Online Mechanism and Virtual Currency Design for Distributed Systems

A dissertation presented by

Chaki Ng

to The School of Engineering and Applied Sciences
in partial fulfillment of the requirements for the degree of
Doctor of Philosophy in the subject of

Computer Science

Harvard University Cambridge, Massachusetts

May 2011
Abstract

As distributed systems increase in popularity and experience resource contention, new resource allocation methods are needed for scalability and manageability. In this thesis, I investigate the use of auctions, a type of market-based methods, as a decentralized resource allocation approach. Agents in auctions individually submit bids that provide critical information, including agents’ private value for resources, for prioritizing resource allocation.

Three issues must be addressed for wider acceptance of using auctions. First, I present empirical data of a deployed market-based resource allocator, called Mirage, for allocating sensor network resources. The data provides observations of agent bidding patterns, which varied across values, sizes, and allocation timing. Furthermore, agents exhibited strategic behaviors that validate the need to mitigate such behaviors, which create complexity for both agents and system administrators.

Second, I design Roller, an online mechanism that is strategyproof with respect to agents’ submitted values and resource sizes. Roller is also configurable and able to provide different tradeoffs in regard to mis-reports of allocation timing. As agents of distributed systems often require responsive decisions, Roller uses a rolling window abstraction that enables the allocation of future resources. When compared to other allocators, Roller provides a high value and high responsiveness environment that is suitable for agents with
dynamic requests. Third, I study monetary policies in regard to the control of virtual currency for distributed systems. By establishing a space for policy design, I analyze the effectiveness of different monetary policies against specific workloads and agent strategies. A framework for identifying symmetric mixed strategy Nash equilibrium is also presented, which allows me to identify a policy that promotes active bidding as being effective in capturing high allocative efficiency. In addition, I observe that agent strategies that rely on agent values tend to dominate other strategies. By focusing on the above three issues, I provide empirical data, a responsive and strategyproof allocation method, and a framework to design and analyze virtual currency that can be useful for systems designers considering scalable resource allocation for distributed systems.
## Contents

Title Page ........................................... i
Abstract ........................................... iii
Table of Contents ................................... v
List of Figures ....................................... vii
List of Tables ........................................ vii
Citations to Previously Published Work ........... xi
Acknowledgments ...................................... xii
Dedication ........................................... xiv

### 1 Resource Challenges in Distributed Systems

1.1 Introduction .................................... 1
1.2 Rise of Distributed Systems .................... 4
1.3 Critical Resource Challenges ................. 8
1.4 Solving Challenges with Markets ............. 13

### 2 Market Deployment Lessons

2.1 Introduction .................................... 17
2.2 Mirage ......................................... 20
2.3 Usage Experience ............................... 24
2.4 Observed Strategic Behaviors ................ 31
2.5 Challenges and Refinements ................... 35
2.6 Answers to Research Questions ................ 37
2.7 Summary ....................................... 38

### 3 Online Mechanism Design

3.1 Introduction .................................... 39
3.2 Requirements ................................... 42
3.3 The Roller Mechanism .......................... 44
3.4 Design Justifications ............................ 51
3.5 Workloads and Metrics ......................... 57
3.6 Tuning Roller .................................. 65
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.7</td>
<td>Varying Supply</td>
<td>76</td>
</tr>
<tr>
<td>3.8</td>
<td>Comparisons with Other Allocators</td>
<td>84</td>
</tr>
<tr>
<td>3.9</td>
<td>Late Allocation</td>
<td>91</td>
</tr>
<tr>
<td>3.10</td>
<td>Summary</td>
<td>98</td>
</tr>
<tr>
<td>4</td>
<td>Virtual Currency Design</td>
<td>101</td>
</tr>
<tr>
<td>4.1</td>
<td>Introduction</td>
<td>101</td>
</tr>
<tr>
<td>4.2</td>
<td>Monetary Policy Design</td>
<td>103</td>
</tr>
<tr>
<td>4.3</td>
<td>Agent Models</td>
<td>111</td>
</tr>
<tr>
<td>4.4</td>
<td>Equilibrium Analysis</td>
<td>118</td>
</tr>
<tr>
<td>4.5</td>
<td>Basic Strategic Interactions</td>
<td>123</td>
</tr>
<tr>
<td>4.6</td>
<td>Understanding the Effects of Distribution Methods</td>
<td>130</td>
</tr>
<tr>
<td>4.7</td>
<td>Understanding the Effects of Money Supply</td>
<td>134</td>
</tr>
<tr>
<td>4.8</td>
<td>Understanding the Effects of Monetary Policy</td>
<td>139</td>
</tr>
<tr>
<td>4.9</td>
<td>Summary</td>
<td>146</td>
</tr>
<tr>
<td>5</td>
<td>Related Work</td>
<td>148</td>
</tr>
<tr>
<td>5.1</td>
<td>Market-based Systems</td>
<td>148</td>
</tr>
<tr>
<td>5.2</td>
<td>Online Mechanism Design</td>
<td>150</td>
</tr>
<tr>
<td>6</td>
<td>Conclusion</td>
<td>154</td>
</tr>
<tr>
<td>6.1</td>
<td>Summary</td>
<td>154</td>
</tr>
<tr>
<td>6.2</td>
<td>Future Work</td>
<td>155</td>
</tr>
<tr>
<td>A</td>
<td>Roller Proofs</td>
<td>158</td>
</tr>
<tr>
<td>A.1</td>
<td>Introduction</td>
<td>158</td>
</tr>
<tr>
<td>A.2</td>
<td>Allocation and Payment Rule</td>
<td>159</td>
</tr>
<tr>
<td>A.3</td>
<td>Establishing Strategyproofness</td>
<td>160</td>
</tr>
<tr>
<td>A.4</td>
<td>Roller With No Late Departure</td>
<td>162</td>
</tr>
<tr>
<td>A.5</td>
<td>Roller With Possible Late Departures</td>
<td>163</td>
</tr>
<tr>
<td>B</td>
<td>Replicator Dynamics</td>
<td>165</td>
</tr>
<tr>
<td>B.1</td>
<td>Basics</td>
<td>165</td>
</tr>
<tr>
<td>B.2</td>
<td>Computing Payoff Matrix</td>
<td>166</td>
</tr>
<tr>
<td>B.3</td>
<td>Strategy Populations</td>
<td>166</td>
</tr>
<tr>
<td>B.4</td>
<td>Iterations</td>
<td>166</td>
</tr>
<tr>
<td>B.5</td>
<td>Finding and Verifying Equilibrium</td>
<td>167</td>
</tr>
<tr>
<td>Bibliography</td>
<td></td>
<td>169</td>
</tr>
</tbody>
</table>
List of Figures

1.1 PlanetLab load example. 5th, average, and 95th percentile load average on 220 nodes leading up to the OSDI 2004 submission deadline [31].......... 10

2.1 Mirage map. 148 sensors deployed throughout the Intel Berkeley Research Lab. ................................................................. 20

2.2 Virtual currency policy. .............................................................. 24

2.3 Cumulative distribution of nodes requested by agents. ............... 25

2.4 Cumulative distribution of duration requested by agents. ............ 26

2.5 Cumulative distribution of delay and patience. ............................ 27

2.6 Testbed utilization: daily usage for the 97 MICA2 and 51 MICA2DOT nodes. ................................................................. 27

2.7 Bid value distributions: submitted total values by all agents. ...... 28

2.8 Bid value distributions: submitted values per node hour for the seven most active agents. ...................................................... 29

2.9 Median market prices (per nodenslot). ......................................... 30

1 number of nodes and
SR
R, or t3

6 rolls in. L R

1

.1 Rolling window abstraction: with NR

4

vii

number of slots, at same t. All allocations must start on example. maxdur is 3, thus all bids with slots = 3 can start

Advancing rolling window: At t. All columns “shift” to the right while column t (in gray color) phases out while column t also rolls right and is

............... 46

3.3 Single slot windows. ................................................................. 52

3.4 Multiple slot windows. ............................................................. 52

3.5 Rolling window. ..................................................................... 53

3.6 Rolling window with laststart time. ............................................. 55

3.7 Laststart: how much the windows are filled at beginning of each time period. 56

3.8 Two monotonically increasing total value curves. Curve A exhibits marginally increasing values as the nodeslot increases, and curve B exhibits marginally decreasing values. ................................................................. 60

3.9 Unit value distributions [y: y + z] for different nodeslots. ............ 61
List of Figures viii

3.10 Effects of $S_{R/a}$ on different $\lambda$ ranges: with marginally increasing ($\uparrow$) values. 67
3.11 Effects of $S_{R/a}$ on different $\lambda$ ranges: with marginally negative ($\downarrow$) values. . 68
3.12 Effects of $S_{R/a}$ on different $n$ ranges. ........................................ 70
3.13 Effects of $S_{R/a}$ on different $s$ ranges. ........................................ 71
3.14 Effects of $S_{R/a}$ on different $\Delta$ ranges. ........................................ 72
3.15 Workload example that yields higher $a$ with a larger window size. For
the left window, bids can start only in the first slot. For the right window, bids
can start in the first four slots. ‘-‘ indicates a sold slot and ‘x’ an unallocatable
slot. ................................................................. 75
3.16 $a$ and $\beta$ with varying node size $N_{R/a}$............................................ 78
3.17 Comparing value and revenue with $\lambda = 3$. ................................. 79
3.18 Comparing value and revenue with $\lambda = 5$. ................................. 81
3.19 Comparing value and revenue with $\lambda = 10$. ............................... 82
3.20 Reserve price effects. ........................................................................ 83
3.21 Roller vs. Greedy. Marginally increasing value distribution.............. 86
3.22 Roller vs. Greedy. Marginally decreasing value distribution............. 87
3.23 Roller vs. FCFS vs. SJF. Marginally increasing value distribution..... 88
3.24 Roller vs. FCFS vs. SJF. Marginally decreasing value distribution..... 89
3.25 Roller vs. FCFS vs. SJF: number of nodes and total values. $\lambda = 5$.
Marginally increasing values................................................................. 91
3.26 Roller vs. FCFS vs. SJF: number of nodes and total values. $\lambda = 10$.
Marginally increasing values................................................................. 92
3.27 Effects of $\rho$ late % on average payoffs: across different multiples of over-
report departures. ............................................................................. 94
3.28 Effects of $\rho$ late % on system value $a$ : across different multiples of over-
report departures. ............................................................................. 96
3.29 System value $a$ captured by each $\rho$ late %: driven by multiples that de-
liver highest agent payoffs. ............................................................... 97
3.30 Total revenue captured by each $\rho$ late %: among all over-report multiples. 98
3.31 Total revenue captured by each $\rho$ late %: driven by multiples that deliver
highest agent payoffs. ........................................................................ 99
4.1 Monetary policy dimensions. .............................................................. 104
4.2 D2: Stable agent weight calculation. ............................................. 109
4.3 Replicator dynamics to finding symmetric mixed strategy Nash equilibrium. 122
4.4 Two-strategy payoff matrix for D1: Uniform. ................................. 124
4.5 Time-based patterns for profile (4,6) with D1:Uniform. The top graph shows average bid values submitted by agents playing each strategy. An average market price line is added as reference. S1:Greedy agents submit decreasing bid values because they constantly spend high amount of currency. The middle and bottom graphs show the ending balance of agents and the system, respectively. Balance of S2:Jobs agents are high due to their more conservative bidding nature. 126

4.6 Two-Strategy equilibrium for D1:Uniform. Starting with equal percentage of agents playing each strategy, replicator dynamics quickly converge to a symmetric mixed strategy Nash equilibrium of (0.4, 0.6), in which each agent will play S1:Greedy with a 40% chance. 127

4.7 Time-based patterns for profile (2,3,2,3) with D2:Stable. The top graph shows average bid values submitted by agents playing each strategy. An average market price line is added as reference. S1:Greedy agents submit decreasing bid values because they constantly spend high amount of currency. The middle and bottom graphs show the ending balance of agents and the system, respectively. Balance of S2:Jobs agents are high due to their more conservative bidding nature. 128

4.8 Four-strategy equilibrium for D2:Stable. Starting with equal percentage of agents playing each strategy, replicator dynamics quickly converge to a symmetric mixed strategy Nash equilibrium of (0, 0, 0.3, 0.7), in which agents will play S3:Values with a 30% chance and S4:Prices with a 70% chance. 129

4.9 Time-based patterns for profile (0,0,3,7) with D2:Stable. 130

4.10 D1:Uniform and pure strategy profile (2,3,2,3). Resource and value captured by the policy for each of the 10 agents, under each of the four workloads. 131

4.11 D2:Stable and pure strategy profile (2,3,2,3). Resource and value captured by the policy for each of the 10 agents, under each of the four workloads. 132

4.12 D3:Active and pure strategy profile (2,3,2,3). Resource and value captured by the policy for each of the 10 agents, under each of the four workloads. 134

4.13 D4:Urgent and pure strategy profile (2,3,2,3). Resource and value captured by the policy for each of the 10 agents, under each of the four workloads. 135

4.14 W1:Common and D1:Uniform. Distribution intervals and symmetric mixed strategy Nash equilibrium. 136

4.15 W1:Common and D1:Uniform. Distribution intervals and value efficiency. 137

4.16 W3:Common and D3:Active. Distribution intervals and symmetric mixed strategy Nash equilibrium. 138

4.17 W3:Combo and D3:Active. Distribution intervals and value efficiency. 138
List of Tables

2.1 Example of strategy S4 (auction sandwich attack) on 97 MICA2 nodes. 35
3.1 Responsiveness of different bid instances. On the left column, each number represents time-to-win for a winning bid and x represents a losing bid. For example, 22x means there are three bids, with one of the bids being a losing bid. 64
3.2 Summary of workload parameter effects. Window size S is fixed. ↑, ↓, ↔ means increase, decrease, and no change, respectively. 73
3.3 Effects of demand on metrics: given fixed SRR. 78
4.1 Distribution methods: overview. 108
4.2 Workloads for virtual currency experiments: Grey boxes highlight differences between two types of agents in a workload. 114
4.3 Agent strategies: overview. 115
4.4 Payoff matrix example: for a 3-strategy game with a total of 3 agents. |x| and |y| represents the number of agents playing the 1st and 2nd strategies, respectively. The number of agents playing the 3rd strategy can be inferred from 3 - |x| - |y|. Each entry represents the average payoffs (%) of a specific pure strategy profile. ‘-‘ indicates no agent plays the strategy for the particular profile. 121
4.5 Workload efficiency at equilibrium for different policies. Maximum and minimum pure strategy profile efficiencies are included for comparison. Highest equilibrium efficiencies are in bold. 140
4.6 Symmetric mixed strategy Nash equilibrium profiles that correspond to the “equilibrium” entries in Table 4.5. Profiles with highest value efficiencies for each workload are in bold. 144
4.7 Corresponding pure strategy profiles that generate the “maximum” entries in Table 4.5. Profiles with highest value efficiencies for each workload are in bold. 146
B.1 Example of a payoff matrix. 166
Citations to Previously Published Work

The bulk of Chapter 2 have appeared in the following paper:


The design in Section 3.3 is extended from the following paper:

Acknowledgments

My long journey to learning academic research has been an unforgettable life experience. I cannot imagine a better place to do research than at Harvard, with great teachers, students, and resources. This thesis is special to me because it is interdisciplinary. I am thankful to have not one, but two wonderful and caring advisors in David Parkes and Margo Seltzer. It is an understatement to say that they gave me the utmost and unconditional support throughout this journey.

Thanks to David, who made it exciting for me to think about applying economics to computer science problems. He helped me succeed through his attention to details, deep knowledge of various literature, and openness to provide as much guidance as I needed.

Thanks to Margo, who was the ultimate coach to help me focus on the key questions, ideas, and tasks. Her ability to be a successful professor, entrepreneur, and parent simultaneously is a constant reminder that I can always do better.

I would like to thank Jim Waldo, my other committee member, who played a key role in the development of the thesis, including ensuring that the thesis language is compatible with a systems audience.

The collaborative projects as part of the thesis were truly critical and exciting. I thank Brent Chun, Amin Vahdat, Alex Snoeren, Alvin AuYoung, Saul Youssef, John Brunelle, and Jeff Shneidman for the many hours we spent together in discussing and implementing ideas. Thanks to friends at Harvard, including Adam Juda, Jason Woodard, Shawn Hsiao, Uri Braun, Rohan Murty, Jonathan Ledlie, Lex Stein, John Dias, and many others. They made the graduate school experience enjoyable, especially during crunch times and paper deadlines.

Thanks to Kevin Polk for always supporting my constant juggling school and work and
to colleagues at Interactive Constructs. Thanks to my friends from high school to business school and from Hong Kong to Boston who have been supportive of my academic endeavors. Last but not least, my family plays a huge role in this journey. It would not have started if not for my parents, who let me pursue school, work, and life thousands of miles away from them. Thanks to my sister who takes good care of them. Thanks to Shirley, who has a big heart to always keep the family running and everyone happy. And thanks to my girls, who give me the laughters and joy to realize how lucky I am with everything.
To my parents and my family
Chapter 1

Resource Challenges in Distributed Systems

1.1 Introduction

In this thesis, I explore market-based resource allocation methods for distributed systems [68] in the presence of selfish agents. In recent years, distributed systems (e.g., grid computing, cloud computing, and peer-to-peer systems) have become popular in research and commerce. A key feature of distributed systems (or simply “systems”) is the ability to coordinate many complementary types of computational resources (e.g., servers, sensors, storage, and network bandwidths) for ease of shared use by multiple agents over space and time. This sharing is economical for single agents with occasional needs for these resources.

1 I use the terms selfish, self-interested, and strategic interchangeably.

2 The term agent refers to either human users or software-based intelligent agents.
To facilitate sharing among agents, a system must include a method for resource allocation. There are a wide range of methods that can be used when the resources available exceed the resources requested. However, available methods are more limited when the situation is reversed, and resource allocation must be prioritized. A common method is to authorize a centralized entity (e.g., a person, a committee, or a software program/scheduler) to prioritize. This centralized method becomes less and less effective as the number of variables in determining priority increases (i.e., does not scale when there is a wide array of agents, each with their individual needs, some perhaps more important than others). The result of ineffectively allocating resources also carries a negative economic impact for the distributed system owners. Therefore, this thesis focuses on market-based methods of prioritizing resource allocation.

The market used for examples in this thesis is auctions. Through auctions [52], agents try to obtain resource allocation by submitting a “bid.” Through some common language, each agent submits a bid which describes the desired resource and the value it is willing to pay. The objective of an auction is to allocate resources to bids with the highest values, while also meeting resource constraint requirements of the distributed system. An auction that captures a high amount of value is called “allocatively efficient.” By having agents submit bids individually, an auction decentralizes the resource allocation process and this eases centralized bottlenecks [33].

Because maximizing value is a key motivation for using an auction, it is important to define what value is. Value is the “maximum willingness to pay” for some resource by an agent [97, 36], measured in some currency. For example, an agent may be willing to pay a

Walker [102] offered a parallel view of value: “The term 'value' always implies power in exchange, and nothing else. Value is the exchange power which one commodity or service has in relation to another.”
maximum of $100 for using a server. I distinguish two types of values for each agent in this thesis. First, an agent attaches a true value to some resource as soon as its demand arises. This value is privately known only to the agent and is fixed. Second, the agent decides how close to the true value he wants to pay and submits this amount as a bid value to the resource allocation auction. Thus, bid values an auction receives are not necessarily the true values of the agents. Agents’ true values and bid values can each be based on either a real (e.g., USD) or a virtual currency, depending on the system. In this thesis, value-based metrics for experiments are based on true values, unless otherwise noted. While the idea of using market-based methods for distributed systems is by no means a new idea (dating back to the PDP-1 futures market [94]), many open issues still remain that limit its adoption by systems designers. In this thesis, I make contributions to the following open issues:

• Lack of empirical data: There is a serious lack of empirical data on how market-based methods behave in distributed systems. One of the reasons is most of the market-based methods do not collect usage data. This makes it hard for systems designers, as well as agents, to evaluate market-based methods. To remedy this situation, I introduce Mirage, a deployed market-based resource allocator for a sensor network testbed that provides real usage data, particularly in regard to strategic behaviors exhibited by agents.

• Allocating dynamic resources: Traditional market-based methods (e.g., auctioning a vase on eBay) are not designed for resources that are available dynamically over time. These metrics are applicable because I generate agents’ true values in my experiments. In the real-world, a system cannot generate these true value-based metrics.
time (e.g., using a server now vs. 2 hours from now). Furthermore, the set of agents varies at different times, creating different sets of bids that can be exploited strategically. To address this need I have designed Roller, an online mechanism that uses a rolling window abstraction to auction resources across space and time, and can make responsive, early decisions on allocations that occur in a future time.

• Elicit true information from selfish agents: A system should maximize the aggregate true values of agents. This requires that agents have an incentive to report their private true values as bid values. Similarly, agents should be encouraged to report truthful resource needs in space and time. Furthermore, for mechanisms like Roller that run continuously, the agents’ desire to strategize across a series of auctions should be mitigated. I have designed Roller to address these three barriers to eliciting true information.

• Unclear virtual currency characteristics: Not all systems can use real currency for bids and payments. Examples include non-profit and internal corporate systems. Instead, they must create virtual currency for use with market-based methods. However, it remains unclear how much currency a system should create, as well as how it should distribute currency to agents. I will present a framework for designing and analyzing monetary policy for different agent workloads and strategies.

1.2 Rise of Distributed Systems
Distributed systems have emerged as an important computing paradigm in recent years. One aspect that is important for many applications is the ability to scale and coordinate
large amounts of computational resources for multiple agents to share over both space and time.

Since the early days of the computer industry, agents have been sharing computational resources. Things have changed dramatically since the days of logging into the shared corporate-owned mainframes. The arrival of personal computers in the 1980s, and with them new classes of applications for productivity, business, and entertainment, have allowed agents to work privately at any given time. Nonetheless, there is never a shortage of applications or usage needs that demand resources beyond that which any individual personal computer can provide, despite computing capabilities having constantly increased as described by Moore’s Law [9].

In this thesis, I focus on systems that are administrated by a single trusted domain. Each system is available over time and serves an array of agents (e.g., employees of a company, researchers of a university), each with different resource needs over time. If designed properly, such a system can provide many benefits, such as:

- Statistical multiplexing: By having multiple agents sharing, a distributed system can serve the agent demands over space and time.

- Economy of scale: A properly designed distributed system enables adding new resources over time. Thus, a system can start small and grows as the needs of agents arise.

- Resource heterogeneity: A system may include resources of different types (e.g., servers for computation and disks for storage).

- Fault tolerance: The failure of some resources will not render the whole system un-
Variations of distributed systems that have offered these benefits have evolved over the years. Three of these variations are the motivation for this thesis and are described below.

1.2.1 Grid Computing

In many scientific disciplines, including astronomy, biology, and physics, scientists often need large amounts of computational capabilities to process and analyze data-intensive experiments. These data are generated from equipment such as the Large Hadron Collider [7], which can produce around ten TB (terabytes) of data every eight hours [17]. These data are then distributed to physicists as required. The amount of resources required to process these enormous amounts of data are significant and unaffordable by virtually all individual scientists and administrative domains (e.g., universities). In addition, because most large-scale computational needs only arise periodically, sharing is more economical.

Because different scientists need this type of large resource at different times, the concept of grid computing was created, to enable multiplexing distributed resources. A computational grid is a “hardware and software infrastructure that provides dependable, consistent, pervasive, and inexpensive access to high-end computational capabilities.” [38] Today, these grids are being deployed in different large-scale centers around the globe, including ATLAS [3] and Open Science Grid [10].

1.2.2 Network Testbeds

In computer science, testbeds are indispensable as they provide a platform to perform experiments on the next generation of computational technologies. As computational sys-
tems often span across the globe, the need to access heterogeneous resources both locally and remotely is great. Nonetheless, deploying remote resources (e.g., in Asia) presents huge barriers for most research projects, which do not have budgets or personnel to operate remotely.

As a result, initiatives to form testbeds by pooling resources owned by different universities are being widely developed. PlanetLab [80] is a well-known project that includes over 1,000 nodes from hundreds of universities and institutions around the globe. Researchers use PlanetLab to access “slices” of the whole network of nodes easily (e.g., use a slice to access 5% of all nodes). Some experiments on PlanetLab include content distribution networks [39] and global network traffic monitoring and management [65].

Another example is sensor network testbeds. Wireless sensor networks are important as they apply to physical applications such as environmental monitoring [63] and health care [64]. Since many sensors are location-specific and thus heterogeneous (e.g., collection of real-time weather in Boston vs. Seattle), some research groups put together different testbeds for sharing [69, 30, 66].

1.2.3 Cloud Computing
In recent years, cloud computing has become popular for organizations of all sizes to access hosted computational resources on a “pay as you go” basis. Many web applications today are hosted on public cloud services such as Amazon Web Services [1] and Google App Engine [6]. There are a wide range of resources available, including data storage, computational power, and databases. The ability for an organization to avoid acquiring, installing, and maintaining such complex resources is a key reason why cloud computing
has taken off. Nonetheless, there are still barriers to using cloud computing, especially for large or-

organizations that are concerned with reliability and security. Outages of cloud computing services can occur, which can affect popular websites and applications and their millions of users [2]. Reliability issues like this will further encourage large organizations to build and deploy their own private clouds. A private cloud [8] is accessible only by authorized agents and provides more fine-grained control.

Many of these private clouds involve virtual machines [15] that each serve one or more agents at a time. Administrators manage these clouds by dynamically allocating agents’ requests of computational resources to different virtual machines, each with its own resource constraints. New tools and paradigms for managing private clouds have only begun to emerge as their complexity grows.

1.3 Critical Resource Challenges
The power of multi-agent sharing resources such as grids and testbeds comes with a price. As the amount of resources and the number of agents increase, the number of possible ways for a system to allocate resources to agents with diverse needs increases exponentially. Because there is no single, accepted rule as to how and when certain resources should be allocated to different agents, an important task for systems designers is to define objectives and design resource allocation methods that support such objectives. This task is complex due to several critical, interrelated resource challenges, that are discussed below.
1.3.1 Dynamic Resource Needs

Distributed systems are attractive because they provide resources that are available on an on-demand basis. Thus, resource allocation methods must recognize agent requests (that can arrive at any time) and provide the requested resources immediately or at the requested time. Some agents may have infrequent requests, while others may make a request every two hours. Thus, a good resource allocation method should be able to handle these dynamic requests in real-time, and fill these requests as soon as possible without incurring significant delays for agents. Delays can be problematic in many cases. For example, in Grid Computing scientists often spend hours or even days preparing the data and code before running an experiment on a system. Systems often are different (e.g., different versions of operating systems) meaning the preparation cannot start until the scientists know for sure which resources will be available. Thus, allocators that inform them of reserved resources in advance would help avoid the risk of not being ready to use the resources, because the data is still being prepared.

In addition, resources are also designed to be available in different configurations in order to satisfy different agents who desire diverse combinations of resources. In the context of a sensor network, one agent may demand half of the sensors for one hour, and another agent may demand all of the sensors for one day. Systems must be able to take different requests and make allocation decisions based on the various constraints of the resources, as well as individual agent goals, while meeting the objectives of the system. Resource allocation methods that consider space and time from the perspectives of systems and agents are critical to successfully address these dynamic resource needs.
1.3.2 Resource Contention

A serious problem that all systems must tackle is resource contention, which arises when agents collectively request resources that exceed total system resources. For example, PlanetLab usage often increases significantly in the days leading up to conference deadlines (see Figure 1.1 for an example). The key challenge is to decide which requests should be granted and which denied. Systems must adapt resource allocation methods that can resolve this challenge.

Figure 1.1: PlanetLab load example. 5th, average, and 95th percentile load average on 220 nodes leading up to the OSDI 2004 submission deadline [31]. One contributing factor to such explosive demand is the lack of a method to encourage individual agents with less-urgent needs to back off during periods of time when there are insufficient computational resources. Traditional resource allocation methods often just
drop certain requests randomly, as they don’t have additional agent information in regard to values or preferences.

Consider the real-life example of an emergency waiting room in a hospital with a room full of ill patients, where only half of the patients can be served. Assume that half of the patients have serious conditions and will die if not served. If the method is to simply randomly select which patients are to be served, then likely many patients will end up dying, which is definitely not the goal of either the hospital or the patients.

1.3.3 Ignored Agent Values

The previous hospital example highlights this next challenge. Every agent has some kind of value attached to the resources it seeks, and a system should respect and leverage such value for resource allocation. Two agents who have otherwise identical requests (e.g., ten sensors immediately for one hour) may have different values (e.g., $100 for one and $5 for the other). If these values are somehow revealed to a system, it can use them to decide which requests are more valuable and should be granted resources.

The challenge for systems is threefold: (i) to determine the currency with which all reported values should be based on (e.g., real vs. virtual currency) and to create and manage such currency if necessary, (ii) to enable a way for all agents to express their values in the system, and (iii) to adapt allocation methods that take such represented values as first-order factors.
1.3.4 Presence of Selfish Agents

Last but not least, systems must embrace the fact that agents may be selfish. Selfish behaviors are common in society [57], and agents are usually primarily interested in their own self-interests and want to maximize their individual gains. For public goods, economists have long observed these behaviors [58]. Their effects are also well-documented in many distribution systems [91]. In peer-to-peer systems, the ideal goal is to have agents that both contribute and consume resources. However, tragedy of the commons [45] (or freeriding) is a well-known problem in which agents often consume much more than they contribute. In grid computing, scientists with urgent tasks often cannot obtain resources because they are delayed by long-running jobs of lower priority that ideally should be suspended to make room for the urgent jobs [29].

Yet, traditional resource allocation methods assume agents are either obedient (i.e., following the prescribed algorithm) or perhaps adversarial (i.e., intentionally behaving badly mainly to hurt the outcomes of others) [37]. The concept of selfish agents has not been fully embraced by systems designers, although it can easily disrupt allocations and goals. Consider the hospital example again in which all patients look ill and it is impossible to tell who is dying. Assume the hospital adopts a method that asks each patient to answer the question of whether they are dying, and admits only those who say yes. In order to get admitted, some non-dying patients who may be in serious pain might respond yes for selfish reasons. Even those non-dying patients who planned to tell the truth might lie and say yes if they realize the chance to be admitted is slim otherwise, since most others have replied yes. Selfish behaviors in this case totally disrupt the system.
1.4 Solving Challenges with Markets

The challenges from the previous section involve agents who are selfish, with individual values and preferences regarding resources, and with resource needs that are combinatorial and time-varying from each other. As systems grow in popularity, these challenges become even more critical to address. Unfortunately, few traditional allocation methods address these challenges, as the following examples demonstrate.

1. TeraGrid is shared by a large number of physicists [14]. Resources are divided into "service units" of one CPU hour each. The allocation requests are submitted manually by agents and are then reviewed by different committees. The committees will review the requests according to some qualitative schemes. This type of process is not scalable, and has the potential to be manipulated.

2. The resource abstraction of PlanetLab [80] is a slice, which is a "horizontal cut" of a number of machines to include a certain amount of CPU, memory, storage, etc. Agents on PlanetLab obtain slices without physical limits. Thus, during resource contention periods, few agents get their desired amount of resources, even though everyone still has its slice. While scheduling tools such as Sirius [13] and Bellagio [19] allow agents to reserve a certain amount of slice resources in advance, the main allocation method PlanetLab adopts does not capture or use agent values.

3. Heuristic schedulers such as First-Come First-Serve (FCFS), Proportional Sharing (PropShare), Shortest-Job-First (SJF), and Earliest-Deadline-First (EDF) are fast in determining allocations. However, they do not consider combinatorial resources or agent values. In addition, agents can easily manipulate the scheduler, for example,
Chapter 1: Resource Challenges in Distributed Systems

by reporting an early deadline for an EDF scheduler.

The idea of using markets and using value maximization as an objective to allocate computational resources, for either centralized or distributed systems, is not new. There has been much work done on resource allocation in a broad range of systems, including clusters [100, 29], computational grids [104, 54], parallel computers [92], and Internet computing systems [60, 84]. Nonetheless, many obstacles remain.

In this thesis, I contribute by addressing three major obstacles that, once solved, could increase the interest level of systems designers in considering market-based methods of resource allocation. These three major obstacles are: lack of empirical data, lack of focus to address self-serving strategic behavior, and lack of virtual currency understanding.

1.4.1 Lack of empirical data

Despite the benefits of markets that have been discussed by previous works as listed above and in the real world (e.g., stock exchanges, FCC), markets for resource allocation of distributed systems still have not become commonplace for real-world deployments. A key reason is lack of empirical data to support their benefits. In Chapter 2, I address this obstacle via Mirage, an auction-based market designed to allocate resources for a sensor network testbed. Mirage uses a repeated combinatorial auction for resource allocation of 148 nodes over time. The usage data collected show agents submitting diverse bids and exhibiting strategic behaviors. These include manipulating bid values as well as the space and time attributes of resource requested. These data are valuable for motivating the design of market-based methods that mitigate such behaviors.
1.4.2 Lack of focus to address strategic behavior

With the presence of strategic behaviors, resource allocation of a distributed system becomes more complex. This applies to both agents who participate in it, as well as the systems designers and administrators who need to maintain orderly transactions. Most current market-based systems do not address these behaviors as a first-order priority. Furthermore, strategies can span across an array of market attributes, including bid values, resource sizes, and timing, making it hard to bootstrap with some simple mechanism with the hope of fixing strategic behaviors over time.

In Chapter 3, I address this obstacle by designing Roller, an online mechanism that uses a rolling window abstraction for resource allocation and enables agents to submit bids with values and resource requirements over space and time. It addresses the shortcomings of Mirage with allocation and payment rules that mitigate strategic behavior while maintaining the ability to make fast decisions. Specifically, Roller is a type of strategyproof mechanism in which reporting truthfully is the best strategy for agents. I show that Roller captures a high amount of total value within a responsive environment.

1.4.3 Lack of virtual currency understanding

Many systems discussed in this chapter do not and cannot use real currency. In fact, many proposed market-based methods assume the use of virtual currency. However, the characteristics of virtual currency are not yet fully understood. How much currency to create and how to distribute it to agents are among the open questions that need to be addressed. Without a properly designed virtual currency, market-based systems will be negatively affected because all of their transactions depend on it.
In Chapter 4, I explore designing a monetary policy for virtual currency. Specifically, I introduce a design space of policy dimensions that includes: money supply; distribution methods; and distribution intervals. I build a closed system model with agents receiving and spending currency over time, subject to a set of agent bid value strategies. To study the behavior of this model, I use equilibrium analysis in search of a steady state. Overall, different types of policies have varying effects on systems and specific workloads. Some of these policies are able to capture high total value for the systems.
Chapter 2

Market Deployment Lessons

2.1 Introduction
Despite the number of market-based methods currently proposed for distributed systems, few have been deployed and even fewer have provided data on agent characteristics, resource allocation efficiency, or value efficiency. These data are critical for validating the usefulness and design of market-based systems. One way to increase the amount of empirical data is by deploying these systems in industrial or research settings.

In this chapter, I present the design of and empirical data collected from Mirage.
the type, locations, and communication frequencies of the nodes. Therefore, each agent request includes a combination of nodes over space and time and sometimes requires a set of complementary notes.

Each agent is allocated virtual currency to use in bidding for the testbed resources. Each bid specifies the resource combinations of interest (e.g., “any 32 nodes for 8 hours anytime in the next two days”) and a bid value in virtual currency, indicating the maximum the agent is willing to pay. Mirage accepts bids on an ongoing basis and runs an auction periodically. Winning bids in the auction are determined by maximizing the total bid value obtainable for available resources. Because of the combinatorial nature of this resource supply and demand, combinatorial auctions [35] are used in Mirage to fulfill the multiple resource specifications of agents. Unlike a single-item auction, a combinatorial auction is able to consider all resource requirements in a single agent request as well as resource constraints when making resource allocation decisions. This mitigates the problem of “exposure.” For example, an agent wants a pair of shoes but is forced to bid in two auctions, one for the left shoe and one for the right shoe. The agent is likely to win only one shoe in a noncombinatorial auction setting, resulting in zero value.

Empirical data presented in this chapter were collected from the operation of Mirage during the initial four-month period, which had the most agents participating and the highest number of bidding activities of the testing period. Using these data, I answer the following two key research questions.

2.1.1 Research Questions
Do markets work?
I want to validate whether or not a market-based resource allocation scheme is necessary. Currently, there is not enough empirical data to decide whether market-based methods could work well for distributed systems. To determine necessity, there are two basic questions to answer. Can agents adapt to interacting with markets, especially submitting requests along with bid values? Do agents indeed bid different values for resources, especially when virtual currency is used? Positive answers to these questions provide valuable evidence that markets can work as an environment for agents to request for resources.

Do agents game? Traditional resource allocation methods generally do not pay any attention to strategic behaviors motivated by the self-interest of agents. Even certain market-based methods do not place strategic behaviors as a primary allocation factor (e.g., first-price auctions which can be manipulated). To highlight this issue, real data showing real strategic behaviors and their negative effects will be crucial for designers to justify spending the time and energy to devise more appropriate resource allocation methods, for example, mechanism design [73].

2.1.2 Chapter Overview
In Section 2.2, I provide an overview of the Mirage system, including the repeated combinatorial auctions and virtual currency system. I discuss how the usage data obtained helps address the question, “Do markets work?” in Section 2.3. Similarly, I address the question, “Do agents game?” via empirical data in Section 2.4. The findings from the deployment of Mirage bring to light further challenges and refinements that can be used to design a more robust resource allocation model. I discuss these in Section 2.5. Finally, I present my conclusions and summary in Sections 2.6 and 2.7.
Chapter 2: Market Deployment Lessons

2.2 Mirage

An opportunity for resource allocation design arose during the construction of a 148-node sensor network testbed at the Intel Research Laboratory in Berkeley, CA. This testbed is comprised of two types of sensors: 97 Crossbow MICA2 and 51 Crossbow MICA2DOT series sensor nodes, or “nodes,” mounted uniformly in the ceiling of the lab (see Figure 2.1). The testbed is intended to be a non-profit service provided for free to researchers of sensor networks inside and outside of the lab. From a market standpoint, Intel can be viewed as the “seller” and the researchers as the “buyers.”

Figure 2.1: Mirage map. 148 sensors deployed throughout the Intel Berkeley Research Lab.

Mirage [30] is a microeconomic resource allocator designed specifically for this testbed. Note that there are two important assumptions being made here. The first is that the primary goal of the system is to maximize aggregate value. This assumption is predicated on the fact that the second assumption is true in that agents who use Mirage each have value associated with the desired resources. To achieve the primary goal, the system employs a repeated combinatorial auction [35, 72] to schedule allocations. In such an auction, an
agent submits bids specifying resource combinations of interest and the amount of virtual currency the agent is willing to pay. Periodically, the auction clears, a set of winning bids is computed, and trades are settled through payments to a central bank. Next, I discuss the design of the repeated combinatorial auctions and virtual currency.

2.2.1 Repeated Combinatorial Auctions

Mirage uses a first-price, repeated combinatorial auction to allocate resources to competing agents over time. The auction uses a heuristic algorithm in order to clear auctions quickly and regularly. In this setting, an auction is run periodically. During each round, there are multiple buyers (the competing agents) and a single seller (Intel) who sells resources on the system’s behalf. All bids submitted prior to the start of a round are considered. The auction will then calculate winning bids based on per nodeslot value, collect payments (which equal agents’ submitted bid values), and make associated resource allocations. It is important to note that the version of Mirage used here is not strategyproof due to the use of a first-price method. Thus, agents can gain by mis-reporting bid values in Mirage. We chose this method because we did not have a suitable fast and strategyproof algorithm available at the time of deployment. I present the details of the combinatorial auction for this testbed below.

Each Mirage node is allocated for use in 1-hour slots. Agents may bid for 1, 2, 4, 8, 16, or 32-hour slots (durations). To allow agents to plan ahead, the auction sells resources up to

2. I discuss current challenges of using optimal combinatorial auctions in Section 2.5.

3. For example, consider two agents seeking the same resource with true values of 10 and 5, respectively. The first agent can gain (and still win) by submitting any bid value between 5 and 10, such as 6, and committing to a lower payment.
three days in advance, which is a total of 72 slots. Thus, the resources being allocated at any time can be viewed as a matrix of 148 nodes by 72 slots (the next 72 slots from the current time). I refer to each cell in the matrix as a single “nodeslot.” When the system starts up, all nodeslots are available. Over time and as auctions are run repeatedly, nodeslots become occupied as bids are allocated and new nodeslots become available as the “window” of nodeslots opens up. Note that each nodeslot can be used by only one agent.

Agents can submit bids to the auction whenever their needs arise. Each bid includes resource combinations of interest in space (nodes) and time (slots), along with a maximum value (bid value) the agent is willing to pay. Formally, a bid $b_i$ is specified as follows:

$$b_i = (v_i, s_i, t_i, d_i, f_{\text{min}}, f_{\text{max}}, n_i, ok_i).$$

Bid $b_i$ indicates that the agent is willing to pay up to $v_i$ units of virtual currency for any combination of $n_i$ nodes from the preferred subset of nodes $ok_i$, for a duration of $d_i$ hours (1, 2, 4, ..., 32), a start time between $s_i$ and $t_i$ hours, and a frequency in the range $[f_{\text{min}}, f_{\text{max}}]$. $s_i, f_{\text{min}}$ represents the delay of when an allocation can be first considered from the time of bid submissions. The difference between $s_i$ and $t_i$ represents the patience during which an allocation can be assigned. $ok_i$ is a subset of the 148 nodes that the agent prefers to choose from. For example, an agent may only want nodes that are near one side of the Lab or nodes that are at least 10 feet apart from each other. Each sensor supports different frequencies, thus an agent may desire a dedicated frequency for its allocated sensors to communicate wirelessly in order to avoid conflicts with sensors allocated to other agents. In practice, distinct frequencies have not been a scarce resource and thus rarely present problems.

As an example, an agent might request “any 64 nodes, operating on an unused frequency”

Agents use a separate resource discovery service to identify desired nodes.
in the range [423MHz, 433Hz], for 4 consecutive hours anytime between the next 6 hours and the next 24 hours, for up to 99 units of virtual currency.” If the agent needs 100 unique nodes out of the total 148 that meet such resource specifications, then the corresponding bid would be:

\[ b = (99, 6, 24, 4, 423, 443, 64, \text{[a list of 100 nodes]}) \]  

(2.2)

2.2.2 Virtual Currency

Because the testbed is offered as a free public service to select researchers, charging real currency in Mirage is impractical. Instead, Mirage relies on virtual currency and a central bank to enforce currency policy. Because agents in Mirage have no way to earn currency, the system must decide how to distribute virtual currency, both when a new agent joins and over time as agents need additional currency to buy resources.

The virtual currency policy, shown in Figure 2.2, assigns two numbers to each agent’s bank account: a baseline value and a number of shares. When created, each bank account is initialized to its baseline value (i.e., a number of virtual currency is credited to the agent account). Once funded, an agent can then begin to bid and acquire testbed resources through Mirage. In each round of the auction, accounts for winning bids are debited and the proceeds are redistributed through a profit-sharing policy based on the proportional shares of each agent. The primary purpose of this policy is to reward agents who refrain from using the system during times of peak demand and penalize those who do not. These rewards result in transient bursts of credit.

Another mechanism, a savings tax, prevents idle agents from sitting on large amounts of excess credit for extended periods of time (a “use it or lose it” policy). Periodically, agents
whose balance is above their baseline values will be taxed by some tax rate: a portion will be returned to the bank which then distributes the currency to all agents based on shares. Unfortunately, empirical data to evaluate the effectiveness of different tax rates was not collected properly for analysis.

In the deployment, an administrator sets the virtual currency policy. Bank accounts for external agents were assigned a baseline value of 1,000, while bank accounts for the two internal agents (i.e., employees of the Intel Lab) were assigned larger allocations with baseline value of 2,000. For simplicity, the shares of these agents are set to 1,000 and 2,000, respectively. Savings tax is collected every 4 hours, at a rate of 5% of an agent’s account savings. These parameters were chosen to ensure that an exhausted bank account can recover half of its balance within a few days, and the full amount in a week.

2.3 Usage Experience
Mirage began operation in December, 2004 and ran well over a year. In this section, I report on usage data collected over the initial four-month period since this was the period.
of highest activity. As of April 8, 2005, a total of 312,148 node hours were allocated across 11 agents (each representing a separate project).

2.3.1 Dynamic Resource Needs

![Cumulative distribution of nodes requested by agents](image)

Figure 2.3: Cumulative distribution of nodes requested by agents. The results indicate that agents have dynamic and combinatorial resource needs in Mirage over both space and time. Figure 2.3 shows a CDF of the number of nodes (n) that agents requested. The range of nodes is distributed evenly, from a single node to the full set. Similarly, agents do seek different durations (d_i, number of slots) as well, as illustrated in Figure 2.4. Last but not least, agents do have different delay (s_i) and patience (t_i - s_i), highlighting that their needs do vary over time, and that they do take advantage of the ability to submit requests in advance (Figure 2.5).
In summary, these usage data are important in that they substantiate the need for allocation methods that address the challenge of dynamic resource needs in Chapter 1. Therefore, the effort to design and adopt methods that handle these varying needs over value, space, and time efficiently is worthwhile.

2.3.2 Resource Contention

During the initial four months, several periods of significant resource contention took place including the SIGCOMM 2005 (due on 61st day) and SenSys 2005 (due on 120th day) conference deadlines. Figure 2.6 shows the utilization of the MICA2 and MICA2DOT nodes over the four months, plotted on the x-axis as number of days since Mirage was first deployed. It depicts periods of significant utilization (y-axis, near 100%) extending over...
Figure 2.5: Cumulative distribution of delay and patience.

Figure 2.6: Testbed utilization: daily usage for the 97 MICA2 and 51 MICA2DOT nodes. Multiple consecutive days, in particular days around those two deadlines. This confirms that the challenge of resource contention in Chapter 1 must be addressed, because few
Cumulative Fraction of Bids

Delay

Patient
requests could be allocated during such times.

2.3.3 Diverse Agent Values

![Bid value distributions: submitted total values by all agents.](image)

Figure 2.7: Bid value distributions: submitted total values by all agents.

Next, I look at whether agents bid different values and, if so, how they are distributed. Figure 2.7 plots the CDF of bid values (total value for a bid) submitted by all agents. It shows that bid values for testbed resources vary substantially, spanning orders of magnitude. Figure 2.8 plots distributions of bid values per node hour (per unit value for a bid) for the seven most active agents in the system. Bid values of each agent are distributed relatively evenly, suggesting that these ranges are not due to a few anomalous bids over the relatively lengthy four-month period. Furthermore, Figure 2.9 plots the median per nodeslot clearing price for both MICA2
and MICA2DOT nodes over time. These prices are computed by dividing the bid value $v$ of a winning bid by the requested $n$ nodes and $s$ slots (in hours). Unallocated nodeslots are assigned a price of zero. For a given hour, prices of all MICA2 nodes are examined and the median nodeslot price for that hour is plotted. A similar procedure is performed for MICA2DOT nodes.

Of particular interest in this graph are the two sequences of prices from days 45 to 60 and days 105 to 120 (i.e., periods leading up to conference deadlines). These sequences show that the value of testbed resources, as measured by market prices for nodes, increased exponentially (logarithmic y-axis) during times of peak contention. This further suggests that allowing agents to express valuations for resources to drive the resource allocation process is important for making effective use of the testbed (e.g., to distinguish important...
Cumulative fraction of bids
use from low-priority ones). However, it also suggests that agents become more and more desperate to acquire resources as deadlines approach. As it turns out, it is precisely during these times that agents will try their hardest to strategize and perhaps game the system (as shown in Section 2.4). Collectively, these graphs indicate that agents are willing to assign bid values that reflect their own changing priorities. Therefore, the benefit of addressing the challenge ignored agent values in Chapter 1 can be significant.

Figure 2.9: Median market prices (per nodeslot).
2.4 Observed Strategic Behaviors

As mentioned before, the first-price repeated combinatorial auction is not strategyproof. This design decision results in agents exhibiting strategic behaviors in the system. In this section, I discuss these behaviors that emerged in Mirage and that validate the presence of selfish agents challenge in Chapter 1.

During the first four months of operation, Mirage employed two versions of the auction mechanism, A1 and A2, and observed four primary types of strategic behaviors from agents. The first auction mechanism, A1, was deployed from December 9, 2004 to March 28, 2005 (110th day). This auction is open-bid in which all outstanding bids submitted are visible by all agents. For every auction, the resources available for sale are a 148-node × 72-hour window. Agents can submit bids for resources that start and end between any of the 72-hour slots.

In response to the strategic behaviors observed in the agents, a modified mechanism A2, was deployed on March 29, 2005 (111th day). There are two changes in A2. First, it is a sealed-bid auction (no agent can see bids of other agents), in response to the first two strategies S1 and S2. Second, it extends the window to 148-node × 104-hour, but with a laststart time at the 72nd hour, in response to strategy S3. This means that no bid could start (but it could end) between the 73rd to 104th hour. Details of these changes and their motivations will be made clear in the following discussion.

The data presented in previous sections come from both A1 and A2, for their respective time frames. The following are descriptions of the four primary strategic behaviors observed during the A1 phase and the changes made to mitigate them, if any, in the A2 phase.
2.4.1 S1: underbidding based on current demand

In A1, all outstanding bids were publicly visible. Sometimes, especially when resource contention was not serious, some agents would bid dramatically lower amounts rather than their recently submitted values. As an extreme example, one user would frequently bid values of 1 or 2 when few other bids were present. Many bids were similarly lower than typically submitted values.

While underbidding to try to pay the minimum, in the absence of competition, is not unreasonable, it does raise two issues. First, most agents will end up doing it, increasing the need for each agent to figure out how much to bid. This means instead of submitting a bid value equal to the agent’s true value, an agent’s process becomes more complicated as they must look at previous bids in order to submit a lowest winning value. Second, with agents consistently bidding below their true value (except when contention is high), the system outcome is not allocatively efficient (not maximizing aggregate value).

2.4.2 S2: Adaptive bidding

More sophisticated agents further enhanced their bidding strategy from S1. Instead of just looking at previous prices, some agents adaptively refined their bid values in response to how the other agents were refining theirs. In theory, this should have little effect: agents with true high values should eventually outbid those with lower values, after sufficient adjustments. In reality, however, the problem is that not all agents behave this way. Most agents in Mirage bid just once and then logged out (to do other things and wait for results), and some stayed online longer but modified their bids only once. Only a small number of agents
Chapter 2: Market Deployment Lessons 33

Stayed online long enough to make last minute modifications to narrowly “beat out” the other bids. In summary, S1 and S2 both point out the issues of agents manipulating their bids [81],

with both the system and most of the agents suffering from sub-optimal outcomes. The A 2 mechanism was deployed specifically to minimize the ability of agents to base their own bid values on those of other agents. This increase in complexity is unwarranted and is a good reason to push towards a strategyproof auction mechanism. In a strategyproof auction, an agent’s optimal strategy is to always submit its true value as the bid value.

2.4.3 S3: Rolling window manipulation

As resources in Mirage are partitioned by space and time, strategic behaviors are not specific to values (as in S1 and S2). This next strategy, S3, was not anticipated when Mirage was first designed. It occurred often whenever the entire window of resources was almost fully allocated (e.g., before a conference deadline).

As an example, assume that the entire 148-node × 72-hour window is allocated at time t. For each upcoming auction at t + 1, t + 2, ..., a total of 148-node × 1-hour slots are freshly available. An agent bidding for 32 hours thus must wait 32 (or more) hours before the system has enough resources to possibly fulfill his bid. Until then, any other bids with fewer than 32 hours, regardless of values, will be prioritized over this agent’s bid.

The problem here is that agents can exploit this by simply bidding for durations shorter than those of other bids in the system. Bidder i can break his desired duration of 32 hours into 2 sequential 16-hour bids, thus winning and blocking j’s 32-hour bid. Of course, other

An example can be found in Table 2.1. Agent U2 submitted at 3:58PM and left while strategic agent U1 submitted three separate bids in the next few hours.
agents such as j can also follow the strategy and begin to break down their bids into 8-hour increments.

However, this competition will quickly lead to durations getting shorter and shorter, finally reaching 1-hour slots. This totally disrupts the system's goal of offering agents the ability to reserve enough resources in advance with sufficient duration. Instead, agents have to bid every period, making participating in the system extremely labor intensive. Furthermore, there is no guarantee that an agent can win all the required slots. An agent can easily win only some of the 32-hour slots it seeks, making the allocations useless.

To mitigate S3, auction design A 2 was deployed with a larger 104-hour window and a laststart time at the 72nd hour. No bid can be allocated after the 72nd hour. Again, assume the window is fully allocated. Just as in A 1, a bid for 32 hours will have to wait 32 hours before it can be considered. However, bids of all other durations also have to wait 32 hours as well, eliminating the effects of manipulating reported job durations altogether.

2.4.4 S4: Auction sandwich attack
The last strategy observed involves agents who exploit two pieces of information: (i) historical information on previous winning bids to estimate the current workload, and (ii) the heuristic nature of the auction clearing algorithm. Some agents employed a strategy of spreading their needs across several bids for the same auction, all of which combined to win all the nodes but only one of which had a high value per node hour. The high value bid is set such that it will rank higher over all other agents' single bids for all the nodes (this behavior usually occurred before deadlines). Thus, no other agents' bids can backfill the remaining slots (after some nodes are claimed by the first agent). As a result, the first
agent’s remaining low value bids win easily, filling those slots at extremely low prices. An actual occurrence is shown in Table 2.1.

<table>
<thead>
<tr>
<th>Time</th>
<th>Bid</th>
<th>Agent</th>
<th>Value</th>
<th>Nodes</th>
<th>Hours</th>
</tr>
</thead>
<tbody>
<tr>
<td>04-02-2005</td>
<td>03:58:04</td>
<td>#1</td>
<td>U2</td>
<td>1590</td>
<td>97</td>
</tr>
<tr>
<td>04-02-2005</td>
<td>05:05:45</td>
<td>#2</td>
<td>U1</td>
<td>5</td>
<td>24</td>
</tr>
<tr>
<td>04-02-2005</td>
<td>05:28:23</td>
<td>#3</td>
<td>U1</td>
<td>130</td>
<td>40</td>
</tr>
<tr>
<td>04-02-2005</td>
<td>06:12:12</td>
<td>#4</td>
<td>U1</td>
<td>1</td>
<td>33</td>
</tr>
</tbody>
</table>

Table 2.1: Example of strategy S4 (auction sandwich attack) on 97 MICA2 nodes.

Agent U1 submitted three bids, the key one being bid #3. Although the bid is only 130, the per-node value is 0.813 (130/(4 \cdot 40)). In contrast, U2’s bid of 1590 produces a node value of only 0.512 (1590/(32 \cdot 97)). Since the auction considers the per nodeslot value, U1 wins. Once bid #3 won and was allocated 40 nodes, there was no way bid #1 could be allocated at all, since not all 97 nodes were available. As a result, agent U1 backfilled the remaining 57 nodes with bids #2 and #4, a 24-node bid and a 33-node bid, both at extremely low values.

### 2.5 Challenges and Refinements

Designing an appropriate auction mechanism is key to addressing strategies such as those in the previous section. Ideally, the goals for such a mechanism include: (i) strategyproofness, (ii) computational tractability, and (iii) allocative efficiency. The Generalized Vickrey Auction (GVA) [47, 96] is the only known combinatorial mechanism that provides both strategyproofness and efficient allocations. However, it is also computation-
ally intractable, as it is NP-hard to calculate the allocation and individual payments. Other GVA-based mechanisms exist that replace the allocation algorithms in GVA with approximate ones to provide tractability. In this case, however, strategyproofness is no longer available [74]. The goals of tractability and strategyproofness are thus in conflict in general [55], and one must make design tradeoffs to achieve them. Additionally, GVA is a static mechanism and is therefore not suitable for dynamic systems such as Mirage. With these in mind, below is a two-phase roadmap for improving Mirage. The first phase involves short-term improvements to the current mechanism to mitigate the effects of existing agent strategies. The second phase involves designing a new mechanism that approximately achieves the above three goals simultaneously.

2.5.1 Short-term improvements
Reasonable short-term improvements would augment the auction with additional rules and fees to further mitigate strategic behaviors. To further mitigate S1 and S2, transaction fees can be applied to every bid submission as well as modification. These fees would serve as a disincentive for an agent who understates a bid and intends to iteratively refine it. To eliminate S4, a system may restrict each agent to having one outstanding bid at a time and/or mandate that agents not have multiple overlapping allocations in time. Another approach to eliminating S4 is to modify the heuristic algorithm such that if an agent does have bids in which allocations could overlap in time, then those potential allocations are considered differently; for example, such bids are sorted from lowest to highest value per node hour, or simply sorted randomly. In effect, this allows bids for overlapping allocations but creates a disincentive for agents to place such bids.
2.5.2 Towards a strategyproof mechanism

Clearly, it is important to evaluate the goals and identify tradeoffs in designing a new mechanism. Computational tractability is a fundamental requirement for operational reasons - optimal combinatorial auctions can take too long (e.g., fail to clear within an hour for Mirage), lowering the promise of providing a dynamic service.

Even with a tractable mechanism, there are likely to be better strategies that we can discover. The experience from Mirage shows that identifying new types of strategic behaviors in the first place is hard. Therefore, it is important to design new mechanisms that are strategyproof with respect to all attributes that an agent expresses in its bids, including value, size (nodes and slots), and timing information. A strategyproof mechanism can lower complexity for agents and system designers. To address this, in Chapter 3, I use online mechanism design to introduce a mechanism that makes fast decisions and is strategyproof across the bid attributes discussed in this chapter and across multiple auctions.

2.6 Answers to Research Questions

2.6.1 Do markets work?
The Mirage deployment provides data that indicates agents do participate. The data also shows that there is enormous unpredictability of workload demands. Through an expressive bidding language, agents fully used it to submit bids with different values, sizes, and timing. While Mirage does not solicit true information from agents (e.g., values), the varying information provided by each agent shows that a system must provide flexibility in how it allocates resources.
2.6.2 Do people game?

Agents exhibited various strategic behaviors in Mirage. It is important to point out that the agents were not total strangers—it was a relatively tight-knit environment where most agents knew each other. Nevertheless, some agents started applying strategies and gained an edge. What is important, however, is that as a result, many of the other agents also started applying strategies. This is indicative of how selfish behavior can quickly spread as everyone simply wants to obtain the desired amount of resources for the best price. The result of this behavior is mostly negative for everyone involved, for example, strategy S3 (multiple bids for shorter time slots to ensure lowest price, and a higher chance of allocation of contiguous slots) hurts total value and increases the complexity for every agent.

2.7 Summary

Despite using a repeated combinatorial auction known not to be strategyproof, Mirage has provided empirical data on both usage patterns of market-based methods and strategic behaviors. The observations of significant resource contention and a wide range of bid values, sizes, and timings suggest that auction-based schemes can deliver large improvements in aggregate value when compared to traditional approaches such as proportional share allocation or batch scheduling. Fully realizing these gains, however, requires designing strategyproof mechanisms that can mitigate strategic behaviors and can handle the dynamic nature of such systems. This will be addressed in the next chapter.
Chapter 3

Online Mechanism Design

3.1 Introduction

In Chapter 2, I described how using data generated from agents using Mirage provided valuable insights into the use of markets for distributed systems. Specifically, Mirage provides real-world examples of how agents bid and behave in the repeated combinatorial auctions. However, there are several shortcomings of Mirage. In particular, it is exploitable since agents who deployed an array of strategies via manipulation of bid values, resource sizes, and allocation timing in placing bids succeeded in gaining more resources than the non-strategic agents. Thus, the ability and wherewithal of some agents to game the system through the use of these strategies renders a system less productive for agents unwilling or unable to be strategic.

To make matters worse, frustrated non-strategic agents may start behaving strategically. Over time, an increase in the number of agents behaving strategically will increase the number and variety of bids for the same finite set of resources and eventually clog the
system. This, in turn, will lead to the need for administrators to monitor and intervene, which defeats the goal of having a mechanism in the first place.

In this chapter, I describe the design and implementation of Roller, a mechanism that mitigates strategic behaviors as a first-order requirement. If strategic behaviors are mitigated, the complexity of using the system will be dramatically lowered for agents and administrators, making the system scalable and effective.

Note: From this point forward, I will use a general language to describe resources that are to be allocated in order to emphasize the generalizability of Roller to systems offering various types of resources (e.g., sensor network, cloud computing systems). I will use the term node to denote the resource to be allocated and slot to indicate the length of time an agent requires the resource. I will use size or nodeslot to indicate the combination of nodes and slots an agent is requesting. The allocation timing defines the periods during which an agent desires an allocation and consists of three parameters: arrival, departure, and patience refer to the first time period, last time period, and the total number of time periods over which the allocation can be made, respectively.

3.1.1 Research Questions

†There are two main questions to address. First, is Roller strategyproof with respect to value, size, and allocation timing submitted by agents? An agent can simply submit its true value as its bid value with a strategyproof mechanism. Because the total number of agents as well as their individual demands for size and allocation timing vary over time, a mechanism that is strategyproof only with respect to value is not sufficient. Thus, the
†As illustrated by some of the strategies in Chapter 2.
goal of a designer is to consider as many strategies as possible and create mechanisms to minimize the potential agent gains (e.g., obtaining more size for lower value) for deploying such strategies.

Second, can Roller achieve high value and responsiveness? Achieving high value requires the system to consider different allocation combinations and determine the optimal one. This, in turn, increases response time to agent requests, defeating a critical element of distributed systems. Thus, any new mechanism must be designed to strike an acceptable balance between these competing goals.

3.1.2 Chapter Overview

To answer the research questions, I explore different aspects of designing Roller. First, I outline the requirements of the mechanism in Section 3.2. Next, I describe the Roller mechanism, including the bidding language and the determination of winners and payments in Section 3.3. Then, I present a justification of the decisions made in designing the Roller mechanism in Section 3.4. In Section 3.5, I specify the workloads and metrics required for the various experiments. These include tuning Roller when the resource supply is either fixed (Section 3.6) or varying (Section 3.7), and comparing Roller with other algorithms (Section 3.8). I explore the $\rho$-late allocation rule that is essential for strategies related to allocation timing in Section 3.9. Finally, I discuss future refinements and testing for Roller in Section 3.10.
3.2 Requirements

In this section, I describe the key design requirements for the Roller mechanism. These requirements are based primarily on the lessons learned from the Mirage experience and are intended to address the shortcomings of Mirage.

3.2.1 Strategyproof

In order to make the mechanism strategyproof, the mechanism must mitigate an agent’s incentive to manipulate the following parameters to gain an edge over other agents. First, the bid value submitted by an agent should equal its true value. Second, the size information submitted should be truthful. This includes nodes (number of nodes needed) and slots (duration of time the nodes are needed). Third, the timing that the agent needs an allocation to start should reflect true arrival (the first time period an allocation can be considered) and departure (the last time period to be considered). I refer to the number of periods between arrival and departure as the patience of the agent.

3.2.2 Maximize Value and Revenue

The mechanism should aim to achieve high “allocative efficiency” by selecting resource allocation outcomes that maximize the total true value of agents in the system. For “forprofit” systems, a secondary goal can be to extract reasonable revenue. In an ideal world, the mechanism would compare the true values of different agent bids to make decisions. However, this is not feasible because true values are private information to their respective agents. Therefore, to maximize value the mechanism can only use submitted bid values.
This highlights the importance of the previous requirement—a strategyproof mechanism can treat bid values as true values.

### 3.2.3 Responsive and Computationally Tractable

To support agents with dynamic requests, mechanisms with long clearing times such as combinatorial auctions may be unsuitable. A mechanism with long clearing times often appears “busy” or “unavailable” to agents. In fact, if the mechanism can make decisions quickly, the agents are notified sooner and thus do not have to monitor or join more auctions than necessary. When there are other competing systems offering similar services to agents, a non-responsive system can easily lose agents’ interests. Thus, a system should seek to be responsive by using mechanisms that are computationally tractable.

### 3.2.4 Other Assumptions

In this chapter, I assume the use of real currency in the form of U.S. dollars (USD) throughout for both agents’ true values and submitted bid values. Virtual currency can be used by Roller as well, but will be discussed in Chapter 4. Here I apply standard mechanism design assumptions [97], in that agents have a private true value model and have no budget constraints, i.e., all agents can afford to pay their submitted bid values. Agents’ true values, desired resource sizes, and allocation timing, are independent and identically distributed. Furthermore, agents’ true values are non-negative and once determined, are not affected by any external events (e.g., how much the other agents are bidding). Lastly, each agent has single-minded preference (i.e., the agent desires at most a single set of resources at any one time).
3.3 The Roller Mechanism

I now introduce Roller, a mechanism designed specifically for distributed systems such as Mirage. Roller is strategyproof with respect to value, size and is configurable in regard to providing strategyproofness for different aspects of allocation timing. Roller is comprised of the following key components that are fundamental to mechanism design [73]: a resource space that defines an abstraction of time-based resources for allocation, a bidding language that acts as the interface for agents to submit bids and describe resource needs to the system, an allocation rule that determines winning bids for resources, and a payment rule that calculates how much winning bids should pay the system.

Roller employs an ongoing sequence of auctions to support dynamic requests and accepts bids from agents at any time. In brief, several major steps are taken periodically. Agents first submit bids to Roller as soon as their needs arise. An auction is run periodically, and nodes that are partitioned into time-based slots are for sale. During each auction, all bids that qualify for the specific nodeslots available are compared—those with the highest unit values and a demand that fits the available nodeslots are the winners. Their prices are calculated across multiple auctions to ensure strategyproofness with respect to allocation timing. I expand on the details of the above components and steps below, and provide proofs of Roller's strategyproofness in Appendix A.

3.3.1 Resource Space
In Roller, the set of resources available for allocation is represented by the rolling window abstraction. Consider a system with \( N \) identical nodes available over time periods...
\[ 0, T \]. I denote the rolling window as:
\[
R = (N^R, S^R, L^R). \tag{3.1}
\]

Visually, as shown in Figure 3.1, the window is a grid of \( N^R \) nodes by \( S^R \) (call this size \( \text{maxdur} \)). Each slot can start at \( L^R \) and end on \( S^R \), or \( t, \ldots, t + S^R - 1 \) are available. Slots. Each slot is of an arbitrary size (e.g., an hour) chosen by the system administrator. \( L^R \) is the last start time, i.e., all bids must be allocated on or before this time. \( L^R \) is used so that all bids can compete fairly. Specifically, \( L^R \) is set such that bids with the largest acceptable slot size \( s \). I will explain more about the rationale in detail in Section 3.4.

Figure 3.1: Rolling window abstraction: with \( N^R \) number of nodes and \( S^R \) number of slots, at time \( t \). All allocations must start on or before \( L^R \) in this example. \( \text{maxdur} = 3 \), thus all bids with slots = 3 can start at \( L^R \)

I use \( T \) so that the experiments can have a finite scope. In real-world systems, \( T \) would be 8 because these systems run continuously.
These slots represent the resources of future periods of time, thus allowing Roller to allocate resources to agents in advance, shortening response time. The window is “rolling” because as it advances to another time period $t_{n+1}$, it “rolls” to the right, and slots of $t_n$ are all disabled (since those times have passed) and new slots of $t_{n+1}$ are added at the far right end. The following is an example. Example: Consider $R = (4, 5, 3)$. At $t_1$, the slots are $[t]$ (Figure 3.1). Column $t_3$, $t_2$, $t_4$, $t_5$, $t_6$ is the last start time, where all bids will be considered for slots starting on or before $t_1$, $t_2$, but not after. For instance, a bid for $1 \times 2$ nodeslots can be considered for $(t)$, and $(t_2, t_3, t_4)$, but not $(t_4, t_5)$. As time passes, the window “rolls” to the right. At $t_3$, column $t_1$ phases out and column $t_6$ rolls in, making $[t_2, t_3, t_4, t_5, t_6]$ now available. Lalso shifts and is now column $t_4$. See Figure 3.2 for an illustration.

Figure 3.2: Advancing rolling window: At $t_2$. All columns “shift” to the right. Column $t_1$ (in gray color) phases out while $t_6$ rolls in. $t_4$ also rolls right and is now column $t_6$. 

![Figure 3.2](image-url)
3.3.2 Bidding Language

The bidding language is the interface for agents to submit bids to the system. When a resource need arises for agent $i$, it submits a bid $b_i$ to the system in the form of:

$$b_i = (w_i, n_i, s_i, a_i, d_i),$$

where bid value $w_i > 0$ is the maximum amount of USD that agent $i$ is willing to pay in order to get exactly $n_i$ nodes for a set of contiguous $s_i$ slots (shorthanded as $n_i \times s_i$ or $n_3 i$ node slots; $0 < n_i \leq N$ and $0 < s_i \leq 3$ and no later than departure time $d_i$ and $t_3$). If $i$ wins in $t_3 = \maxdur$, the nodeslots must be available for use starting no earlier than arrival time $a_i$.

Example: Assume a testbed with $N = 1$ node that runs over $T = 10$ periods. A bid $b = ($100, 1, 2, 3, 5) can be interpreted as follows. Agent $i$ is willing to pay at most 100 USD for $1 \times 2$ nodeslots that start between $t_3$, the actual nodeslots assigned will be $t_3(2$ slots). Similarly, if $i$ wins in $t_5$, the nodeslots will be $t_5$ and $t_6$. The Roller bidding language has some minor differences compared to Mirage. In Mirage, agents could specify a particular subset of acceptable nodes for allocation consideration, whereas Roller always considers all nodes for every bid. Agents in Mirage also can limit consideration to nodes of certain frequency range; Roller ignores this feature to simplify the discussion. Implementing these features in Roller in the future is feasible, as the space of possible allocations in Mirage is a subset of that of Roller and thus more limited.

3.3.3 Allocation Rule

The allocation rule determines winning bids and their respective allocated resources. For the basic allocation rule, at every time $t_n$, an auction will allocate resources that are
Chapter 3: Online Mechanism Design 48

nodes over the next $S_R$ and $t_2$ is allocated to a future slot $t_3$

comprised of $N_R$

1, agent $i$ submits $b_i$

A greedy algorithm is used to determine winners (based on the scheme in Lehmann et. al. [59]). First, all bids are sorted by $w$. This ranks bids by reported unit values. Starting with the highest bid, the algorithm evaluates whether nodeslots that
satisfy the bid are available. If nodeslots are found, the bid will be declared a winner, the agent notified, and the assigned nodeslots marked for the agent. It is also possible that agents may be informed of winning decisions early, despite not being able to start using the allocated resources until a future period. Example: At time \( t = (100, 1, 1, 1, 3) \) to a system with window \( R = (1, 3, 3) \). Assume at the window slots for \( t \) are already allocated to another agent (e.g., that agent won before \( t \)). Thus, the bid \( b \), but the agent is notified now in \( t \).

### 3.3.4 Payment Rule

The payment algorithm I use is the VirtualWorlds [71] scheme. The goal is to find the best possible price for which a bid can win in the different auctions in which it qualifies during its stated allocation timing (the periods between arrival \( a \) and departure \( d \)). The motivation is to mitigate time-based strategies related to the allocation timing.

1. At the end of each auction at time \( t \), each winner \( i \)'s price equals the unit bid price \( v \) of the first agent \( j \) who loses out because of \( i \)'s allocation. In other words, if \( i \) had not participated in the auction, \( j \) would have won. Denote this price as \( p_i \).
2. A VirtualWorld is created for each winner \( i \). This is an abstract state of the system without \( i \). That is, if \( i \) had not yet shown up at current time \( t_n \), what bids would be present in the system and which nodeslots would have been allocated?

3. For the remaining auctions that occur within \( i \)'s allocation timing, run VirtualWorld for each \( i \) at the beginning of each auction at some time \( t \). The goal is to answer the question, “If \( i \) joined at time \( t \) instead, would it have won?” This is tested by adding new bids that arrive at \( t \), removing expired bids, and adding \( i \) to the VirtualWorld bid list, thus essentially running an imaginary auction that does not affect the actual allocations.

\[
\begin{aligned}
\text{(as in Step 1).} & \quad = p_v \\
\text{If } p_i & \quad \text{in} \in S_i \\
\end{aligned}
\]

5. By the end of the agent's departure time \( d_i \), \( p_i \) (unit price times number of nodeslots).

3.3.5 Late Allocation

4. If \( i \) wins in the VirtualWorld, calculate its price \( p_v \)

\[
= \text{ then set } p_v. \text{ In words, whenever a lower price is found in a VirtualWorld auction, } i \text{'s obligated price is updated.}
\]

have paid across its stated allocation time.
With VirtualWorlds, agent i has no incentive to choose which auction to join. Intuitively, this is because it does “join” all auctions within its allocation timing automatically via VirtualWorlds and receives the best possible price. One last strategic issue remains. Since bidder i will be exposed to all auctions in its specified allocation timing periods in [a], it can potentially game by specifying a longer
patience (i.e., the difference between departure and arrival, \(d_i - a_i + 1\)) by over-reporting \(d_i\). In this case, \(i\) gets to participate in more VirtualWorlds auctions and have a higher chance of further lowering its payment while still being allocated early enough.

The idea introduced here is to establish a \(\rho\)-late allocation rule. With probability \(\rho\) (0 to 100%), a winning bidder \(i\) will have its resources allocated at the end of its submitted departure. For example, bidder \(i\) submits allocation timing \([1, 10]\). Instead of finding resources starting from period 1 and towards period 10, the algorithm tries to allocate as close to period 10 as possible, given the constraints of the current allocations to other bids.

This change will not affect a truthful bidder because the allocation, if any, is still within patience. For a strategic bidder that over-reports its departure (e.g., real patience is from time periods 1 to 5 but is reported as 1 to 10), it risks getting resources that have no value but still must be paid for. For example, if the agent’s true value is $100 for receiving the resource between time \([1, 5]\), the resource is worthless if allocated in time \([6, 10]\). If the payment is $50, the agent ends up spending USD for nothing (zero true value realized). Thus, the agent must decide whether the risk presented by the \(\rho\)-late allocation rule is worthwhile.

In considering the use of a \(\rho\)-late allocation rule, there are also various tradeoffs: (a) possibly lower allocative efficiency due to less effective utilization of resources; and (b) new opportunity for manipulation by reporting an early arrival. For (b), agents can obtain lower payments if their allocations are made “late” as it is compatible with their true departures. However, without the \(\rho\)-late allocation rule, agents have no incentive to report early arrival because the allocations will yield no value.
3.4 Design Justifications

There are several basic and implicit design decisions made for Roller that warrant discussion. In this section, I will describe and justify the decisions made to answer the following questions. I also discuss some design differences between Roller and Mirage.

1. Why use some arbitrary window slot size instead of a size of one? Or infinity?

2. Why use a rolling window instead of simply a series of static and disjointed ones?

3. Why restrict bids with a laststart time?

3.4.1 Window Slot Size

The simplest way to partition the resources over time is probably by selling just the current time slot at every time period \( t \) (see Figure 3.3). Computationally, it is easy because every bidder will be looking for the same slot size. There is also no need to worry about selling resources beyond the current time \( t \). Nonetheless, setting a single slot size in this way does not work for several reasons. For agents that want multiple time slots, they must submit multiple bids, each winning a single slot. Due to competition, there is no guarantee that an agent that wants \( s \) slots will indeed get them all. The outcome is worse if these slots must be consecutive—winning all but one slot, for example, will still render no value to the agent. This is the well-known problem of “exposure” [35] in non-combinatorial auction for bidders seeking complementary resources. Therefore, I focus on selling multiple slots at any time period instead (see Figure 3.4).

Another window slot size option would be to sell all possible future slots at any time period, i.e., have essentially an infinite window. Intuitively the upside seems obvious, since
agents can essentially bid for any number of slots for any future time periods. However, there are several issues with this scheme. First, the seller may not be in a

\(3\) position to offer all of its resources up front. For example, the seller cannot guarantee that it will have the resources available far in the future. Second, an infinite window will likely adversely affect value and revenue. By allocating to a bidder that wants slots far in the future, say 1,000 periods from current time \(t\), the system gives up opportunities to allocate the same slots to agents with higher bid values in the future. Thus, a system that tries to maximize value or revenue suffers with an infinite window. Overall, offering infinite slots does not seem practical, but the actual preferred window slot size really depends on the goals of individual systems.

### 3.4.2 Rolling vs. Static Window

Next, I want to justify why a rolling window is preferred over a static window that is available every few periods. Figure 3.5 shows what a rolling window looks like.

\(\text{Compare}\)

\(\text{Example: the seller leases the resources from another company that requires renewal on a yearly basis.}\)
it with a static window, such as the one in Figure 3.4. The static window sells resources in distinct blocks of slots that never overlap. The key is that for a static window, a slot at time $t$ is offered at only one auction. For a rolling window, however, that same slot will be offered during multiple time periods, essentially equal to window slot size $S_R$. For example, slot $t_3$ is offered in three auctions that begin in $t_1$, $t_2$, and $t_3$.

The other benefit of a rolling window is that it can handle dynamic requests more frequently. By using a static window with a size of $S_R$, bids essentially are considered only every $S$ time periods. For example, with a 10-slot window, auctions are only run in periods $t$, or $t + 10$, or $t + 20$. When the window gets bigger, bids will have to wait even longer for the system to make a decision. Instead, by moving to a rolling window that runs every time period, allocation decisions are made frequently, making the system more responsive to agent requests.
3.4.3 Laststart Time

Third, I want to justify why the laststart time is critical. Because the system sells resources on a rolling window basis, the window rolls every time period, as shown in Figure 3.5. Specifically, for a window with $S_R$ slots, as the system moves from time $t$ to $t + 1$, the following two steps occur:

1. The system removes the column of slots corresponding to $t$; this is the leftmost column visually. This is important because $t$ is now past and no longer refers to valid resources.

2. The system adds a new column corresponding to $t + S_R$; this is the rightmost column visually. This represents the new nodeslots that are now available because of the new time $t + 1$. This step is essential to maintain a valid window size of $S_R$.

The laststart time is critical for all bids to be considered equally. I start with an example of why the system will not work properly when there is no laststart. Imagine at the end of time $t$, all nodeslots of the rolling window are fully allocated. As soon as a new column is rolled in (the rightmost column), the nodeslots of this new column become the solely available resources for auctioning. How will Roller allocate at this time? It will scan the bids and consider only the ones that request exactly one time slot. These bids, regardless of bid values, will be allocated. All other bids will not be considered because the size does not match up. If this goes on for several periods, then only single-slot bids will win. Any bids requesting multiple slots will essentially be denied, as discussed in Chapter 2 (see “S3: Rolling window manipulation”).
Now imagine a window with a laststart time $t$ (see Figure 3.6). No bids will be considered for starting after $t$. Thus, all bids, including single-slot bids, will not be considered. Again, for a window that is fully allocated, it will take several time periods before these slots are available again. At that time, all bids will be considered equally.

![Figure 3.6: Rolling window with laststart time.](image)

I now use a simple experiment to illustrate the use of laststart time. I run Roller for just 15 time periods with a rolling window $N_R = 1$ and $S_R = 3$. Agents all bid for $n = 1$ node. They ask for either 1, 2, or 3 slots ($s_i$) with bid values 1, 10, and 30, respectively. Thus, bids with $s_i = 3$ should theoretically always win. Bids arrive uniformly every time period. Figure 3.7 shows how many nodeslots are already allocated for the rolling window at the beginning of each time period. On the left, it shows the results for a rolling window with no laststart. The window is almost always filled up and the smallest bids with salways win. On the right, results for laststart show a different story. All bids can compete, thus the larger bids $s_i = 3$ win often. The shape of the curve is up and down because $s_i = 3$ wins
every several time periods.

![Graph showing filled window slots at start of period](image)

Time

Figure 3.7: Laststart: How much the windows are filled at beginning of each time period.

### 3.4.4 Comparisons with Mirage

Both Roller and Mirage use the rolling window abstraction for allocating resources, including details like laststart. However, Mirage uses a first-price method to calculate payment, which is not strategyproof and results in agents using the strategies discussed in Chapter 2. Roller, on the other hand, is strategyproof across multiple bid attributes. However, agents in Roller do not know the actual payment until after its departure due to the payment rule.

Roller is still subject to advanced strategies, such as the “auction sandwich attack” observed in Mirage. Specifically, Roller cannot identify collusion [21] among multiple agents or detect false-name bids [105] submitted by a single agent. With collusion, two or more agents can coordinate submitting bids with value or timing that can alter allocation and payments to their benefit. For example, instead of both agents submitting at time t and competing against each other, they arrange to submit at different times to increase the odds of a) both bids getting allocated and b) lowering their payments due to less competition.
With false-name bids, an agent can perform similar tactics by submitting multiple bids under different identities.

A simple example for false-name bids is as follows. Given a 1x4 rolling window, agent i seeks 1x4 nodeslots for a total of $40 (or $10 per nodeslot) and agent j also seeks 1x4 nodeslots for a total of $20 (or $5 per nodeslot). Assume both agents have zero patience and need a decision immediately. To win, agent j submits 2 bids under different identities (e.g., “j” and “k”). The first bid from “j” is $11 for 1x1 nodeslot and the second “k” is $9 for 1x3 nodeslots (or $3 per nodeslot). The result is “j” wins with the highest unit bid value. Agent i will not be allocated as it seeks more resources than are available (1x3 left after “j”), and thus “k” also wins. Thus agent j locks out agent i.

### 3.5 Workloads and Metrics

#### 3.5.1 Workload

The workloads for experiments in this chapter are artificially generated. A workload defines the set of agents’ jobs that arrive over time to request resources from the system. In order to evaluate experimentally whether Roller is capable of serving systems such as Mirage from several perspectives, I specify parameterized workloads for use by different experiments.

**Specification**

A workload $L$ is generated from a collection of parameters,

$$ L = (T, P, \lambda, [n^\text{low} : n^\text{high}], [s^\text{low} : s^\text{high}], [\Delta^\text{low} : \Delta^\text{high}], w (m, [x^\text{low} : x^\text{high}]), (3.3) $$
where:

• $T$ is the number of periods the workload covers (e.g., 500). In my experiments, I run Roller for a set number of time periods $T$, although in real life, the mechanism should run “forever.”

• $P$ is the probability distribution for new job arrivals. By default, I use Poisson distributions in this chapter unless otherwise noted. $\lambda$ is the average new job arrival rate for the Poisson distribution.

$\Delta_{\text{low}} : n_{\text{high}}$ and $[s_{\text{low}} : s_{\text{high}}]$ means nodes $n_i$ define the range of bid patience. For example, $[1:5]$ means bid patience is uniformly drawn between 1 and 5. The arrival of each bid is set to the time period the bid is created. The departure is calculated by adding patience to the arrival.

$\Delta_{\text{low}} : [n_{\text{low}}]$ specify the ranges of nodes and slots. For example, $[n_{\text{low}}]$ are drawn uniformly from $[1,2,3]$ and slots $s$ from $[2,3,4]$. Thus the space of possible nodeslot pairs are $[(1,2), (1,3), (1,4), ..., (3,2), (3,3), (3,4)]$.

• $x_{\text{high}}$] is a function to generate a range for bid values $w$.

Parameters $T$, $P$, and $\lambda$ define the high-level aspects of a workload, i.e., how many bids arrive over time. The other parameters define the bid-level aspects, such as what nodeslots...
each bid seeks. To generate a workload, I take function L with a set of parameter inputs to programmatically generate a list of bids that can then be used as the input feed for any experiments.

Essentially, the program runs for $T$ time periods. For each time period $t \in T$, a number of new bids will be created based on $P$ and $\lambda$. For each bid, numbers are then drawn from $[n_{\text{low}}: n_{\text{high}}]$ to generate nodes $n$ and slots $s$, and $[\Delta]$ to generate $\Delta$ and in turn $\Delta_{\text{low}}: \Delta_{\text{high}}$ (see equation 3.2 on page 47, in which arrival time $a$ is simply the current time period $t$ and departure time $d_{i} = a + \Delta$). Finally, $w(m, [x_{\text{low}}, x_{\text{high}}])$ is called to generate the bid value $w_{i}: x_{\text{high}}$ for the bid as a last step. Each bid is then appended to a text file for use as input to be used in experiments.

Value Generations

Note that I assume true value equals bid value for bids in Roller, because of its strategyproof nature with respect to value. The $w(\cdot)$ method produces distributions that result in expected total values that are generally monotonically increasing. Monotonically increasing means $n_{k} > n_{i} \Rightarrow \bar{w}_{k} > \bar{w}_{i}$, i.e., agents have a higher expected value for larger nodeslots than smaller ones. The difference between the two $m$ methods is whether the values marginally increase or marginally decrease, as $n_{i}$ increases. In other words, how does the rate of change of value $w_{i}$ change as $n_{i}$ changes? Figure 3.8 illustrates the difference.

The $w(m, [x_{\text{low}}, x_{\text{high}}])$ function generates a range $[w_{\text{low}}: w_{\text{high}}]$ on value $w_{i}$. It takes two parameters: (1) $m$, which specifies one of two possible methods used to generate $w$, that is, marginally increasing (“$\uparrow$”) or marginally decreasing (“$\downarrow$”); and (2) $[x_{\text{low}}, x_{\text{high}}]$.
Chapter 3: Online Mechanism Design

Figure 3.8: Two monotonically increasing total value curves. Curve A exhibits marginally increasing values as the nodeslot increases, and curve B exhibits marginally decreasing values.

which specifies a range of unit values. Given the bid nodeslots ns as well as the maximum and minimum nodeslots (ns\textsubscript{min} and ns\textsubscript{max}), both methods m generate a range of valuations as:

$$[w_l : w_{hi}] = ns_i \cdot [y : y + z]. \quad (3.4)$$

The value of y depends on whether the method is marginally increasing or decreasing:

$$y = \begin{cases} \frac{ns}{ns_{min}} \cdot (x_{hi} - x_{ow}) + x_{ow} & \text{if } m = \uparrow \\ ns - \frac{ns_{min}}{ns_{max}} \cdot (x_{hi} - x_{ow}) + x_{ow} & \text{if } m = \downarrow \end{cases} \quad (3.5)$$

$$z = 0.3 \cdot (x_{hi} - x_{ow}). \quad (3.6)$$

The role of y and z is to map and transform the given unit value distribution [x: x\textsubscript{hi} hi] into w\textsubscript{ow}. This is illustrated in Figure 3.9. The resulting values are intentionally designed to be only generally (and not absolutely) monotonically increasing, because as the
figure shows, the distributions do overlap among similar nodeslot sizes to provide some randomness.

**Marginally Increasing Marginally Decreasing**

Next, I explain the distinction between supply and demand. Then, I will define metrics to measure value and responsiveness.

### 3.5.2 Supply and Demand

In this section, I define a model of supply and demand as a reference for the experiments. The demand is the total number of nodeslots requested over \([0, T]\) for workload \(L\) and is entirely driven by workloads and has no dependency on any components of Roller. Demand is estimated as

\[
demand = \lambda \cdot \overline{n} \cdot \overline{s} \cdot T,
\]

where \(\lambda\) is the number of new bids arriving in each time period, \(\overline{n} \cdot \overline{s}\) is the average

\[
bids\ nodeslots\ (ns)
\]

\[
ns_{\text{min}} \quad ns_{\text{max}}
\]
nodeslot size requested by these bids, and $T$ is the total number of time periods.
Supply is the number of total nodeslots made available by the rolling window over $T$ periods. This is calculated by multiplying the number of nodes with the total number of time slots over which each node is available:

$$\text{supply} = N \cdot T. \quad (3.8)$$

Lastly, Demand/Supply, or $\frac{D}{S}$, refers to the ratio between demand and supply and is denoted with the times (“×”) notation, e.g., $1\times$, $1.5\times$, $2\times$. The ratio of $\frac{D}{S}$ indicates how heavily loaded the system is. When $\frac{D}{S} = 1\times$, the number of nodeslots demanded equal the number of slots supplied. When $\frac{D}{S} > 1\times$, there are more jobs than can possibly be scheduled. When $\frac{D}{S} < 1\times$, some resources will go idle. Formally it is simplified as:

$$\text{demand} = \lambda \cdot \bar{n} \cdot \bar{s} / N. \quad (3.9)$$

Example: Given a system with $N_R = 8$ nodes and a workload with $\lambda = 2$ and $\bar{n} = \bar{s} = 2$, $\frac{D}{S} = 1\times$. For a workload with $\lambda = 4$ and $\bar{n} = \bar{s} = 2$, $\frac{D}{S} = 2\times$.

### 3.5.3 Metrics

As stated in the requirements, achieving high values and high responsiveness are important goals for Roller. In this section, I define two metrics that will be used for most experiments: system value $a$ and responsiveness $\beta$. In theory, the two metrics have reciprocal effects because getting a high value requires batching of bids and more computation.

4The supply is an estimate, as I exclude the few remaining slots offered during the last time period at $T$, when the rolling window also comprises slots $T + 1$, $T + 2$, ..., $T + S - 1$. I choose to exclude these as $T$ is quite large compared to $S_R$.\textsuperscript{R}
hence responsiveness to agents suffers. Thus, I aim to evaluate them together to find a balance in different situations.

System Value (a)
System Value $a$ is defined as the average true value captured per unit supply of the system:

$$a = \frac{\sum_{w \in N_{R}} w \cdot T}{\text{supply}}$$

(3.10)

where $w_{\text{in}}$ is the sum of the total true value of all winning bids, and supply is based on equation 3.8. The unit of $a$ is USD, the same as true values. System value $a$ allows the comparison of systems of different nodeslot sizes. The metric is not affected by varying demand/supply ratios.

The second key metric concerns how quickly allocation decisions are made for winning bids. For example, an optimal allocation algorithm may maximize total value by batching and thus become unresponsive. By informing winners as soon as possible, systems can further lower the overhead for participating in the mechanism. I define the responsiveness metric as

$$\beta = \frac{1}{\text{ttwi}}$$

An alternative metric to consider is to divide $w_i$ by the total true value of all bids received. When demand/supply is high (e.g., 10×), the metric will be extremely low, as there are many bids but few winners. The system may look like it is not performing in high demand, when in fact it may have captured a similar amount of winning values compared to a low demand/supply (e.g., 1×).
Table 3.1: Responsiveness of different bid instances. On the left column, each number represents time-to-win for a winning bid and x represents a losing bid. For example, 22x means there are three bids, with one of the bids being a losing bid. Where \( t_{twi} \) is the time-to-win (the number of time periods it takes for \( i \) to win; or the number of time periods it takes Roller to make such a decision), \( 1 = t_{tw} \). Thus, \( \beta \) is a number in the range \((0, 1]\). A small \( t_{twi} \) is good; \( t_{twi} = \Delta_i = 1 \) means bid \( i \) wins in the first possible time period. \( t_{tw} \) normalizes the number so that its range is from 0 to 1. A decision is most responsive for a bid if this value is 1 and least responsive as it approaches 0 (e.g., a bid with extremely long patience, such as 100 time periods, that gets allocated late). The
sum of $\sum_{i} t_{tw}$ divided by the number of winning bids $|K|$ gives us $\beta$, which measures the average responsiveness achieved by the system.

Table 3.1 shows a list of bid scenarios with different $\beta$ values. Each row specifies an example bid scenario and the corresponding $\beta$. The bid scenario represents all the bids in a complete mechanism run: a number represents the TTW of a winning bid, whereas an X represents a losing bid (thus has no TTW). For example, 1xx means there are a total of three bids, with two losing bids and one winning bid with TTW=1. Responsiveness $\beta$ is not affected by demand/supply because losing bids are not a factor in the expression.

3.6 Tuning Roller

Establishing parameters for the size of the rolling window is necessary in order to address the remaining requirements from Section 3.2: “to extract value and revenue” and to be “responsive and computationally tractable.” As the configuration of the rolling window dictates what resources are available at each time period, it directly affects the allocation and payment processes.

I approach this question of how to determine rolling window size by studying each of the window parameters separately. In this section, I fix supply $N$ and study the effects of different rolling window sizes $S_{RR}$. In Section 3.7, I will evaluate varying supply $N_R$.

3.6.1 Fixing Supply
I first present experiments to answer the question “given fixed supply $N_R$ and specific workloads, what window size $S_{RR}$ gives us the most balanced $a$ and $\beta$ performance?”
evaluate $S_R$ by comparing it against different individual workloads that are differentiated by one parameter of interest at a time. I specifically avoid varying multiple parameters simultaneously in order to get clear individual effects on metrics. For each of the following tests, I always work with $N_R = 8$, thus, supply $= 8 \cdot T$. Workloads vary for each test but are all derived from this base workload $L$, given by:

$$L = \left( T, P, \lambda, [n_{low} \cdot hig], [s_{low} \cdot hig], [\Delta{l}_{ow} \cdot \Delta{hig}], w, [x_{low} \cdot x_{hig}], R \right)$$

(3.12)
In descriptive terms, I run each experimental workload for $T = 500$ time periods. A Poisson distribution is used for bid arrivals where $\lambda$ is 3. The bid parameters of $N$ and $S$ are drawn uniformly between 1 and 3, while patience is 10. Bid value $w$ (which equals true value) is generated using both the marginally increasing and decreasing methods, with unit bid values $x$ drawn uniformly between 1 and 10.

### 3.6.2 Arrival Rate $\lambda$

The first experiments involve testing different workloads against rolling window size $S$ by varying only $\lambda$. $\lambda$ is a significant parameter because it directly affects demand/supply. Starting with base workload $L$, I vary $\lambda$ with the following set to generate six different workloads:

$$\lambda \in [1, 2, 3, 4, 5, 10].$$

Each workload is tested against a set of different window sizes $S$, from 3 to 20. The smallest $S$ size has to be 3 because $\text{maxdur} = s = 3$. I collect values of $a$ and $\beta$. 

- [5 w 0 0 ( P / Poisson [ 1 3 : , 1 0 : ) : ) ]
- [10 ]

3.
for each workload and $SR_R$ (e.g., $SR_R$)

$R_i = 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23$

Figure 3.10 shows the results for marginally increasing values. Each line represents a workload with a specific $\lambda$ and the points on each line are for a specific Ssize. First, for a given $S = 6$, observe that increasing $\lambda$ from 1 to 10 leads to higher $a$ and lower $\beta$.

For a given $\lambda$ (e.g., any one of the lines such as $\lambda = 4$), observe the following:
• For $\lambda = 3$, $a$ decreases gradually as $S$ increases. For example, with $\lambda = 4$, $a$ decreases from around 9.7 when $S_{RR} = 3$ to around 9 when $S_R = 10$. After that, $a$ drops off more rapidly when $10 = S = 13$. For $S_{RR} = 13+$, $a$ exhibits minimal decrease and remains virtually constant. On the other hand, for $\lambda = 1, 2$, $a$ remains
Chapter 3: Online Mechanism Design 68

virtually constant for all size SR s.

- For β, the effects are inverse, i.e., β increases as S increases, for λ = 3. For example, with λ = 4, β starts at about 0.55 when SR = 3. It increases to about 0.7 when SR = 10. After that, it further increases and finally reaches 1.0 when SR = 13+. For λ = 2, however, it starts with a much higher β at around 0.7 and quickly reaches 0.9+ between 5 = S = 12. For λ = 1, it reaches β = 1.0 as soon as SR = 5.

It turns out a larger rolling window does not result in value increase for the tested workloads. In fact, it does not even affect the actual amount of resources sold. This is because bids are simply allocated future resources with a larger rolling window. This feature hurts value as higher value bids that arrive later will not be considered. For results with
marginally decreasing (⇓) values, the patterns are similar (Figure 3.11). \(\alpha\) is lower overall for all points, whereas \(\beta\) is higher overall for all points (every \(\beta\) is near or above 0.8).

### 3.6.3 Nodes, Slots, and Patience

I next explore varying ranges for nodes, slots, and patience. The workloads are again based on L(equation 3.12), with arrival rate \(\lambda = 3\), and marginally increasing values. Window size \(S\) again will vary from 3 to 20. First, I use the following ranges for nodes requested by agents:

\[
[n_{\text{low}} : n_{\text{high}}] \in ([1 : 1], [1 : 2], [1 : 3], [1 : 4], [1 : 5], [1 : 6], [1 : 7], [1 \text{ low} : 8]).
\]

Figure 3.12 shows the results. For any given \(\lambda\) curve, the patterns largely resemble those for arrival rate \(\lambda\), that is, decreasing \(\alpha\) and increasing \(\beta\) as \(S\) increases. While \(\lambda\) in this case remains constant, higher \([n_{\text{low}} : n_{\text{high}}]\) ranges represent an increase in demand as well. Thus, higher node ranges achieve higher system value \(\alpha\), given \(S\). Next, I use the following ranges for slots requested by agents:

\[
[s_{\text{low}} : s_{\text{high}}] \in ([1 : 1], [1 : 2], [1 : 3], [1 : 4], [1 : 5], [1 : 6], [1 : 7], [1 \text{ low} : 8]).
\]

I vary bidder \(S\) from \(s_{\text{high}}\) to 20 (instead of the usual 3 to 20). For example, for \([1 : 5]\), \(S\) will be tested from 5 : 20. The reason is that by using a smaller \(S\), the larger bids will be excluded. The results are shown in Figure 3.13. Overall, the patterns are similar to both \(\lambda\) and node results. However, the results fluctuate more. This is due to the fact that with slot size potentially higher than node size (fixed at 8 nodes), there are more combinations of

Similar to results in the previous section, results for marginally decreasing values are similar for this section as well and are not shown.
allocations possible on a slot basis. When the number of bids in the system is high, some "gap" slots may arise. These are smaller slots that are created in between winning bids but not large enough for the remaining bids.

Finally, Figure 3.14 shows results for the following ranges for patience expressed by agents:

\[ [\Delta_{\text{low}} : \Delta_{\text{high}}] \in ([1 : 5], [1 : 10], [1 : 15], [1 : 20], [1 : 25]) \] .

Overall, patience \( \Delta \) has minimal effect on value \( a \) . Higher patience does not lead to higher \( a \) . The reason for this is that all the \( \Delta \) ranges still share the same level of demand/supply, unlike nodes or slots. \( \beta \) follows similar but tighter patterns as those in the previous \( \lambda \) section.
3.6.4 Analysis

I present my analysis of the results for the previous experiments in this section. First, all four parameters exhibit very similar patterns for both system value $a$ and responsiveness $\beta$. Table 3.2 provides a summary for marginally increasing values and for a fixed window slot size $S_R$. When the workload changes and results in increasing agent demand (e.g., increase in arrival rate, nodes, or slots), $a$ should increase because there are more valuable bids for the allocation rule to choose from. However, responsiveness suffers because there are more competitions for every bid. Most of the results presented considered the effect of window size $S_R$ while varying aspects of the workload. Intuitively, a larger rolling window with high $S_R$ should result in higher $a$ —after all, more bids should be captured resulting in higher value.
But this
Figure 3.14: Effects of $S_R$ on different $\Delta$ ranges.

The rationale is that $S_R$ does not affect the real amount of resources available. Instead, the window simply enables the sale of more future resources at every time period. Because my results are workload specific, I discuss whether alternative workloads may yield a potentially different conclusion in Section 3.6.5. This decrease in system value can be further explained. Bids of lower values are able to win allocations in earlier auctions, before bids of higher values arrive in a later period. This happens more frequently as $S$ increases, since more lower value bids can be assigned early, as long as their patience is long enough (i.e., to be assigned to the “far right” of the window). This results in those slots being unavailable as time passes, blocking future (high
value) bids from winning. When demand is greater than supply (e.g., $\lambda = 3$), the desirable range for window slot sizes seems to be between $10 = S_{\text{R}} - 12$. These sizes achieve the most balanced $a$ and $\beta$ combinations. Based on this, a good range to use for the rolling window slot size $S_{\text{R}}$ is

$$S_{\text{R}} = \Delta \hat{\Delta} [s], \quad (3.13)$$

which equals mean patience (e.g., 10) plus mean bid slot sizes (e.g., 2 for [1:3]). When demand is less than or close to supply, the system is able to capture most of the submitted values for various window sizes $S_{\text{R}}$. While it seems logical to use a smaller $S_{\text{R}}$ to keep things simple, I recommend preparing for high demand rather than choosing a small window size that will suffer when demand does rise.

For responsiveness, the general results is:

- Higher $S_{\text{R}}$ results in better responsiveness $\beta$.

When demand is low, most bids can be allocated immediately, resulting in a very high $\beta$. However, as demand increases, there are more and more bids competing, which leads
to many bids experiencing a delay in getting a decision from Roller. By using a higher rolling window size $S_R$, more bids can be accepted for allocation if demand is fixed. In other words, if nothing else changes, then a larger rolling window implies offering more future resources for sale now. This means that more bids can be accepted, thus improving responsiveness. However, as soon as $S_R$ is beyond the reach of any bid’s patience, then the extra slots will remain unallocated and unpaid for.

Finally, the main difference between marginally increasing and decreasing values lies in responsiveness. For the latter, the winning bids are those with smaller nodeslots. They do not take up large blocks of slots and do not create unusable “gaps” as much as bids with larger nodeslots, hence responsiveness is better because more bids can be allocated more often.

3.6.5 Alternative Workloads

The workloads used have consistently shown that increasing window slot size $S_R$ results in a lower system value $a$. This does not, however, always apply to all possible workloads. I provide in this section a special workload case that potentially can yield higher $a$ with a higher $S_R$. Consider a single node $N_{RR}$ for sale and a very simple workload with the following bids:

1. (“777”,1,1)
2. (“555”,1,4)
3. (“66”,4,4)
4. (“777”,7,7)
5. ("555", 7, 10)

6. ("66", 10, 10).

For visual illustration purposes, the bids are represented in a different format. Here, the first field represents the unit true value and number of slots sought for the single node. For example, "777" means the bid is seeking 3 slots for $7 each (and hence has a total true value of $w_i = 21). The second and third fields refer to arrival and departure times.

Figure 3.15: Workload example that yields higher a with a larger window size. For the left window, bids can start only in the first slot. For the right window, bids can start in the first four slots. ‘-’ indicates a sold slot and ‘x’ an un-allocatable slot.

Figure 3.15 shows how the allocation plays out for a window with slot size $S = 3$ and another with $S_{RR} = 6$. $laststart$ is period 1 and period 4, respectively. For the smaller window, the “777” bids always win over the “555” bids. Then, the “66” bids arriving in between different “777” bids are also allocated. The system value can be calculated by simply adding the numbers in the figure. For fair comparison, I only calculate a with the first ten slots and consider unallocated slots to have a zero value. Thus, for the smaller window, $a = \frac{7 \times 3 + 6 \times 2 + 0 + 7 \times 3 + 6}{10} = 6$.

In the larger window, “555” bids are able to win before the “66” bids arrive. Here, a equals $\frac{7 \times 3 + 5 \times 3 + 7 \times 3 + 5}{10} = 6.2$. Thus, in this example, a increases with a
larger $S_R$. In the smaller window, lower unit value bids “555” lose out to higher unit value bids “66”, despite having a long patience. While this result is consistent with how Roller is designed, “66” bids actually contribute less to $a$ because the smaller slot size of two leaves an extra "gap" slot that cannot be allocated (the "x" slot for time period six). Taking all three slots into consideration, the bid “66” contributes not a unit value of 6 but $6 \times \frac{2}{3} = 4$ to system value $a$.

On the contrary, “555” bids are not affected by “66” bids in the larger window because they arrive earlier. Thus, the unit value of 5 for all three slots contributes to a higher $a$.

Putting together these and earlier observations, the effect of changing rolling window size are dependent on workloads. While a larger window leads to high responsiveness $\beta$ in both cases, system value $a$ can be higher or lower. A useful extension of this work would be to enable adaptive window sizing that can adjust to workload on the fly. The key is to monitor the metrics and change the window size to capture more value, while maintaining good responsiveness.

3.7 Varying Supply
The results in the previous section all assume a fixed supply $N_R$. This assumption may not be realistic, as number of agents and their demand can grow over time, and thus the system may need to expand its supply. In this section, I study the effects of varying $N_R$. Specifically, I will look at effects on metrics $a$ and $\beta$. Can more value be captured and can responses be faster by investing in more nodes? Furthermore, I will introduce and look at revenue, a metric that is relevant for for-profit systems. The key goal is to find out how
many nodes are appropriate, and whether at some point additional nodes would not yield more benefits.

3.7.1 Effects on $a$ and $\beta$

I first analyze the effects of varying $N_R$ on system value and responsiveness. The base workload $L$ is used. For window slot size, I use $S_R = 12$ per Equation 3.13, and vary $N_R$ between 5 and 50 to collect $a$ and $\beta$. Figure 3.16 shows results validating the effects. First, note that by increasing $N_R$, $a$ worsens. The reason is that with more nodes, there are now more resources for the same number of bids, effectively lowering demand/supply ratios. Thus, there will be more winners and fewer losers. $a$ suffers as a result, as it now includes not just the highest value bids but more and more low value bids in calculating per unit value. Thus, investing in additional nodes is worthwhile only if the specific workload enables capturing new winning bids that have high values.

Responsiveness $\beta$, on the other hand, increases quickly. In fact, for $L$, $\beta$ reaches 1.0 as soon as $N_R$ is about 15. Per these metrics, the motivation to increase $N$ would be to improve responsiveness, when demand exceeds supply. Table 3.3 summarizes the general $N_R$ effects.

An example is a type of workload in which half the agents have very high values and the other half very low values. The ideal node size is one that allocates precisely to all the high value bids only.
3.7.2 Revenue Analysis

Next, I will explore another important metric: revenue. Revenue equals the total payments by the winners to the system. For a “social” or “non-profit” system such as Mirage, revenue is not relevant since the system uses virtual currency and agents pay nothing to receive such currency. Nonetheless, in “for-profit” cases systems must charge agents USD in order to keep the services running, as well as make a profit. This is true for systems such
as Amazon Web Services [1], as well as some enterprise systems that need to charge back operating expenses to different companies, departments, and projects.

To address this, I run Roller and study how revenue is affected in different settings. I use the same standard workload $L$ (Equation 3.12) but vary arrival rate $\lambda$ and use only marginally increasing values. Results for $\lambda = 3$ are shown in Figures 3.17. First, note that these are not the same $a/ß$ graphs as before. Instead, at the top, I show $a$ and compare it with “Price Per Nodeslot.” At the bottom, I show “Total Value” and “Total Revenue,” both aggregate measures of winning bids. Total value equals the sum of all winning bid values $w_i$, total revenue equals the sum of total payments made by winning bids, and price per nodeslot is the average per unit nodeslot price as calculated by the payment rule.

![Figure 3.17: Comparing value and revenue with $\lambda = 3$.]
First, observe that total value > total revenue. Total value increases quickly from 5 nodes to 15 nodes. After 15 nodes, total value remains at around 8,500, rendering additional new nodes worthless. The reason is that at 15+ nodes, there is much more supply than demand, thus the same set of bids are allocated and the extra new slots un-allocated. As a result, total values appear unchanged.

Total revenue tells a different story. As nodes increase, there is less competition and thus the payment for each winning bid will decrease. Prices actually go to zero at \( N= 15+ \). Recall the Roller payment rule: a winning bid pays the bid value of the first bid that it replaces. When there is no bid that a winning bid replaces, the price is zero. Thus, with a larger \( N_{RR} \), the price and thus revenue are both zero, since all winning bids are not replacing any bids at all.

Next, I will show results when \( \lambda \) increases. Figure 3.18 (\( \lambda = 5 \)) appears similar overall to 3.17, except it achieves higher total value at higher \( N_{RR} \) size (20). However, there is one new pattern observed. Note that total revenue rises from \( N_{RR} = 5 \) to \( N= 10 \). After that, it drops as \( N_{RR} \) increases. Figure 3.19, with \( \lambda = 10 \), shows an even clearer trend. Total revenue increases steadily and tops off at around \( 20 = N= 25 \), and then gradually drops off until it reaches 0 when \( N_{RR} = 40 \). Thus, revenue is maximized when \( N_{RR} \) is about 20 to 25—further increasing nodes results in negative marginal returns.

The rise in revenue towards the peak occurred because supply was too low (\( N \) was low), making many bids with good values lose. The values of these same bids determine what the winning bids’ payments should be. Thus, the revenue increases due to the sum of these losing bids. However, as \( N_{RR} \) reaches a certain level, these bids do win, making other
3.7.3 Reserve Price

As shown in previous graphs, revenue drops off and approaches 0 when the
supply is high enough to ease competition. Zero revenue is likely unacceptable for most systems. To resolve this issue, I introduce a simple technique to control prices and thus revenue.

The concept is reserve price, denoted by $r_R$. The reserve price is the minimum unit price
Figure 3.19: Comparing value and revenue with $\lambda = 10$.

set for each nodeslot in the rolling window. A bid with unit price less than the minimum price will not be considered for allocation. In addition, a bid that wins will pay at least the reserve price. Essentially, I replaced step 1 of the payment rule (Section 3.3.4) with:

$$p_i = \max(r, v_i), \quad (3.14)$$

where $v_i$ is the price

$$r > v$$

1

R

ee

s

r
$R_i, S_R$ is the previously stated “unit bid price of the first bid that lost because of i’s allocation.” Example: consider a simple unit size rolling window with $(N= s) = (1, 1)$ and bids for unit nodeslots $(n_i = 1)$. Assuming two bids $i$ and $j$ with unit bid price $v_i = 10$ and $v_j = 5$, the following two cases can be considered.

- If $v_i < v_j$, $i$ wins (since $v_i < v_j$) and must pay unit price $p = \max(0, v_i)$.
Thus, i pays $p_i = 5$. Note this has the same outcome as before when there was no reserve price.

$i$ wins and pays unit price $p_i$.

In this case, given the reserve price.

I now re-run the experiment of Figure 3.19 to show that reserve price can help us avoid drop-offs in revenue. I use the same workload distribution, with $\lambda = 1$, and again vary different $N$ sizes. This time, I create a reserve price $r = 5$. I decide on this price by looking at the price per nodeslot for $N$. 

\[ \text{Price Per Nodeslot} \]
serve price effects.

Figure 3.20 shows the result. Note immediately that the total revenue curve now flattens for N= 30+, instead of dropping off. Thus, reserve price does indeed help us maintain
desirable revenue levels. Overall values are higher as well due to the reserve price (compared to Figure 3.19). Similarly, price per nodeslot flattens at $N_R = 30+$ and equals about 5. This confirms the correctness of the reserve price calculation.

3.8 Comparisons with Other Allocators

In this section, I compare Roller with other allocators. Specifically, I separate other allocators into two classes: value-based and non-value-based. Value-based allocators, such as Roller, aim to obtain value information from agents to make allocation decisions, whether such information is truthful or not. There are a wide range of non-value-based allocators, including traditional systems allocators such as First-Come First-Serve (FCFS) and Earliest Deadline First (EDF) algorithms. Instead, these allocators use information other than value, such as job sizes and timing, to make allocation decisions.

While comparing Roller with another value-based allocator is logical, comparing fairly with the non-value-based allocators is hard, because these, by default, will perform poorly with metrics such as system value $a$. However, they probably can do better than Roller regarding responsiveness $\beta$. I compared both for a more in depth understanding. Moreover, I ran additional experiments to find out how many nodes are required for different allocators to achieve the same level of value.

3.8.1 Value-Based Allocator

The goal of a value-based allocator is to maximize aggregate values. In this section, I compare Roller with a straightforward value-based method (referred to here as the “Greedy”
method) to see whether or not Roller outperforms. The Greedy method works as follows:
1. Only nodeslots starting with the current period are available.
   \[ w_i \text{ and patience } \Delta \text{ regardless of nodeslot size } \]
2. At every \( t \), all bids are sorted by bid value \( w_i \).
3. Starting from the top of the bid list, a bid is allocated if there are enough nodeslots available, starting at current time \( t \).
4. Winning agents pay their own reported bid value \( w_i \).

The Greedy method focuses on bid value alone. While it is not an optimal allocator, which can potentially extract even more value by considering nodeslot constraints, the Greedy method is a reasonable benchmark as it likely extracts a competitive amount of value compared to an optimal allocator and far more value than a non-value allocator. While the above resembles the Roller mechanism, there are several key differences. First, total value rather than unit value is used. Second, it uses first-price (\( w_i \)) and hence the method is not strategyproof. Third, the rolling window concept is not used and thus no selling of future resources is allowed.

Workload \( L \) is again used with \( \lambda = [1, 2, 3, 4, 5, 10] \). The number of nodes is 8.

For Roller, window size \( S \) is 12, per the results of Section 3.6. For each \( \lambda \), I ran Greedy and Roller to collect metrics \( a \) and \( \beta \). Figures 3.21 and 3.22 show results for marginally increasing and decreasing values, respectively.

System Value a Comparison. Roller and Greedy achieve virtually identical \( a \) with marginally increasing values. This is because both methods prioritize bids with both high
total value and high unit value for allocations, due to marginally increasing values. For marginally decreasing values, Roller outperforms Greedy because the high unit value bids win in the former, but lose in the latter. These high unit values derive high $\alpha$ for Roller, but not for Greedy.

Responsiveness $\beta$ Comparison. Roller beats Greedy by about 0.2 throughout for either marginally increasing or decreasing values. This is mainly due to the use of the rolling window, whereas bids for Greedy simply have to wait multiple periods for decisions to be made.

![Figure 3.21: Roller vs. Greedy. Marginally increasing value distribution.](image)

In summary, Roller captures good total value while enabling a responsive experience. For distributed systems, this good response time is an important consideration for those choosing between a fast and strategyproof allocator like Roller and an allocator with per-
Figure 3.22: Roller vs. Greedy. Marginally decreasing value distribution.

haps higher value but low responsiveness.

3.8.2 Non-Value-Based Allocators

The next comparison is with traditional, non-value-based allocators. I compare Roller to two popular and simple allocators: First Come, First Serve (FCFS) and Shortest Job First (SJF). For FCFS, bids that have the earliest arrival time will be ranked first in every time period. For SJF, bids will be ranked by nodeslot size $n_i$, in which bids that request the smaller nodeslots have higher priority. In addition, FCFS and SJF will only consider bids to start at $t$ and do not use an allocation window for future slots.

The workloads used are based on $L$ (Equation 3.12) with $\lambda = (1, 2, 3, 4, 5, 10)$. The number of nodes $N$ is fixed at 8. For the rolling window, I use a fixed window slot size $b$.
Roller Greedy
Chapter 3: Online Mechanism Design

\[ S = \Delta + s = 12 \] for all tests. Results for marginally increasing and decreasing values are shown in Figure 3.23 and Figure 3.24, respectively.

![Figure 3.23: Roller vs. FCFS vs. SJF. Marginally increasing value distribution.](image)

System Value a Comparison. First, I will discuss \( a \) for marginally increasing values. For Roller, system value \( a \) increases gradually as arrival rate \( \lambda \) increases. This is consistent with our results per Table 3.2. For FCFS and SJF, \( a \) increases for \( 1 = \lambda = 3 \) but decreases slightly as demand increases per rising \( \lambda \). System value \( a \) for Roller is almost twice as good as that of SJF for \( \lambda = 3 \), while FCFS results are in the middle.

The results are not surprising because FCFS and SJF do not consider values at all. For FCFS, its \( a \) is close to the average of all bid values, since it only considers arrival time. For SJF, it is worse because it selects the smallest jobs which are also the lowest value jobs for marginally increasing value distributions.
Figure 3.24: Roller vs. FCFS vs. SJF. Marginally decreasing value distribution.

For marginally decreasing values, a for Roller and FCFS are similar as before, but with lower values in general. SJF, on the other hand, has virtually identical a as Roller. The reason is that the highest value jobs now are the smallest ones, which SJF prioritizes exclusively.

Responsiveness ß Comparison. Figures 3.23 and 3.24 both show that Roller is very responsive. SJF is second, but competitive. By selecting the smallest jobs, SJF allows more slots to be available to various requests and thus increases responsiveness. However, FCFS performs poorly as soon as $\lambda > 1$. This can be explained as follows. Because of the lack of a rolling window, time-to-win in general suffers. With FCFS, every time bids with longer slots ($s_i$) are allocated, the slots are “blocked” for several time periods, further delaying time-to-win for other bids. This effect is similar to the effect of not using a laststart bar, as
discussed in Section 3.4. Nodes and Values Comparison. Finally, I want to see how many nodes \( N_R \) are required for these three systems to achieve a certain level of values. I use \( \lambda = (5, 10) \) for workload L. This time, I run the allocators against different numbers of nodes: \( N_R = (1, 2, 3, 4, 5, 10) \). Graphs of total value as well as \( a \) and \( \beta \) are plotted in Figure 3.25 for \( \lambda = 5 \) and Figure 3.26 for \( \lambda = 10 \).

Total values for all allocators increase gradually as supply \( N_R \) increases. They all, at some point, reach a ceiling (e.g., 30,000 for \( \lambda = 10 \)) because every bid is accepted given the large amount of nodes supplied. Adding more nodes does not help the system generate more value and thus will not make good use of resources.

To achieve a certain level of value, Roller uses fewer nodes than FCFS or SJF. For example, if total value of 20,000 is desired for a \( \lambda = 10 \) workload, then 20 nodes are needed for Roller, while approximately 27 and 32 nodes are needed for FCFS and SJF, respectively.

The \( a \) and \( \beta \) graphs again confirm the general behaviors of Roller. \( a \) for FCFS and SJF is more interesting: it rises and then drops off. The rise is due to increased winners in general, thus boosting \( a \). However, it drops off precisely at the ceiling level. Because there are excessive nodes, \( a \) will decrease because it takes the total number of nodes into account.

To summarize:

- Roller is more responsive, generates higher value than FCFS and higher or similar value as SJF.
- Roller uses fewer nodes to achieve the same level of value as FCFS and SJF.
3.9 Late Allocation

In this section, I describe experiments to test the \( \rho \) -allocation rule. Again, the purpose of \( \rho \) is to probabilistically allocate late to limit agents that submit over-reported departure time \( d \), in order to receive lower payments through getting extra VirtualWorlds periods. When a bid is created from the workload, it has a \( \rho \) probability ( \( \rho \in [0, 100]\% \)) of being assigned as a “late” bid by Roller. When an agent mis-reports his true patience, I call that over-reporting, I denote the amount of over-reporting in terms of the multiple between over-report departure ( \( d_{\text{over}} \)) and actual departure ( \( d \))—denoted by 1.5x, 2x, and so forth. For example, if \( d = 10 \) and \( d_{\text{over}} = 5 \), then it is a “2x over-report.” For the experiments, I use the average payoff metric to quantify the outcomes to agents that over-report. A payoff to an agent \( i \) equals its captured total true value minus its total

<table>
<thead>
<tr>
<th>Value</th>
<th>Roller</th>
<th>FCFS</th>
<th>SJF</th>
</tr>
</thead>
<tbody>
<tr>
<td>10000</td>
<td>12000</td>
<td>14000</td>
<td>16000</td>
</tr>
<tr>
<td>2000</td>
<td>4000</td>
<td>6000</td>
<td>8000</td>
</tr>
</tbody>
</table>

Figure 3.25: Roller vs. FCFS vs. SJF: number of nodes and total values. \( \lambda = 5 \).

Marginally increasing values.
payment. For example, if true value is $10 and total payment is $4, then payoff equals $6 (positive) for a truthful agent but -$4 (negative) for a late allocation. An average payoff is computed from both kinds of bids. Note that identifying over-report bids is hard or even impossible by a system—but is feasible in an experimental setting as it decides which agents over-report and by how much.

Late allocations have no effect on agents reporting true departures. On the other hand, an agent with a "late" winning bid with over-reported departure captures none of its true value \( w \), as such value is valid only for receiving the resource on or before its true departure \( d \). Moreover, it is obligated to making payments for winning the resources, essentially resulting in the agent paying for something that it no longer desires. Its payoff, as a result, is negative payment.