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

Dissertation advisors

David C. Parkes and Margo I. Seltzer

Author

Chaki Ng

Online Mechanism and Virtual Currency Design for Distributed Systems

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

List of Figures

List of Tables

Citations to Previously Published Work

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

> *Addressing Strategic Behavior in a Deployed Microeconomic Resource Allocator*, Chaki Ng, Philip Buonadonna, Brent N. Chun, Alex C. Snoeren, and Amin Vahdat, 3rd Workshop on the Economics of Peer-to-Peer Systems, August 2005. `http://doi.acm.org/10.1145/1080192.1080195`

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

> *Virtual Worlds: Fast and Strategyproof Auctions for Dynamic Resource Allocation*, Chaki Ng, David C. Parkes, and Margo Seltzer. ACM Conference on Electronic Commerce, June 2003, page 238–239.

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 [1] agents [2]. 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.[3] For example, an agent may be willing to pay a

[3]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 [4].

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

[4] 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-

available.

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 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.

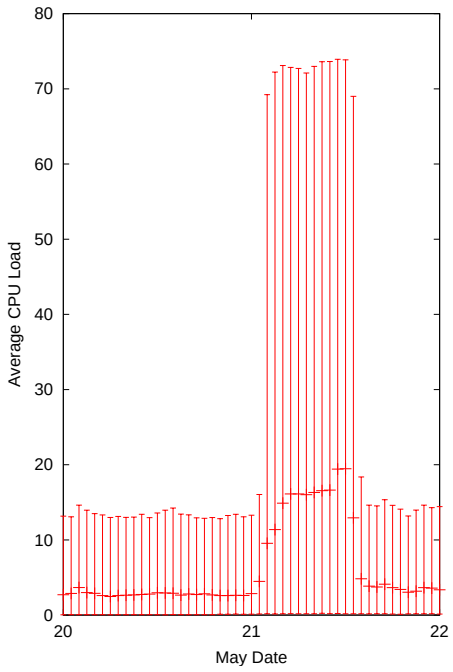

Figure 1.1: **PlanetLab load example**. 5th, average, and 95th percentile load average on 220 nodes leading up to the OSDI 2004 submission deadline [31].

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,

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* [1], an auction-based market for allocating the resources of a wireless sensor network testbed. The testbed is composed of 148 individual wireless sensor nodes and used by researchers (agents) across the USA. Almost all agents require multiple nodes to conduct their sensor network experiments. Some of their experiments have specific requirements such as

[1] Mirage was a joint project with collaborators from Harvard University, Intel Research Lab, and UCSD [30].

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 non-combinatorial 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.

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."

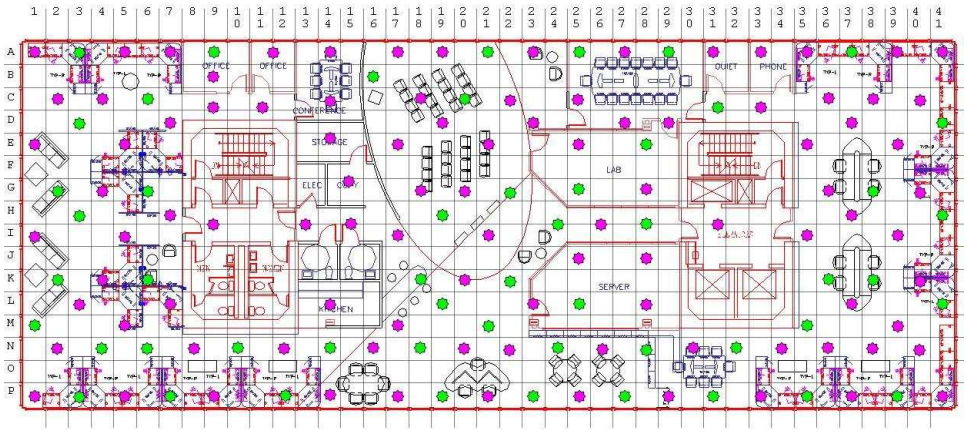

Figure 2.1: **Mirage map**. 148 sensors deployed throughout the Intel Berkeley Research Lab.

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 [2] 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 [3]. 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:

$$b_i = (v_i, s_i, t_i, d_i, f_{min}, f_{max}, n_i, ok_i). \tag{2.1}$$

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 $[f_{min}, f_{max}]$. s_i 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 [4]. 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

[4] 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:

$$b_i = (99, 6, 24, 4, 423, 443, 64, \text{[a list of 100 nodes]}). \tag{2.2}$$

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

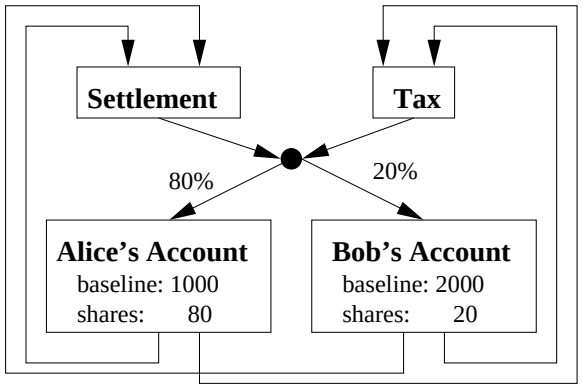

Figure 2.2: **Virtual currency policy.**

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

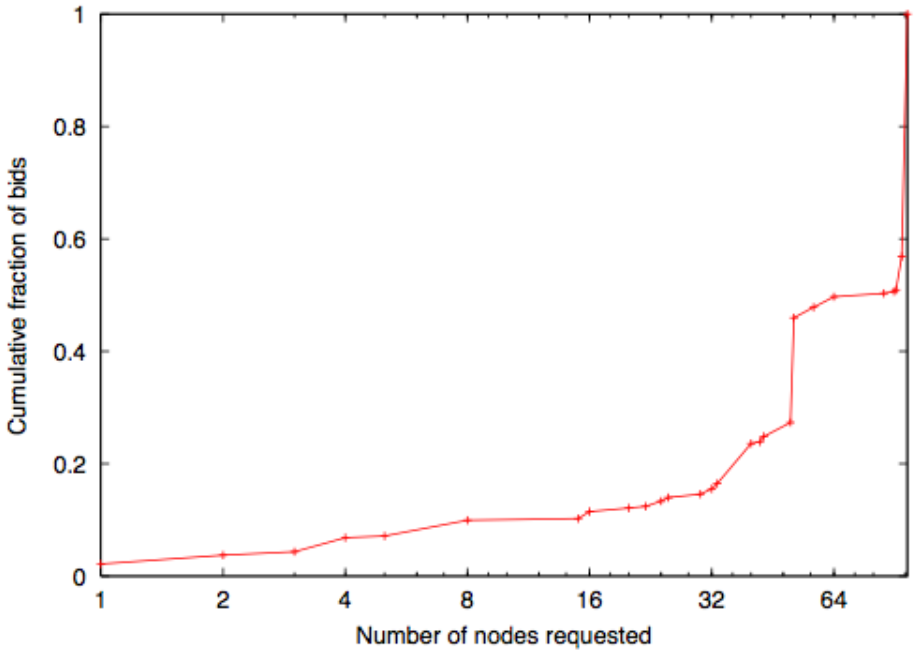

Figure 2.3: **Cumulative distribution of nodes requested by agents.**

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_i) 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).

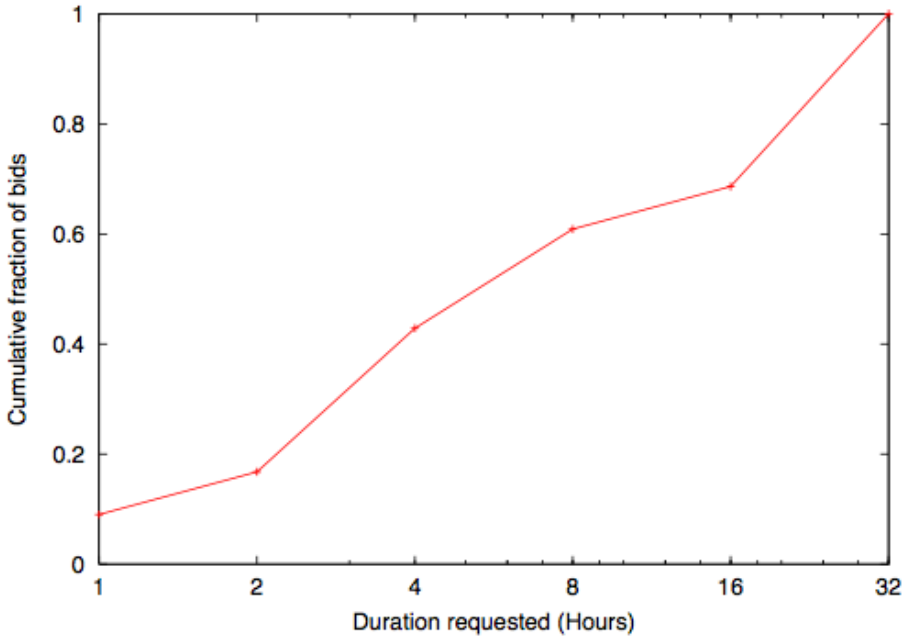

Figure 2.4: **Cumulative distribution of duration requested by agents.**

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

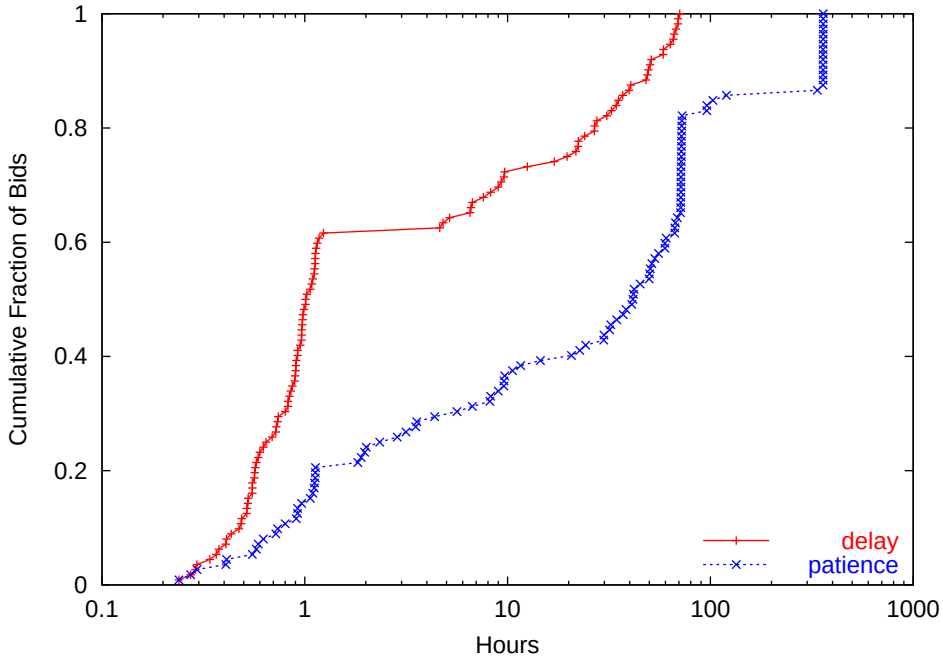

Figure 2.5: **Cumulative distribution of delay and patience.**

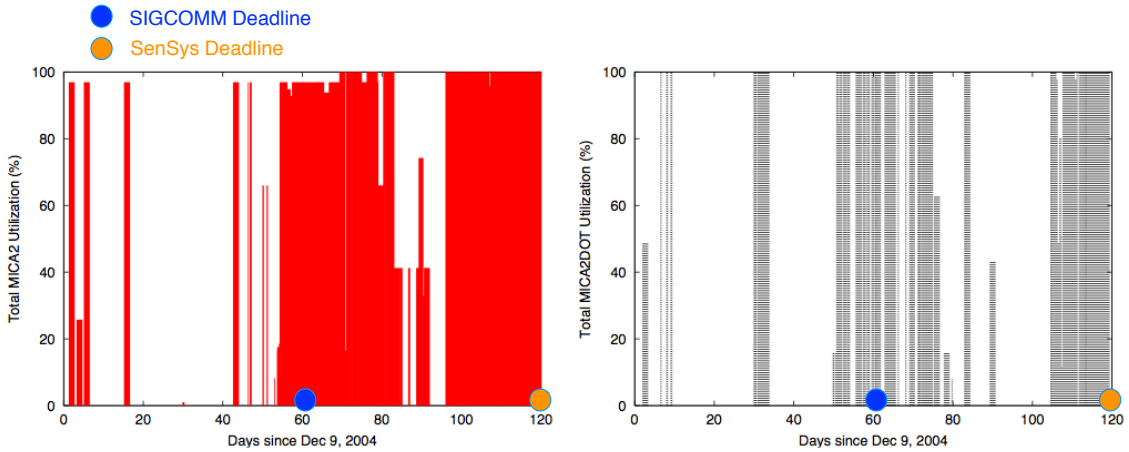

Figure 2.6: **Testbed utilization:** daily usage for the 97 MICA2 and 51 MICA2DOT nodes.

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

requests could be allocated during such times.

2.3.3 Diverse Agent Values

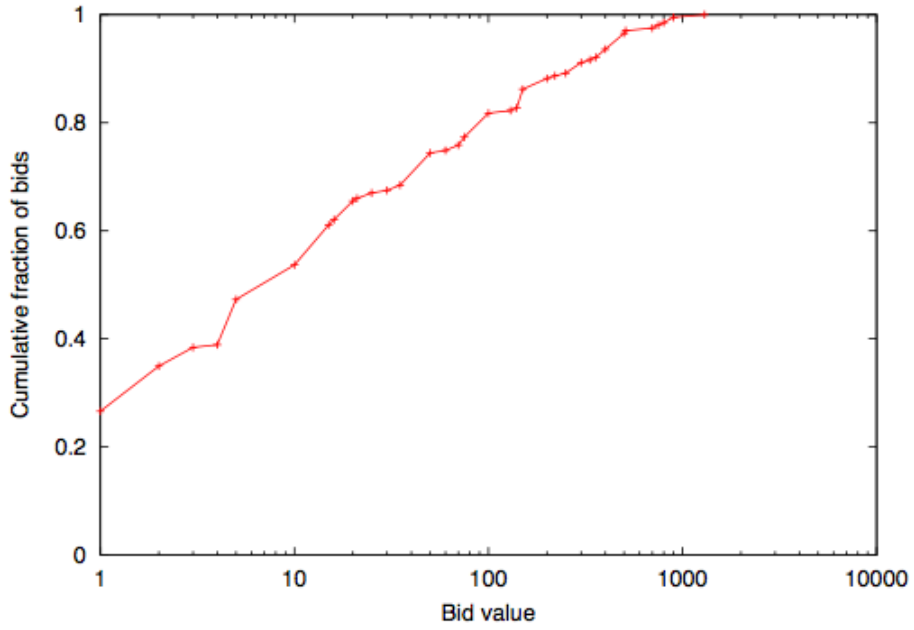

Figure 2.7: **Bid value distributions:** submitted total values by all agents.

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

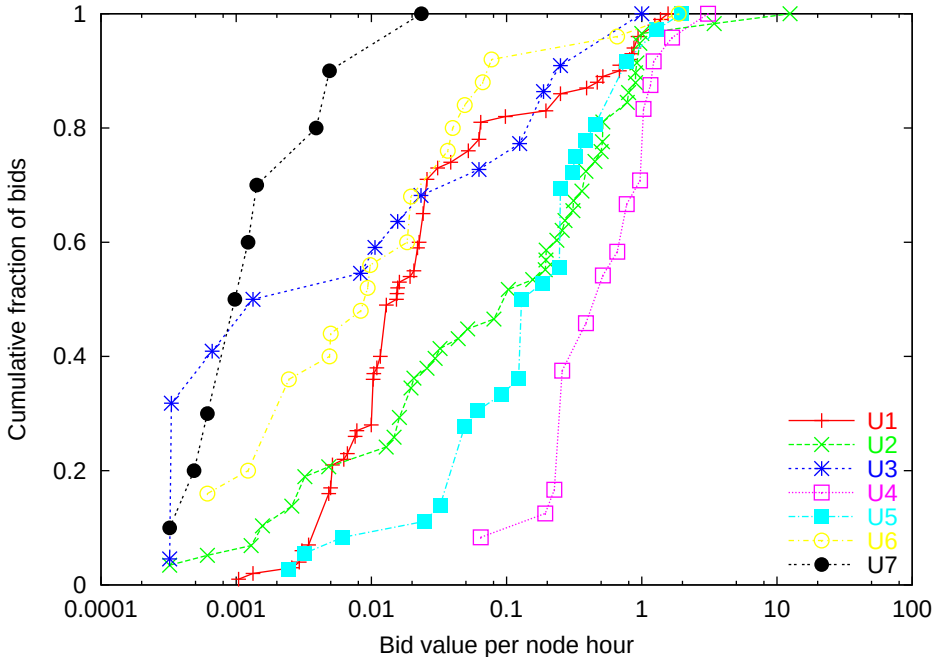

Figure 2.8: **Bid value distributions:** submitted values per node hour for the seven most active agents.

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

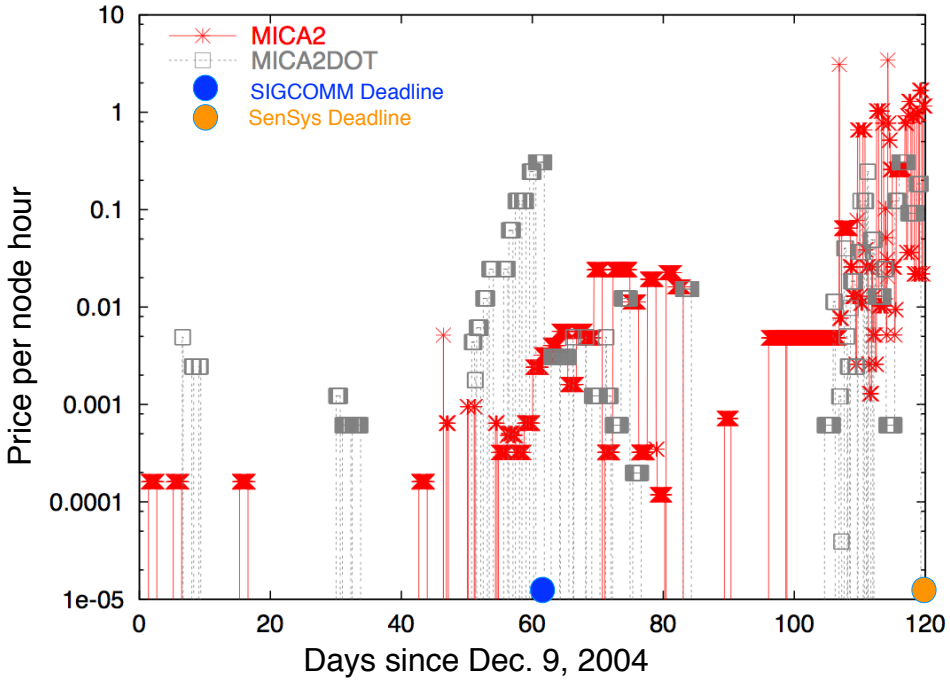

Figure 2.9: **Median market prices (per nodeslot).**

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.

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 $\times$ 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 $\times$ 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

stayed online long enough to make last minute modifications to narrowly "beat out" the other bids [5].

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 $A2$ 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 $\times$ 72-hour window is allocated at time t. For each upcoming auction at $t+1, t+2, ...$, a total of 148-node $\times$ 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

[5] 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 $A2$ 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 $A1$, 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.

Time	**Bid Number**	**Agent**	**Value**	**Nodes**	**Hours**
04-02-2005 03:58:04	#1	U2	1590	97	32
04-02-2005 05:05:45	#2	U1	5	24	4
04-02-2005 05:28:23	#3	U1	130	40	4
04-02-2005 06:12:12	#4	U1	1	33	4

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 consider 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 [1]. Thus, the

[1] 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 ρ-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 "for-profit" 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_R identical *nodes* available over time periods

$[0, T]^2$. 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 slots. Each slot is of an arbitrary size (e.g., an hour) chosen by the system administrator. L_R ($\leq S_R$) is the *laststart* 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 (call this size *maxdur*) can start at L_R and *end* on S_R. I will explain more about the rationale in detail in Section 3.4.

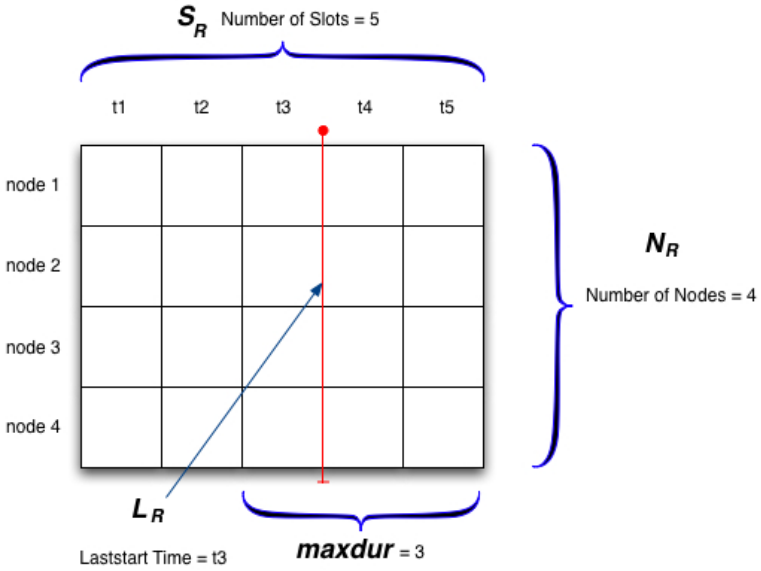

Figure 3.1: **Rolling window abstraction:** with N_R number of nodes and S_R number of slots, at time t_1. All allocations must start on or before L_R, or t_3 in this example. *maxdur* is 3, thus all bids with slots ≤ 3 can start at L_R.

At any time t_n, the next S_R slots (specifically, [t_n, t_{n+1}, ..., $t_n + S_R - 1$]) are available.

[2]I use T so that the experiments can have a finite scope. In real-world systems, T would be ∞ because these system 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 $t_n + S_R$ are added at the far right end. The following is an example.

Example: Consider $R = (4, 5, 3)$. At t_1, the slots are $[t_1, t_2, t_3, t_4, t_5]$ (Figure 3.1). Column t_3 is the laststart time, where all bids will be considered for slots *starting* on or before t_3, but not after. For instance, a bid for 1×2 nodeslots can be considered for (t_1, t_2), (t_2, t_3), and (t_3, t_4), but not (t_4, t_5). As time passes, the window "rolls" to the right. At t_2, column t_1 phases out and column t_6 rolls in, making $[t_2, t_3, t_4, t_5, t_6]$ now available. L_R also shifts and is now column t_4. See Figure 3.2 for an illustration.

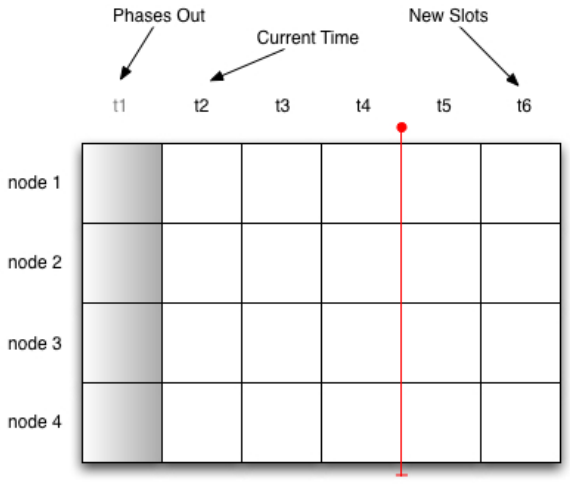

Figure 3.2: **Advancing rolling window:** At t_2. All columns "shift" to the right. Column t_1 (in gray color) phases out while column t_6 rolls in. L_R also rolls right and is now t_4.

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:

$$b_i = (w_i, n_i, s_i, a_i, d_i), \tag{3.2}$$

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 ns_i nodeslots; $0 < n_i \leq N_R$ and $0 < s_i \leq$ *maxdur*). The nodeslots must be available for use starting no earlier than arrival time a_i and no later than departure time d_i.

Example: Assume a testbed with $N = 1$ node that runs over $T = 10$ periods. A bid $b_i = (\$100, 1, 2, 3, 5)$ can be interpreted as follows. Agent i is willing to pay at most 100 USD for 1×2 nodeslots that start between t_3 and t_5. If i wins in t_3, the actual nodeslots assigned will be t_3 and t_4 (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

comprised of N_R nodes over the next S_R available slots. All bids that are submitted before the auction begins and that have allocation timing that coincides with the rolling window slots will be included in the auction.

A greedy algorithm is used to determine winners (based on the scheme in Lehmann et. al. [59]). First, all bids are sorted by w_i/ns_i. 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_1, agent i submits $b_i = (\$100, 1, 1, 1, 3)$ to a system with window $R(1, 3, 3)$. Assume at t_1 the window slots for t_1 and t_2 are already allocated to another agent (e.g., that agent won before t_1). Thus, the bid b_i is allocated to a future slot t_3, but the agent is notified now in t_1.

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_i and departure d_i). The motivation is to mitigate time-based strategies related to the allocation timing.

1. At the end of each auction at time t_n, each winner i's price equals the unit bid price v_j 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.

4. If i wins in the *VirtualWorld*, calculate its price p_v (as in Step 1). If $p_i \geq p_v$ then set $p_i = p_v$. In words, whenever a lower price is found in a *VirtualWorld* auction, i's obligated price is updated.

5. By the end of the agent's departure time d_i, p_i will be the lowest unit price i would have paid across its stated allocation timing. The final payment for i is thus $p_i n_i s_i$ (unit price times number of nodeslots).

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.

3.3.5 Late Allocation

One last strategic issue remains. Since bidder i will be exposed to all auctions in its specified allocation timing periods in $[a_i, d_i]$, 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 ρ-late allocation rule. With probability ρ (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 ρ-late allocation rule is worthwhile.

In considering the use of a ρ-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 ρ-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-know 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

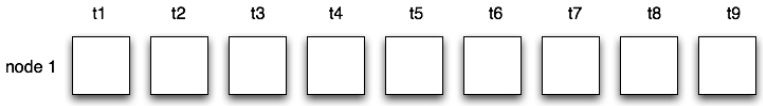

Figure 3.3: **Single slot windows.**

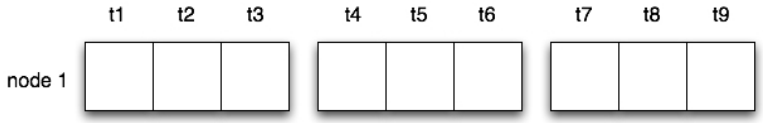

Figure 3.4: **Multiple slot windows.**

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 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 [3]. 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. Compare

[3]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.

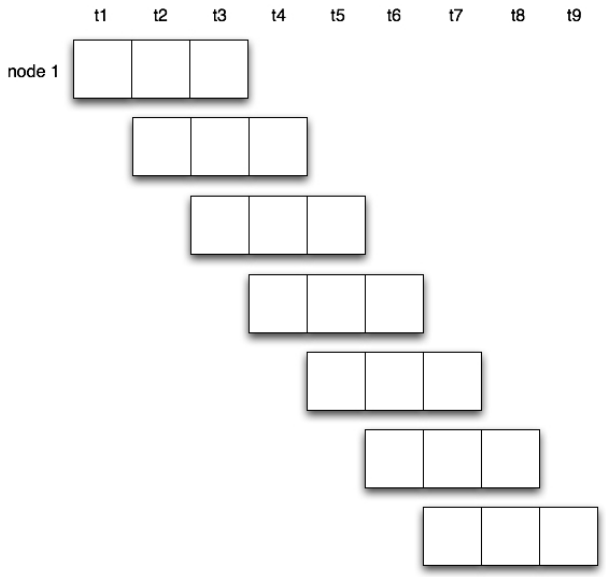

Figure 3.5: **Rolling window.**

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.

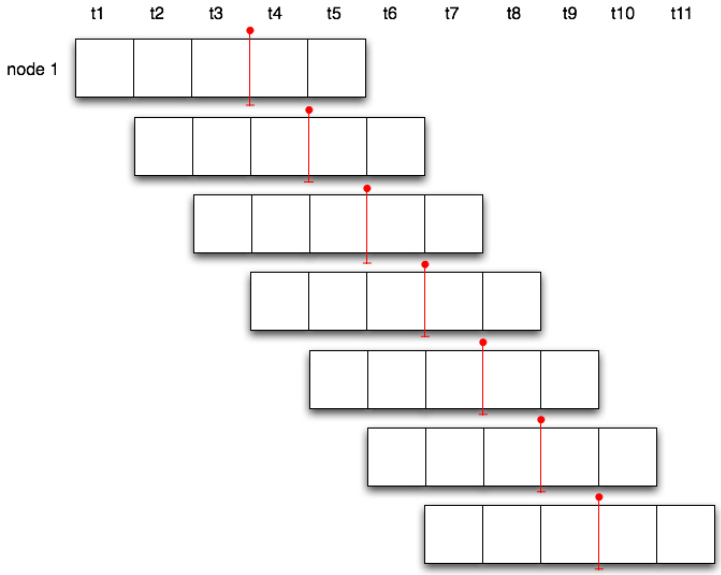

Figure 3.6: **Rolling window with laststart time.**

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 $n_i = 1$ 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 s_i always 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.

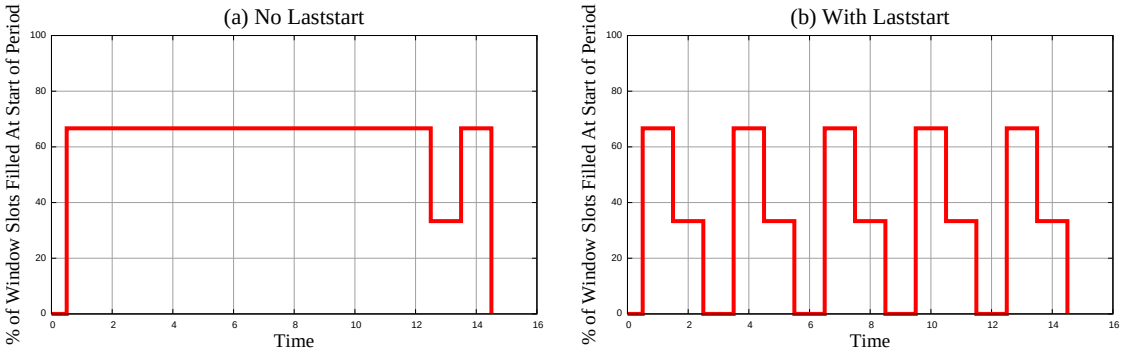

Figure 3.7: **Laststart**: How much the windows are filled at beginning of each time period.

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,

$$L = (T, \mathcal{P}, \lambda, [n_{low} : n_{high}], [s_{low} : s_{high}], [\Delta_{low} : \Delta_{high}], w(m, [x_{low} : x_{high}])), \quad (3.3)$$

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."

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

- $[n_{low} : n_{high}]$ and $[s_{low} : s_{high}]$ specify the ranges of nodes and slots. For example, $[n_{low} : n_{high}]$=[1:3], $[s_{low} : s_{high}]$=[2:4] means nodes n_i are drawn uniformly from [1,2,3] and slots s_i from [2,3,4]. Thus the space of possible nodeslot pairs are [(1,2), (1,3), (1,4), ..., (3,2), (3,3), (3,4)].

- $[\Delta_{low} : \Delta_{high}]$ defines 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.

- $w(m, [x_{low} : x_{high}])$ is a function to generate a range for bid values w_i. Note that true value equals bid value in this case, as Roller is strategyproof with respect to value. m is the method to generate values while $[x_{low} : x_{high}]$ is the base range of unit bid value. The goal is to produce monotonically increasing values, which I will explain in the next section.

Parameters T, $\mathcal{P}$, and λ 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 $\mathcal{P}$ and λ. For each bid, numbers are then drawn from $[n_{low} : n_{high}]$ $[s_{low} : s_{high}]$ to generate nodes n_i and slots s_i, and $[\Delta_{low} : \Delta_{high}]$ to generate Δ_i and in turn a_i and d_i (see equation 3.2 on page 47, in which arrival time a_i is simply the current time period t and departure time d_i is $a_i + \Delta_i$). Finally, $w(m, [x_{low} : x_{high}])$ is called to generate the bid value w_i 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 $ns_k > ns_i \Rightarrow \bar{w_k} > \bar{w_j}$, 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 ns_i increases. In other words, how does the rate of change of value w_i change as nodeslot ns_i changes? Figure 3.8 illustrates the difference.

The $w(m, [x_{low} : x_{high}])$ function generates a range $[w_{low} : w_{high}]$ on value w_i. It takes two parameters: (1) m, which specifies one of two possible methods used to generate w_i, that is, marginally increasing ("$\Uparrow$") or marginally decreasing ("$\Downarrow$"); and (2) $[x_{low} : x_{high}]$,

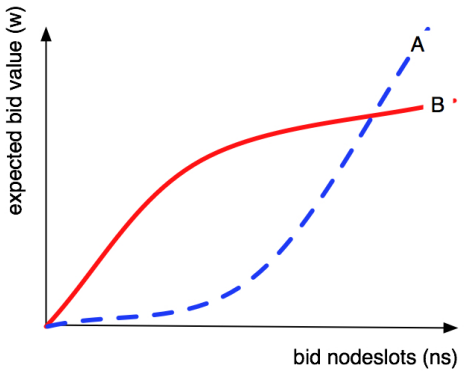

Figure 3.8: **Two monotonically increasing total value curves**. Curve *A* exhibits marginally increasing values as the nodeslot increases, and curve *B* exhibits marginally decreasing values.

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

$$[w_{low} : w_{high}] = ns_i \cdot [y : y + z]. \tag{3.4}$$

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

$$y = \begin{cases} \frac{ns_i - ns_{min}}{ns_{max} - ns_{min}} \cdot (x_{high} - x_{low}) + x_{low} & \text{if} \quad m = \Uparrow, \\ \frac{ns_{max} - ns_i}{ns_{max} - ns_{min}} \cdot (x_{high} - x_{low}) + x_{low} & \text{if} \quad m = \Downarrow . \end{cases} \tag{3.5}$$

Finally, the value of z is determined as:

$$z = 0.3 \cdot (x_{high} - x_{low}). \tag{3.6}$$

The role of y and z is to map and transform the given unit value distribution $[x_{low} : x_{high}]$ into w_i. 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.

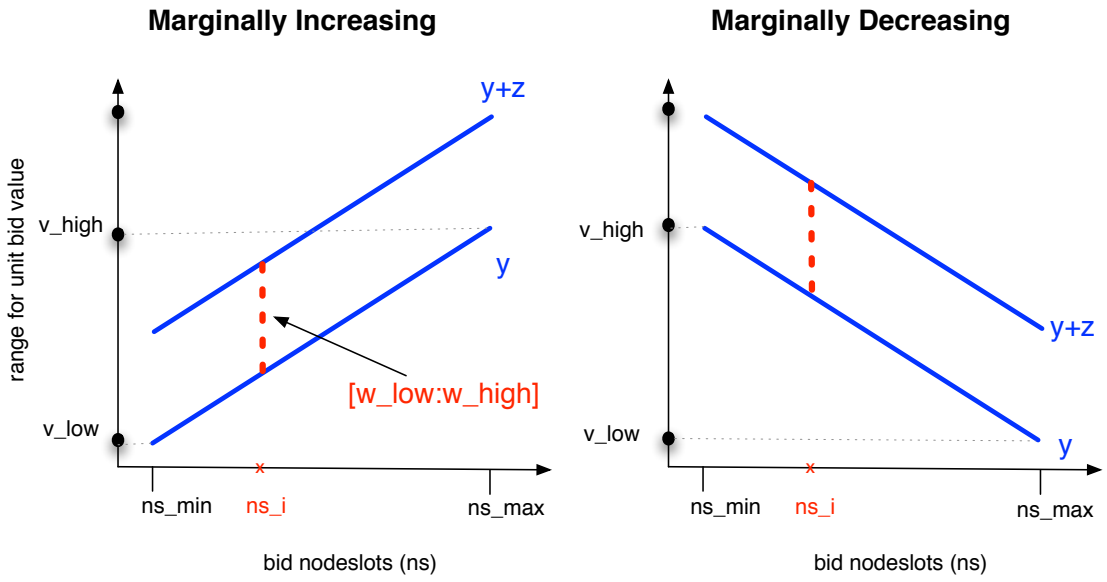

Figure 3.9: **Unit value distributions [$y : y + z$] for different nodeslots.**

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, which provides a useful reference for the experiments. The *Demand* is the total number of nodeslots requested over $[0, T]$ for workload L. Demand is entirely driven by workloads and has no dependency on any components of Roller. Demand is estimated as

$$\text{demand} = \lambda \cdot \bar{n}\bar{s} \cdot T, \tag{3.7}$$

where λ is the number of new bids arriving in each time period, $\bar{n}\bar{s}$ is the average

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: [4]

$$\text{supply} = N_R \cdot T. \tag{3.8}$$

Lastly, *Demand/Supply*, or 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 D/S indicates how heavily loaded the system is. When $D/S = 1\times$, the number of nodeslots demanded equal the number of slots supplied. When $D/S > 1\times$, there are more jobs than can possibly be scheduled. When $D/S < 1\times$, some resources will go idle. Formally it is simplified as:

$$\text{demand} = \lambda \cdot \bar{n}\bar{s}/N_R. \tag{3.9}$$

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

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 α and responsiveness β. In theory, the two metrics have reciprocal effects because getting a high value requires batching of bids and more computation,

[4]The *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 $T+1, T+2, ..., T+S_R-1$. I choose to exclude these as T is quite large compared to S_R.

hence responsiveness to agents suffers. Thus, I aim to evaluate them together to find a balance in different situations.

System Value (α)

System Value α is defined as the *average true value captured per unit supply of the system:*

$$\alpha = \frac{\sum w_{win}}{\text{supply}} = \frac{\sum w_{win}}{N_R \cdot T}, \tag{3.10}$$

where $\sum w_{win}$ is the sum of the total true value of all winning bids, and supply is based on equation 3.8. The unit of α is USD, the same as true values. System value α allows the comparison of systems of different nodeslot sizes. The metric is not affected by varying demand/supply ratios. [5]

Responsiveness (β)

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

$$\beta = \frac{\sum \frac{1}{ttw_i}}{|K|} \tag{3.11}$$

[5] An alternative metric to consider is to divide $\sum w_{win}$ by the *total true value of all bids received*. When demand/supply is high (e.g., $10\times$), 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., $1\times$).

Bids' Time-to-Win	β
1xx	1/1 = 1
11x	(1+1)/2 = 1
111	(1+1+1)/3 = 1
12x	(1+1/2)/2 = 0.75
13x	(1+1/3)/2 = 0.67
22x	(1/2+1/2)/2 = 0.5
1xxxx	1/1 = 1
2xxxx	(1/2)/1 = 0.5
11xxx	(1+1)/2 = 1
12xxx	(1+1/2)/2 = 0.75
13xxx	(1+1/3)/2 = 0.67
22xxx	(1/2+1/2)/2 = 0.5
111xx	(1+1+1)/3 = 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 ttw_i is the *time-to-win* (the number of time periods it take for i to win; or the number of time periods it takes Roller to make such a decision), $1 \leq ttw_i \leq \Delta_i$. Thus, β is a number in the range $(0, 1]$. A small ttw_i is good; $ttw_i = 1$ means bid i wins in the first possible time period. $\frac{1}{ttw_i}$ 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 $\frac{1}{ttw_i}$ divided by the number of winning bids $|K|$ gives us β, which measures the average responsiveness achieved by the system.

Table 3.1 shows a list of bid scenarios with different β values. Each row specifies an example bid scenario and the corresponding β. 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 β 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_R and study the effects of different rolling window sizes S_R. 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_R gives us the most balanced α and β performance?" I

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:

$$\begin{aligned} L' &= (T, \mathcal{P}, \lambda, [n_{low} : n_{high}], [s_{low} : s_{high}], [\Delta_{low} : \Delta_{high}], w(m, [x_{low} : x_{high}])) \\ &= (500, \text{Poisson}, 3, [1:3], [1:3], [10:10], w(\Uparrow / \Downarrow, [1:10])). \end{aligned} \tag{3.12}$$

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

3.6.2 Arrival Rate λ

The first experiments involve testing different workloads against rolling window size S_R by varying only λ. λ is a significant parameter because it directly affects demand/supply. Starting with base workload L', I vary λ 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_R, from 3 to 20. The smallest S_R size has to be 3 because *maxdur* $= s_{high} = 3$. I collect values of α and β

for each workload and S_R combination. Note the corresponding demand/supply for $\lambda \cong 0.5\times, 1\times, ..., 5\times$.

Figure 3.10 shows the results for marginally increasing values. Each line represents a workload with a specific λ and the points on each line are for a specific S_R size. First, for a given S_R (e.g., $S_R = 6$), observe that increasing λ from 1 to 10 leads to higher α and lower β.

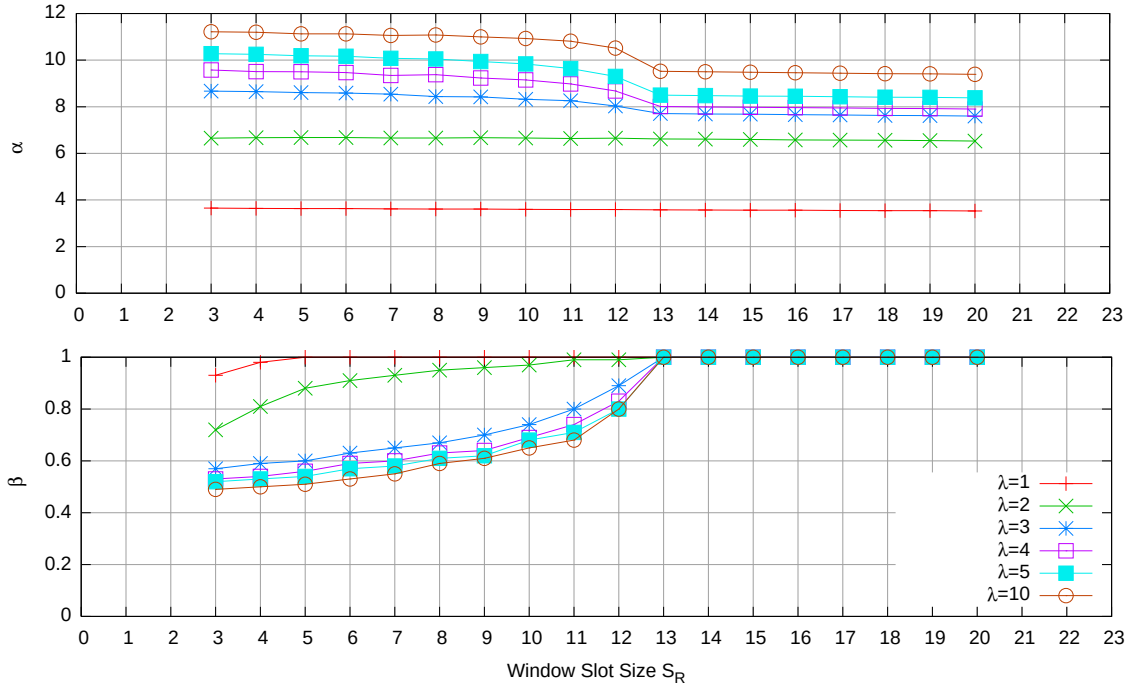

Figure 3.10: **Effects of S_R on different λ ranges**: with marginally increasing ($\Uparrow$) values.

For a given λ (e.g., any one of the lines such as $\lambda = 4$), observe the following:

- For $\lambda \geq 3$, α *decreases gradually as* S_R *increases*. For example, with $\lambda = 4$, α decreases from around 9.7 when $S_R = 3$ to around 9 when $S_R = 10$. After that, α drops off more rapidly when $10 \leq S_R \leq 13$. For $S_R = 13+$, α exhibits minimal decrease and remains virtually constant. On the other hand, for $\lambda = 1, 2$, α remains

virtually constant for all S_R sizes.

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

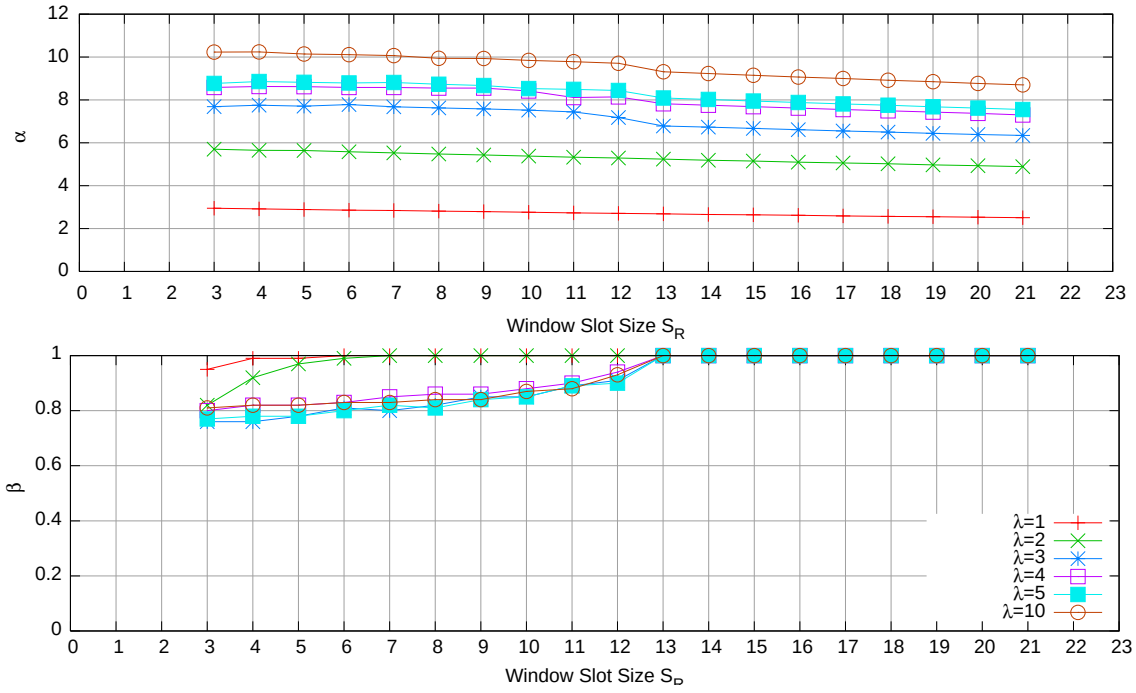

Figure 3.11: **Effects of S_R on different λ ranges**: with marginally negative ($\Downarrow$) values.

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 ($\Downarrow$) values, the patterns are similar (Figure 3.11). α is lower overall for all points, whereas β is higher overall for all points (every β 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[6]. Window size S_R again will vary from 3 to 20.

First, I use the following ranges for nodes requested by agents:

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

Figure 3.12 shows the results. For any given λ curve, the patterns largely resemble those for arrival rate λ, that is, decreasing α and increasing β as S_R increases. While λ in this case remains constant, higher $[n_{low} : n_{high}]$ ranges represent an increase in demand as well. Thus, higher node ranges achieve higher system value α, given S_R.

Next, I use the following ranges for slots requested by agents:

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

I vary bidder S_R from s_{high} to 20 (instead of the usual 3 to 20). For example, for $[1:5]$, S_R will be tested from $5:20$. The reason is that by using a smaller S_R, the larger bids will be excluded. The results are shown in Figure 3.13. Overall, the patterns are similar to both λ 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

[6]Similar to results in the previous section, results for marginally decreasing values are similar for this section as well and are not shown.

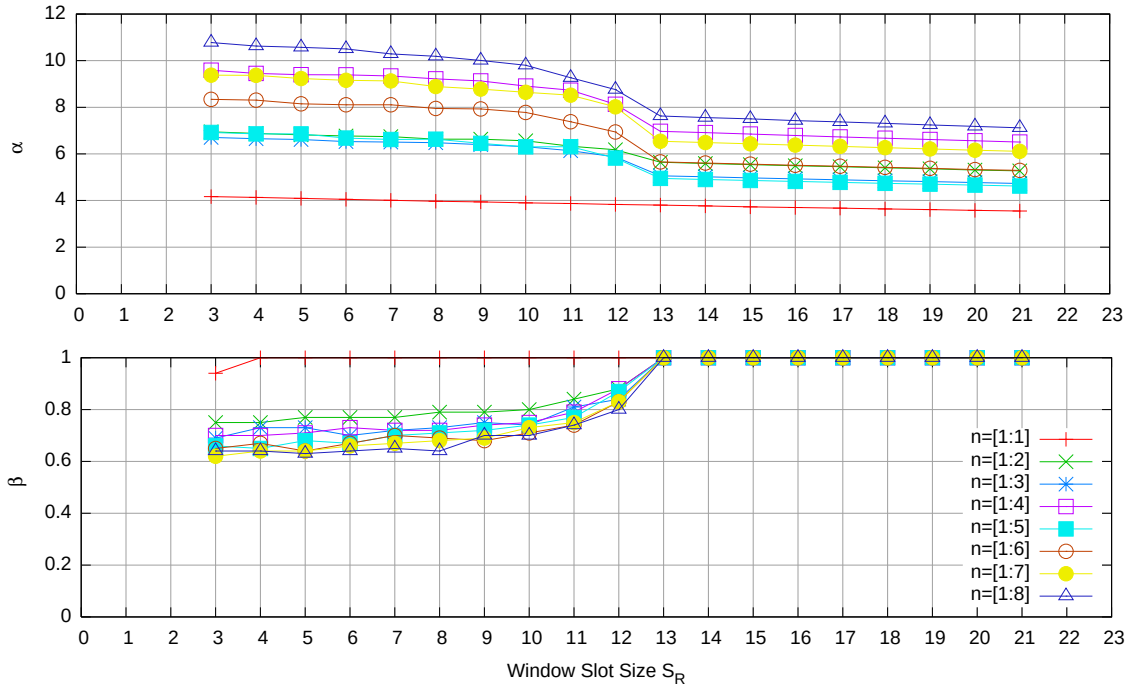

Figure 3.12: **Effects of S_R on different n ranges.**

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:

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

Overall, patience Δ has minimal effect on value α. Higher patience does not lead to higher α. The reason for this is that all the Δ ranges still share the same level of demand/supply, unlike nodes or slots. β follows similar but tighter patterns as those in the previous λ section.

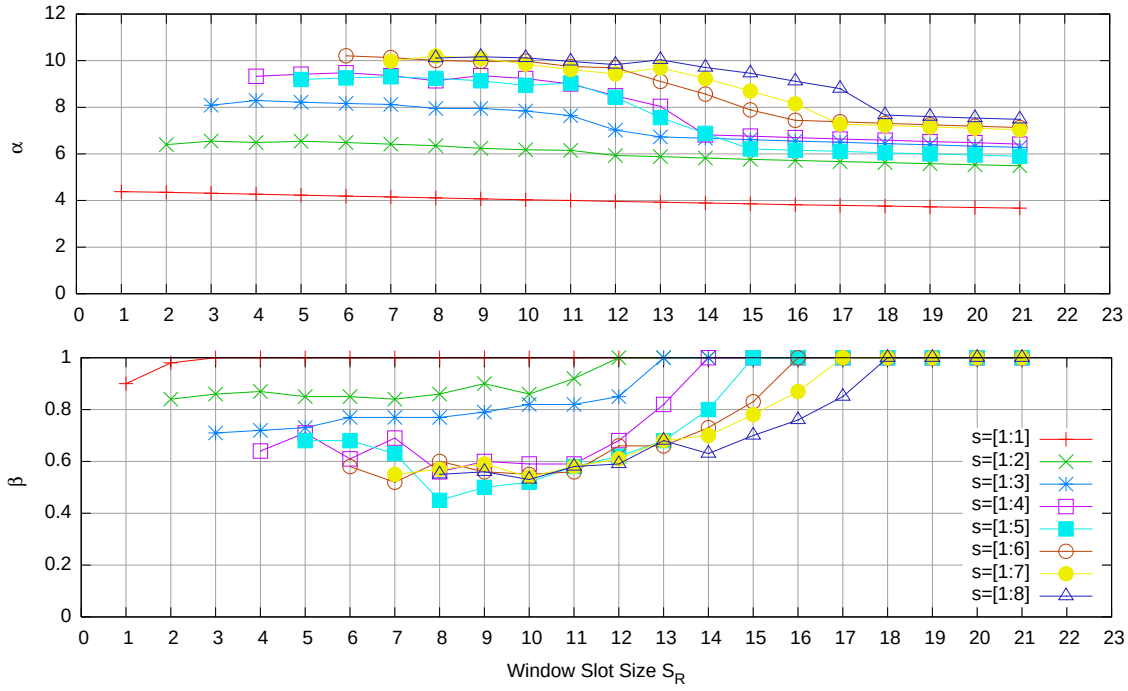

Figure 3.13: **Effects of S_R on different s ranges.**

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 α and responsiveness β. 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), α 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 α—after all, more bids should be captured resulting in higher value. But this

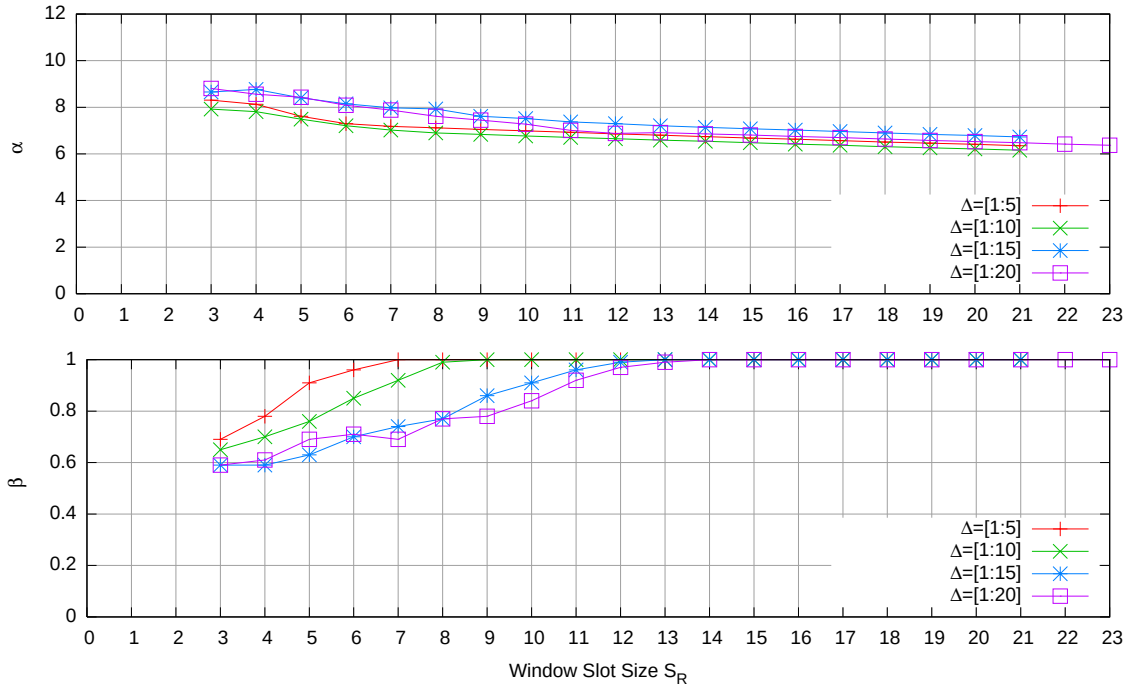

Figure 3.14: **Effects of S_R on different Δ ranges.**

intuition is incorrect for the tested workloads, and instead:

Higher S_R results in a lower system value α.

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_R 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

Parameter	α	β
Arrival Rate $\lambda \Uparrow$	$\Uparrow$	$\Downarrow$
Nodes $[n_{low} : n_{high}] \Uparrow$	$\Uparrow$	$\Downarrow$
Slots $[s_{low} : s_{high}] \Uparrow$	$\Uparrow$	$\Downarrow$
Patience $[\Delta_{low} : \Delta_{high}] \Uparrow$	$\leftrightarrow$	$\Downarrow$

Table 3.2: **Summary of workload parameter effects**. Window size S_R is fixed. $\Uparrow, \Downarrow, \leftrightarrow$ means increase, decrease, and no change, respectively.

value) bids from winning.

When demand is greater than supply (e.g., $\lambda \geq 3$), the desirable range for window slot sizes seems to be between $10 \leq S_R \leq 12$. These sizes achieve the most balanced α and β combinations. Based on this, a good range to use for the rolling window slot size S_R is

$$S_R = \bar{\Delta} + [\bar{s}], \tag{3.13}$$

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_R. While it seems logical to use a smaller S_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_R results in better responsiveness β.

When demand is low, most bids can be allocated immediately, resulting in a very high β. 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 α. This does not, however, always apply to all possible workloads. I provide in this section a special workload case that potentially can yield higher α with a higher S_R.

Consider a single node N_R 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 a_i and departure d_i times.

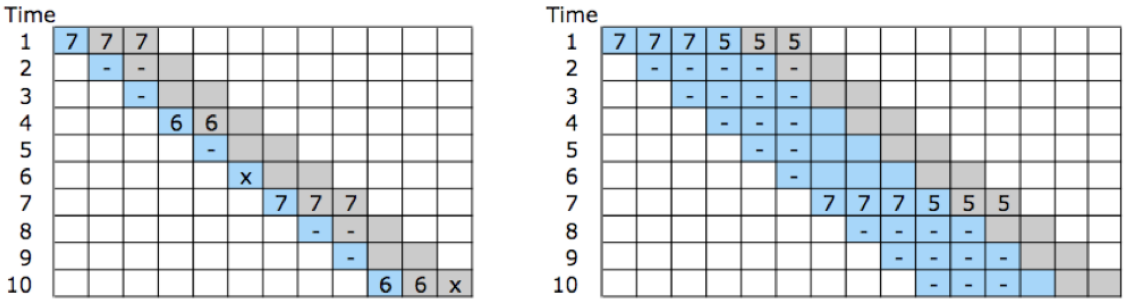

Figure 3.15: **Workload example that yields higher α with a larger window size.** For the left window, bids can start only in the first slot. For the right window, bids can start in the first four slots. ‘-’ indicates a sold slot and ‘x’ an un-allocatable slot.

Figure 3.15 shows how the allocation plays out for a window with slot size $S_R = 3$ and another with $S_R = 6$. *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 α with the first *ten* slots and consider unallocated slots to have a zero value. Thus, for the smaller window, $\alpha = (7 \times 3 + 6 \times 2 + 0 + 7 \times 3 + 6)/10 = 6$.

In the larger window, “555” bids are able to win before the “66” bids arrive. Here, α equals $(7 \times 3 + 5 \times 3 + 7 \times 3 + 5)/10 = 6.2$. Thus, in this example, α *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 α 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 $6 \times 2/3 = 4$ to system value α.

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 α.

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 β in both cases, system value α 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 α and β. 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 α and β

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 α and β.

Figure 3.16 shows results validating the effects. First, note that by increasing N_R, α 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. α 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. [7]

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

[7] 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.

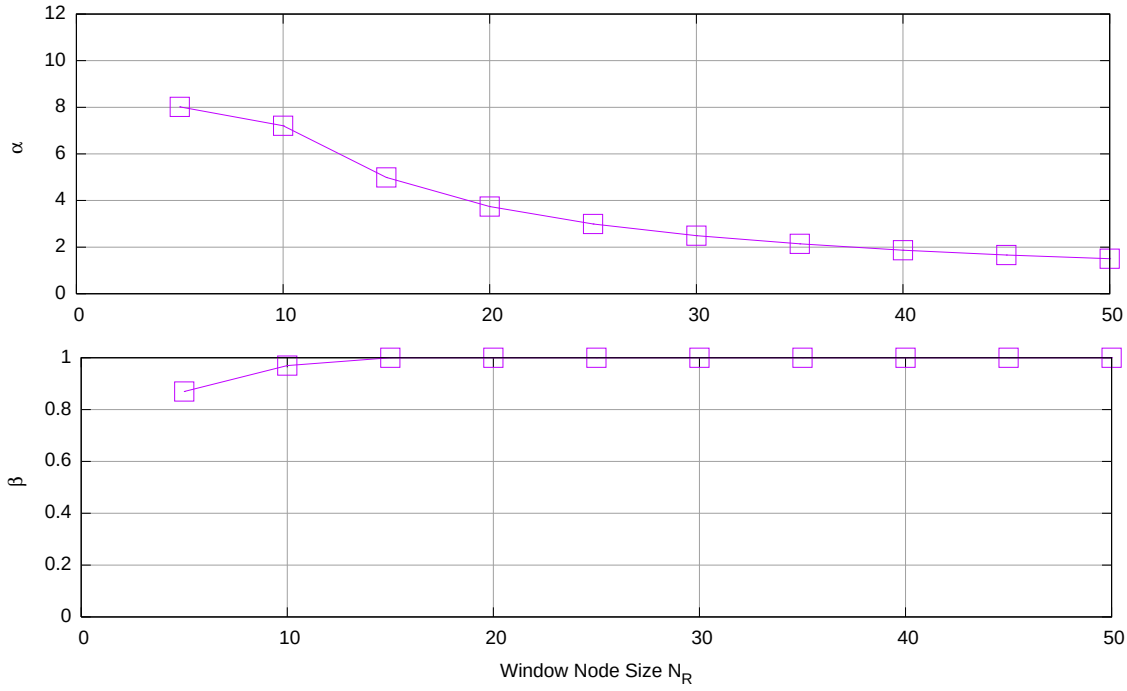

Figure 3.16: α **and** β **with varying node size** N_R**.**

Variable	**Demand/Supply**	α	β
$N_R \Uparrow$	$\Downarrow$	$\Downarrow$	$\Uparrow$

Table 3.3: **Effects of demand on metrics**: given fixed S_R.

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 λ and use only marginally increasing values.

Results for $\lambda = 3$ are shown in Figures 3.17. First, note that these are not the same α/β graphs as before. Instead, at the top, I show α 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.

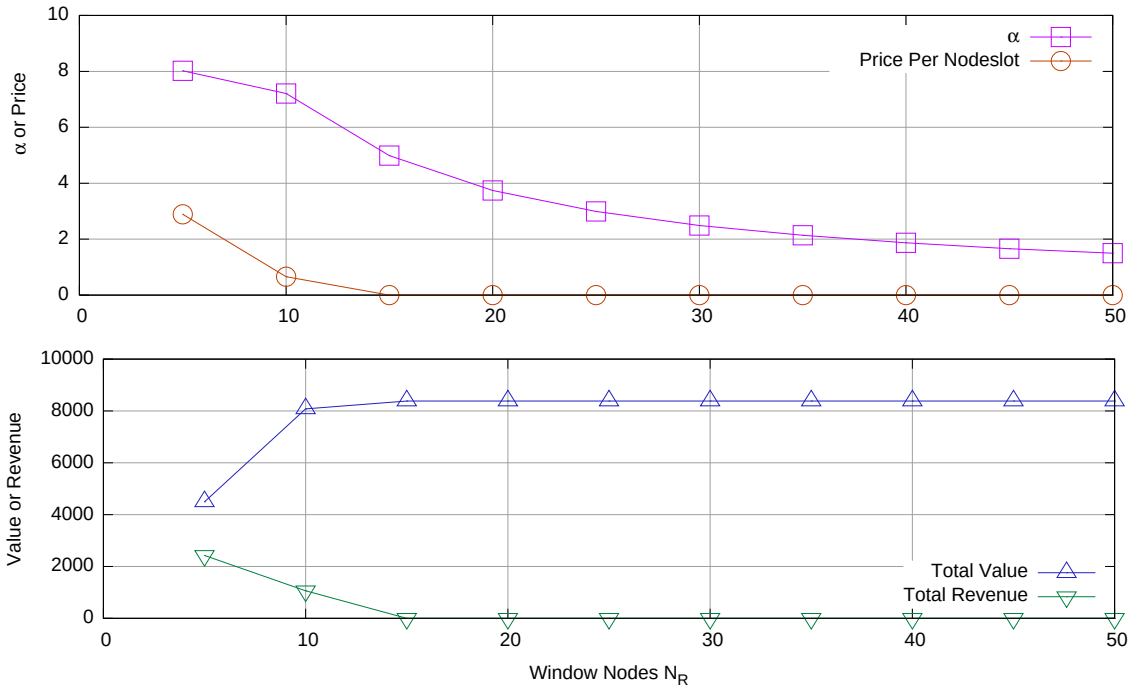

Figure 3.17: **Comparing value and revenue with $\lambda = 3$.**

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 $N_R = 15+$. 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_R, 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 λ increases. Figure 3.18 ($\lambda = 5$) appears similar overall to 3.17, except it achieves higher total value at higher N_R size (20). However, there is one new pattern observed. Note that total revenue rises from $N_R = 5$ to $N_R = 10$. After that, it *drops* as N_R increases.

Figure 3.19, with $\lambda = 10$, shows an even clearer trend. Total revenue increases steadily and tops off at around $20 \leq N_R \leq 25$, and then gradually drops off until it reaches 0 when $N_R = 40$. Thus, revenue is *maximized* when N_R 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_R 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_R reaches a certain level, these bids do win, making other

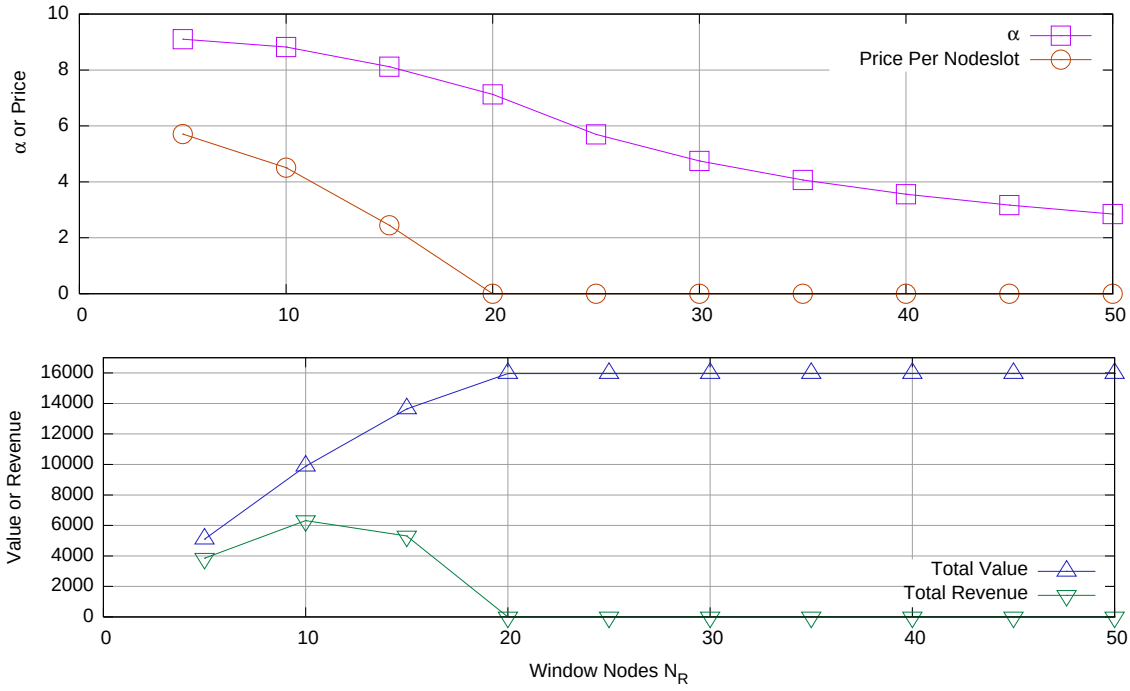

Figure 3.18: **Comparing value and revenue with $\lambda = 5$.**

low value bids lose. These losing bids result in low or even zero payments for a lot of the winners, hence leading revenue lower.

In summary, increasing N_R can generate additional revenue but the effect depends on demand. The goal of a system administrator is to monitor the demand trends before investing in additional nodes.

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*

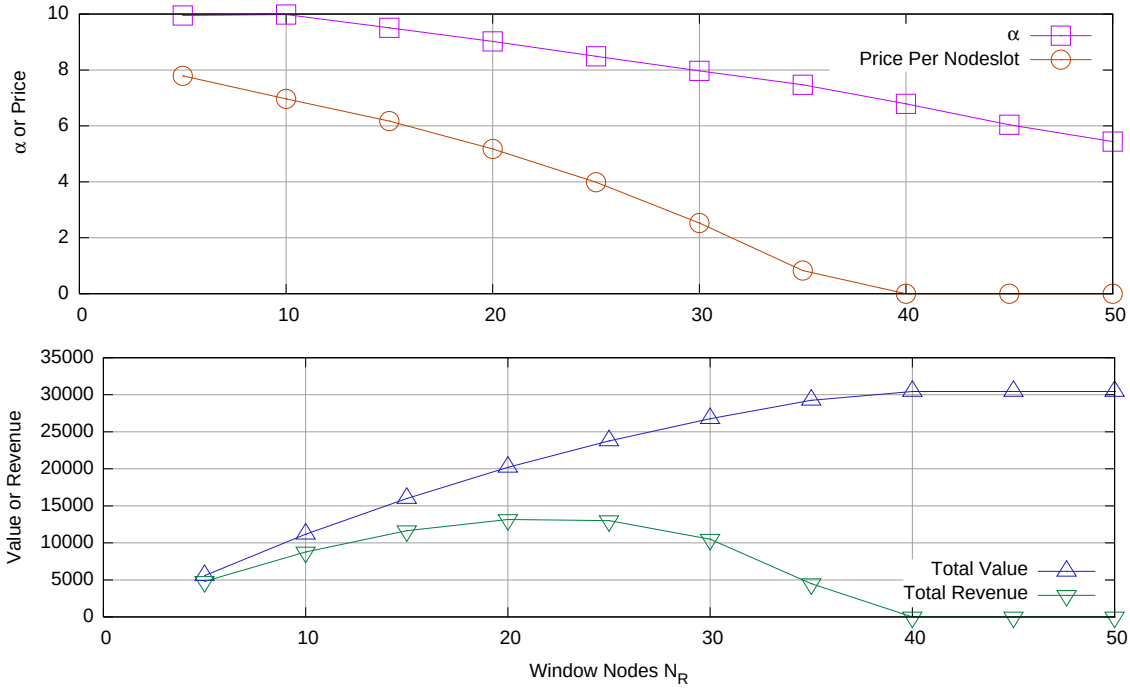

Figure 3.19: **Comparing value and revenue with** $\lambda = 10$**.**

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:

$$p_i = \max(r_R, v_j), \tag{3.14}$$

where v_j 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 $(N_R, S_R) = (1, 1)$ and bids for unit nodeslots ($n_i = s_i = 1$). Assuming two bids i and j with unit bid price $v_i = 10$ and $v_j = 5$, the following two cases can be considered.

1. Reserve price $r_R = 0$. i wins (since $v_i > v_j$) and must pay unit price $p_i = \max(0, v_j =$

5). Thus, i pays $p_i = 5$. Note this has the same outcome as before when there was no reserve price.

2. Reserve price $r_R = 7$. i wins and pays unit price $p_i = \max(7, 5) = 7$. Thus, i is charged more than v_j 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 $\lambda = 10$. I again vary different N_R sizes. This time, I create a reserve price $r_R = 5$. I decide on this price by looking at the price per nodeslot for $N_R = 20$ in Figure 3.19.

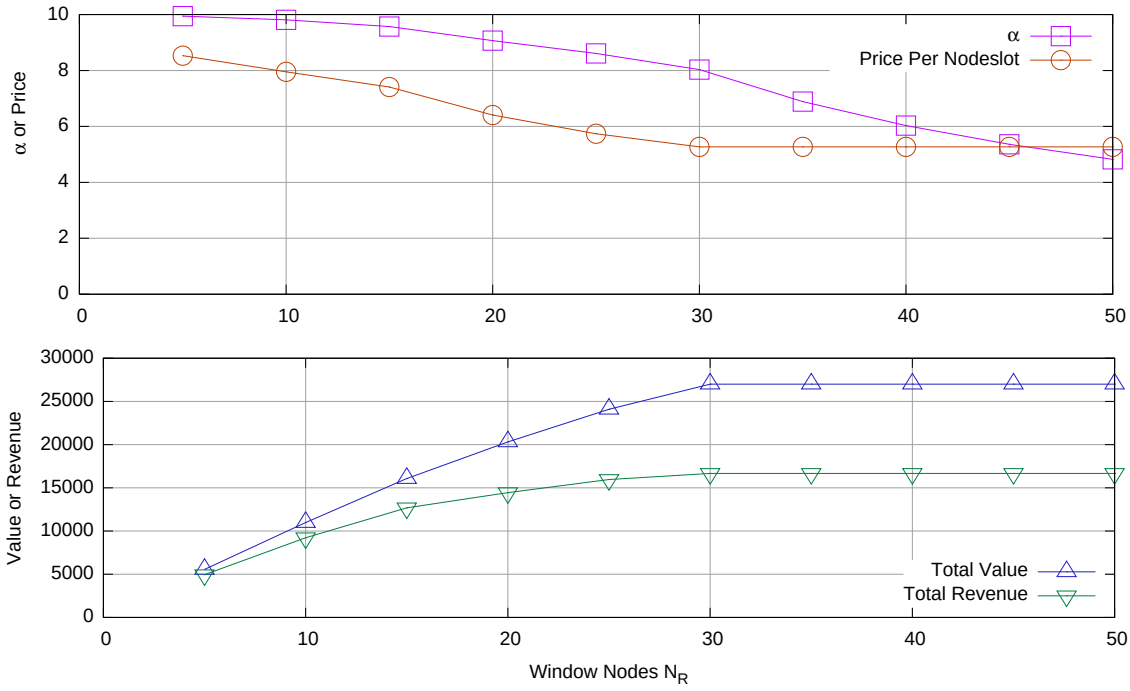

Figure 3.20: **Reserve price effects.**

Figure 3.20 shows the result. Note immediately that the total revenue curve now flattens for $N_R = 30+$, 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 α. However, they probably can do better than Roller regarding responsiveness β. 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.
2. At every t, all bids are sorted by bid value w_i regardless of nodeslot size ns_i and patience Δ_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 N_R is 8. For Roller, window size S_R is 12, per the results of Section 3.6. For each λ, I ran Greedy and Roller to collect metrics α and β. Figures 3.21 and 3.22 show results for marginally increasing and decreasing values, respectively.

System Value α Comparison. Roller and Greedy achieve virtually identical α 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 α for Roller, but not for Greedy.

Responsiveness β 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.

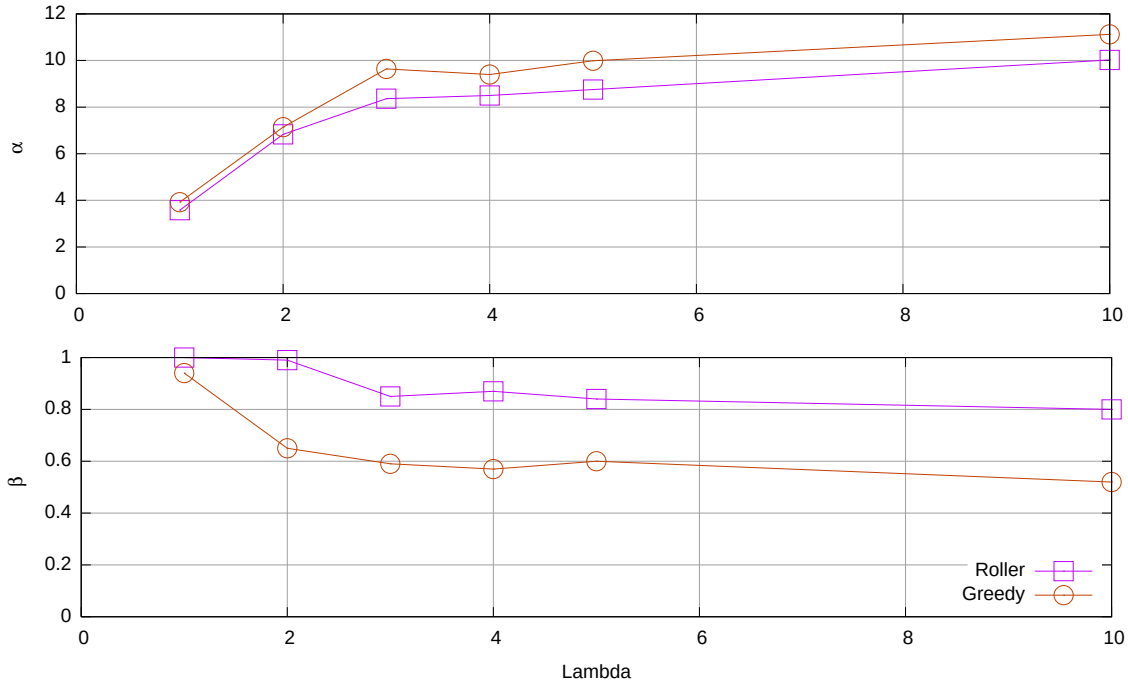

Figure 3.21: **Roller vs. Greedy. Marginally increasing value distribution.**

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-

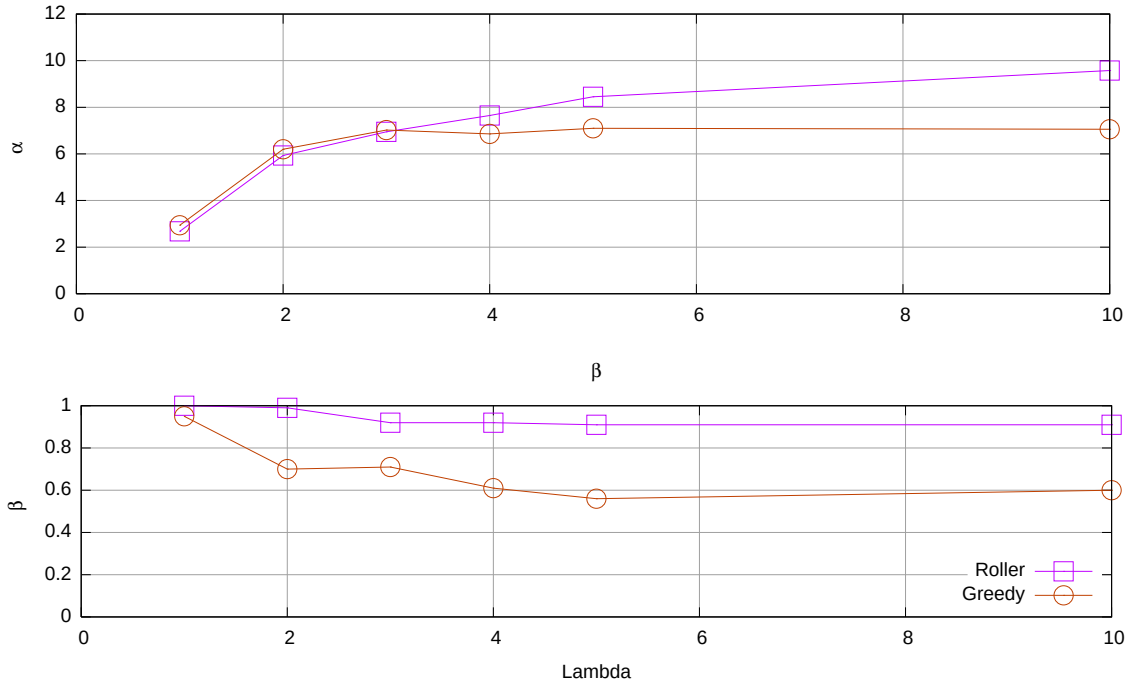

Figure 3.22: **Roller vs. Greedy. Marginally decreasing value distribution.**

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 a_i will be ranked first in every time period. For SJF, bids will be ranked by nodeslot size $n_i s_i$, 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_R is fixed at 8. For the rolling window, I use a fixed window slot size

$S_R = \bar{\Delta} + \bar{s} = 12$ for all tests. Results for marginally increasing and decreasing values are shown in Figure 3.23 and Figure 3.24, respectively.

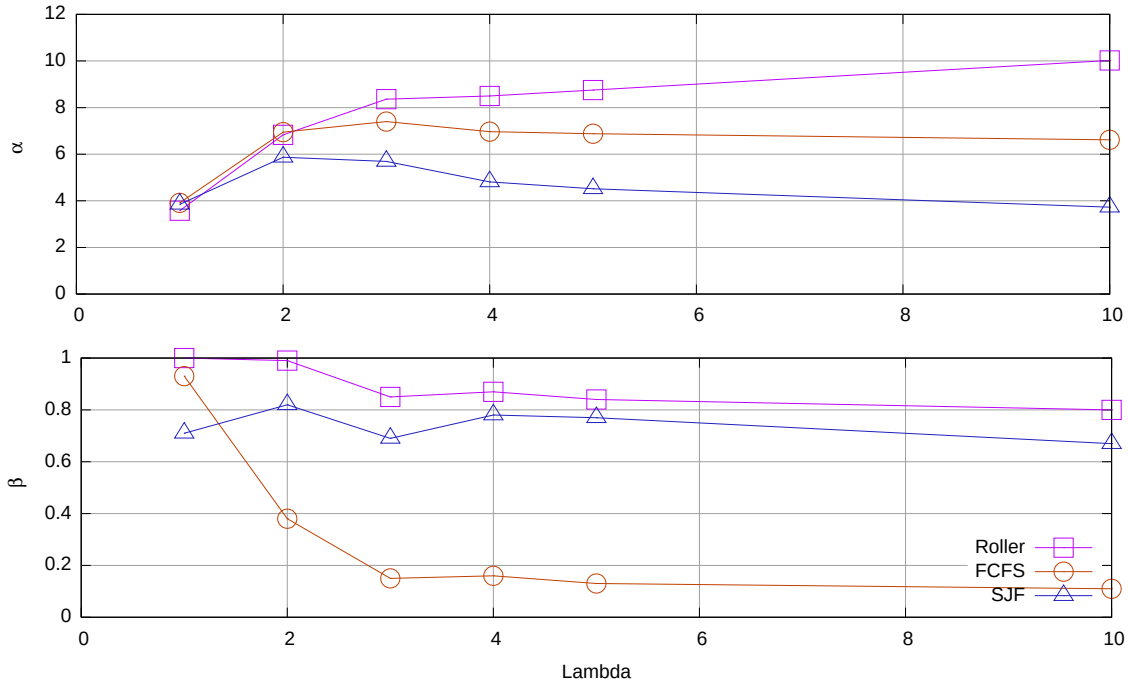

Figure 3.23: **Roller vs. FCFS vs. SJF. Marginally increasing value distribution.**

System Value α Comparison. First, I will discuss α for marginally increasing values. For Roller, system value α increases gradually as arrival rate λ increases. This is consistent with our results per Table 3.2. For FCFS and SJF, α increases for $1 \leq \lambda \leq 3$ but decreases slightly as demand increases per rising λ. System value α for Roller is almost twice as good as that of SJF for $\lambda \geq 3$, 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 α 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.

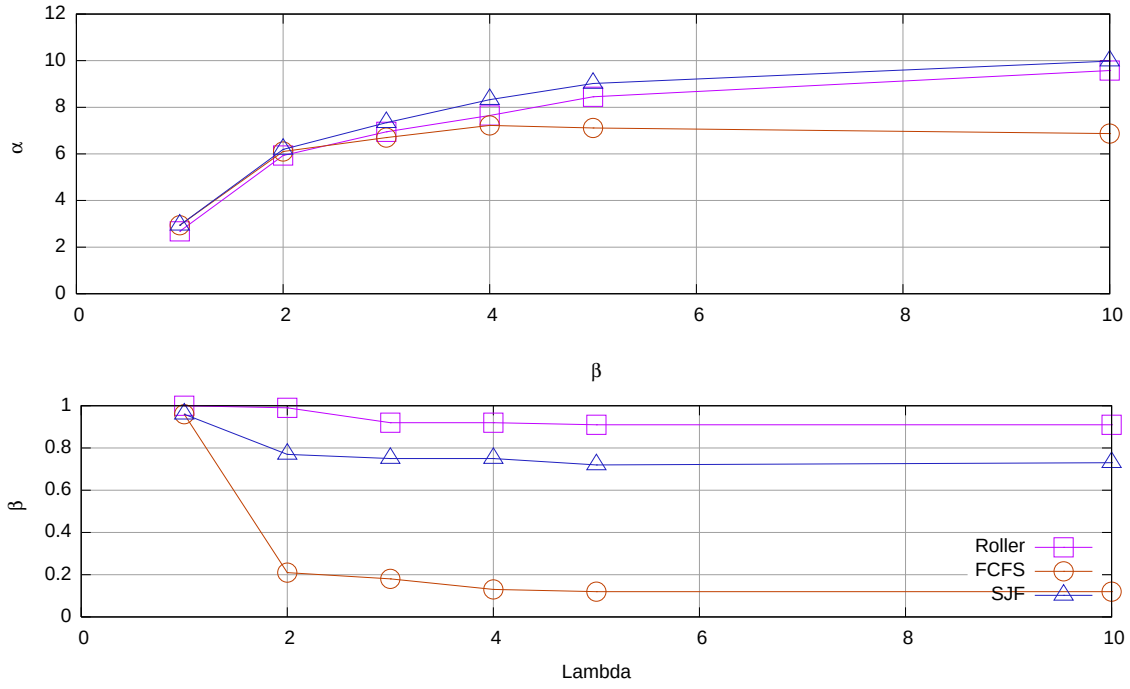

Figure 3.24: **Roller vs. FCFS vs. SJF. Marginally decreasing value distribution.**

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

Responsiveness β 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 α and β 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 α and β graphs again confirm the general behaviors of Roller. α for FCFS and SJF is more interesting: it rises and then drops off. The rise is due to increased winners in general, thus boosting α. However, it drops off precisely at the ceiling level. Because there are excessive nodes, α 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.

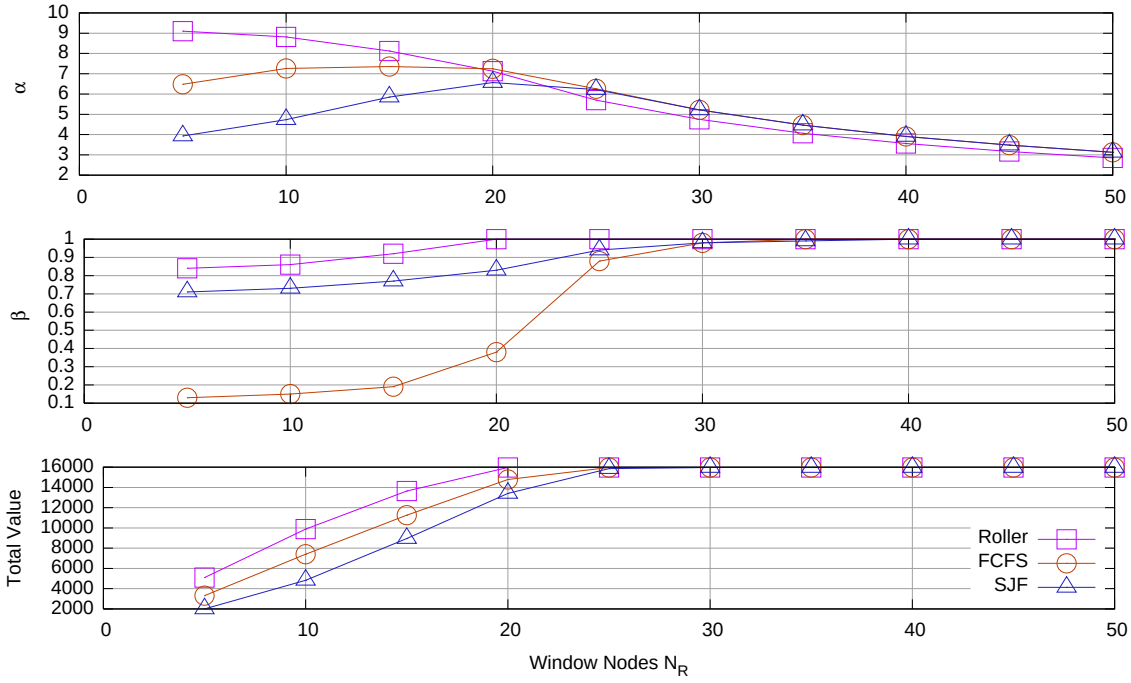

Figure 3.25: **Roller vs. FCFS vs. SJF: number of nodes and total values.** $\lambda = 5$**. Marginally increasing values.**

3.9 Late Allocation

In this section, I describe experiments to test the ρ-allocation rule. Again, the purpose of ρ 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 ρ 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') and actual departure (d)—denoted by 1.5x, 2x, and so forth. For example, if $d' = 10$ and $d = 5$, 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

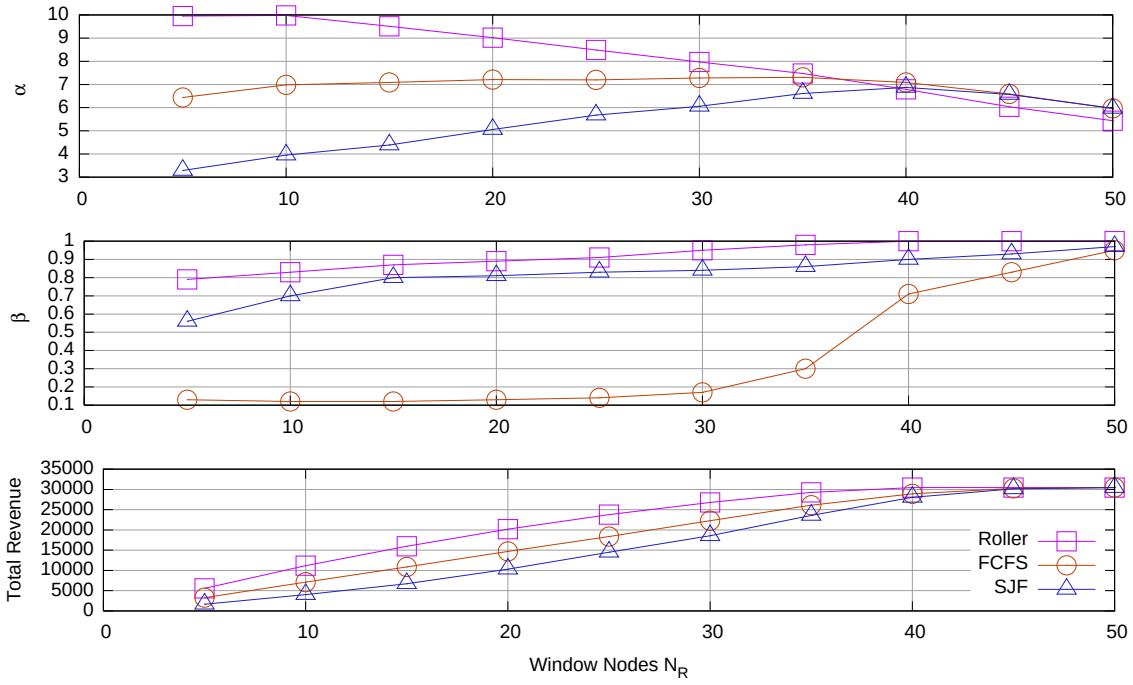

Figure 3.26: **Roller vs. FCFS vs. SJF: number of nodes and total values.** $\lambda = 10$**. Marginally increasing values.**

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_i, as such value is valid only for receiving the resource on or before its true departure d_i. 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.

I raise the following questions for experiments. Given an over-report multiple, what are the effects on average agent payoffs under varying ρ set by Roller? Similarly, what are the effects on system value α? For a given ρ, what is the over-report multiple that returns the highest average agent payoff? What is the corresponding optimal system value?

For the rolling window, the number of nodes N_R is 8 and window slot size S_R is 12. The percentage of over-reported bids in a run is fixed at 20% of all bids. Both over-report multiple and ρ vary for different experiments. I use base workload $L' = (T, \mathcal{P}, \lambda, [n_{low} : n_{high}], [s_{low} : s_{high}], [\Delta_{low} : \Delta_{high}], w(m, [x_{low} : x_{high}])) = (500, \text{poisson}, 3, [1 : 3], [1 : 3], [5 : 5], w(\Uparrow, [1 : 10]))$ for all of the experiments.

The first experiments address the questions regarding average payoffs. I generate a workload with a 20% population of over-reporting bids, each with d' based on a chosen multiple (1x to 2x, with 1x representing all agents are "truthful" and acting as a reference). Then I submit the workload to Roller using one of six different ρ ([0,20,40,60,80,100]) percentages. The payoff of each over-reporting bid is tracked, in order to generate an average payoff at the end of a run. Each run is performed 100 times.

Figure 3.27 shows the results. Each line represents a specific over-report multiple. The points of a line indicates a specific ρ and the average payoff achieved. The 2x over-report line, which represents the most aggressive over-reporting, has the most dramatic pattern. It starts off with very high average payoff when $\rho = 0$. Basically, when no bid is penalized for over-reporting, agents have strong incentives to over-report. Also, the average payoff is significantly higher than that of the 1x line because every 2x bid enjoys twice the number of VirtualWorlds periods to lower their payments, resulting in more chances to get low prices.

As ρ increases towards 100%, the average payoff drops for all over-reporting levels.

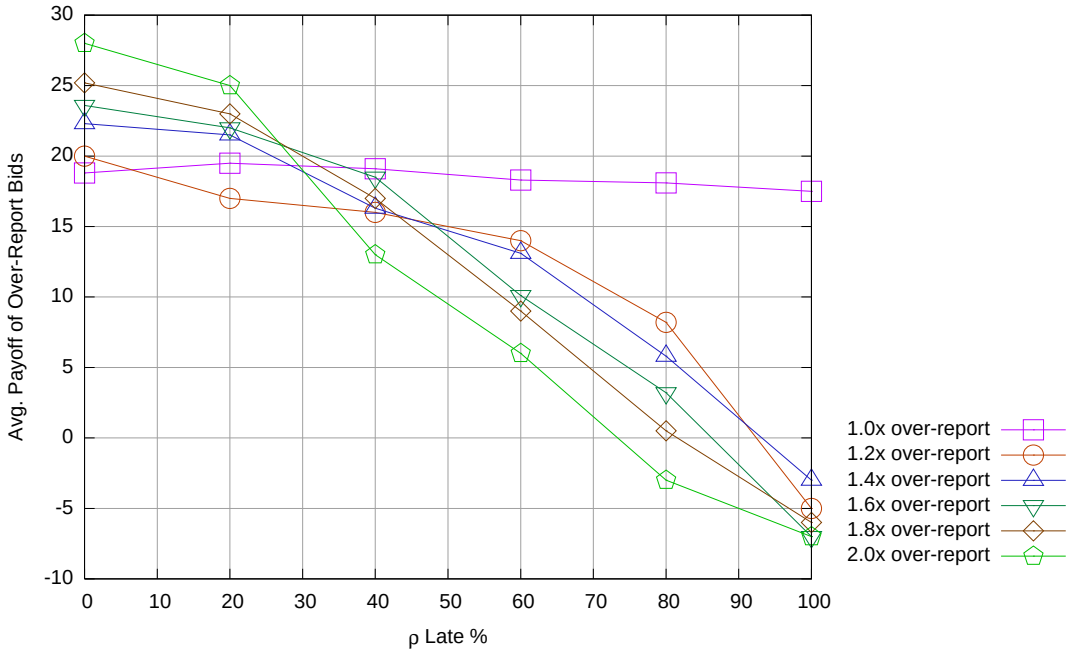

Figure 3.27: **Effects of ρ late % on average payoffs**: across different multiples of over-report departures.

However, the average payoff declines most steeply for those with the most dramatic over-reporting (i.e., 2x). This is because some of the bids will be assigned for late allocation. For example, for $\rho = 20\%$, every bid (whether over-report or not) has a 1 in 5 chance to be assigned late. Clearly, not all over-represented bids will be selected. Only those selected contribute possible negative payoffs, and thus lower average payoffs.

When $\rho = 100\%$, any over-reporting results in negative average payoffs and thus the incentive is removed. The payoffs for different multiples overlap in different places due to varying effects of ρ and other workload conditions, such as the payments available given other normal bids. Finally, for certain ρ such as 80%, agents with greater over-reporting (e.g., 2x) get hurt more than lower ones (e.g., 1.2x). With higher number of over-reporting periods, those agents with the greatest over-reporting, are penalized the most.

From the perspective of an agent thinking about over-reporting, it can reference the vertical points across a fixed ρ. For example, when $\rho = 0\%$, it is best to over-report with a high multiple (2x). When ρ is over 50%, it is best not to over-report (1x) and receive negative payoffs. I denote the highest point of each ρ as the "optimal multiple."

In Figure 3.28, I plot a similar graph but with system value α as the y-axis. System values for $\rho > 0\%$ are lower than that of $\rho = 0\%$, because for α only the true value is counted (i.e., no payment involved). When $\rho = 0\%$, any over-report bids still risk the chance that they may be allocated in an over-report departure, hence the lower overall system value. As ρ increases, the number of over-report bids that are allocated in over-report departure increases, further lowering system value. Another reason is that with demand greater than supply (i.e., $\lambda = 3$), it becomes more and more difficult to allocate for a bid, over-report or not, through late allocation. As a result, many bids in high ρ lose, resulting in lower value captured. Lastly, for the 1x multiple, system value decreases as ρ increases. First, when $\rho = 100\%$, many of the slots at startup time are wasted. Second, with more late bid allocations, the set of allocatable slots in the window shifts dramatically and the number of available slots decreases due to such shifts. For 1x multiple, my experiments show that the number of slots allocated when $\rho = 100\%$ is 9% less than that of $\rho = 0\%$. This implies that some slots and bids were left un-allocated, resulting in lost value.

With results from the first two figures, I address the optimal amount of system value α for each ρ percentage. First, the "optimal multiple" points are identified previously in Figure 3.27. These indicate what agents would do for a given ρ. Now, with the multiple identified, I look up the system value α captured by this multiple for the same ρ in Figure 3.28. For example, as $\rho = 0\%$ the optimal multiple is 2x, the system value for the 2x

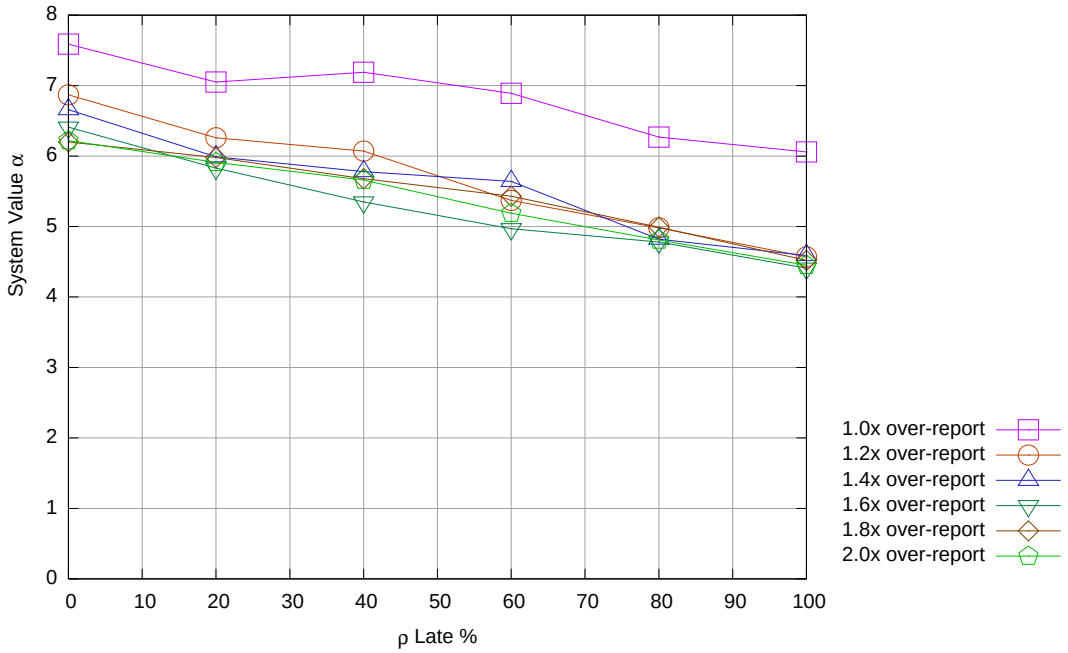

Figure 3.28: **Effects of ρ late % on system value** α: across different multiples of over-report departures.

line under $\rho = 0$ is used. I plotted these system values for each ρ to generate Figure 3.29. The graph suggests that the system should set $\rho = 40\%$. This is because when it is too low (e.g., $\rho = 0\%$), over-reported bids will be submitted and these agents have an edge. In addition, if set too high (e.g., $\rho = 100\%$), every bid suffers.

In Figure 3.30, I plot total revenue collected with each ρ percentage, across different over-report multiples. In general, for a given multiple, revenue increases as ρ increases. When more bids are considered "late," it becomes more competitive to win. This results in an increase in prices for the "late" bids while the rest of the bids face less competition. The revenue for 1x multiple is highest throughout because it captures higher system value (see Figure 3.28) and thus has more value to capture as revenue. In Figure 3.31, I plot a total revenue graph based on the optimal multiples points that deliver the highest agent payoffs

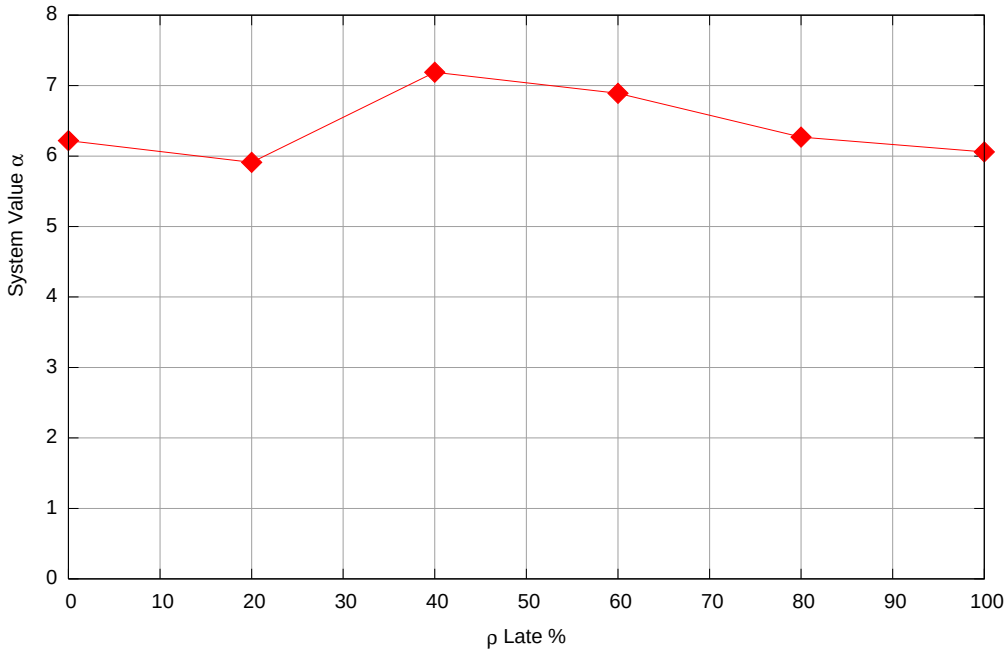

Figure 3.29: **System value α captured by each ρ late %**: driven by multiples that deliver highest agent payoffs.

(see Figure 3.27). Similar to Figure 3.29, the first two points with ρ equals 0% and 20% are driven by 2x multiples, and the rest by 1x multiple. In general, total revenue increases as ρ increases. This presents a tradeoff to systems designers as using ρ to maximize total revenue ($\rho = 100\%$ in Figure 3.31) will not lead to a maximum system value ($\rho = 40\%$ in Figure 3.29).

One last thing of interest to note is the mis-report of "early arrival" by agents. Assume these agents always report true departures. Agents have no incentive to mis-report early arrival. For example, an agent with true needs for resources starting in time 4 mis-reports an arrival time of 1. This is because resources won in these periods yield zero true value to the agents. However, if selected as "late" with $\rho > 0$, such agents will not be allocated in these mis-reported early periods and still achieve value for their allocations. In this case,

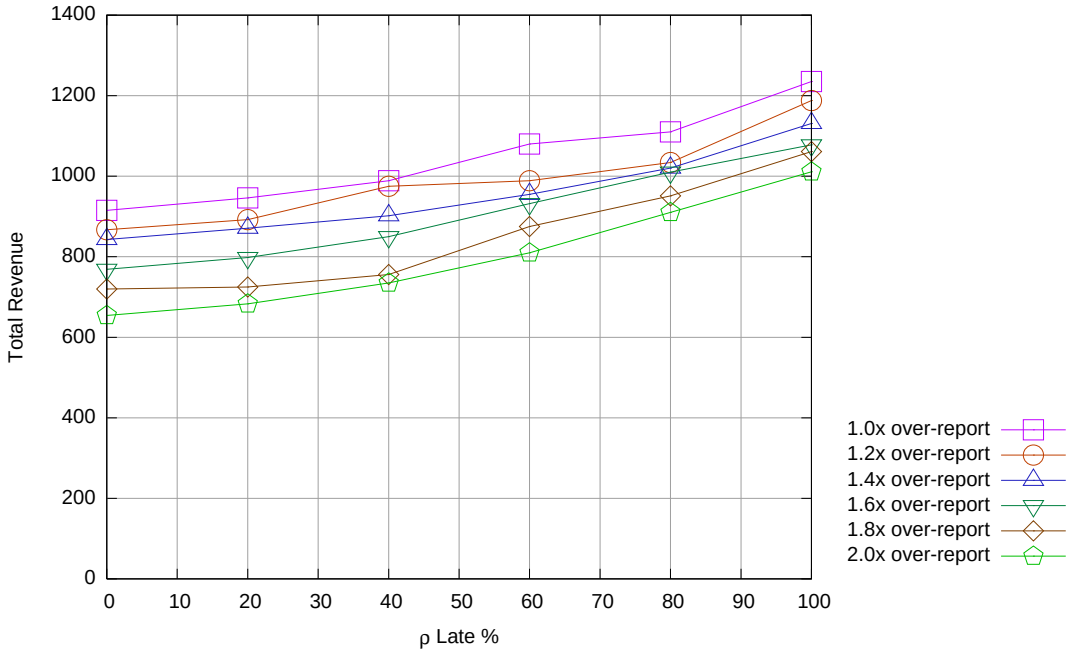

Figure 3.30: **Total revenue captured by each ρ late %**: among all over-report multiples.

agents have incentives to under-report arrival, in order to obtain more VirtualWorlds periods for lowering payment. Thus, with late allocation Roller is not strategyproof for these cases (a tradeoff). Nonetheless, early arrivals are debatable as a realistic model, as agents usually do not know of their needs until such needs do arise.

3.10 Summary

In this chapter, I introduced Roller and made a number of discoveries that address the two research questions. First, *Roller is strategyproof with respect to value and size, and is configurable in regard to providing strategyproofness for different aspects of allocation timing*. Its allocation and payment rules mitigate strategic behaviors for these dimensions.

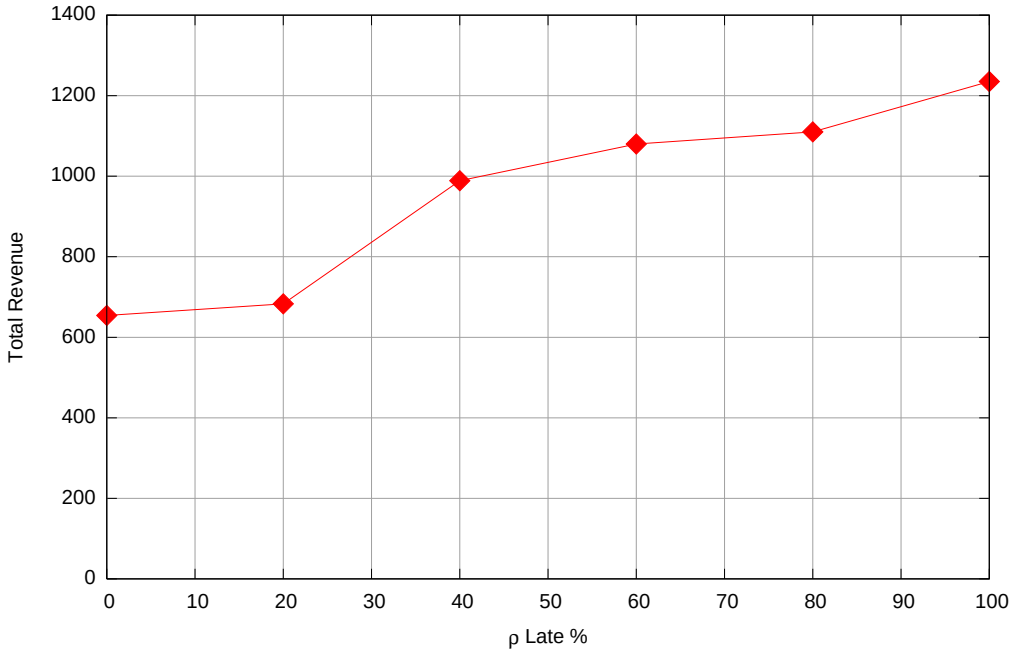

Figure 3.31: **Total revenue captured by each ρ late %**: driven by multiples that deliver highest agent payoffs.

For example, agents have no incentive to determine which auction may yield the highest utilities, as the payment rule provides the optimal situations across an agent's allocation timing.

Second, *Roller captures a competitive amount of value and is highly responsiveness* in the tested workloads. The key is to select a rolling window size that best balances value and responsiveness. Compared to a value-based allocator, Roller is more responsive and competitive in terms of value captured. Compared to non-value-based allocators, Roller commands respectable responsiveness, while generating higher value.

There is much future work to do, the most important being to optimize Roller allocations while at the same time not sacrificing its strategyproof properties. Exploring Roller's performance with new classes of workloads is also crucial, especially for opportunities to

identify classes of workloads that can achieve high α and β simultaneously with the use of a larger rolling window. Other work includes exploring alternative resource abstractions. For example, some systems may only want to offer a partial amount of future slots (e.g., 50%) at any given time.

Chapter 4

Virtual Currency Design

4.1 Introduction

In this chapter, I describe virtual currency [1], artificially created objects that mirror the role of real currency such as the US Dollar (USD) and Japanese Yen, as a medium of exchange for resources of a system. I also denote virtual currency as "tokens" and the symbol "@" interchangeably. In recent years, many systems have adopted virtual currency, including different market-based systems, as well as online services such as games and social networks (e.g., World of Warcraft [16], Second Life [12], Facebook [5]).

Why use virtual currency? In real-life, agents already use real currency to pay for goods and services, including buying cloud computing resources [1]. Unfortunately, some systems cannot adopt real currency. Examples include non-profit systems such as PlanetLab [80] and internal corporate systems. Asking agents to pay USD for free resources provided by an organization or entity is often infeasible and may even have complex financial and tax

[1] Also referred to as virtual money, virtual cash, artificial money, scrip, credit, share, token, among others.

implications. However, in order to use auction-based solutions such as Roller, some form of exchange medium is required. Virtual currency offers these systems an alternative medium of exchange for agents to use in bidding and the system to consider when determining and executing allocations and payments.

However, adopting virtual currency comes with a price. The system must specify and implement, from scratch, the infrastructure and monetary policies that enable a virtual currency environment. The infrastructure addresses necessary elements including the form (e.g., digital secure tickets [41] or database-driven as in Mirage [30]), creation process, storage, transactions, as well as security of virtual currency. The monetary policies specify operational rules including how much currency to create, and how much each agent should receive. Essentially, these are analogous to real-world examples such as the USD (form), US Mint (creation process), the banking system (storage and transactions) and the Fed (monetary policies). Thus, it takes serious efforts to create a virtual currency environment.

In this chapter, I focus solely on the effects of monetary policies. This topic has not been widely studied in market-based computer systems. Understanding the effects of monetary policies is important for designers of market-based systems to mitigate undesirable outcomes such as currency crises that have been seen in online worlds [2]. In fact, some for-profit adopters of virtual currency have hired traditional economists to manage policies, suggesting a demand for a systematic study of this topic. The main research questions of the chapter are as follows:

Given a set of reusable resources and agents with specific workloads and strategies, how do different monetary policies affect aggregate value and resource allocation?

[2] SecondLife [18] suffered inflation by printing too much currency.

And how do they influence the effects of different agent strategies?

To address these questions, I devise a framework with the following three components:

- **Monetary Policy Design Space** (Section 4.2): A monetary policy [3] should inform the amount of currency to create and the way to distribute it both initially and then on a continuing basis. I will parameterize different policies through a set of policy dimensions.

- **Agent Models** (Section 4.3): Agents have a demand for resources over time. How agents value their jobs, and the strategies used by agents to submit bids, impacts the effectiveness of the monetary policy.

- **Equilibrium Analysis** (Section 4.4): For a given monetary policy, I compute the symmetric mixed strategy Nash equilibrium for a fixed set of agent strategies and workload, as well as the associated value captured by the system at equilibrium.

Using the framework, I run a series of experimental studies in Sections 4.5 through 4.8. These experiments use Roller as the underlying method for resource allocation [4]. I finish the chapter with a summary in Section 4.9.

4.2 Monetary Policy Design

In this section, I establish a design space for monetary policies. My approach is to create different policies through a set of policy dimensions: $P = (D, M, F)$ (see Figure 4.1),

[3] Also referred to as simply "policy" in this chapter.

[4] Roller is only for resources and is not involved with any virtual currency components.

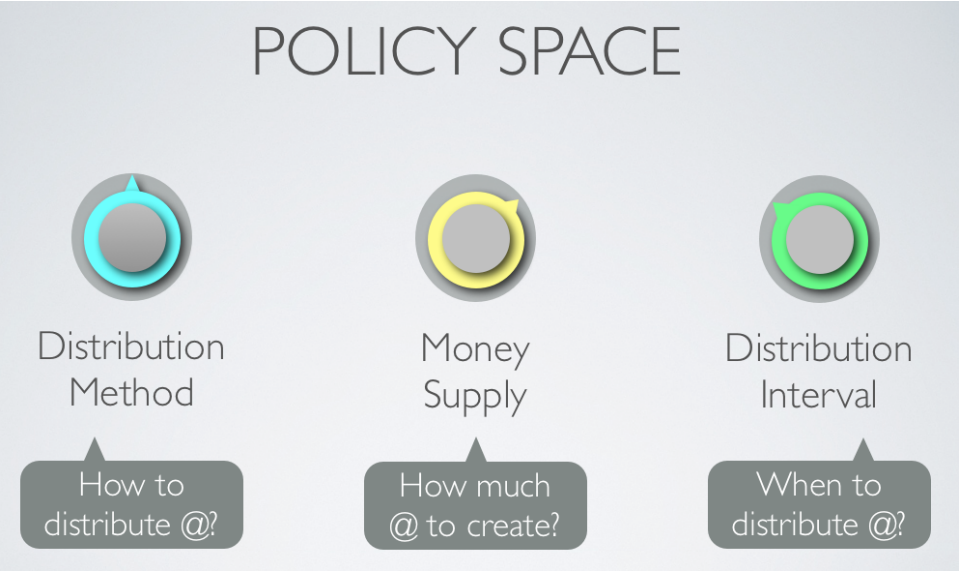

Figure 4.1: **Monetary policy dimensions.**

where D is the distribution method (*how to distribute currency*), M is the money supply (*how much currency to create*), and F is the distribution interval (*when to distribute currency*), respectively. D is a set of four methods, all of which are non-parameterized. A single policy is created by parameterizing M and F and selecting one of the four methods. A sound monetary policy must address all three components in the framework to ensure the virtual currency system functions as desired.

I first provide a model of a virtual currency system to provide some context for policies and dimensions. Then, I describe each of the three policy dimensions in detail.

4.2.1 System Model of Virtual Currency

For this chapter, I am making the following assumptions. The virtual currency system is supported by a simple centralized infrastructure. It is entirely software-based, with tokens in the form of digital records stored in a database. A bank uses the database to create and

manage a system account as well as accounts for each agent. The bank also handles all transactions between the system and agents. Thus, every agent relies on the bank for all currency-related activities.

By design, the system I am modeling is a closed economy. It supports only one currency and agents cannot use any other currency to bid for resources. Because the resources are system-owned, transfers among agents are not necessary.

At initialization, the system chooses a policy by setting each of the policy dimensions. The bank accordingly creates a number of "tokens" based on the money supply M and credits them to the system account (with a bank balance of B^t at any time t). Based on the chosen distribution method D and the distribution interval F, the bank deducts tokens from the system account and distributes them to different agent accounts (with balance b_i^t for agent i at time t). Every time an auction clears, the bank debits tokens from agent accounts for payments and credits them back to the system account.

Different workloads describe the jobs that seek resource allocation by the system over time. Clear workload definitions are important for the design of monetary policy, as many of the policy dimensions take workload information as inputs. For example, a policy may distribute currency to agents based on how many jobs they have submitted to the system.

In this discussion, jobs originate from a fixed set of agents $\mathcal{A}$, each of which has a repeated job demand for system resources over time. I simulate all activities of every agent, in order to study its behaviors over time. Every few periods, each agent learns about a number of new jobs. Each job has node size (n_i), slot size (s_i), arrival time (a_i), and departure time (d_i). In this simulation, upon learning of a new job, an agent immediately attaches a true value w_i (in USD and is private to the agent) to the job. The job is then appended to the

agent's job queue, which holds jobs that are active (not yet allocated and not expired). The number of jobs in agent i's queue at time t is denoted by k_i^t. Jobs in the queue are sorted in order of arrival time and are labeled $\{1,2,...,k\}$. No two jobs of an agent overlap in time including their arrival and departure times.

The above model shares the same attributes as Roller. Neither support more detailed node descriptions such as frequencies. The new job arrival process above differs from both Roller and Mirage, with the former using a Poisson process. Roller also uses a more sophisticated value distribution.

The system accepts bids from agents on an ongoing basis and runs a Roller auction every time period t (e.g., an hour) as described in Chapter 3. Every period, each agent checks whether its first job in the queue is valid for the current time period. If it is, the agent submits a bid value $\hat{w}_i^1$ [5]. If the bid is successful, the bank deducts payment from the agent's account and credits the system account. Winning or expiring jobs are removed from the agent's job queue.

4.2.2 Money Supply (M)

The money supply (M) addresses the question: "how much currency to create?" This includes the balance of the system account as well as the individual balances of each agent. In the real-world, money supply affects many things such as inflation. In this chapter, I study fixed money supply, where the total number of tokens in the system is created once at initialization and fixed over time. These tokens cycle through the system as agent payments and bank distributions.

[5] I denote $\hat{w}_i^1$ as bid value (in tokens) and w_i^1 as the true value (in USD). The superscript ^{1} denotes the first job in agent is queue.

4.2.3 Distribution Interval (F)

The distribution interval (F) answers the question,"when to distribute currency?" Because agents spend tokens to acquire resources in Roller, but have no way to earn tokens, they have a net outflow of currency. Therefore, the system must redistribute currency from its own account back to the agents.

The time periods for distributions are determined by the distribution interval F. Starting at time period 0, distributions occurs every F time periods (i.e., 0, F, $2F$, $3F$, ..., etc.). The auction runs every time period t, irrespective of what F is. If $F = 1$, then every time period the auction runs and the bank distributes currency. If $F > 1$, then there are multiple auctions between each distribution.

Specific values of F can have major effects on the system, given fixed money supply. For example, a system that has $F = \infty$ will likely drive all agents to "bankruptcy," regardless of M and B. I will explore the ramifications of different ranges of F on the system.

4.2.4 Distribution Method (D)

The distribution method (D) addresses the question "how to distribute currency?" A distribution method determines how much virtual currency each agent will receive. There are a total of four, non-parameterized, methods D1-D4, summarized in Table 4.1 that are discussed below.

D1:Uniform

For this method, the bank simply distributes its total current balance *uniformly* to all agents. I use $\mathcal{A}$ to denote the set of agents, $|\mathcal{A}|$ as number of agents, and b^{t-1} and b^t to

Method	**Details**
D1:Uniform	Divide bank balance equally for every agent.
D2:Stable	The lower the standard deviation of an agent's bid values, the more tokens it receives.
D3:Active	Agents with a higher cumulative number of jobs submitted to-date receive more tokens.
D4:Urgent	Agents with low average patience receive 2x the tokens.

Table 4.1: **Distribution methods**: overview.

denote the account balance of an agent i *before* and *after* a distribution. The bank distributes an equal portion of its balance to every agent at every distribution period. Similarly, B^{t-1} is the balance of the system account, all of which is available for distribution, right before a distribution occurs (after which it goes to zero).

$$b_i^t = b_i^{t-1} + B^{t-1}/|\mathcal{A}|, \qquad \forall i \in \mathcal{A} \tag{4.1}$$

D2:Stable

For systems with agents that bid randomly or aggressively, this method rewards agents with more predictable bidding behaviors. Specifically, the bank distributes more currency to agents with more stable bid price patterns. I use σ_i^{t-1} to denote the standard deviation of bid values by agent i between time 0 and $t-1$. This time range captures the complete

bidding history of each agent [6]. Each agent will receive tokens based on the inverse of the natural $\log$ of the standard deviation. At every distribution period, D2:Stable distributes currency as follows:

$$b_i^t = b_i^{t-1} + \frac{\log(\sigma_i^{t-1} + 2)^{-1}}{\sum_{j=1}^{|\mathcal{A}|} \log(\sigma_j^{t-1} + 2)^{-1}} \cdot B^{t-1} \tag{4.2}$$

Thus agents with lower standard deviations receive more tokens. Mathematically, adding 2 to the standard deviation is to ensure the denominator is positive (e.g., to avoid calculating inverses of $\log(0)$ and $\log(1)$). At time 0, the method uses D1:Uniform as default as no agent has submitted any bid.

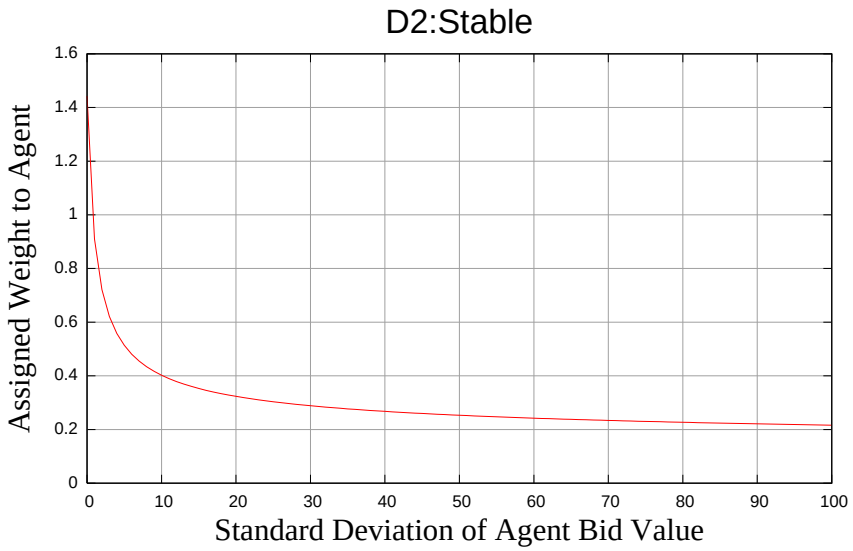

Figure 4.2: **D2:Stable agent weight calculation.**

[6] It is conceivable to use a shorter time range, for example, the past several periods.

D3:Active

For this method, agents with a higher long-term demand for jobs receive proportionally more tokens. These are agents that submit more bids over time (whether those bids win or not are irrelevant). This is applicable to systems such as PlanetLab, which prefers active users (e.g., researchers that often have lots of experiments). Therefore, agents receive tokens in proportion to the cumulative numbers of unique bids submitted from time 0 to $t-1$, denoted as h_i^{t-1}. Formally, D3:Active distributes currency to agents as follows at every distribution period:

$$b_i^t = b_i^{t-1} + \frac{h_i^{t-1}}{\sum_{j=1}^{|\mathcal{A}|} h_j^{t-1}} \cdot B^{t-1} \tag{4.3}$$

As before, at time 0, the method defaults to D1:Uniform.

D4:Urgent

For this method, agents with shorter average patience ($d_i - a_i$) receive proportionally more tokens. These agents essentially have more jobs with earlier deadlines. I denote the maximum patience allowed in a system as Δ_{max}. D4:Urgent assigns a score of 2 to a job with patience less than $\Delta_{max}/2$ and a score of 1 otherwise. An average score for each agent is calculated from the score of the previous 5 jobs. I denote this score as g_i^{h-1}. D4:Urgent distributes currency to agents at every distribution period as follows:

$$b_i^t = b_i^{t-1} + \frac{g_i^{t-1}}{\sum_{j=1}^{|\mathcal{A}|} g_j^{t-1}} \cdot B^{t-1} \tag{4.4}$$

Thus, an agent with short patience for all of its previous 5 jobs will receive twice as many tokens as another agent with long patience. As before, at time 0, the method defaults

to D1:Uniform.

4.2.5 Mirage

The monetary policy design discussed above differs from that of Mirage. In Mirage, each agent is assigned a fixed number of shares and a baseline value. The bank in Mirage distributes currency proportionally using these shares. Thus, there is only one distribution method. Mirage assigns different baseline values to different agents, with internal staff receiving twice the amount of external researchers. In addition, Mirage uses a "use it or lose it" policy that taxes agents with account balances over their baseline value. In this chapter, I do not explore tax effects because the agent models used here do not including hoarding agents.

4.3 Agent Models

In this section, I build agent models based on three aspects: 1) *Currency basis* refers to whether the agents use virtual or real currency to reason about the intrinsic value of a job and for making transactions in a system; 2) *Workloads* define the set of jobs submitted by agents to the system over time; and 3) *Agent strategies* determine how agent represent jobs with USD values as token-based bids into the system.

4.3.1 Currency Basis

The currency basis, in terms of real currency ($) and virtual currency (@), that every agent uses to value resources and to make bids and payments is critical to model a system.

In most real-world systems, agents hold a true value for resources in real currency (e.g., USD), and bid and pay for resources in USD. This "all real currency basis" is simple for both system and agents.

However, a system that uses virtual currency involves agents that bid and pay in tokens, while still holding a true value for those resources in USD. In this scenario of "mixed currency basis," each agent makes implicit and individual decisions privately in terms of how much virtual currency to submit as a bid value for a job that has a certain true value in USD. This is the currency basis for this chapter.

Lastly, there is also an "all virtual currency basis" in which agents also value in the virtual currency. However, like any country in the early days of using its own fiat currency, it will take a long time for a system to create trust and stability for many agents to endorse virtual currency.

4.3.2 Workloads

In this chapter, I use the following four types of workloads.

- W1:Common. For this workload, all jobs from every agent are drawn from the same set of ranges for all job variables.
- W2:Jobs. The workload consists of jobs drawn from two different job arrival ranges. Some agents receive new jobs from the *high jobs* range, while the remaining agents receive jobs from the *low jobs* range.
- W3:Combo. This workload extends W2:Jobs. Some agents receive new jobs from the *high values* range. In addition, these new jobs draw from a high value range. Thus,

the more jobs an agent has, the more valuable these jobs are.

- W4:Patience. The workload consists of jobs drawn from two different patience ranges. Some agents receive new jobs form the *low patience* range. These low patience jobs also draw from a high value range. Thus, low patience agents have high value jobs.

The full details of the ranges of different job variables for each of the workload are described in Table 4.2. "# New Jobs" is the number of new jobs each agent learns of every five time periods. "True value" is for the whole job and is in USD. The true value (in USD) of each job is private to an agent and unknown to everyone else including the system. The true value is fixed permanently when an agent first learns of a job and is not affected by any future events, including any virtual currency allocated to the agent. For *nodes* and *slots*, jobs for all workloads draw from the range [1:3] and are not listed.

On the supply side, the Roller mechanism is used for resource allocation (Chapter 3). For the rolling window settings, $R = (N_R, S_R, L_R) = (8, 8, 5)$. In other words, the rolling window is 8 nodes by 8 slots, with a laststart time of 5. The experiments all run for a total of $T = 500$ periods. Every experiment is repeated 50 times.

These workloads differ from those used in evaluating Roller. First, in the Roller experience of Chapter 3, we did not use differently typed workloads. Second, we are now generating true values with a less sophisticated method. Third, we do not parameterize workloads here as we did in the Roller experiments. Overall, my goal is to have a set of simpler and more predictable workloads to use to study the policy dimensions.

WORKLOAD	AGENT TYPE		RANGE		
	Type	*# of Agents*	*# New Jobs*	*Patience*	*True Value*
W1:Common	N/A	10	[1:3]	[0:5]	$[1:100]
W2:Jobs	High Jobs	5	[3:4]	[0:5]	$[1:100]
	Low Jobs	5	[1:2]	[0:5]	$[1:100]
W3:Combo	High Values	5	[3:4]	[0:5]	$[100:200]
	Low Values	5	[1:2]	[0:5]	$[1:100]
W4:Patience	Low Patience	5	[1:3]	[0:2]	$[100:200]
	High Patience	5	[1:3]	[3:5]	$[1:100]

Table 4.2: **Workloads for virtual currency experiments**: Grey boxes highlight differences between two types of agents in a workload.

4.3.3 Agent Strategies

With a mixed currency basis, agents have to choose strategies for bidding some number of tokens on jobs that have true values in USD. The type of strategies that can be used is unbounded, as an agent can strategize using any number of factors it chooses. An agent is free to choose a strategy, such as the ones discussed below, that depend on some standard variables, or ones that depend on some ad-hoc variables (e.g., roll of dice, day of the month, etc.). I focus on a small set of strategies for testing the policies.

The motivation to analyze monetary policy with agent strategies in experimental studies originates from empirical game theory [48]. For my experiments, there are a total of four strategies, as listed in Table 4.3. A *strategy set* S includes one or more of these strategies and is denoted by $S = \{S1, S2, S3, S4\}$. These strategies differ in the factors they adopt

to make bid value decisions. Some are more aggressive while others consider future job demand. Again, none of the strategies are parameterized for the experiments. Bid values submitted by an agent must be positive real numbers and not exceed the available balance of an agent at the time of submission.

Strategy	**Bid Value Calculation**
S1:Greedy	Current balance.
S2:Jobs	Current balance divided by number of jobs in the queue.
S3:Values	Current balance multiplied by weighted USD job value.
S4:Prices	110% of moving average of past 5 winning prices.

Table 4.3: **Agent strategies**: overview.

S1:Greedy

The first strategy, S1:Greedy, is simple, yet aggressive. An agent simply bids all of its current balance b_i^t for the first job, without concern for future needs. Formally, agent i submits the following bid value $\hat{w}_i^1$ for the first job in its queue that is valid at time t:

$$\hat{w}_i^1 = b_i^t \tag{4.5}$$

S2:Jobs

The second strategy, S2:Jobs, is more conservative than S1:Greedy as the agent is more forward-looking. By spending only a portion of its balance on the first job, an agent hopes to reserve tokens for future jobs in the queue.

For the first job at time t, an agent submits a bid value equal to its balance divided by number of jobs in its queue k_i^t. Formally:

$$\hat{w}_i^1 = b_i^t / k_i^t \tag{4.6}$$

S3:Values

This strategy is similar to S2:Jobs in that both strategies consider the state of the job queues. For S3:Values, an agent submits a bid value $\hat{w}_i^t$ for the first jobs in the queue based on the weighted average of true values (in USD) of all jobs in the queue. Denote the true value of job j as w_i^j.

$$\hat{w}_i^1 = b_i^t \cdot w_i^1 / \sum_{j=1}^{k_i^t} w_i^j \tag{4.7}$$

Example: Agent i has two jobs $\{1, 2\}$ in the queue with values $\{\$50, \$100\}$, thus the first job has a weight of 1/3. Current balance is @90. Thus, i submits $\hat{w}_i^t$ = @30.

S4:Prices

Agents that play this strategy submit bids that are 10% above the *moving average* of the recent five average per-unit prices [7]. While simple, agents leverage public price information and employ a statistical model. Because the prices are influenced by all agents, this strategy explicitly takes into consideration the actions of everyone. This is in contrast from the other three strategies, in which each agent only considers its own bank balance and job needs.

[7]For a winning job with a price of 20 for 4 nodeslots, the per unit price is 20/4=5. The average per-unit price of time t is the average of all such per unit prices. For example, per unit prices of $\{2,4,6\}$ equals 4

Denote p^t as the average per-unit price for time t, the moving average per-unit price $\bar{p}$ of the past 5 periods, $\{t-1, t-2, ..., t-5\}$, is:

$$\bar{p} = \frac{\sum_{j=1}^{5} p^{t-j}}{5} \tag{4.8}$$

With $\bar{p}$, the agent calculates its bid value for the first job in the queue as follows:

$$\hat{w}_i^1 = \max(n_i s_i \cdot \bar{p} \cdot 110\%, b_i^t) \tag{4.9}$$

Multiplying $n_i s_i$ is necessary since $\bar{p}$ is only per-unit. I choose 110% arbitrarily for agents to bid slightly above the moving average. Finally, if the calculated bid value exceeds the agent's available balance, then the agent will submit its balance as bid value instead.

As an example, if the past five per unit prices were @$\{1,2,3,4,5\}$, then $\bar{p} = 3$. If agent i has $n_i s_i = 4$, then it submits a bid value @13.2 if it is less than the agent balance.

4.3.4 Mirage

As noted in the previous section, I do not study hoarding agents in this chapter. Nonetheless, the use of a savings tax in Mirage was intended to prevent agents from hoarding. To model such agents, a basic strategy would be for an agent to submit the minimum amount of bid value (e.g., @1) for jobs. A more sophisticated strategy would be for the agent to play S1:Greedy for jobs with the highest values, but to bid the minimum for all others.

There are distribution methods that can be considered for hoarding strategies. For example, a method that distributes more currency to agents with low running balance, as these are likely non-hoarding agents. Another possibility is to distribute based on cumulative bid values, since these would be very low for bidding agents. The S2:Stable strategy can po-

tentially have some effects as well if hoarding agents do occasionally bid very high values (i.e., resulting in high standard deviations).

4.4 Equilibrium Analysis

I model the virtual currency system with empirical game theory [48]. The game consists of a finite set of strategies and is symmetric. A game is symmetric if all agents have the same strategy set, and the payoff to playing a given strategy depends only on the strategies being played, not on who plays them [28] [8]. Furthermore, agents' actions on choosing strategies are non-deterministic.

A game in which agents' actions are non-deterministic is the mixed extension of the strategic game [76]. An agent is said to use a mixed strategy whenever he or she chooses to randomize over the set of available actions (of choosing which strategy to play). Formally, a mixed strategy game is a probability distribution that assigns to each available action a likelihood of being selected [87]. Essentially, before the game begins, each agent "rolls a dice," based on these probabilities, to decide which action to play. Contrast this with a deterministic agent, who has pre-determined a specific action to play before the game begins.

A symmetric mixed strategy Nash equilibrium (NE) is a strategy profile with the property that no single agent can, by deviating unilaterally to another strategy, produce an outcome that it finds strictly preferable. Every finite strategic game has a mixed strategy Nash equilibrium [88]. There are several ways to view NE. One of which is that of a steady state of a population of agents [76].

[8] An example of a symmetric game is the Prisoners' Dilemma.

My goal is to find NE for each monetary policy and workload combination. The tool I use for finding NE is *replicator dynamics*, which originates in evolutionary biology [75, 86, 95]. The particular set of procedures I use is based on work by Cheng et. al. [28], which iteratively adjust strategy populations by comparing expected agent payoffs with respect to the current mixture of strategies in the population.

I now describe the steps required for each experimental run, which involves a workload and a monetary policy instance. In Appendix B, I provide additional details.

1. **Build Payoff Matrix**: Given a set of agents each playing a specific strategy, what is the expected payoff for each agent? The goal is to explore the different combinations of agents and strategies and populate a payoff matrix for each combination (known as a pure strategy profile).

2. **Search for Equilibrium**: With the payoff matrix, the goal is to find a symmetric mixed strategy Nash Equilibrium can be found.

3. **Calculate Metrics**: Once an equilibrium is found, what are the metrics for this steady state? The goal is to collect different metrics for each experimental run, and then be able to use the metrics to compare different policies.

4.4.1 Build Payoff Matrix

Suppose there are n agents, each playing a strategy from the set $S = \{x, y, z\}$ [9]. A *pure strategy profile*, denoted by $p \in P$, captures the exact strategy chosen by each agent: $p = (n_x, n_y, n_z)$, where each element represents the number of agents that play a respective

[9] I use {x,y,z} for illustration purposes. In my experiments the strategy set includes {S1,S2,S3,S4} as discussed before

strategy. For example, if exactly 5 agents play x, 1 plays y and 4 play z, then $p = (5, 1, 4)$. Thus, the number of unique pure strategy profiles in P is based on the number of strategies and agents and is finite.

The *payoff* of a particular strategy profile p equals the *percentage* of true value captured by the agents (the USD-based true value of winning jobs divided by USD-based true value of all jobs for each agent, multiplied by 100%) and is denoted as $u(p)$. Thus, the highest payoff an agent can receive is 100%. I represent the payoff as $u(p) = (u_x, u_y, u_z)$. u_s represents the *average* payoffs for all the agents that play strategy s. For example, for strategy profile $p = (3, 2, 1)$ with 6 agents, u_x is the average payoff for the 3 agents that play the first strategy x. The payoff definition does not involve virtual currency, such as the balance of tokens an agent has. This is because the virtual currency is assumed to be closed and have no intrinsic value outside of the system.

To calculate the payoffs, I run the system 50 times for each of the strategy profiles to obtain average payoffs. Once all the payoffs are collected, the *payoff matrix* is populated with one entry for each pure strategy profile. Table 4.4 is an example for a payoff matrix for a 3-strategy set and 3 agents.

There are 3 rows and 3 columns in the table. Each row represents the number of agents that play the x strategy (between 0 and 3). Similarly, each column represents the number of agents that play the y strategy. Because we have a total of 3 agents, we can deduce the number of agents that play the z strategy as $|z| = 3 - |x| - |y|$. For example, for the entry $|x| = 2, |y| = 1$, the strategy profile is $s = (2, 1, 0)$ (i.e., no agent plays strategy z). The payoff in the table is $u(s) = (45, 15, -)$. This means that the *average payoff* is 45% and 15% for agents playing strategy x and y, respectively.

	$\|y\|$=**0**	$\|y\|$=**1**	$\|y\|$=**2**	$\|y\|$=**3**
$\|x\|$=**0**	(-,-,30)	(-,30,40)	(-,20,10)	(-,25,-)
$\|x\|$=**1**	(30,-,15)	(15,15,20)	(10,10,-)	(-,-,-)
$\|x\|$=**2**	(5,-,15)	(40,15,-)	(-,-,-)	(-,-,-)
$\|x\|$=**3**	(10,-,-)	(-,-,-)	(-,-,-)	(-,-,-)

Table 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 .

4.4.2 Search for Equilibrium

With the payoff matrix created for a specific monetary policy and workload, the next step is to search for a symmetric mixed strategy Nash Equilibrium (NE). The NE is a mixed strategy profile s' that represents the probabilities that each agent will use to select which pure strategy to play.

To find NE with replicator dynamics, I start with equal proportions of agents playing each strategy, e.g., 0.33, 0.33, 0.33, and estimate how these proportions change based on expected payoffs of playing each strategy. This process is iterated over time, until the process stops at a specific strategy profile where there is no "better move." Table 4.3 shows a sample graph where the populations of the mixed strategy profile shift over time in the equilibrium search process.

The calculation of each update in replicator dynamics involves finding out expected payoffs for each strategy and the probabilities that agents will play a certain pure strategy profile. Also, it is important to note that upon convergence the calculated equilibrium is

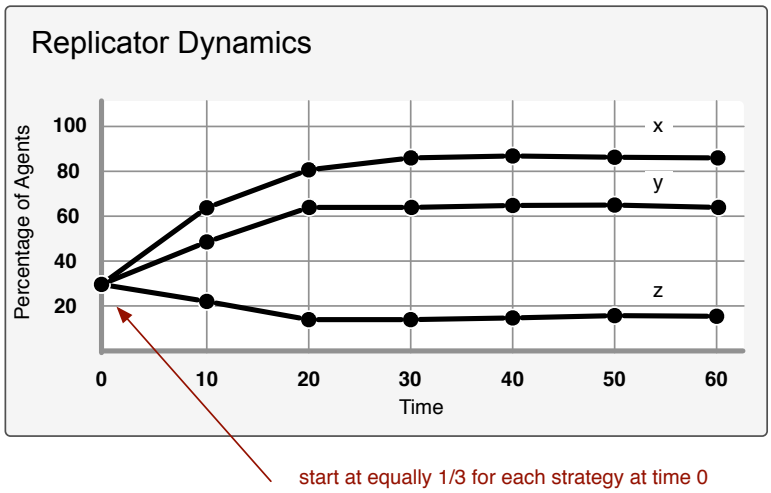

Figure 4.3: **Replicator dynamics** to finding symmetric mixed strategy Nash equilibrium.

verified as a true equilibrium. Specifically, the expected payoffs for those strategies with a non-zero proportion in the mixed strategy profile must all be the highest (compared to those with a zero proportion) and be identical. Finally, note that the above procedures as well as the varying populations during replicator dynamics reflect purely the steps it takes to compute the NE.

4.4.3 Calculate Metrics

Once a mixed strategy Nash equilibrium is computed, the resulting mixed strategy profile can be used to derive metrics. For measuring value, I use the *value efficiency* metric. It is the total true value captured by the system at equilibrium as a percentage of the offline optimal total true value. The latter quantity is obtained by running the same workload through an offline allocator. This allocator receives all the bids at the onset of the simulation and uses linear programming to find the optimal solution. It is expected that the system will not match the value captured by the offline allocator, since all decisions are made online.

To calculate the equilibrium value efficiency, I use the probabilities in the mixed strategy profile and the value efficiency of each pure strategy profile. Specifically, I use the former to derive new profile probabilities, which may exclude some of the pure strategy profiles. I then multiply the profile probabilities with the respective value efficiency for each included pure strategy to obtain a weighted average value efficiency for equilibrium.

The difference between value efficiency and system value (α) in Chapter 3 is as follows: α is the average true value in USD captured per unit of resource (a nodeslot), whereas value efficiency in this chapter focuses on the total true value captured for the whole system and is represented as a percentage by comparing the online allocator value to the value captured by the offline allocator. If the latter is fixed, then an increase in value efficiency should lead to an increase in α.

In the next four sections, I explore the effects of policies against different workloads and agent strategies. My approach is four-fold. First, I study some basic strategic interactions in this section. Second, I explore the effects of the distribution method dimension against different workloads in Section 4.6. Third, I look at the dimensions that affect money supply, the distribution interval and money supply dimensions, in Section 4.7. Fourth, in Section 4.8, I compare the value efficiency achieved by the each policy against every workload at equilibrium.

4.5 Basic Strategic Interactions

For strategic interactions, I begin with a simple strategy set: {S1:Greedy, S2:Jobs}. This set includes a total of 11 strategy profiles: {(10,0), (9,1), (8,2), ..., (0,10)}. For the first step of equilibrium analysis, I generate a payoff matrix in Figure 4.4. Each row represents a pure

strategy profile and its payoff. For example, the second row is the (9,1) pure strategy profile in which 9 agents play S1 strategy and 1 agent plays S2. "S1 V%" and "S2 V%" are the "payoffs," which are the average value captured as a percentage of all value demanded by an agent, for agents that play strategy S1 and S2 in such pure strategy profile, respectively. In other words, 47% is the average payoffs obtained from the payoffs of the 9 agents that play S1.

S1	S2	S1 V%	S2 V%
10	0	51	0
9	1	47	78
8	2	42	81
7	3	36	80
6	4	30	79
5	5	46	56
4	6	60	43
3	7	61	45
2	8	62	47
1	9	65	49
0	10	0	48

Figure 4.4: **Two-strategy payoff matrix for D1:Uniform.**

The payoff matrix reveals the varying payoffs of different pure strategy profiles. When all agents play the same strategy, as in (10,0), no agent can receive a clear advantage as they bid the same and receive the same amount of tokens via D1:Uniform. Thus, all agents capture similar values, which are about half of the total value.

The situation changes as agents switch strategies for adjacent profiles. For (9,1), the lone agent that now plays S2:Jobs captures a significant amount of payoff at 78%. Simultaneously, the remaining 9 agents see their payoffs drop. This trend continues through profile (6,4), with payoffs by S1:Greedy agents dropping to 30%. From (5,5) to (0,10), the situation reverses with S1:Greedy agents capturing higher payoffs and reaching a high of 65%.

S2:Jobs agents payoffs drop but but are still over 40%.

S1:Greedy suffers more than S2:Jobs in the majority profile situations (e.g., (8,2) vs. (2,8)). When there are more S1:Greedy than S2:Jobs, the displaced bids in Roller are probably from an S1:Greedy agent. Thus, S1:Greedy agents may be paying prices set by other losing S1:Greedy bids. This rapidly lowers their bank balance and thus their ability to compete for more jobs, resulting in lower payoffs. This also leads to the minority agents, S2:Jobs capturing relatively higher payoffs.

To further understand the strategy dynamics, I plot time-series patterns for one of the profiles: (4,6) in Figure 4.5. This gives a picture of agents' bid values and the flow of tokens over time. The top section of the graph consists of two lines plotting average bid values submitted by agents playing S1:Greedy and S2:Jobs. A third line traces average prices paid by all agents as a reference. The middle and bottom sections of the graph show the average ending balances of all agents and the system, respectively. The three graphs are inter-related as bid values, prices, and token distributions affect ending balances and vice versa.

The bid value patterns for the two strategies are opposite. For S1:Greedy, agents submit high values at time 0. Bid values gradually decrease over the next few periods, as S1:Greedy agents receive and pay for their allocations. S2:Jobs agents, on the other hand, submit lower bid values that increase gradually, as their jobs are consumed and the average price per job rises. This cycle restarts during every distribution interval, when new tokens are distributed to all agents.

For S1:Greedy, agents win early because of the aggressive style of submitting their full balance as bid values (e.g., the bid value in time 1 equals the ending balance at time 0).

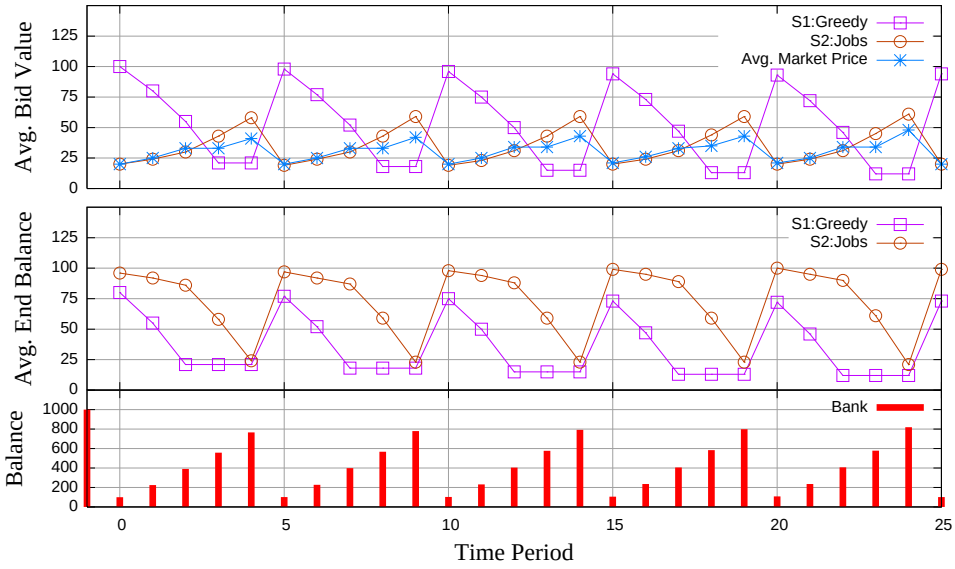

Figure 4.5: **Time-based patterns for profile (4,6) with D1:Uniform.** The top graph shows average bid values submitted by agents playing each strategy. An average market price line is added as reference. S1:Greedy agents submit decreasing bid values because they constantly spends high amount of currency. 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.

The prices they pay are set by the bid values of S2:Jobs agents. This only applies to a scheme such as Roller but not a first-price auction (where S1:Greedy will run out of tokens immediately). As a result, agents have decreasing balances, which lead to decreasing bid values.

Both bid values and balances decrease until S1:Greedy agents start losing to S2:Jobs. This happens at time 3. S2:Jobs agents have barely spent any of their balance up to this point because they have been losing to S1:Greedy. Their bid values are still higher than the nearly depleted balances of S1:Greedy. In addition, the number of jobs in the queue decreases as some jobs either have expired or have already won. As a result, S2:Jobs bid values become higher as time passes. Note that the price curve is below S2:Job bid values

as well, indicating that S2:Jobs are indeed winning.

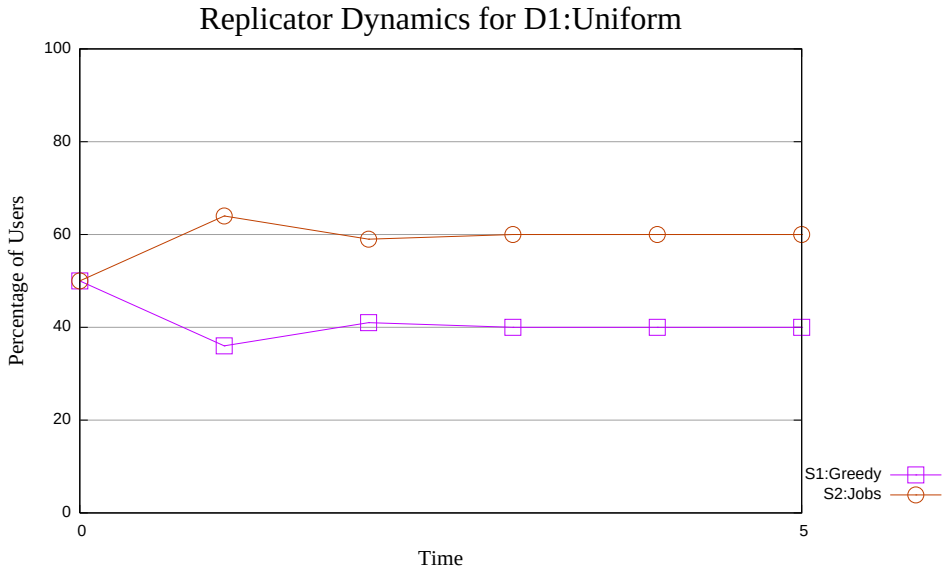

Figure 4.6: **Two-Strategy equilibrium for D1:Uniform.** Starting with equal percentage of agents playing each strategy, replicator dynamics quickly converge to a symmetric mixed strategy Nash equilibrium of (0.4, 0.6), in which each agent will play S1:Greedy with a 40% chance.

Finally, I use replicator dynamics to calculate equilibrium as shown in Figure 4.6. The mixed strategy Nash equilibrium is (0.4, 0.6), in which every agent will play S1:Greedy and S2:Jobs with a 40% and 60% chance, respectively.

4.5.1 Four Strategies

With some understanding of S1:Greedy and S2:Jobs strategies, I next add the remaining two strategies to the mix. Adding S3:Values and S4:Prices to the strategy set for 10 agents, the number of strategy profiles increases exponentially from 11 to 286.

I study the full strategy set with the D2:Stable distribution method. First, I select one of the profiles, (2,3,2,3), to observe general strategy behaviors. In Figure 4.7, I show time-

based patterns for the D2:Stable strategy. The ending balance of S4:Prices is significantly higher than the other agents balances. This is because D2:Stable rewards bid values with a low standard deviation and S4:Prices has the lowest standard deviation among the four strategies. For S1:Greedy, the story is the opposite. These agents' balances quickly depleted over time and the agents were penalized for aggressive bidding.

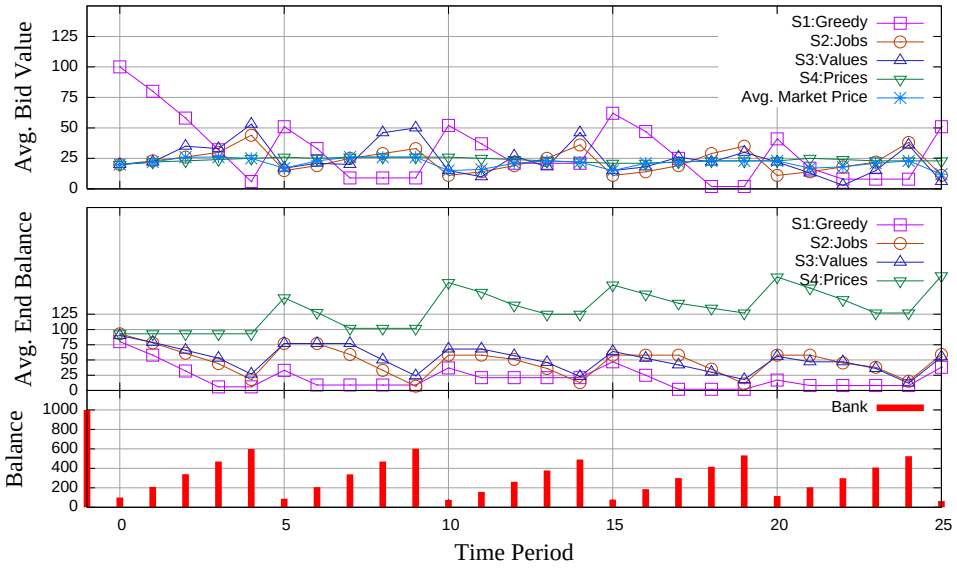

Figure 4.7: **Time-based patterns for profile (2,3,2,3) with D2:Stable.** The top graph shows average bid values submitted by agents playing each strategy. An average market price line is added as reference. S1:Greedy agents submit decreasing bid values because they constantly spends high amount of currency. 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.

Although S4:Prices agents have the most tokens, S3:Values do bid relatively more aggressively to win a good number of jobs. Also note that S3:Values and S2:Jobs receive a decent number of tokens.

Next, I run replicator dynamics to identify the equilibrium, as shown in Figure 4.8. The equilibrium mixed strategy profile is (0,0,0.3,0,7), suggesting that agents play S3:Values

and S4:Prices with a 30% and 70% chance, respectively, and do not play S1:Greedy and S2:Jobs at all. Essentially, the latter two strategies simply do not yield good payoffs for agents under the D2:Stable method. S1:Greedy is too aggressive to receive enough tokens, while S2:Jobs is too conservative to win over S3:Values and S4:Prices.

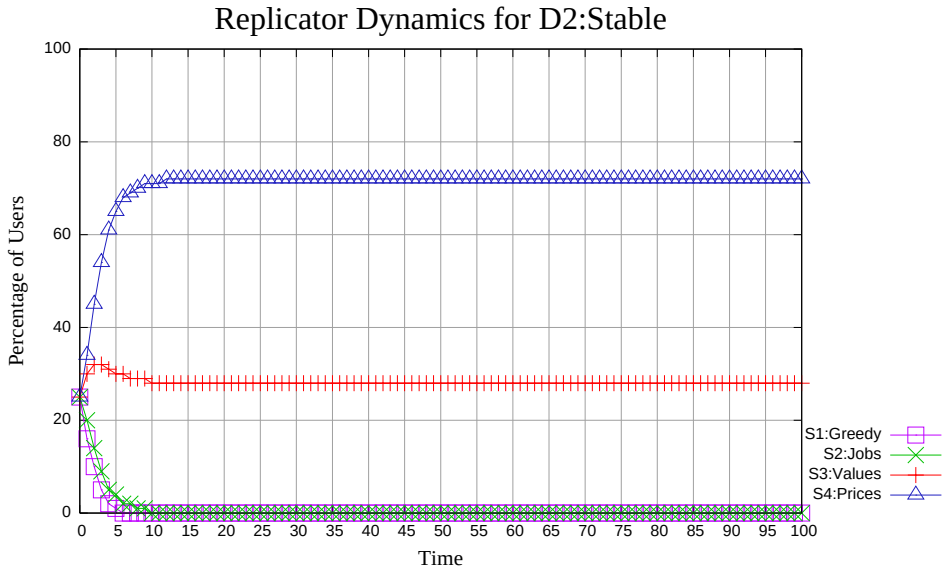

Figure 4.8: **Four-strategy equilibrium for D2:Stable.** Starting with equal percentage of agents playing each strategy, replicator dynamics quickly converge to a symmetric mixed strategy Nash equilibrium of (0, 0, 0.3, 0.7), in which agents will play S3:Values with a 30% chance and S4:Prices with a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

In Figure 4.9, I plot the time-based patterns for (0,0,3,7) as an example. S4:Prices tend to win early (bid values match closely to average prices) while S3:Values win late. This occurs because the bid values of early bids by S3:Values are too low—there are more jobs to consider earlier than later. Both strategies are rather stable in terms of bid values, resulting in a similar number of tokens received.

The results in this section show that a strategy that works in one situation may not perform at all in others. While S1:Greedy and S2:Jobs seem to co-exist when they are the only available strategies, they are completely driven out when S3:Values and S4:Prices strate-

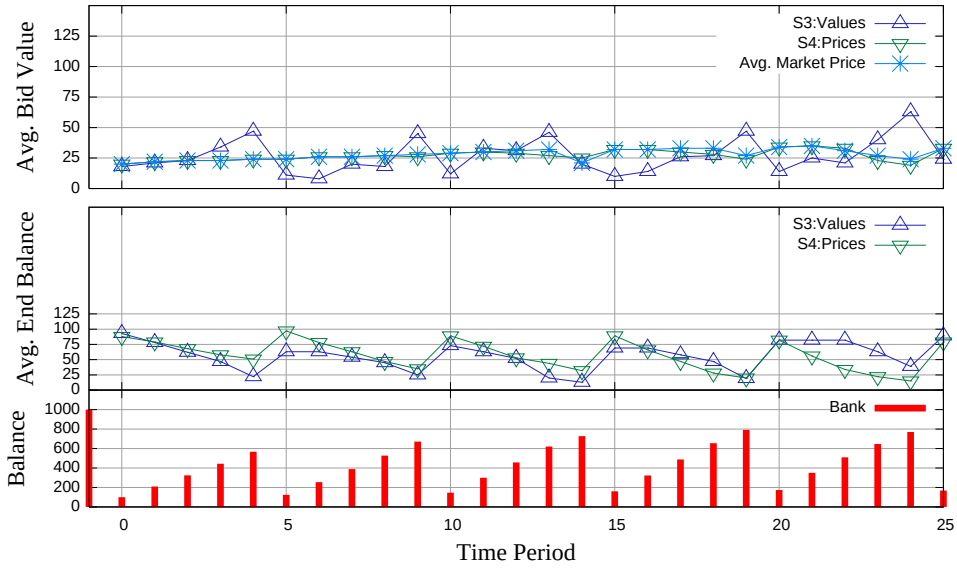

Figure 4.9: **Time-based patterns for profile (0,0,3,7) with D2:Stable.**

gies are added to the mix. It is thus very important to compare strategies across different workload settings when testing monetary policies.

4.6 Understanding the Effects of Distribution Methods

In this section, I evaluate the effects of distribution methods on individual agents. For each method, how much currency does each agent receive? How much resource and value are captured with different amounts of currency received?

My experiment is as follows. I run each distribution method against one pure strategy profile (2,3,2,3) [10] for every workload. This is done 50 times to collect averages. As there are 10 agents, the averages of the following are tracked for each agent in each workload: i)

[10]Using one specific profile is important to track individual agents properly. The profile (2,3,2,3) was chosen for its balanced representations.

the number of tokens received; ii) the amount of resource in terms of nodeslots won by the agent, as a percentage of all nodeslots won; iii) the amount of value captured by the agent, as a percentage of total value captured by all agents. F and M are fixed at 5 and @1,000, respectively. For each distribution method, I plot two graphs that show how the amounts of tokens received by agents affect (ii) and (iii) for all four workloads.

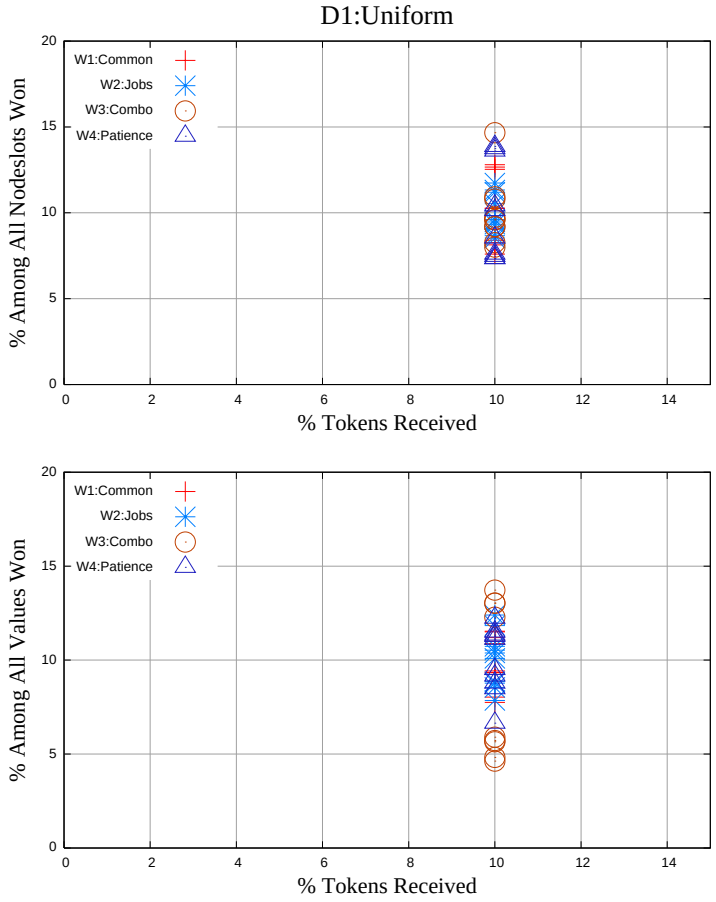

Figure 4.10: **D1:Uniform and pure strategy profile (2,3,2,3)**. Resource and value captured by the policy for each of the 10 agents, under each of the four workloads.

Figure 4.10 shows the results for the D1:Uniform method in the form of two graphs. For the top graph, the x-axis represents the percentage of tokens an agent receives and the y-axis represents the proportion of nodeslots won by the agent, as a percentage of all

nodes won by all agents. For the bottom graph, the y-axis represents the proportion of value captured by the agent. Results for all 4 workloads and a data point for each of the 10 agents are plotted on the graphs.

Every agent receives the same amount of tokens (at 10% each) per the D1:Uniform method, for all four workloads. The amount of nodeslots received by agents in all workloads varies between 7 to 15%, due to different strategies played.

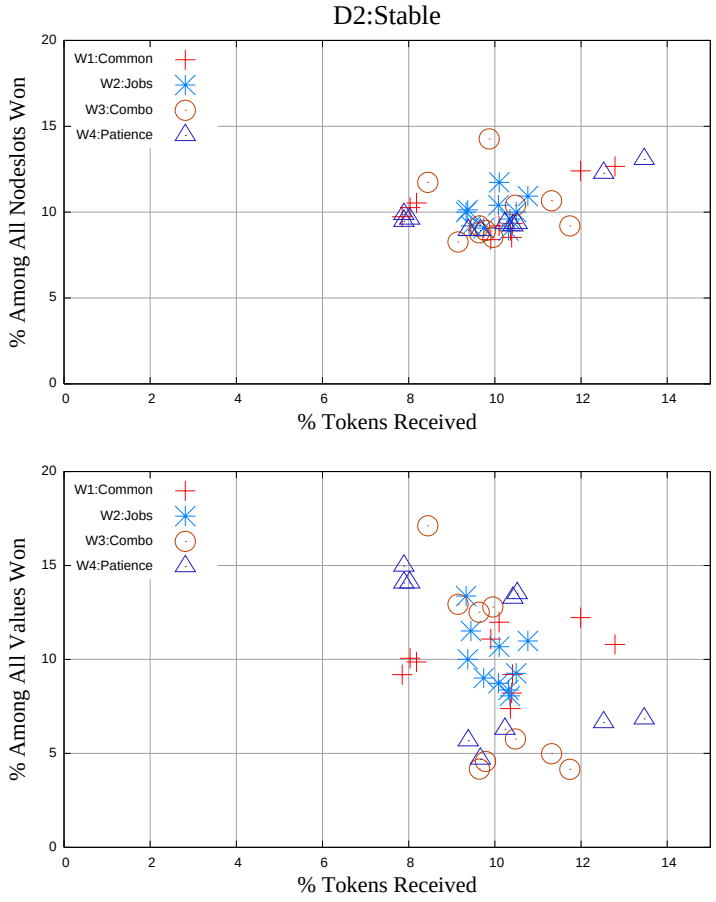

Figure 4.11: **D2:Stable and pure strategy profile (2,3,2,3)**. Resource and value captured by the policy for each of the 10 agents, under each of the four workloads.

Next, Figure 4.11 shows the results for the D2:Stable distribution method. Here, the 10 agents receive a wide range of tokens. However, there is no clear correlation between

the number of tokens received vs. the amount of resource and value captured. This is because D2:Stable is strategy-oriented—it distributes currency in a way that depends on agent strategies. For example, an S2:Jobs agent receives a good amount of tokens from using D2:Stable, but always bids only a fraction of the received tokens.

Figure 4.12 shows the results for the D3:Active distribution method. First, note that agents receive a wide range of tokens. This is due to the way that D3:Active distributes tokens. For W2:Jobs and W3:Combo, half of the agents have more jobs and thus receive a high number of tokens. Because jobs in W1:Common and W4:Patience share the same distribution, the agents receive a similar amount of tokens at about 10%. D3:Active shows that by giving agents more tokens in this particular way, they can receive proportionally more nodeslots and value.

Finally, the results for D4:Urgent (Figure 4.13) are similar to those of D3:Active, except the main workload of interest is W4:Patience. The other workloads have the same patience distribution for all jobs, thus agents receive a similar amount of tokens. W4:Patience agents, however, receive either a high or low number of tokens, depending on their type.

In summary, distribution methods that are workload dependent (i.e., D3:Active and D4:Urgent) distributes more tokens to agents that are either more active or less patient. These agents are able to leverage the extra tokens to capture more value and resources. A distribution method that is strategy-oriented (i.e., S2:Stable), on the other hand, does not necessarily give agents with extra tokens an edge to obtain more value or resource than agents with fewer tokens.

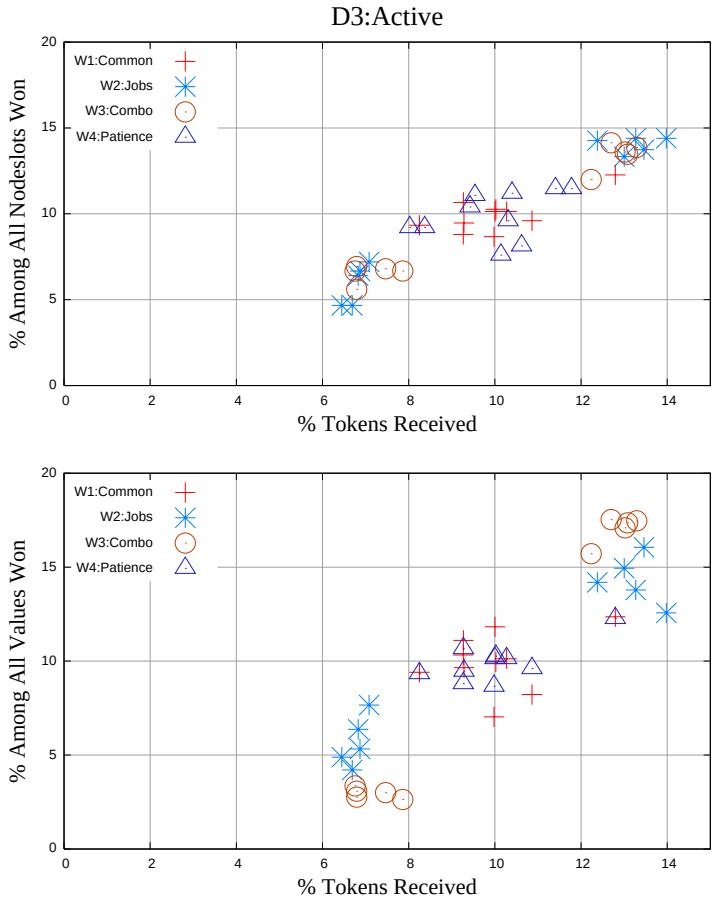

Figure 4.12: **D3:Active and pure strategy profile (2,3,2,3)**. Resource and value captured by the policy for each of the 10 agents, under each of the four workloads.

4.7 Understanding the Effects of Money Supply

In this section, I explore the two dimensions that affect money flow and money supply: Distribution Interval F and Money Supply M.

4.7.1 Distribution Interval

To see the effects of the distribution interval policy dimension on the mixed strategy Nash equilibrium and value efficiency, I start with the W1:Common workload and

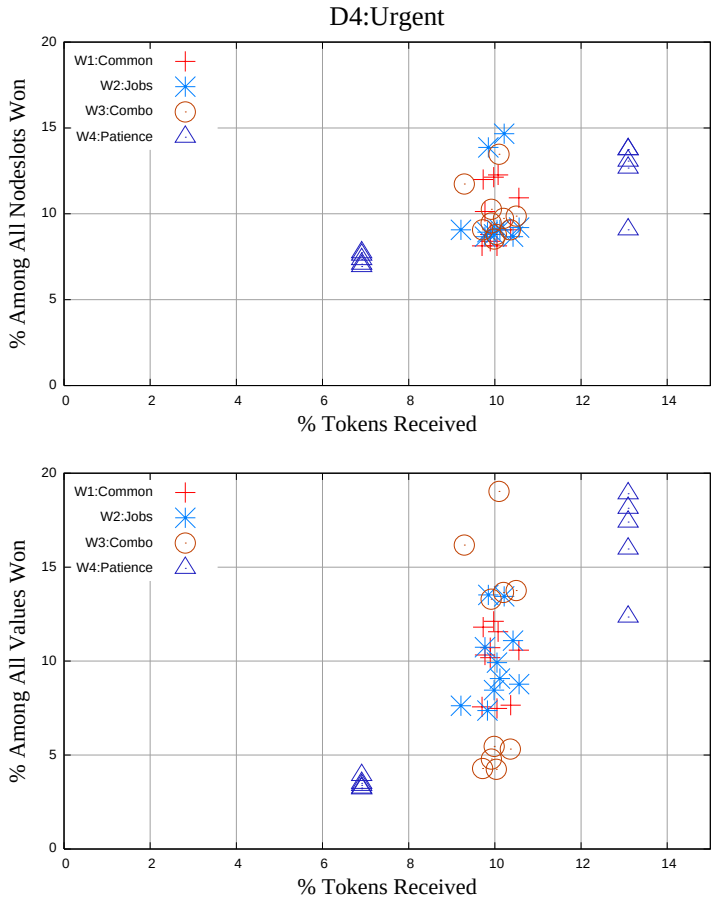

Figure 4.13: **D4:Urgent and pure strategy profile (2,3,2,3)**. Resource and value captured by the policy for each of the 10 agents, under each of the four workloads.

D1:Uniform distribution method. Money supply is fixed at @1,000. By varying distribution interval F from 1 to 25, I calculate the mixed strategy Nash equilibria as shown in Figure 4.14.

When F is at 1, agents receive tokens every time period. This favors S1:Greedy the most because it bids the highest, and the policy keeps providing it with more tokens every time period. This results in over 40% of S1:Greedy agent population in the mixed strategy Nash equilibrium. S2:Values and S3:Prices agents round out the populations.

However, as F increases, the proportion of S1:Greedy agents decreases rapidly. This

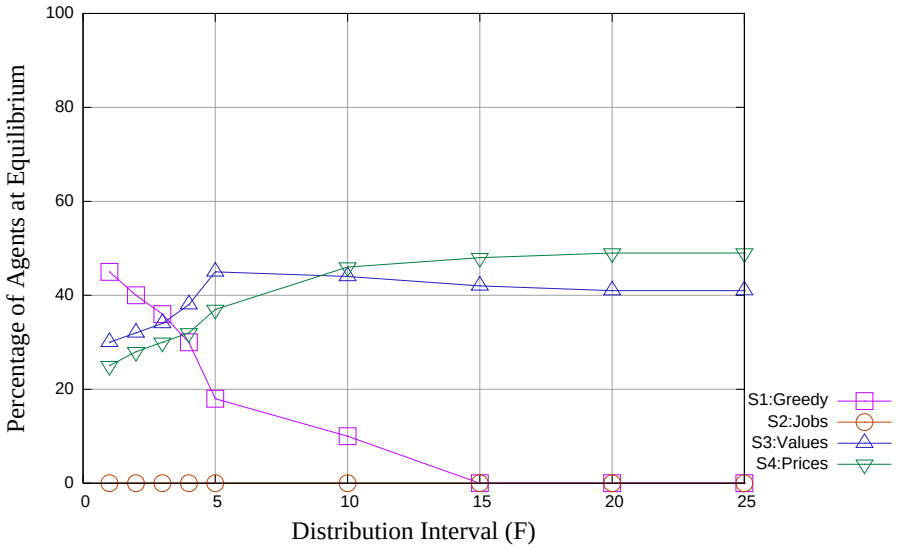

Figure 4.14: **W1:Common and D1:Uniform.** Distribution intervals and symmetric mixed strategy Nash equilibrium.

is because these agents no longer have the advantage when the interval increases, because they spend most of their tokens very quickly and in some cases go bankrupt. This zero-sum effect leads to a gradual increase in the proportions of S2 and S3 agents in the mixed strategy Nash equilibrium. For $F > 15$, the S1:Greedy strategy is completely driven out in the equilibria.

For value efficiency (Figure 4.15), it increases gradually as F increases from 1 to 5. As more S3:Values agents are represented at the mixed strategy Nash equilibrium, more high value jobs will be counted towards value efficiency. The maximum value efficiency is at 5, which matches the arrival rate of new jobs. For $F > 5$, the value efficiency starts to drop. This is because more and more agents will be out of tokens and can no longer submit bids, while a few have hardly any tokens left to compete meaningfully against agents that play other more conservative strategies. Thus, as the distribution interval increases, there are an increasing number of bids not submitted or lost, resulting in value not captured by

the system [11].

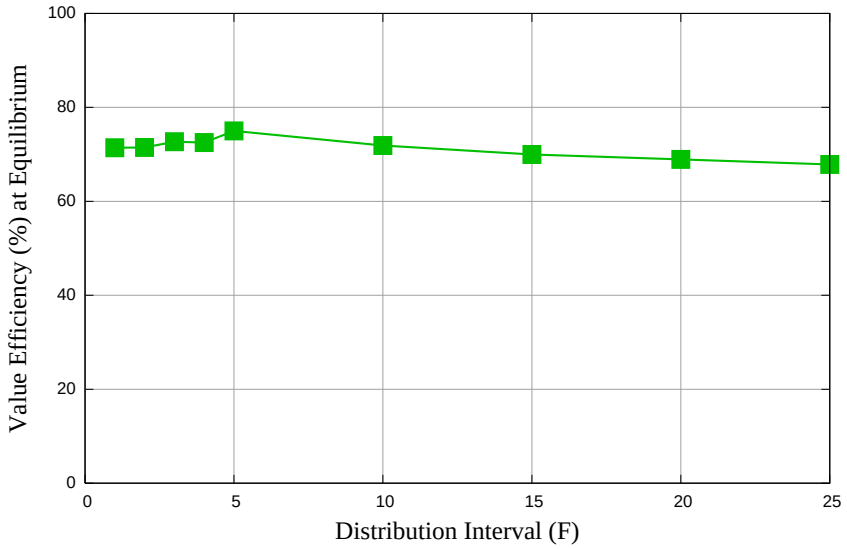

Figure 4.15: **W1:Common and D1:Uniform.** Distribution intervals and value efficiency.

Next, I repeat the experiment for W3:Combo and D3:Active workloads. Figures 4.16 and 4.17 show the results. For mixed strategy Nash equilibrium, notice that the only strategies represented are S3:Values and S4:Prices. When F is low, there are more agents that play S4:Prices. This is because when there are lots of tokens being distributed, S4:Prices agents submit high bid values since all agent submits high bid values, making prices stay high. However, as F increases, prices decrease as agents run out of tokens, making S4:Prices less competitive than S3:Values, which is a relatively more conservative bidding strategy. In terms of value efficiency, the high-level trends are similar to W1:Common/D1:Uniform—efficiency increases and peaks at $F = 5$. Of course, the level of value efficiencies of all F points are higher in this case, because D3:Active is able to capture many high value jobs

[11]There are however still agents, aside from those that play S1:Greedy strategy, that are able to win resources with their extremely low bank balances.

from the W3:Combo workload.

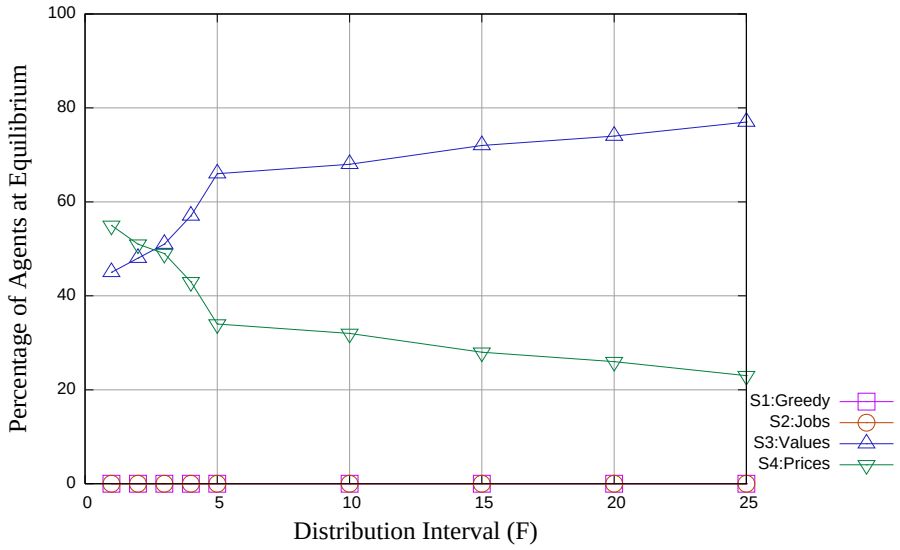

Figure 4.16: **W3:Common and D3:Active.** Distribution intervals and symmetric mixed strategy Nash equilibrium.

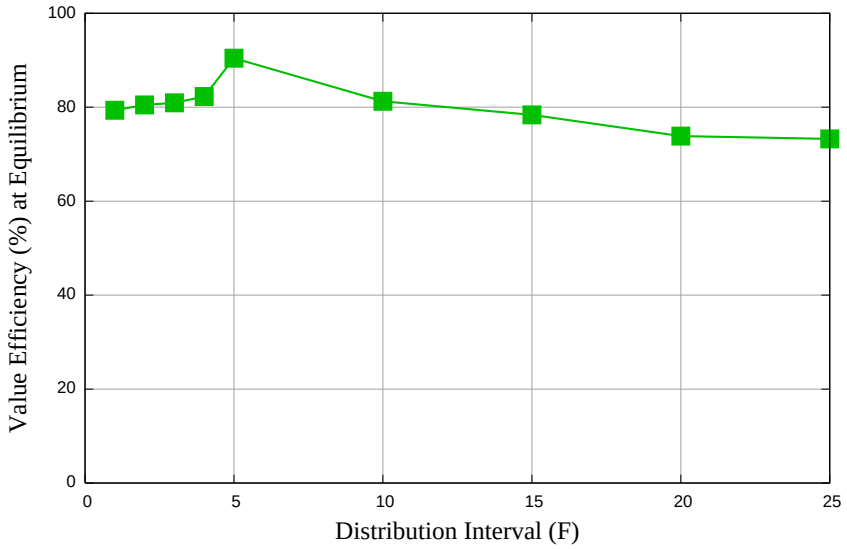

Figure 4.17: **W3:Combo and D3:Active.** Distribution intervals and value efficiency.

4.7.2 Money Supply

The level of money supply M has zero effect on the symmetric mixed strategy Nash equilibrium and metrics that depend on true values in USD. The main reason is because the "mixed currency basis" (see Section 4.3), in which the agents use USD for true values and tokens for bid values. Specifically, all strategies discussed produce bid values based on whatever amount of tokens each agent has. For example, if money supply is doubled, then all bid values generated by every strategy will be simply doubled. This results in the same outcomes for the system including equilibrium as well as metrics such as value efficiency.

The results would have been different for the "all virtual currency basis." Consider an agent with a job true value of @100. If the system distributes @50 to it, it can only bid up to @50 and likely will lose to other agents with lower true values. This results in lower total value captured (in tokens) by the system.

4.8 Understanding the Effects of Monetary Policy

In this last experimental section, I study how different monetary policies affect overall value captured in equilibrium. I approach it by comparing results at equilibrium for each workload against each of the four different monetary policies. Following the preceding analysis, each policy is set with money supply $M = @1,000$, distribution interval $F = 5$. As there are four workloads and four distribution methods, the comparison includes a total of sixteen sets of results. I first present the results specific to value efficiency, followed by results for strategies. These are captured in the mixed strategy Nash equilibrium.

WORKLOAD	EFFICIENCY	POLICY			
		D1:Uniform	*D2:Stable*	*D3:Active*	*D4:Urgent*
W1:Common	Maximum	86.71%	84.91%	86.4%	86.63%
	Equilibrium	**75.1%**	70.7%	74.14%	74.87%
	Minimum	57.2%	58.55%	57.88%	57.5%
W2:Jobs	Maximum	79.09%	78.82%	81.77%	78.58%
	Equilibrium	77.78%	68.99%	**81.77%**	76.35%
	Minimum	57.76%	58.08%	57.18%	57.76%
W3:Combo	Maximum	87.46%	87.04%	93.78%	87.96%
	Equilibrium	82.63%	80.63%	**90.44%**	82.48%
	Minimum	70.01%	68.02%	66.49%	64.75%
W4:Patience	Maximum	89.96%	91.06%	91.09%	95.91%
	Equilibrium	81.55%	79.91%	82.72%	**91.38%**
	Minimum	59.17%	67.33%	58.83%	65.00%

Table 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.

4.8.1 Effects on Value Efficiencies

To understand how policies affect overall value captured for a specific workload, I compare the value efficiency achieved at equilibrium for each monetary policy. I follow the equilibrium analysis steps to calculate the number. Additionally, I present the *maximum* and *minimum* efficiencies captured by pure strategy profiles in the payoff matrix. Because

the value efficiency at equilibrium is derived from the payoff matrix, these two efficiencies define the possible range for the value efficiency at equilibrium. A good policy for a workload should produce value efficiency towards the maximum.

Table 4.5 summarizes the results. Each row represents one workload and each column represents one policy's chosen distribution method. For example, the first cell is for W1:Common and D1:Uniform. Each cell has the three efficiencies, maximum, equilibrium, and minimum, arranged in descending order for comparison. For each workload (row), the highest value efficiency is marked in bold. In the following, I review the results from the perspectives of the workloads and policies.

Workload Perspective

First, there is no clear winning policy for the **W1:Common** workload. The value efficiency at equilibrium is basically a tie for D1:Uniform, D2:Active, and D4:Urgent at nearly 75%. The reason that no policy stands out, or captures efficiency closer to the maximum, is that *every job shares the same value distribution* of $[1:100]. A policy may distribute more currency to an agent, but the values of its jobs are similar to those of the other agents. In addition, jobs are not differentiated in terms of job arrival rate or patience, and so D3:Active and D4:Urgent distribute uniformly since the weight in both formulae is similar for all agents. Thus, the two policies behave quite similarly to D1:Uniform for W1:Common workload. This is indicated by the range of possible efficiencies, as the maximum and minimum entries are virtually identical for all three policies.

Second, for **W2:Jobs** workloads, the best value efficiency improves slightly to over 80% by D3:Active. For agents of "high jobs" type, they receive more tokens to support

bidding for resources. If they play strategies such as S3:Values, then those jobs with higher values can have a higher chance of winning. Even though the value distribution is identical across all agents, the extra currency provides a higher chance for some high value jobs to contribute to the captured value.

Third, a clear winning policy can finally be found in workload **W3:Combo**. This is the first workload with *distinct value distributions among different agents*. D3:Active captures the highest value because it seeks out the agents with the most number of jobs, which turn out to be the highest value jobs. Thus, even though values are private to the agent, D3:Active is an example showing that *a policy can capture high value if it targets the workload variables that are correlated to value*. This is further supported by the results of the **W4:Patience** workload. Job values in this workload correlate to patience, which is the deciding variable for the D4:Urgent policy.

Policy Perspective

From a policy perspective, the results show that **D1:Uniform** and **D2:Stable** are not the most effective for any of the workloads. They fail because they do not affect how the high value jobs are being allocated. Specifically, these policies fail to acknowledge the differences among agents and are not good policies for the workloads, especially W3:Combo and W4:Patience. Nonetheless, the results do not preclude the existence of *some* workload that can benefit from either of these policies.

D2:Stable is special because it is designed to support certain types of agent strategies more than the others. Specifically, its use of bid value standard deviation is targeted to counter bids with wild ranges, such as those from S1:Greedy agents. However, it turns

out that this creates limitations on the currency system. Those agents that receive the most currency, such as S2:Jobs, also tend to bid very low. This indirectly creates a hoarding effect as these agents continue to accumulate more currency than they can spend. This results in having a Money supply that can run low at times, resulting in lost value not captured from some jobs. Therefore, D2:Stable is an example that *policies that target agent strategies may not necessarily increase value*.

D3:Active and **D4:Urgent** are the only policies that are clear-cut winners for at least one workload. They capture 90.44% (standard error of 2.1%) and 91.38% (standard error of 1.97%) value efficiency, respectively, and beat the next best policy for the respective workload by 7 to 9%. The reason for their success is the ability to recognize and thus allocate more currency to agents with high value jobs, through some value-correlated variable (e.g., job rate and patience). Thus, the success of any policy hinges on whether such a correlation exists and whether or not the policy distributes with respect to the correlation. Of course, these policies still do not always succeed in delivering the highest value, as the results are workload driven. For example, D3:Active does not work well on D4:Urgent (shorter patience jobs have higher values), since the policy targets a variable (number of jobs) that in this case does not correlate in value.

Finally, the value efficiencies achieved by some of the above policies are approaching maximum efficiency. This indicates that the policies are highly effective for the types of workloads and agent strategies involved.

WORKLOAD	POLICY			
	D1:Uniform	*D2:Stable*	*D3:Active*	*D4:Urgent*
W1:Common	**(0.18,0,0.45,0.37)**	(0,0,0.29,0.71)	(0.2,0,0.43,0.37)	(0.17,0,0.43,0.4)
W2:Jobs	(0.06,0,0.94,0)	(0,0,0.31,0.69)	**(0,0,1,0)**	(0.11,0,0.89,0)
W3:Combo	(0.27,0,0.4,0.33)	(0,0,0.28,0.72)	**(0,0,0.63,0.37)**	(0,0,0.59,0.41)
W4:Patience	(0.28,0,0.44,0.28)	(0,0,0.3,0.70)	(0,0,0.58,0.42)	**(0,0,0.53,0.47)**

Table 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.

4.8.2 Effects on Strategies

Lastly, I look at the symmetric mixed strategy profiles in Nash equilibrium for each of the workload and policy combinations from the same data set as before. Table 4.6 shows the 16 different profiles, each represented by a cell that corresponds to the same combination in Table 4.5.

If a symmetric mixed strategy profile consists of non-zero proportions for all pure strategies, then agents play all of these strategies at equilibrium based on the proportions. However, if a proportion is zero, then the respective pure strategy s will not be played by any agent at equilibrium. This further implies that any pure strategy profile that includes s in the payoff matrix will be driven out because of the selected policy.

For example, the **S2:Jobs** strategy is zero for every equilibrium. This means that pure strategy profiles with non-zero number of S2 agents, such as (0,10,0,0), (1,2,3,4), and (3,5,1,1), are driven out by all four policies for every workload. This is due to the fact that S2 is too conservative for any agent to successfully compete for jobs against the other

strategies.

In comparison, **S1:Greedy** strategy performs slightly better than S2:Jobs. Its aggressive bidding strategy enables an agent to win *some* jobs, especially during distribution intervals. Nonetheless, its presence is limited among the different mixed strategy equilibria, in which the proportions of S1:Greedy agents are typically either 0% or around 20%.

The most well-represented strategies are **S3:Values** and **S4:Prices**. S3:Values in particular is included in all 16 profiles, as it is the only strategy that takes agent values into account. As a result, the expected payoffs for this strategy are generally higher than those of the other strategies. Higher expected payoffs attract more agents to adapt the strategy as the system iterates over time towards a steady state.

4.8.3 Value Effects

In Table 4.7, I plot the corresponding pure strategy profile for the "maximum" value efficiency entries in Table 4.5. For example, (0,0,10,0) pure strategy profile is the best for W1:Common and D1:Uniform, in terms of capturing the most value. This profile is one in which every agent plays S3:Values only. In fact, most of these maximum profiles are weighted heavily or exclusively with S3:Values. The reason is that this is the only strategy that correlates to an agent's true value. Thus, high value jobs are represented by high value bids, contributing to the value efficiency. Therefore, one can see the goal of monetary policy as one of promoting the "value" strategy in equilibrium.

WORKLOAD	POLICY			
	D1:Uniform	*D2:Stable*	*D3:Active*	*D4:Urgent*
W1:Common	**(0,0,10,0)**	(0,0,10,0)	(0,0,10,0)	(0,0,10,0)
W2:Jobs	(0,0,9,1)	(0,0,10,0)	**(0,0,10,0)**	(0,0,10,0)
W3:Combo	(0,0,8,2)	(0,0,9,1)	**(0,1,8,1)**	(0,0,9,1)
W4:Patience	(0,0,10,0)	(1,0,8,1)	(0,0,9,1)	**(0,0,10,0)**

Table 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.

4.9 Summary

In this chapter, I explored monetary policies. First, I provided a design space for creating monetary policies by varying a set of policy dimensions. Second, I provided a model for agents in terms of workloads and bidding strategies. Third, using the replicator dynamics technique, I showed how to calculate a symmetric mixed strategy Nash equilibrium profile, and value efficiency at equilibrium for different policy and workload settings.

The experiments showed that agents, in general, receive resources and capture value proportional to how much currency they receive. For workloads with jobs that have generally higher true values, policies that distribute currency based on variables that correlate to these true values can improve total value captured in the system. These policies also achieve high value efficiencies, with some approaching the value captured by an offline allocator as well as by the maximum pure strategy profile efficiency. In general, policies that award more currency to more active agents tend to do well. On the other hand, no policy showed the ability to improve total value for workloads with common value distributions.

In addition, absolute money supply level has no effect when agents value in USD but bid in the virtual currency. However, the periods during which currency is distributed matter significantly.

With the mixed currency basis, each agent has some set amount of virtual currency at their disposal at any point in time. As a result, agents have budget constraints. This breaks the no budget constraints assumption in Roller. However, Roller still contributes as the underlying mechanism with its strategyproof properties. First of all, mitigating incentives to mis-report nodeslots and departure are good for the system irrespective of virtual currency and budget constraints. In terms of valuation, once an agent has decided that its true value for some resources is some number of virtual currency (e.g., @5), then with strategyproofness of Roller the agent can simply bid @5 and not other values. This provides stability to the system.

The work in this chapter only covers the surface of understanding monetary policies. The future challenge hinges on designing new types of policies that can capture value for a wide variety of agent workloads and strategies. This requires more deployments of real market-based systems in order to learn about different classes of workloads and strategies. Further, it would be more realistic to use policies that are less static but more adaptive to agent workloads during a run. Finally, studying the ability of agents to value resources in virtual currency is critical and should be an ongoing activity.

Chapter 5

Related Work

5.1 Market-based Systems

For years, there has been a wide range of research on using market-based methods for resource allocation in systems. A wide variety of market-based methods are discussed in surveys [89, 33, 23]. Others have advocated use of markets compared with traditional resource allocation [90, 53]. The earliest work known is perhaps Harvard's PDP-1 [94], a futures market for CPU time. Subsequent work has been applied to a broad range of distributed systems including clusters [100, 29], data centers [27], computational Grids [104, 54], parallel computers [92], and Internet computing systems [60]. Many of these, such as Mirage, use auctions to allocate resources. However, none addressed nor provided data on strategic behaviors. A few classic systems are discussed further as follows.

Spawn [100] is a system for allocating dedicated CPU time for reservations. Workloads are assumed to be tree-based, concurrent applications. These applications have funding rates associated with different nodes of the tree which are used to purchase CPU resources

for tasks associated with leaves beneath those nodes. CPU resources are purchased by participating in Vickrey auctions for dedicated slices of CPU time which are held independently by each node. There was no payment scheme that tied the auctions together. Another CPU time work is Stoica [92]. It uses a first price auction to allocate dedicated CPU time for parallel jobs. Jobs are assigned saving accounts for purchasing CPU resources. A centralized auction allocates CPU time on all processors.

In POPCORN [84], applications are decomposed into small Java programs that can then be run on servers in a distributed system. Buyers submit requests along with their values to be matched with sellers via a centralized market. Three mechanisms in POPCORN were tested, including a repeated Vickrey auction and two Double Auction variations. The bidding language in POPCORN was not as expressive as that in Mirage. Mariposa [93] is another system for distributed computing. It specifically enables a bidding platform for self-interested servers to provide query optimizations.

Nimrod/G [25] is a system that allows agents to specify the types of resources needed for a particular price. The diversity of resources is much richer than Mirage and the process of acquiring resources was very complex, involving a chain of resource discovery and negotiations. The complex nature of Grid resources is well-documented, as other work such as MACE [23] begins exploring combinatorial exchange frameworks but did not have empirical data. More traditional grid resource problems are covered in systems such as Condor[61, 82] and G-Commerce [104].

Bellagio [19] is related to Mirage and targeted the PlanetLab [80] allocations. It involves more extensive resource discovery services and focuses more on the servers, which are contributed by the community as opposed to one organization in Mirage. SHARP [41]

and Tycoon [54] are among other systems motivated by the dynamic PlanetLab problem. Finally, there is work that focuses on other emerging aspects of distributed systems. For example, Muse [27] is designed for Internet hosting centers and focuses on energy being the main resources.

5.2 Online Mechanism Design

Mechanism design originates from traditional economics. Seminal work informs the foundations of GVA auctions [98, 32, 42, 96]. It was later proposed for solving systems problems [97]. Nisan and Ronan [73] proposed the study of algorithmic mechanism design, the search of computationally tractable mechanisms that have desirable properties such as strategyproofness, within the computer science community. Feigenbaum et. al. [37] extended the research agenda with a distributed direction, in which the mechanisms are implemented across a network. There were economists that identified issues in traditional mechanism design that are relevant for computer science [85].

In fact, there is a wide range of work on computational tractability that addresses some of these issues [74, 106, 78]. This is especially important for multi-unit and combinatorial settings [51, 35, 67, 20, 83], with the most computational-intensive problems evolve around combinatorial exchanges [62]. Aside from computation, there are other properties of mechanism design that are restrictive in the real-world, such as the budget constraint condition [22, 36].

Online mechanism design [77] extends mechanism design with dynamically arriving and departing agents and resources, applicable to systems such as web access [40] and advertising [34]. The earliest work was proposed by Lavi and Nisan [56]. Starting with a

very simple, single-unit model, the work has gradually grown to include characterization of truthful online mechanisms with expiring items [43, 44]. Another approach to online mechanisms is the model-based framework [79, 26].

5.3 Virtual Currency

While many market-based systems have researched different resource allocation methods, many of these systems have mainly assumed the use of real currency (USD). For the others that assume use of virtual currency, few have discussed in depth the supporting infrastructure or policies. Interestingly, the use of virtual currency is not a novel concept. It was adopted by market-based systems as early as PDP-1 [94]. Other researchers have created currencies for distributed systems such as network testbeds [41], peer-to-peer systems [99], and sensor networks [30]. There are also a few currency-like works that are relevant. One prime example is Lottery Scheduling [101].

Virtual currency has been more popular in fields outside of market-based systems. Second Life [12], an online virtual world where agents can create avatars, generates significant revenue for its Linden Dollars. Users pay Second Life US dollars to exchange for Linden Dollars in order to buy virtual land and to accessorize avatars. Online games such as World of Warcraft [16] and social networks such as Facebook [5] also sell virtual currency for real USD. Because it is very easy for a company such as Second Life to raise prices on virtual goods and to increase money supply (in order to sell for more USD), these systems often suffer hyperinflation and other economic issues [18]. Agents typically are prohibited from exchanging virtual currencies among themselves, though there have been 3rd party companies that enable such exchanges.

In lottery scheduling [101], agents are issued tickets, an alternative form of virtual currency abstraction. The scheduler holds lotteries periodically and selects a winning agent for the current resources. The more tickets an agent owns, the more chances he can win. Tickets are thus similar to the shares in Mirage, as both yield proportional sharing effects. However, agents have few options to acquire more resources. The only influence they can have is to attempt to acquire more tickets, the process of which is not specified in the paper. In addition, agents cannot express when they need resources more than other times. Specifically, agent values are not expressed or captured in a lottery scheduler. Thus, low-value jobs could easily win lotteries even during resource contention, hurting those who have more important needs.

SHARP [41] is a distributed infrastructure supporting the "tickets" abstraction. It supports a wide variety of security and claims protocols. However, SHARP does not address policies specifically. Others have built on top of SHARP to take advantage of its infrastructure. For example, Irwin et. al. [46] introduces a self-recharging virtual currency in their Cereus system. Specifically, spent credits recharge back to an agent after a fixed interval, known as the recharge time, from the time the agent submits a bid. The common currency, called credits, enables agents to bid for resources. Agents do not earn credits, but are assigned budgets of credits that can be spent. Periodically, the credits are automatically replenished, returning an agent to its original budget. This differs from my design as Cereus does not recycle credits among agents. Cereus is inspired by the "yen" currency used in PDP-1 [94], which has a different recharging rule.

KARMA [99] is designed for peer-to-peer resource exchanges (e.g., file sharing). The system includes a set of dynamic nodes, each of which was contributed by an agent and

contains files of interest to others. Agents themselves are also the consumers. The unit of virtual currency is the *karma*, a number that captures the amount of resources an agent has contributed and consumed. A set of nodes called the bank-sets keep track of the various karma numbers. When an agent contributes resources, his karma increases. Similarly, his karma decreases when he consumes resources of others agents. No agent is allowed to consume resources with negative karma, thus forcing everyone to contribute regularly. Because agents and nodes come and go, the total number of karma in the system fluctuates and requires constant rebalancing.

Kash et. al. [50] explored a heterogeneous population of agents in a closed currency setting. To receive service from others, agents must earn scrips by providing services to others. Each agent plays a threshold strategy in equilibrium: below the threshold, an agent is motivated to volunteer to earn more scrips; above the threshold, the agent will not volunteer. The paper characterizes the distribution of money in equilibrium, as a function of the fraction of agents of diverse types. The analysis demonstrates the possibility of a monetary crash, in which too much monetary supply leads to no agent wanting to work.

Chapter 6

Conclusion

6.1 Summary

In this thesis, I addressed three critical areas for resource allocation in distributed systems. First, I presented the design and deployment of Mirage, a market-based resource allocator for use by real agents. The resulting empirical data support using agent value as a first-order decision variable, especially for resource contention. With expressive bids, the system could offload decision making by having agents request resources across space and time. In addition, agents exhibited several types of strategic behaviors, all of which justified the need for designing auctions that mitigate such behaviors.

Second, I introduced online mechanism design for systems such as Mirage. Roller is designed specifically to embrace selfish agents and has an emphasis on a responsive, on-demand experience via the use of computationally tractable algorithms. It allocates combinatorial resources and is strategyproof with respect to value, space, and different aspects of allocation timing depending on its configuration. This is accomplished through the use

of carefully designed allocation and payment rules. I examined experimentally different parameterizations to identify ways to balance system value with responsiveness as well as to manage supply level. Empirical results demonstrate good value capture when compared with an offline optimal allocator.

Lastly, I analyzed monetary policy of virtual currency through a three-component framework: a design space for policies that consists of a set of dimensions; a model for agents in terms of workloads and bidding strategies; and a way to identify equilibria for different policy and workload settings using replicator dynamics. This framework enables comparisons among different policies and provides a first step for designing policies to address agent behaviors. For example, distribution methods that prefer stability mitigate aggressive bidding, while those that award active usage promote value-based bidding, and higher allocative efficiency, in equilibrium.

My work in this thesis provides a small step towards the full realization of market-based systems in the real-world. To design systems that provide a strategyproof experience and configurable tools for managing policies is no simple task. There are many future tasks ahead that require serious attention. I shall conclude this thesis with a few of these tasks.

6.2 Future Work

Open Empirical Data: The data collected from Mirage is valuable for further understanding the benefits and mechanics of market-based systems. However, this represents a very small sample. The community must strive for creating a rich set of data and a rich knowledge base for systems designers. A database that is open, updatable, and accessible by any researchers, can help scale analysis and enable new designs. One approach is to

gather sets of empirical data for major types of distributed systems. For example, how do agents behave in a global file backup system where identities are anonymous compared to a corporate computation system in which there is a hierarchy? With enough data and knowledge in the long-run, the market-based systems community can look to create design patterns to benefit future systems.

Online Mechanism Extensions: As Roller focuses on strategyproofness in a responsive environment with agents arriving and departing, it sacrifices high system value. Further exploration of how to capture more value by fine tuning the allocation rule is important. Other approaches that are interesting to explore include alternative ways to sell resources. For example, could "buy-it-now" pricing as seen on eBay [4] possibly benefit both agents of low patience and the system? In addition, it will be important to relax the single-minded assumption; e.g., to allow an agent to represent a group of collaborating researchers. Designing new online mechanisms that address advanced scenarios such as these are important for real-world deployments.

Adaptive Policies: As new types of workloads and strategies are explored and studied, new types of policies must be designed. Ultimately, it is important for systems to consistently identify different workload patterns. This helps administrators to select policies effectively. In the long-run, the goal should be to automate policy management with learning-based services. These services can analyze workload changes dynamically and adjust policies on the fly. Without such smart services, policy management can be too manual and non-scalable as workloads become more complex. Market-based systems can be enhanced with good automatic policy management to go with the simple, strategyproof experience provided by online mechanisms.

Open Mechanisms: In this thesis, each system is administrated by a single domain. The systems are thus "closed" in the sense that each has its own set of agents, allocation mechanism, and virtual currency. Agents of one system cannot access resources of another system and vice versa. While this remains practical today, more and more systems are looking to join forces for common benefits (e.g., cope with load spikes, cache data in Asia from the East Coast). Some of these include PlanetLab [11] and ATLAS [3]. In such open environments, however, strategyproofness cannot be maintained without new types of infrastructure support. Specifically, every domain must publish its own language, preference, and statistics to allow agents of all domains to observe whether it is strategyproof [70]. This verifiable infrastructure will be key to scale the simple experience provided by strategyproof mechanism to a global scale [49].

Exchangeable Virtual Currencies: Virtual currencies must be extended from addressing policies of one local domain to those of federated systems. How can agents acquire resources with local virtual currencies across domains? One solution is to replace all with a super-currency that every domain and agent will adopt. It is unclear, however, how local domain policies can be maintained. Furthermore, having a central body to replace the administrative bodies of every local domain seems unrealistic. An alternative is to consider keeping one virtual currency per system and then find a way for them to be exchanged. An agent in domain A can use resources in domain B if the two domains are federated and agree on how exchange rates are established. Defining a method for establishing these agreements and managing exchanges remains an important challenge [24].

Appendix A

Roller Proofs

A.1 Introduction

In this section, I provide proofs and discussions for several strategyproof properties claimed in the Roller mechanism. A mechanism is strategyproof if truth telling by an agent is a dominant strategy and maximizes its expected payoff, regardless of what the other agents do. Because the bidding language of Roller includes several attributes, strategyproofness must be shown for all attributes. The goal of Roller is to elicit truthful reports of these attributes from agents, including value, resource size, and allocation timing.

The rest of the section is as follows:

- In Section A.2, I provide a short review of the allocation and payment rules.
- In Section A.3, I establish the conditions required for any mechanism to be considered strategyproof.
- In Section A.4, I prove that Roller is strategyproof using the standard allocation rule,

with the assumption that agents do not mis-report departures.

- In Section A.5, I discuss the late allocation rule that address tradeoffs in regard to mis-reports of allocation timing.

A.2 Allocation and Payment Rule

An agent submits bid b_i to Roller, with $b_i = (w_i, n_i, s_i, a_i, d_i)$. w_i is the bid value for getting $n_i s_i$ nodeslots (size), and the allocation of nodeslots must start between arrival a_i and departure d_i.

The basic allocation rule works as follows: In period t, of all the bids not yet allocated, sort the bids in decreasing order of $w_i/n_i s_i$ (breaking ties in favor of earlier arrivals). Working from the top of the list of bids and reviewing rolling window nodeslots starting from the "leftmost" slots, if the nodeslots requested by a bid can be fulfilled by the current rolling window then make an allocation.

The payment rule works as follows: compute the unit price to bidder i as the minimal value v_i' that the bidder could have stated and still received an allocation in some period of the auction (with nothing else about its bid changed).

A ρ-late allocation rule is also defined, which extends the basic allocation rule to sometimes favor late allocations for a bidder, with late allocations preferred for a bid with uniform probability $\rho \in (0, 1)$. Any agent that over-report its departure risks getting a late allocation that results in zero value capture.

The basic allocation rule tries to allocate a bid as close to the "left" side (earlier times) of the rolling window as possible. The ρ-late allocation rule does the opposite and, with

probability ρ, tries to allocate a bid as close to the "right" side of the rolling window as possible.

A.3 Establishing Strategyproofness

The proof method I use is the monotonicity method [77].

Definition 1 (Type Domination)**.** *We say type* $\theta_i = (w_i, n_i, s_i, a_i, d_i)$ *dominates type* $\theta_j = (w_j, n_j, s_j, a_j, d_j)$*, denoted by* $\theta_i \succ \theta_j$*, if* $w_i > w_j$*,* $n_i \leq n_j$*,* $s_i \leq s_j$*,* $a_i \leq a_j$*, and* $d_i \geq d_j$*. We say* θ_i *is of stronger type whereas* θ_j *is of weaker type.*

In words, a strong type θ_i has higher value (w_i), fewer or equal nodeslots (n_i, s_i), and higher or equal patience (early arrival a_i and/or later departure d_i) than a weaker type θ_j. Holding everything else the same (e.g., $n_i = n_j, ..., d_i = d_j$), a bid with higher value is of stronger type than one with lower value (i.e., willing to pay more for the same resources). Similarly, a bid with the same value and patience but seeking fewer nodeslots is of stronger type (i.e., paying the same for fewer resources). Finally, a bid with the same value and nodeslot size but higher patience is of stronger type (more patient/flexible in terms of allocation timing and hence the allocation decision than the weaker type).

Let $f(\theta_i, \theta_{-i}) \in \{0, 1\}$ denotes whether or not agent i is allocated by allocation rule f, where $\theta_{-i} = (\theta_1, ..., \theta_{i-1}, \theta_{i+1}, ..., \theta_m)$ and m is the number of agents.

Definition 2 (Monotonicity)**.** *An allocation rule* f *is* monotonic *if for every agent* i*, every* $\theta_i \succ \theta_i'$*, and every* θ_{-i}*,* $f(\theta_i, \theta_{-i}) \geq f(\theta_i', \theta_{-i})$*.*

In words, if a weaker type is allocated, a stronger type will also get allocated (e.g., $f(\theta_j) = 1 \Rightarrow f(\theta_i) = 1$ if $\theta_i \succ \theta_j$). If a weaker type is not allocated, a stronger type may

or may not be allocated.

Assume that agents do not mis-report early arrival or late departure, the strategyproof condition is established as follows:

Theorem 1. *[Strategyproof Condition] Let f be an allocation rule of a mechanism. There is a payment rule p such that the mechanism (f, p) is strategyproof if and only if allocation rule f is monotonic.*

This theorem is provided by Hajiaghayi et. al. (Theorem 6 in [43]). Note that the payment rule specified in that paper is equivalent to the following representation of the Roller payment rule:

$$p_i(\theta) = \min\{\hat{w}_i : f_i((\hat{w}_i, n_i, s_i, a_i, d_i), \theta_{-i}) = 1\}, \text{if } f_i(\theta) = 1, \text{and } p_i(\theta) = 0 \text{ otherwise.} \tag{A.1}$$

The reason Roller uses such a payment rule is to mitigate strategic behaviors that span *across* auctions. For example, if the first auction has a single bid of @10 and the following auction also has a single bid of @5, then an agent with a bid value of @20 would win in either auction. However, under a simple second-price rule it would be better off joining only the second auction, and *hiding* from the first. By guaranteeing that the agent will pay the lowest possible price of @5, as is achieved with the Roller payment rule, the agent can instead submit its bid during the first auction.

Because Roller always calculate the lowest possible payment for a winning agent, to show that Roller is strategyproof, all we need to do is to show that the allocation rule is monotonic per Theorem 1. I will show this for both basic and ρ-late allocation rules in the following sections.

A.4 Roller With No Late Departure

In this section, I prove that the basic allocation rule is strategyproof by showing that it is monotonic. Here, I preclude the ρ-late allocation rule, which will be discussed in the next section.

Theorem 2. *The basic allocation rule is monotonic.*

Proof. Let $\theta_i = (w_i, n_i, s_i, a_i, d_i)$ and $\theta_i' = (w_i', n_i', s_i', a_i', d_i')$ be two types and $\theta_i \succ \theta_i'$. If the basic allocation rule is monotonic, then whenever θ_i' is allocated, θ_i *must* be allocated as well. I will show this by reviewing each of the type elements individually (while keeping other elements equal between the two types in each case). Note that the proof also generalizes to changing multiple type elements simultaneously.

- $w_i > w_i'$: the allocation rule sorts based on $v_i = w_i/(n_i s_i)$ and $v_i' = w_i'/(n_i' s_i')$. Thus, θ_i will be ranked higher than θ_i', which means if θ_i' is allocated in period t, then θ_i must also be allocated at least in period t, since if still unallocated in t then competing bids are unchanged from θ_i' and so θ_i will now be allocated.

- $n_i < n_i'$: since n_i is smaller, θ_i will be ranked higher than θ_i'. Thus, if space is available for n_i' (i.e., θ_i' is allocated) in period t, then θ_i will be allocated at least in period t, since if still unallocated in t then competing bids are unchanged from θ_i' and so θ_i will now be allocated.

- $s_i < s_i'$: because smaller s_i has the same effect as n_i in terms of higher ranking of θ_i, this too will lead to θ_i being allocated at least in period t in which θ_i' is allocated.

- $a_i < a_i'$: θ_i will be allocated at least in period t in which θ_i' is allocated, or earlier. This is because a_i and a_i' have no effect on the decision in any given period. In particular, if bid θ_i is still unallocated by period t, then competing bids are unchanged from θ_i' and so θ_i will now be allocated.

- $d_i > d_i'$: θ_i will be allocated in exactly the same period t as θ_i', because neither d_i nor d_i' are used in the allocation decision.

□

A.5 Roller With Possible Late Departures

With the ρ-late allocation rule, there are two cases to evaluate the strategyproof property with respect to departure d. The first case, denoted as "case ρ" is if the bid of an agent is selected to be allocated late. The second case, denoted as "case $1-\rho$" is if the bid is not to be allocated late, and proceeds exactly as above.

For "case ρ," when a bid is allocated "late," need to prove the following condition to show monotonicity:

$$f(\theta_i = (w_i, n_i, s_i, a_i, d_i) = 1 \Rightarrow f(\theta_i' = (w_i', n_i', s_i', a_i', d_i')) = 1 \text{ for } \theta_i' \succ \theta_i$$

Proof. Consider $d_i' > d_i$ and all other elements are the same (i.e., $a_i = a_i'$, etc.). If there is capacity available between t and d_i, in the period t when Roller decides to allocate θ_i, then there must be capacity available between t and d_i', and note that Roller will still decide to allocate θ_i' in the same period, t, because the reported departure does not affect the decision to make an allocation since it does not affect the ranking of a bid. □

The proof also generalizes to changing multiple type elements simultaneously. For "case $1 - \rho$," Theorem 1 holds when agents cannot over-report departure. More generally, strategyproofness also holds for mis-reports of early arrival under the basic allocation rule, because an allocation before true arrival yields zero value for agents despite lower payments.

By switching to ρ-late allocation, strategyproofness with regard to early arrival is traded with robustness in regard to late departure mis-reports. In the extreme case of $\rho = 1$, it is the case that Roller becomes fully strategyproof with regard to late departure mis-reports by the analysis in Hajiaghayi et. al. [43].

Appendix B

Replicator Dynamics

In this appendix, I will show the steps involved with Replicator Dynamics in order to find a symmetric mixed strategy Nash equilibrium (NE). This discussion is based on the Wellman et. al. [103] paper.

B.1 Basics

Denote the number of players by N and the strategy set consisting of M pure strategies by:

$$S = \{a, b, c, ..., z\}$$

.

A pure strategy profile $r = \{n_a, n_b, n_c, ..., n_z\}$ captures the number of agents playing each pure strategy. For example $r = (1, 2, 1)$ or $r = (4, 0, 0)$ for $N = 4$, $M = 3$, and $S = \{a, b, c\}$.

B.2 Computing Payoff Matrix

The payoff matrix u shows the average payoffs for each strategy of every pure strategy profiles. An average payoff is the percentage of value captured among all jobs by agents. Table B.1 shows an example partial payoff matrix (listed with only 3 pure strategy profiles):

n_a	n_b	n_c	Payoffs
1	2	1	(50, 70, 40)
0	2	2	(-, 30, 60)
4	0	0	(80, -, -)

Table B.1: **Example of a payoff matrix**.

The first row represents the strategy profile $r = \{1, 2, 1\}$, where $1, 2, 1$ agents play strategy a, b, c and on average receive payoffs of $50, 70, 40$, respectively. When no agent plays a certain strategy, the payoff will be indicated by "-."

B.3 Strategy Populations

Denote the proportion of population that pure strategy a is played in time t by $p^t(a)$. The sum of these proportions among all strategies is 1. To start calculating equilibrium, these proportions for every pure strategy at time 0 are initialized to be of equal amounts:

$$p^0(a) = 1/|S|, \forall a \in S$$

With the above example, $|S| = 3$ and so $p^0(a) = 1/3, \forall a \in S$.

B.4 Iterations

Once initialized, these proportions will be iterated over time t for calculating each $p^t(a)$:

$$p^t(a) = \frac{q^t(a)}{\sum_{a \in S} q^t(a)}$$

Note that $\sum p^t(a), \forall a \in S$ will equal to 1. Next, $q^t(a)$ is defined as follows:

$$q^t(a) = p^{t-1}(a) \cdot (EP_a^{t-1} - W)$$

where W is the minimal payoff value in the payoff matrix and EP_a^{t-1} is the expected payoff for strategy $a \in S$:

$$EP_a^{t-1} = \frac{\sum_{(r, \forall n_a(r)>0)} Pr^{t-1}(r) \cdot u(a|r)}{\sum_{(r, \forall n_a(r)>0)} Pr^{t-1}(r)}$$

$u(a|r)$ is the payoff to a player with pure strategy a given pure strategy profile r.

Finally, $Pr^{t-1}(r)$ is the probability of pure strategy profile r in iteration $t-1$:

$$Pr^{t-1}(r) = \frac{N!}{n_a \cdot n_b! \cdot \ldots \cdot n_c}! \cdot [p^{t-1}(s_1)]^{n_a(r)}$$

B.5 Finding and Verifying Equilibrium

If t is iterated long enough, the probabilities $p^t(a)$ should all become stable, i.e., their individual values do not change for further iterations. This fixed point is a possible equilibrium. Next, we must check the the equilibrium is indeed a true equilibrium. To do this, we check two conditions. Denote the equilibrium iteration by t^* and the equilibrium strategy profile by p^*:

1. $EP_a^{t^*} = EP_b^{t^*} = ... = EP_z^{t^*}, \forall a \in p^*$

2. $EP_a^{t^*} > EP_\alpha^{t^*}, \forall a \in p^*, \forall b \notin p^*$

Condition 1 states that all pure strategies that are represented in the support of the equilibrium mixed strategy profile p^* must have the same expected payoff value. Condition 2 states that the expected payoff value must be strictly greater than the expected values of all other pure strategies not played in the equilibrium strategy profile. For example, if the mixed strategy profile $p^* = (0, 0.5, 0.5)$, then it must be true that $EP_b^{t^*} = EP_c^{t^*} > EP_a^{t^*}$.

Bibliography

[1] Amazon web services. http://aws.amazon.com.

[2] Amazon web services crash. http://www.infoworld.com/t/iaas/some-data-irrecoverable-after-amazon-web-services-crash-155.

[3] Atlas experiment. http://atlas.web.cern.ch/Atlas/Collaboration/.

[4] ebay. http://www.ebay.com.

[5] Facebook. http://www.facebook.com.

[6] Google app engine. http://code.google.com/appengine/.

[7] Large hadron collider. http://lhc.web.cern.ch/lhc/.

[8] Microsoft private cloud. http://www.microsoft.com/virtualization/en/us/private-cloud.aspx.

[9] Moore's law. http://www.intel.com/technology/mooreslaw/index.htm.

[10] Open science grid. http://www.opensciencegrid.org/.

[11] Planetlab federation. https://www.planet-lab.org/federation.

[12] Secondlife. http://www.secondlife.com.

[13] Sirius. https://snowball.cs.uga.edu/ dkl/pslogin.php.

[14] Teragrid resource allocation policies. http://www.teragrid.org/userinfo/access/allocations.php.

[15] Vmware private cloud. http://www.vmware.com/solutions/cloud-computing/private-cloud/index.html.

[16] World of warcraft. http://us.battle.net/wow/en/.

[17] The god particle and the grid. *Wired*, December 2004.

[18] Money trouble in second life. *Technology Review*, August 2007.

[19] A. AuYoung, B. Chun, A. Snoeren, and A. Vahdat. Resource allocation in federated distributed computing infrastructures, 2004.

[20] Y. Bartal, R. Gonen, and N. Nisan. Incentive compatible multi unit combinatorial auctions, 2002.

[21] R. Betancourt, W. Gillespie, E. Kwerel, L. Martin, D. Vincent, T. Referees, P. Cramton, P. Cramton, P. Cramton, J. A. Schwartz, J. A. Schwartz, and J. A. Schwartz. Collusive bidding: Lessons from the fcc spectrum auctions. *Journal of Regulatory Economics*, 17(17):229–252, 2000.

[22] C. Borgs, J. Chayes, N. Immorlica, M. Mahdian, and A. Saberi. Multi-unit auctions with budget-constrained bidders. In *Proceedings of the 6th ACM conference on Electronic commerce*, EC '05, pages 44–51, New York, NY, USA, 2005. ACM.

[23] J. Broberg, S. Venugopal, and R. Buyya. Market-oriented grids and utility computing: The state-of-the-art and future directions. *Journal of Grid Computing*, 6:255–276, 2008. 10.1007/s10723-007-9095-3.

[24] J. Brunelle, P. Hurst, J. Huth, L. Kang, C. Ng, D. C. Parkes, M. Seltzer, J. Shank, and S. Youssef. Egg: An extensible and economics-inspired open grid computing platform. In *Proc. 3rd International Workshop on Grid Economics and Business Models (GECON'06)*, Singapore, 2006.

[25] R. Buyya, D. Abramson, and J. Giddy. NimrodG: An Architecture of a Resource Management and Scheduling System in a Global Computational Grid. In *Proceedings of the 4th International Conference on High Performance Computing in Asia-Pacific Region*, May 2000.

[26] R. Cavallo, D. C. Parkes, and S. Singh. Optimal coordinated planning amongst self-interested agents with private state. In *Proceedings of the 22nd Conference on Uncertainty in Artificial Intelligence (UAI'06)*, pages 55–62, Cambridge, MA, 2006.

[27] J. S. Chase, D. C. Anderson, P. N. Thakar, A. Vahdat, and R. P. Doyle. Managing energy and server resources in hosting centres. In *Symposium on Operating Systems Principles*, pages 103–116, 2001.

[28] S. Cheng, D. M. Reeves, Y. Vorobeychik, and M. P. Wellman. Notes on equilibria in symmetric games. In *In Proceedings of the 6th International Workshop On Game Theoretic And Decision Theoretic Agents (GTDT*, pages 71–78, 2004.

[29] B. Chun and D. Culler. User-centric performance analysis of market-based cluster batch schedulers. In *Proceedings of 2nd IEEE International Symposium on Cluster Computing and the Grid, Berlin (CCGrid 2002)*, 2002.

[30] B. N. Chun, P. Buonadonna, A. AuYoung, C. Ng, D. C. Parkes, J. Shneidman, A. C. Snoeren, and A. Vahdat. Mirage: A Microeconomic Resource Allocation System for SensorNet Testbeds. In *Proceedings of the 2nd IEEE Workshop on Embedded Networked Sensors*, 2005.

[31] B. N. Chun, C. Ng, J. Albrecht, D. C. Parkes, and A. Vahdat. Computational resource exchanges for distributed resource allocation. Technical report, 2004.

[32] E. H. Clarke. Multipart pricing of public goods. *Public Choice*, 2:19–33, 1971.

[33] S. H. Clearwater, editor. *Market-based control: a paradigm for distributed resource allocation*. World Scientific Publishing Co., Inc., River Edge, NJ, USA, 1996.

[34] F. Constantin, J. Feldman, S. Muthukrishnan, and M. Pal. An online mechanism for ad slot reservations with cancellations. In *Proc. ACM-SIAM Symposium on Discrete Algorithms (SODA'09)*, pages 1265–1274, 2009.

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

[36] S. Dobzinski, R. Lavi, and N. Nisan. Multi-unit auctions with budget limits. In *FOCS '08: Proceedings of the 2008 49th Annual IEEE Symposium on Foundations of Computer Science*, pages 260–269, Washington, DC, USA, 2008. IEEE Computer Society.

[37] J. Feigenbaum and S. Shenker. Distributed algorithmic mechanism design: recent results and future directions. In *DIALM '02: Proceedings of the 6th international workshop on Discrete algorithms and methods for mobile computing and communications*, pages 1–13, New York, NY, USA, 2002. ACM Press.

[38] I. Foster and C. Kesselman. The grid: blueprint for a new computing infrastructure. 1998.

[39] M. J. Freedman, E. Freudenthal, and D. Mazires. Democratizing content publication with coral. In *Networked Systems Design and Implementation*, pages 239–252.

[40] E. Friedman and D. C. Parkes. Pricing wifi at starbucks– issues in online mechanism design. In *Fourth ACM Conf. on Electronic Commerce (EC'03)*, pages 240–241, 2003.

[41] Y. Fu, J. Chase, B. Chun, S. Schwab, and A. Vahdat. Sharp: an architecture for secure resource peering. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 133–148. ACM Press, 2003.

[42] T. Groves. Incentives in Teams. *Econometrica*, 41:617–631, 1973.

[43] M. Hajiaghayi, R. Kleinberg, M. Mahdian, and D. C. Parkes. Online auctions with re-usable goods. In *Proc. 6th ACM Conf. on Electronic Commerce (EC'05)*, 2005.

[44] M. T. Hajiaghayi, R. Kleinberg, and D. C. Parkes. Adaptive limited-supply online auctions. In *Proc. ACM Conf. on Electronic Commerce*, pages 71–80, 2004.

[45] G. Hardin. The tragedy of the commons. *Science*, 162(3859):1243–1248, 1968.

[46] D. Irwin, J. Chase, L. Grit, and A. Yumerefendi. Self-recharging virtual currency. In *P2PECON '05: Proceeding of the 2005 ACM SIGCOMM workshop on Economics of peer-to-peer systems*, pages 93–98, New York, NY, USA, 2005. ACM Press.

[47] M. O. Jackson. Mechanism theory. In *The Encyclopedia of Life Support Systems*. EOLSS Publishers, 2000.

[48] P. R. Jordan, C. Kiekintveld, and M. P. Wellman. Empirical game-theoretic analysis of the tac supply chain game. In *AAMAS '07: Proceedings of the 6th international joint conference on Autonomous agents and multiagent systems*, pages 1–8, New York, NY, USA, 2007. ACM.

[49] L. Kang and D. C. Parkes. Passive verification of the strategyproofness of mechanisms in open environments. In *Proc. Eighth International Conference on Electronic Commerce (ICEC'06)*, 2006.

[50] I. A. Kash, E. J. Friedman, and J. Y. Halpern. Optimizing scrip systems: efficiency, crashes, hoarders, and altruists. In *ACM Conference on Electronic Commerce*, pages 305–315, 2007.

[51] A. Kothari, D. C. Parkes, and S. Suri. Approximately-strategyproof and tractable multi-unit auctions. In *Fourth ACM Conf. on Electronic Commerce (EC'03)*, 2003. To appear.

[52] V. Krishna. Auction theory. book, 2002.

[53] K. Lai. Markets are Dead, Long Live Markets. Technical Report arXiv:cs.OS/0502027, HP Labs, Palo Alto, CA, USA, Feb. 2005.

[54] K. Lai, L. Rasmusson, E. Adar, S. Sorkin, L. Zhang, and B. A. Huberman. Tycoon: an Implemention of a Distributed Market-Based Resource Allocation System. *Multiagent and Grid Systems*, 1(3):169–182, Aug. 2005.

[55] R. Lavi, A. Mu'alem, and N. Nisan. Towards a characterization of truthful combinatorial auctions. In *Proc. 44th Annual Symposium on Foundations of Computer Science*, 2003.

[56] R. Lavi and N. Nisan. Competitive analysis of incentive compatible on-line auctions. In *ACM Conference on Electronic Commerce*, pages 233–241, 2000.

[57] J. O. Ledyard. Incentive compatibility and incomplete information. *Journal of Economic Theory*, 18(1):171–189, June 1978.

[58] J. O. Ledyard. Public goods: A survey of experimental research. Public Economics 9405003, EconWPA, May 1994.

[59] D. Lehmann, L. I. O'Callaghan, and Y. Shoham. Truth revelation in approximately efficient combinatorial auct ions. *Journal of the ACM*, 49(5):577–602, September 2002.

[60] L. Levy, L. Blumrosen, and N. Nisan. On-Line Markets for Distributed Object Services: The MAJIC System. In *Proceedings of the 3rd USENIX Symposium on Internet Technologies and Systems*, March 2001.

[61] M. Litzkow and M. Livny. Experience with the condor distributed batch system. In *Proceedings of the IEEE Workshop on Experimental Distributed Systems*, Huntsville, AL, October 1990.

[62] B. Lubin, A. Juda, R. Cavallo, S. Lahaie, J. Shneidman, and D. C. Parkes. ICE: An Expressive Iterative Combinatorial Exchange. *Journal of Artificial Intelligence Research*, 33:33–77, 2008.

[63] D. Malan, T. Fulford-Jones, M. Welsh, and S. Moulton. Codeblue: An ad hoc sensor network infrastructure for emergency medical care. In *Wearable and Implantable Body Sensor Networks*, 2004.

[64] D. Malan, T. Fulford-Jones, M. Welsh, and S. Moulton. Codeblue: An ad hoc sensor network infrastructure for emergency medical care. In *Wearable and Implantable Body Sensor Networks*, 2004.

[65] M. L. Massie, B. N. Chun, and D. E. Culler. The ganglia distributed monitoring system: design, implementation, and experience. *Parallel Computing*, 30:817–840, 2004.

[66] motelab. Harvard sensor network testbed. web. http://motelab.eecs.harvard.edu.

[67] A. Mu'alem and N. Nisan. Truthful approximation mechanisms for restricted combinatorial auctions: extended abstract. In *Eighteenth national conference on Artificial intelligence*, pages 379–384, Menlo Park, CA, USA, 2002. American Association for Artificial Intelligence.

[68] S. Mullender, editor. *Distributed systems*. ACM, New York, NY, USA, 1989.

[69] R. N. Murty, A. Gosain, M. Tierney, A. Brody, A. Fahad, J. Bers, and M. Welsh. Citysense: A vision for an urban-scale wireless networking testbed. *IEEE Transactions on Reliability*.

[70] C. Ng, D. C. Parkes, and M. Seltzer. Strategyproof computing: Systems infrastructures for self-interested parties. In *Workshop on Economics of Peer-to-Peer Systems*, Berkeley, CA, 2003.

[71] C. Ng, D. C. Parkes, and M. Seltzer. Virtual worlds: Fast and strategyproof auctions for dynamic resource allocation. In *Fourth ACM Conf. on Electronic Commerce (EC'03)*, 2003. Poster version. Longer version at http://www.eecs.harvard.edu/econcs/virtual.pdf.

[72] N. Nisan. Bidding and allocation in combinatorial auctions. In *ACM Conference on Electronic Commerce*, pages 1–12, 2000.

[73] N. Nisan and A. Ronen. Algorithmic mechanism design. In *Proceedings of the 31st ACM Symposium on Theory of Computing*, Atlanta, GA, May 1999.

[74] N. Nisan and A. Ronen. Computationally feasible vcg mechanisms. In *EC '00: Proceedings of the 2nd ACM conference on Electronic commerce*, pages 242–252, New York, NY, USA, 2000. ACM Press.

[75] M. A. Nowak, A. Sasaki, C. Taylor, and D. Fudenberg. Emergence of cooperation and evolutionary stability in finite populations. *Nature*, 428(6983):p646 – 650, 20040408.

[76] M. J. Osborne and A. Rubinstein. *A Course in Game Theory*. The MIT Press, 1994.

[77] D. C. Parkes. Online mechanisms. In N. Nisan, T. Roughgarden, E. Tardos, and V. Vazirani, editors, *Algorithmic Game Theory*, chapter 16. Cambridge University Press, 2007.

[78] D. C. Parkes. *Iterative Combinatorial Auctions:Achieving Economic and Computational Efficiency*. PhD thesis, Department of Computer and Information Science, University of Pennsylvania, May 2001.

[79] D. C. Parkes and S. Singh. An MDP-based approach to Online Mechanism Design. In *Proc. 17th Annual Conf. on Neural Information Processing Systems (NIPS'03)*, 2003.

[80] L. Peterson, D. Culler, T. Anderson, and T. Roscoe. A blueprint for introducing disruptive technology into the internet, 2002.

[81] R. Porter and Y. Shoham. On cheating in sealed-bid auctions. In *ACM EC'03*, 2003.

[82] R. Raman, M. Livny, and M. H. Solomon. Matchmaking: Distributed resource management for high throughput computing. In *HPDC*, pages 140–, 1998.

[83] C. R. Reeves. Modern heuristic techniques for combinatorial problems, 1993.

[84] O. Regev and N. Nisan. The POPCORN market-an online market for computational resources. In *Proceedings of the first international conference on Information and computation economies*, pages 148–157. ACM Press, 1998.

[85] M. H. Rothkopf. Thirteen reasons why the vickrey-clarke-groves process is not practical. *Oper. Res.*, 55:191–197, March 2007.

[86] P. Schuster and K. Sigmund. Replicator dynamics. *Journal of Theoretical Biology*, 100(3):533 – 538, 1983.

[87] R. Sethi. Mixed strategy. In *International Encyclopedia of the Social Sciences*, pages 290–291. 2007.

[88] R. Sethi. Nash equilibrium. In *International Encyclopedia of the Social Sciences*, pages 540–542. 2007.

[89] S. Shetty, P. Padala, and M. Frank. A Survey of Maket Based Appraoched in Distributed Computing. Technical report, University of Florida, 2003.

[90] J. Shneidman, C. Ng, D. C. Parkes, A. AuYoung, A. C. Snoeren, A. Vahdat, and B. Chun. Why markets could (but don't currently) solve resource allocation problems in systems. In *HOTOS'05: Proceedings of the 10th conference on Hot Topics in Operating Systems*, pages 7–7, Berkeley, CA, USA, 2005. USENIX Association.

[91] J. Shneidman and D. Parkes. Rationality and self-interest in peer to peer networks, 2003.

[92] I. Stoica, H. Abdel-Wahab, and A. Pothen. A Microeconomic Scheduler for Parallel Computers. In *Proceedings of the IPPS '95 Workshop on Job Scheduling Strategies for Parallel Processing*, pages 122–135, April 1995.

[93] M. Stonebraker, P. M. Aoki, W. Litwin, A. Pfeffer, A. Sah, J. Sidell, C. Staelin, and A. Yu. Mariposa: A wide-area distributed database system. *VLDB Journal: Very Large Data Bases*, 5(1):48–63, 1996.

[94] I. E. Sutherland. A futures market in computer time. *Commun. ACM*, 11(6):449–451, 1968.

[95] P. D. Taylor and L. B. Jonker. Evolutionary stable strategies and game dynamics. *Mathematical Biosciences*, 40(1-2):145 – 156, 1978.

[96] H. Varian and J. K. MacKie-Mason. Generalized Vickrey auctions. Technical report, University of Michigan, 1995.

[97] H. R. Varian. Economic mechanism design for computerized agents. In *Proceedings of the 1st conference on USENIX Workshop on Electronic Commerce - Volume 1*, pages 2–2, Berkeley, CA, USA, 1995. USENIX Association.

[98] W. Vickrey. Counterspeculation, Auctions and Competitive Sealed Tenders. *Journal of Finance*, pages 8–37, 1961.

[99] V. Vishnumurthy, S. Chandrakumar, S. Ch, and E. G. Sirer. Karma: A secure economic framework for peer-to-peer resource sharing, 2003.

[100] C. A. Waldspurger, T. Hogg, B. A. Huberman, J. O. Kephart, and W. S. Stornetta. Spawn: A distributed computational economy. *Software Engineering*, 18(2):103–117, 1992.

[101] C. A. Waldspurger and W. E. Weihl. Lottery scheduling: Flexible proportional-share resource management. In *Operating Systems Design and Implementation*, pages 1–11, 1994.

[102] A. Walker. *The Science of Wealth: A Manual of Political Economy*. John Wilson And Son, 1866.

[103] M. P. Wellman, J. K. MacKie-Mason, D. M. Reeves, and S. Swaminathan. Exploring bidding strategies for market-based scheduling. In *EC '03: Proceedings of the 4th ACM conference on Electronic commerce*, pages 115–124, New York, NY, USA, 2003. ACM.

[104] R. Wolski, J. S. Plank, J. Brevik, and T. Bryan. Analyzing market-based resource allocation strategies for the computational Grid. *The International Journal of High Performance Computing Applications*, 15(3):258–281, Fall 2001.

[105] M. Yokoo. False-name bids in combinatorial auctions. *SIGecom Exch.*, 7:48–51, December 2007.

[106] E. Zurel and N. Nisan. An efficient approximate allocation algorithm for combinatorial auctions. In *Proceedings of the 3rd ACM conference on Electronic Commerce*, pages 125–136. ACM Press, 2001.