazier, and J. A. Padget. Engineering Large-scale Distributed Auctions. *Proc. of the 7th Intl. Joint Conference on Autonomous Agents and Multiagent Systems*, pages 1311–1314, Estoril, Portugal, May 2008.

[56] P. Gradwell and J. A. Padget. A Comparison of Distributed and Centralised Agent Based Bundling Systems. *Proc. of the 9th Intl. Conf. on Electronic Commerce*, pages 25–34, Minneapolis, USA, Aug 2007.

[57] T. Groves. Incentives in Teams. *Econometrica*, 41(4):617–631, Jul 1973.

[58] A. Gupta, D. Agrawal, and A. E. Abbadi. Distributed Resource Discovery in Large Scale Computing Systems. *Proc. of the 2005 IEEE/IPSJ Intl. Symp. on Applications and the Internet*, pages 320–326, Trento, Italy, Jan 2005.

[59] M. Hauswirth and R. Schmidt. An Overlay Network for Resource Discovery in Grids. *Proc. of the 16th Intl. Workshop on Database and Expert Systems Applications*, pages 343–348, Copenhagen, Denmark, Aug 2005.

[60] J. Huang, R. A. Berry, and M. L. Honig. Auction-Based Spectrum Sharing. *Mobile Networks and Applications*, 11(3):405–418, Jun 2006.

[61] P. Huang, H. Peng, P. Lin, and X. Li. Macroeconomics based Grid Resource Allocation. *Future Generation Computer Systems*, 24(7):694–700, Jul 2008.

[62] A. Iamnitchi, I. T. Foster, and D. Nurmi. A Peer-to-Peer Approach to Resource Location in Grid Environments. *Proc. of the 11th IEEE Intl. Symp. on High Performance Distributed Computing*, page 419, Washington, USA, Jul 2002.

[63] B. Javadi, R. K. Thulasiram, and R. Buyya. Statistical Modeling of Spot Instance Prices in Public Cloud Environments. *Proc. of the 4th IEEE Intl. Conf. on Utility and Cloud Computing*, page (to appear), Melbourne, Australia, Dec 2011.

[64] J. Kappel, T. J. Velte, and A. T. Velte. *Microsoft virtualization with Hyper-V*. McGraw-Hill, 2009.

[65] K. Keahey, M. Tsugawa, A. Matsunaga, and J. A. B. Fortes. Sky Computing. *IEEE Internet Computing*, 13(5):43–51, Sept. 2009.

[66] P. Klemperer. Auction Theory: A Guide to the Literature. *Journal of Economic Surveys*, 13(3):227–286, 1999.

[67] Kuhn, H. W. The Hungarian method of solving the assignment problem. *Naval Res. Logistics Quart.*, 2:83–97, 1955.

[68] J. F. Kurose and R. Simha. A Microeconomic Approach to Optimal Resource Allocation in Distributed Computer Systems. *IEEE Transactions on Computers*, 38(5):705–717, May 1989.

[69] K. Lai. Markets Are Dead, Long Live Markets. *SIGecom Exchanges*, 5(4):1–10, 2005.

[70] K. Lai, B. A. Huberman, and L. R. Fine. Tycoon: A Distributed Market-based Resource Allocation System. Technical Report cs.DC/0404013, HP Labs, Palo Alto, USA, April 2004.

[71] K. Lai, L. Rasmusson, E. Adar, L. Zhang, and B. A. Huberman. Tycoon: An Implementation of a Distributed, Market-based Resource Allocation System. *Multiagent and Grid Systems*, 1(3):169–182, 2005.

[72] L. Lin, Y. Zhang, and J. Huai. Sustaining Incentive in Grid Resource Allocation: A Reinforcement Learning Approach. *Proc. of the 7th IEEE Intl. Symp. on Cluster Computing and the Grid*, pages 145–154, Rio de Janeiro, Brazil, May 2007.

[73] D. Lucking-Reiley. Vickrey Auctions in Practice: From Nineteenth-Century Philately to Twenty-First-Century E-Commerce. *Journal of Economic Perspectives*, 14:183–192, 2000.

[74] M. Mihailescu and Y. M. Teo. Dynamic Resource Pricing on Federated Clouds. *Proc. of the 10th IEEE/ACM Intl. Conf. on Cluster, Cloud and Grid Computing*, pages 513–517, Merlbourne, Australia, May 2010.

[75] D. S. Milojicic, I. M. Llorente, and R. S. Montero. Opennebula: A cloud management tool. *IEEE Internet Computing*, 15(2):11–14, 2011.

[76] D. S. Milojicic and R. Wolski. Eucalyptus: Delivering a private cloud. *IEEE Computer*, 44(4):102–104, 2011.

[77] R. B. Myerson and M. A. Satterthwaite. Efficient Mechanisms for Bilateral Trading. *Journal of Economic Theory*, 29(2):265–281, 1983.

[78] n2n. Layer Two Peer-to-Peer VPN. `http://www.ntop.org/products/n2n/`, Dec 2011.

[79] D. Neumann, B. Schnizler, I. Weber, and C. Weinhardt. Second-Best Combinatorial Auctions - The Case of the Pricing-Per-Column Mechanism. *Proc. of the 40th Hawaii Intl. Conf. on Systems Science*, page 33, Waikoloa, USA, Jan 2007.

[80] G. F. Newell. *Applications of Queuing Theory*. Chapman and Hall, 1982.

[81] T.-W. J. Ngan, D. S. Wallach, and P. Druschel. Enforcing Fair Sharing of Peer-To-Peer Resources. *2nd Intl. Workshop on Peer-to-Peer Systems*, volume 2735 of *LNCS*, pages 16–26, Berkley, USA, February 2003.

[82] S. J. Nielson, S. A. Crosby, and D. S. Wallach. A Taxonomy of Rational Attacks. *Proc. of the 4th Intl. Workshop on Peer-to-Peer Systems*, volume 3640, pages 36–46, Ithaca, USA, Feb 2005.

[83] N. Nisan. Bidding and Allocation in Combinatorial Auctions. *Proc. of the 2nd ACM Conf. on Electronic Commerce*, pages 1–12, Minneapolis, USA, Sep 2000.

[84] N. Nisan and A. Ronen. Algorithmic Mechanism Design. *Proc. of the 31st ACM Symp. on Theory of Computing*, pages 129–140, Atlanta, USA, May 1999.

[85] T. Omtzigt. High-Productivity Cloud computing: Federated Clouds. `http://stillwater-cse.blogspot.com/2008/06/federated-clouds.html`, Jun 2008.

[86] OpenNebula. The Open Source Toolkit for Data Center Virtualization. `http://opennebula.org`, Dec 2011.

[87] Parallel Workload Archives. LLNL Thunder Log. `http://www.cs.huji.ac.il/labs/parallel/workload`, Dec 2011.

[88] D. C. Parkes, J. R. Kalagnanam, and M. Eso. Achieving Budget-Balance with Vickrey-Based Payment Schemes in Exchanges. *Proc. of the 17th Intl. Conf. on Artificial Intelligence*, pages 1161–1168, Seattle, USA, Aug 2001.

[89] S. Parsons, J. A. Rodriguez-Aguilar, and M. Klein. Auctions and bidding: A guide for computer scientists. *ACM Computing Surveys*, 43(2):10:1–10:59, Jan 2011.

[90] Pastry. A scalable, decentralized, self-organizing and fault-tolerant substrate for peer-to-peer applications. http://freepastry.org, Dec 2011.

[91] H. N. Pham, Y. M. Teo, N. Thoai, and T. A. Nguyen. An Approach to Vickrey-based Resource Allocation in the Presence of Monopolistic Sellers. *7th Australasian Symp. on Grid Computing and e-Research*, volume 99 of *CRPIT*, pages 77–83, Wellington, New Zealand, 2009.

[92] A. Pikovsky, P. Shabalin, and M. Bichler. Iterative Combinatorial Auctions with Linear Prices: Results of Numerical Experiments. *Proc. of the 8th IEEE Intl. Conf. on E-Commerce Technology (CEC 2006) and the 3rd IEEE International Conference on Enterprise Computing, E-Commerce, and E-Services*, pages 39–49, Palo Alto, USA, Jun 2006.

[93] PlanetLab. An Open Platform for Developing, Deploying, and Accessing Planetary-scale Services. http://www.planet-lab.org, Dec 2011.

[94] V. Puranik. 7 Easy Tips to Reduce Your Amazon EC2 Cloud Costs. http://aws-musings.com/7-easy-tips-to-reduce-your-amazon-ec2-cloud-costs, Dec 2011.

[95] R. Rajkumar, C. Lee, J. P. Lehoczky, and D. P. Siewiorek. Practical Solutions for QoS-based Resource Allocation Problems. *Proc. of the 19th IEEE Real-Time Systems Symp.*, pages 296–306, Madrid, Spain, Dec 1998.

[96] R. Ranjan and R. Buyya. *Hanbook of Research on Scalable Computing Technologies*, chapter Decentralized Overlay for Federation of Enterprise Clouds. ICI Global, Jul 2009.

[97] O. Regev and N. Nisan. The POPCORN market. Online markets for computational resources. *Decision Support Systems*, 28(1-2):177–189, 2000.

[98] S. Ried and H. Kisker. Sizing The Cloud. Forrester Research, http://forrester.com/rb/Research/sizing_cloud/q/id/58161/t/2, Apr 2011.

[99] RightScale. More servers, bigger servers, longer servers, and 10x of that! http://blog.rightscale.com/2010/08/04/more-bigger-longer-servers-10x, Dec 2011.

[100] B. Rochwerger, D. Breitgand, E. Levy, A. Galis, K. Nagin, I. M. Llorente, R. S. Montero, Y. Wolfsthal, E. Elmroth, J. A. Cáceres, M. Ben-Yehuda, W. Emmerich, and F. Galán. The RESERVOIR Model and Architecture for Open Federated Cloud Computing. *IBM Journal of Research and Development*, 53(4):4, Jul 2009.

[101] G. Rosen. Anatomy of an Amazon EC2 Resource ID. http://www.jackofallclouds.com/2009/09/anatomy-of-an-amazon-ec2-resource-id, Dec 2011.

[102] A. I. T. Rowstron and P. Druschel. Pastry: Scalable, Decentralized Object Location, and Routing for Large-Scale Peer-to-Peer Systems. *IFIP/ACM Intl. Conf. on Distributed Systems Platforms*, volume 2218 of *LNCS*, pages 329–350, Heidelberg, Germany, Nov 2001.

[103] T. W. Sandholm. An Algorithm for Optimal Winner Determination in Combinatorial Auctions. *Proc. of the 16th Intl. Joint Conference on Artificial Intelligence*, pages 542–547, San Francisco, USA, Jul 1999.

[104] ScaleUp Technologies. Developer of a self-service cloud management platform. `http://scaleup.it`, Dec 2011.

[105] B. Schnizler. Java Combinatorial Auction Simulation Environment. `http://www.schnizler.com/jcase.html`.

[106] P. Sempolinski and D. Thain. A Comparison and Critique of Eucalyptus, OpenNebula and Nimbus. *Proc. of the Second Intl. Conf. on Cloud Computing*, pages 417–426, Indianapolis, USA, Dec 2010.

[107] Y. Sheffi. Combinatorial Auctions in the Procurement of Transportation Services. *Interfaces*, 34(4):245–252, 2004.

[108] J. Sherwani, N. Ali, N. Lotia, Z. Hayat, and R. Buyya. Libra: a Computational Economy-based Job Scheduling System for Clusters. *Software, Practice and Experience*, 34(6):573–590, May 2004.

[109] J. Shneidman and D. C. Parkes. Rationality and Self-Interest in Peer to Peer Networks. *Proc. of the 2nd Intl. Workshop on Peer-to-Peer Systems*, pages 139–148, Berkely, USA, Feb 2003.

[110] B. K. M. Sim and K. Mong. A Survey of Bargaining Models for Grid Resource Allocation. *SIGecom Exchanges*, 5(5):22–32, Jan 2006.

[111] SkyBoxz. Federated Cloud Management Platform. `http://skyboxz.com`, Dec 2011.

[112] SpotCloud. Cloud Capacity Clearing House. `http://spotcloud.com`, Dec 2011.

[113] I. Stoica, H. M. Abdel-Wahab, and A. Pothen. A Microeconomic Scheduler for Parallel Computers. *Proc. of the IPPS '95 Workshop on Job Scheduling Strategies for Parallel Processing*, pages 200–218, Santa Barbara, USA, Apr 1995.

[114] I. Stoica, R. Morris, D. Karger, F. Kaashoek, and H. Balakrishnan. Chord: A Scalable Peer-To-Peer Lookup Service for Internet Applications. *Proc. of the ACM SIGCOMM 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, pages 149–160, New York, USA, Aug 2001.

[115] Sun. Grid Compute Utility. `http://www.network.com`, Mar 2010.

[116] Y. M. Teo, V. March, and M. Mihailescu. *Handbook of Research on Scalable Computing Technologies*, chapter Hierarchical Structured Peer-to-Peer Systems. IGI Global, Jul 2009.

[117] Y. M. Teo and M. Mihailescu. Collision Avoidance in Hierarchical Peer-to-Peer Systems. *Proc. of the 7th Intl. Conf. on Networking*, pages 336–341, Cancun, Mexico, Apr 2008.

[118] A. N. Toosi, R. N. Calheiros, R. K. Thulasiram, and R. Buyya. Resource Provisioning Policies to Increase IaaS Provider's Profit in a Federated Cloud Environment. *Proc. of the 13th IEEE Intl. Conf. on High Performance Computing & Communication*, pages 279–287, Banff, Canada, Sep 2011.

[119] M. Verloop and R. Núñez-Queija. Assessing the Efficiency of Resource Allocations in Bandwidth-Sharing Networks. *Performance Evaluation*, 66(1):59–77, Jan 2009.

[120] W. Vickrey. Counterspeculation, Auctions, and Competitive Sealed Tenders. *The Journal of Finance*, 16(1):8–37, Mar 1961.

[121] C. A. Waldspurger, T. Hogg, B. A. Huberman, J. O. Kephart, and W. S. Stornetta. Spawn: A Distributed Computational Economy. *IEEE Transactions on Software Engineering*, 18(2):103–117, Feb 1992.

[122] C. A. Waldspurger and W. E. Weihl. Lottery Scheduling: Flexible Proportional-Share Resource Management. *Proc. of the First USENIX Symposium on Operating Systems Design and Implementation*, pages 1–11, Monterey, USA, Nov 1994.

[123] H. Wang, Q. Jing, R. Chen, B. He, Z. Qian, and L. Zhou. Distributed Systems Meet Economics: Pricing in the Cloud. *Proc. of the 2nd USENIX Workshop on Hot Topics in Cloud Computing*, Boston, USA, Jun 2010.

[124] R. Wolski, J. Brevik, J. S. Plank, and T. Bryan. *Grid Computing: Making The Global Infrastructure a Reality*, chapter Grid Resource Allocation and Control Using Computational Economies, pages 747–772. John Wiley & Sons, 2003.

[125] R. Wolski, J. S. Plank, J. Brevik, and T. Bryan. Analyzing Market-Based Resource Allocation Strategies for the Computational Grid. *The International Journal of High Performance Computing Applications*, 15(3):258–281, Aug 2001.

[126] R. Wolski, J. S. Plank, J. Brevik, and T. Bryan. G-commerce: Market Formulations Controlling Resource Allocation on the Computational Grid. *Proc. of the 15th Intl. Parallel & Distributed Processing Symp.*, pages 46–54, San Francisco, USA, Apr 2001.

[127] C. Wu, B. Li, and Z. Li. Dynamic Bandwidth Auctions in Multioverlay P2P Streaming with Network Coding. *IEEE Transactions on Parallel and Distributed Systems*, 19(6):806–820, Jun 2008.

[128] Xen hypervisor, the powerful open source industry standard for virtualization. `http://www.xen.org`, Dec 2011.

[129] C. S. Yeo and R. Buyya. A Taxonomy of Market-based Resource Management Systems for Utility-driven Cluster Computing. *Software: Practice and Experience*, 36(13):1381–1419, Nov 2006.

[130] S. Zaman and D. Grosu. Combinatorial Auction-Based Allocation of Virtual Machine Instances in Clouds. *Proc. of the Second Intl. Conf. on Cloud Computing*, pages 127–134, Indianapolis, USA, Nov 2010.

[131] M. M. Zavlanos, L. Spesivtsev, and G. J. Pappas. A distributed auction algorithm for the assignment problem. *Proc. of the 47th IEEE Conf. on Decision and Control*, pages 1212–1217, Cancun, Mexico, Dec 2008.

Appendix A

Impact of Rationality in English Auctions

One-sided auctions are preferred by many providers because they are easy to implement and may increase their payments due to the *winner's curse*, which results in the consumer payment being higher than the resource value [89]. Winner's curse occurs in one-sided auctions that have no incentives for users to declare their truthful valuations, such as the English auction. In this section, we study the impact of untruthfulness in the case of English auction by looking at the variation of the number of successful requests in different scenarios. We use our simple auctions simulator in a balanced market, where an event has equal probability to be a provider resource publish or a consumer request. For simplicity, we consider one item from one resource types. The simulator runs with a number of 1,000 users, where a user is both provider and consumer. We vary the number of untruthful users from 10% to 30%, indicated by *untruthful*. An untruthful user declares a false reserved price, which differs from the truthful price according to a uniform distribution with the mean given by *price change*. The truthful price is sampled from an exponential distribution. Figure A.1 shows our results, with a price change of 10% and 20% for each number of untruthful users, 10% and 30%

respectively. We contrast these results with the number of allocations obtained by a strategy-proof auction, given by *truthful*. We can see that the number of successful requests degrades proportionally with the number of untruthful users in the system. For example, the number of successful consumer requests is similar when 10% of users are untruthful, both when the price change was 10% and 20%, and close to 90% of the number of successful requests performed by the strategy-proof scheme. Similarly, for 30% untruthful users, the number of successful requests was close to 70% of the ideal case. Our experiment shows that there is a direct relation between the number of successful requests and the degree of untruthful users in the system. This result motivates the need of a strategy-proof mechanism where participants are incentivized to be truthful.

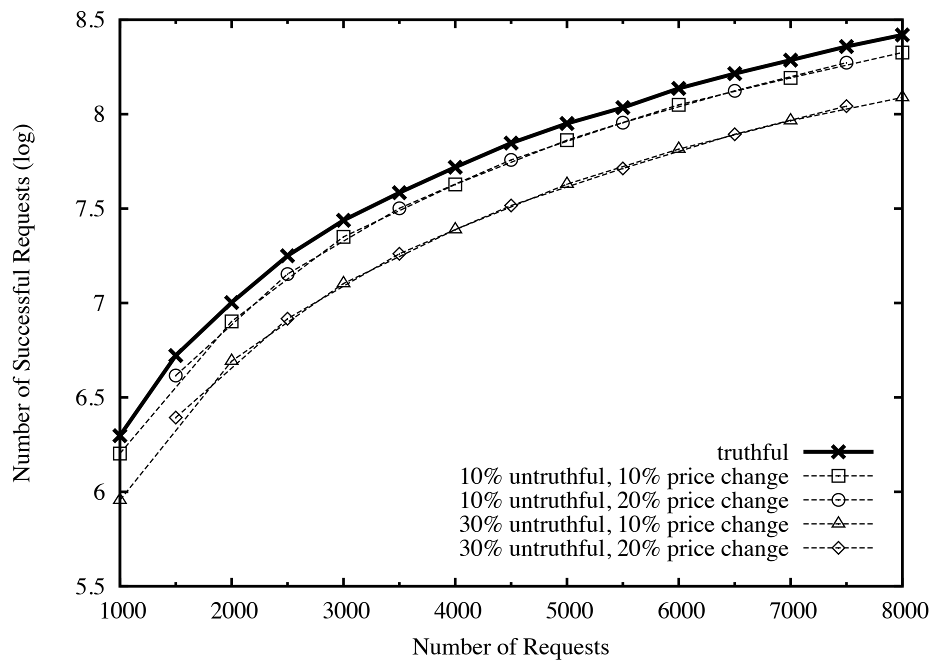


Figure A.1: Varying the Number of Untruthful Users

Appendix B

Prototyping Using FreePastry

FreePastry [90] is an open-source Java implementation of the Pastry protocol that can be deployed over the Internet. We have implemented a prototype of our pricing mechanism as a FreePastry application, and tested the deployment on PlanetLab. In addition, FreePastry allows running applications in a simulator with minimum code change. Figure B.1 shows the minimum code to start an overlay network. This is achieved by setting up the factory that generates the peer node identifiers (`nidFactory`), which is used then to create a factory that generates the nodes (`factory`). Finally, a node can be created using the factory, and booted. If the application is running in the simulator, which is indicated by the flag `SIMULATOR`, a simulator object is created which is used by the node factory instead of the public port used for communication. After the node is booted, the application can be started on the node. FreePastry applications are created by implementing the `Application` interface in the `commonapi` package. All the classical peer-to-peer operations are made available by FreePastry to the application. For example, a message from an application on the current node can be routed by the overlay to the node responsible by an identifier `id` using the method `node.route(Id, Message, null)`. The last parameter represents a node handle, thus the `route` method can be used to send a direct message to a node in the overlay. We make use of the

```
// set-up random NodeIds
NodeIdFactory nidFactory = new RandomNodeIdFactory(env);

// construct the PastryNodeFactory
PastryNodeFactory factory;
if (SIMULATOR) {
    NetworkSimulator simulator = new EuclideanNetwork(env);
    simulator.setMaxSpeed(1.0f);
    factory = new DirectPastryNodeFactory(nidFactory, simulator, env);
} else {
    factory = new SocketPastryNodeFactory(nidFactory, bindport, env);
}

// construct a node using the factory
PastryNode node = factory.newNode();

// boot node, start ring with NULL bootaddress
if (SIMULATOR) {
    bootaddress = node.getLocalHandle();
}
node.boot(bootaddress);
if (SIMULATOR) {
    simulator.setFullSpeed();
}
```

Figure B.1: Initializing a FreePastry Overlay

overlay routing when publishing resources and requests, and when performing resource lookups, while the subsequent communication between nodes is using direct messages.

When running our FreePastry application, the user is presented with a prompt, where he can type several commands, including:

1. `load <file>`, to load and execute an input file
2. `pause`, in the simulator, to pause the execution
3. `resume`, in the simulator, to resume the execution
4. `dynamic`, to enable dynamic pricing
5. `fixed`, to enable fixed pricing

6. `statistics`, on the root node, to show the number of peers in the overlay, the number of provider publish events, the number of allocated provider resources, the total and successful number of consumer requests, the total welfare, and total allocation time
7. `loglevel <N>`, to set the desired level of verbosity for the log messages displayed on the console
8. `exit`, to quit the application

The input to the application is given in an XML file, which is interpreted using the `simple-xml` v.1.6.2 and `stax` v.1.2 Java libraries. An example of an input file is given below in Figure B.2. The `types` tag defines the total number of resources in the system. The `bundle` tag is used to specify the number of resource types in a consumer, and the number of resource types published by a provider, respectively. For example, in Figure B.2, each consumer request or provider publish will have a number of resource types generated randomly between 1 and 5. The `items` tag represents the number of items for each type in the consumer request or provider publish. Next, the `price` tag represents the price per item. For a consumer request, the total price is computed as the sum of all the items of each resource type in the request. The publish and request events are generated using the `events` tag, which specifies the number of events, the interarrival time, and the probability for the event to be a user join, user leave, provider publish or consumer request. In addition to generated events, we allow the user to manually add specific events using the `event` tag, with the simulator time when the event will run, the type of the event, such as `publish` or `request`, the node id that will trigger the event, and the list of resources.

```xml
<workload simulator="DynamicPricing" version="2.0">
   <resource>
      <types class="nus.soc.cslab.FixedValue">
         <value>10</value>
      </types>
      <bundle class="nus.soc.cslab.UniformDistribution" seed="2010">
         <low>1</low>
         <high>5</high>
      </bundle>
      <items class="nus.soc.cslab.ExponentialDistribution" seed="2010">
         <mean>10</mean>
      </items>
      <price class="nus.soc.cslab.UniformDistribution" seed="2010">
         <low>80</low>
         <high>120</high>
      </price>
   </resource>
   <events number="10000" interarrival="100">
      <join>100</join>
      <leave>0</leave>
      <publish>0</publish>
      <request>0</request>
   </events>
   <events number="600000" interarrival="100">
      <join>0</join>
      <leave>0</leave>
      <publish>50</publish>
      <request>50</request>
   </events>
   <event time="5000">
      <publish uid="1">
         <resource type="1" items="3" price="95"/>
         <resource type="2" items="3" price="90"/>
      </publish>
   </event>
   <event time="5500">
      <request uid="2">
         <resource type="1" items="2" price="100"/>
         <resource type="2" items="2" price="100"/>
      </request>
   </event>
</workload>
```

Figure B.2: Input File for the FreePastry Prototype

Appendix C

SkyBoxz Organization

Each SkyBoxz [111] account is assigned to a user role, which can be *consumer*, *provider*, or *developer*. The consumer role has the basic functions of creating requests, running jobs and monitoring the state. In addition to the above, a provider can add new clouds, administer the virtual machine images, and modify the instance types available for each cloud, such as the number of cores, memory, disk, and price. Lastly, the developer can modify the implementation of SkyBoxz and add new application platforms that connect using the PHP API.

1. **Consumer**

   (a) **Jobs** – shows the status of the latest jobs started by the consumer. If there are no active jobs, it presents the user to start a new job

   (b) **Resources** – shows a list with the current request and, for each request, the running instance details, such as instance type, image, IP address, and cloud provider (as in Figure C.1). Allows the user to inspect monitoring information for each instance type (example in Figure C.2)

   (c) **Clouds** – displays the list with the clouds accessible to the user, together with the instance types in each cloud (see Figure C.3)

(d) **Terminal Login** – displays a list with the running instances, and allows the user to start a remote shell on the selected instance

(e) **Acquire Resources** – the user selects the number of instance types in the request, the cloud (by default it can allocate on all clouds), the pricing policy (fixed pricing or dynamic pricing), the reserved price, and, for each instance type, the number of cores, size of memory and storage, VM image, and number of items (see Figure C.4)

(f) **Projects** – presents the list with the user projects and the status of the latest jobs. If resources have been acquired, the user can start a job by selecting the allocated request identifier (Figure C.5)

(g) **Add Project** – The user can create new projects by uploading source files and specifying the commands of the job

2. **Provider**

   (a) **Add Cloud** – the provider adds a new cloud by specifying the cloud access point IP address and protocol, and access credentials for the users (under development)

   (b) **VM Images** – allows the provider to define define virtual machine images

   (c) **VM Settings** – displays a list with the instance type for each cloud, where the provider can modify the number of cores, memory size, storage, number of items available, and the price per item

3. **Developer**

   (a) **Add Content** – the developer can modify existing pages, add new pages and new applications (example of SNAP test results in Figure C.6)

SkyBoxz

LATEST NEWS | WHY SKYBOXZ | HOW IT WORKS | FORUMS | ABOUT US

You are logged in as: demo with a balance of: V$1000.

Service Provider

PROVIDER

- VM Images
- VM Settings

DASHBOARD

- Cloud Status
  - Jobs
  - Resources
  - Clouds
- Direct Access
  - Terminal Login
  - Acquire Resource
- My Workspace
  - Projects
  - Add Project
- User Setting
  - SkyBoxz Account
  - Log Out

Home » Cloud Status

Resources

Resouce ID:211

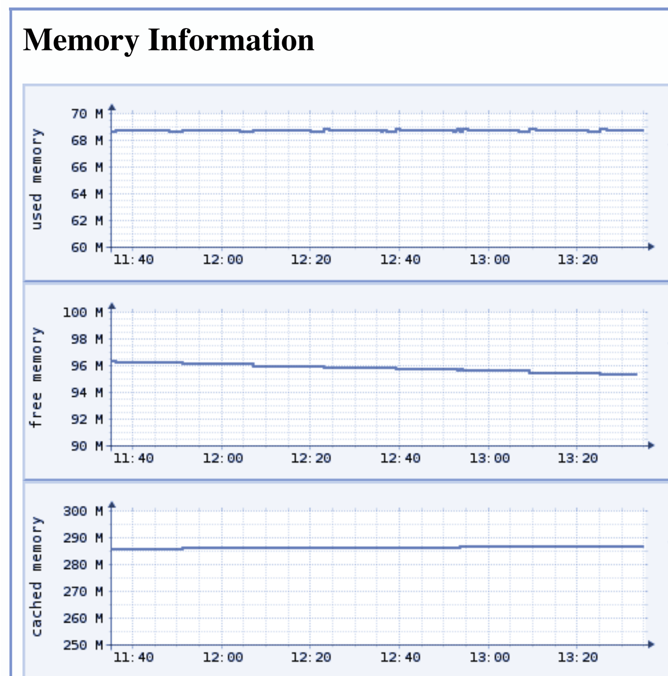
View Status

from **2012-01-08 17:30:23**

	Instance Id	Instance Type	VM Image	IP Address	Cloud	Status
☐	i-535608E9	m1.small	emi-1F7E1575	192.168.1.106	soc-nimbus	View Detail
☐	i-35D5070B	m1.large	emi-1F7E1575	192.168.1.104	soc-nimbus	View Detail
☐	i-37E0064F	m1.small	emi-1F7E1575	192.168.1.105	soc-nimbus	View Detail

Terminate selected instances

Figure C.1: Resources on SkyBoxz



SkyBoxz Instance (ID=i-535608E9) Memory Information

Page 2 of 3

Figure C.2: Monitoring on SkyBoxz

SkyBoxz

LATEST NEWS | WHY SKYBOXZ | HOW IT WORKS | FORUMS | ABOUT US

You are logged in as: demo with a balance of: V$1000.

Service Provider

PROVIDER
- **VM Images**
- **VM Settings**

DASHBOARD
- **Cloud Status**
  - **Jobs**
  - **Resources**
  - **Clouds**
- **Direct Access**
  - **Terminal Login**
  - **Acquire Resource**
- **My Workspace**
  - **Projects**
  - **Add Project**
- **User Setting**
  - **SkyBoxz Account**
  - **Log Out**

Home » Cloud Status

Clouds

- SOC-NIMBUS
- AMAZON-US-EAST
- CLOUD-OF-SARI

SOC-NIMBUS	CPU (cores)	RAM (MB)	disk (GB)
m1.small	1	512	5120
c1.medium	2	512	5120
m1.large	2	1024	10240
m1.xlarge	4	2048	20480
c1.xlarge	8	2048	20480

AMAZON-US-EAST	CPU (cores)	RAM (MB)	disk (GB)
m1.small	1	1740	163840
c1.medium	2	1740	358400
m1.large	2	7680	870400
m1.xlarge	4	15360	1730560
c1.xlarge	8	7168	1730560

Figure C.3: Cloud Providers on SkyBoxz

SkyBoxz

LATEST NEWS | WHY SKYBOXZ | HOW IT WORKS | FORUMS | ABOUT US

You are logged in as: demo with a balance of: V$1000.

Service Provider

PROVIDER
- **VM Images**
- **VM Settings**

DASHBOARD
- **Cloud Status**
  - **Jobs**
  - **Resources**
  - **Clouds**
- **Direct Access**
  - **Terminal Login**
  - **Acquire Resource**
- **My Workspace**
  - **Projects**
  - **Add Project**
- **User Setting**
  - **SkyBoxz Account**
  - **Log Out**

Home » Direct Access

Acquire Resource

Pricing and Allocation

Number of instance types: 2

Allocate on cloud(s): All Available

Pricing policy: fixed pricing

Maximum price/hour: 10

Request Instances

Number of cores: 2

Memory size: 1024 MB

Disk capacity: 5 GB

Number of instances: 1

VM image: amazon-soc

Number of cores: 1

Memory size: 512 MB

Disk capacity: 5 GB

Number of instances: 2

VM image: amazon-soc

Submit | Reset

Figure C.4: Acquire Resources on SkyBoxz

SkyBoxz

LATEST NEWS | WHY SKYBOXZ | HOW IT WORKS | FORUMS | ABOUT US

You are logged in as: demo with a balance of: V$1000.

Service Provider

PROVIDER
- VM Images
- VM Settings

DASHBOARD
- Cloud Status
  - Jobs
  - Resources
  - Clouds
- Direct Access
  - Terminal Login
  - Acquire Resource
- My Workspace
  - Projects
  - Add Project
- User Setting
  - SkyBoxz Account
  - Log Out

Project Information

Project Name

SkyBoxz Ping

Description

Example C program that pings SkyBoxz with instance name and IP

Files

ping_message.c

Customized Command

Pre-run Command:

gcc -o ping_message ping_message.c; ./ping_message

Post-run Command:

Previous Result

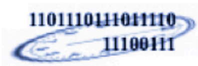
Submission Time	Status
2011-12-12 15:25:33	Completed

Running Instances

Resource - 211 | Submit this job | Delete this job

Figure C.5: Project Information on SkyBoxz

Welcome, teoym! You have a balance of 938¢.



SNAP
Snapshots of Program Execution on Different Computation Platforms

HOME | WHY SNAP | HOW IT WORKS | NEWS | ABOUT US

DASHBOARD

- Setup Experiment
- View Experiments
- Forums
- My account
- Create content
- Log out

View Experiments

No	Program File	Experiment Type	Instance Type	Run Output	Compile Output	
1	sin.c	sequential - gcc	Ubuntu 9.4, default RAM	0.041863219312105931	i-3FFE07B31.outputCompile	Download
2	sin.c	sequential - gcc	Ubuntu 9.4, default RAM	0.041863219312105931	i-437E07702.outputCompile	Download
3	vectoraddssh.c	parallel - mpich	Ubuntu 9.4, default RAM - 2 nodes	34.41476821899414062	3.outputCompile	Download
4	vectoraddssh.c	parallel - mpich	Ubuntu 9.4, default RAM - 8 nodes	10.000000000000000000	4.outputCompile	Download
5	vectoraddsshscs.c	parallel - mpich	Ubuntu 9.4, default RAM - 2 nodes	0.384322643280029296	5.outputCompile	Download
6	vectoraddsshscs.c	parallel - mpich	Ubuntu 9.4, default RAM - 8 nodes	2.028853893280029296	6.outputCompile	Download

Figure C.6: View Experiments Results on SNAP