iCDA: Continuous Double Auction on the Web

Brandon Diamond*

April 30, 2009

Abstract

This paper describes the implementation and design of a "web-scale" electronic continuous double auction system. The intention of this system is to facilitate multiple simultaneous participants in the form of autonomous trading agents; an individual agent may only trade in a single market at a time, though a single commodity may be traded in multiple markets by multiple agents simultaneously. The distributed system– itself comprised of distinct servers for hosting marketplace, agent, and load balancing software– is designed to run on the commodity hardware offered by services like Google App Engine, Amazon Web Services, Linode, and SliceHost. The system accepts new requests via a RESTful API, and the process of starting, running, and terminating a job is entirely automated. The system also includes a web-based reporting interface for testing and analytics purposes.

*btd@brown.edu (Brown University)

Contents

1 Introduction

Numerous researchers and institutions have committed significant resources toward the development of electronic continuous double auction systems. Some of these projects are designed to facilitate human traders; still others are designed with trading software in mind. Many such endeavors amount to academic exercises[6]: the resulting systems emphasize theoretical accuracy and measurability with the goal of elucidating the properties and consequences of particular market policies, designs, and interactions. In light of this objective, many *continuous double auction* (CDA) systems impose significant constraints on the end user[5]; flexibility is sacrificed for purity and correctness. Unfortunately, this trade-off makes these systems difficult to adapt to unpredictable, loosely structured environments where users do not always behave rationally and ease-of-use exceeds theoretical accuracy in importance.

The world wide web is a prime example of such an environment, consisting of hundreds of thousands of distinct software programs vying for the user's limited time and attention. Any website that presents a poor or cumbersome interface– even to a useful service– is likely to replaced by a far simpler and therefore superior site in mere weeks[1]. On the web, those applications that thrive and grow tend to be those that fulfill one purpose, and do so well. As a result, a developer wishing to utilize a typical CDA system as part of a novel web application would be required to devote a considerable amount of time and energy into sanitizing user interactions: it is essential that the user interface of any serious web application be simple and intuitive, even if the market back-end itself is ugly and sophisticated. Furthermore, as web applications are typically required to support massive amounts of concurrent traffic, the underlying market system will likely become a primary performance bottleneck unless this system has been designed to scale -– a property not typically found in academic software.

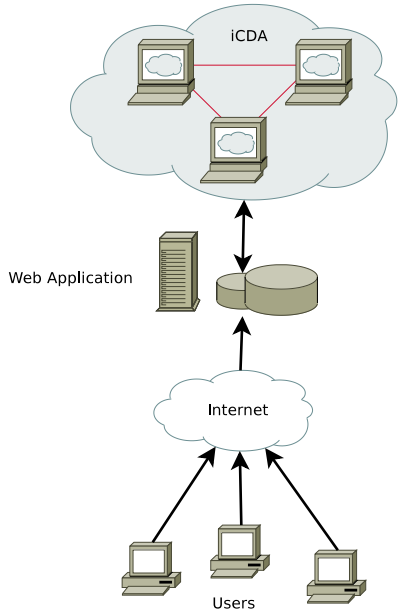

Figure 1: The system in context.

An ideal CDA system for the web would (1) present a flexible and clean interface to the developer; (2) scale easily and seamlessly; (3) support untrustworthy, naïve users; and (4) degrade predictably and gracefully under heavy usage. The remainder of this paper describes iCDA, a system seeking to address each of these desiderata (figure 1).

2 Background

iCDA is a distributed system written primarily in the Java™ programming language (JDK 6.0) that is capable of concurrently supporting multiple continuous double auctions. Where possible, a modular approach is taken in the design and implementation of this system so as to retain a strong focus on achieving a flexible and highly functional CDA environment. For instance, a great deal of computation is offloaded from the market module to an external database

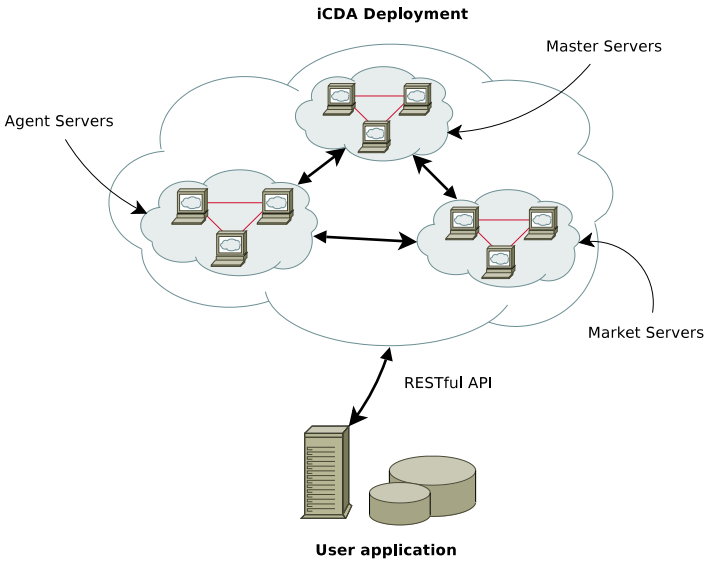

Figure 2: iCDA organization.

server supporting the SQL-92 standard, or better. This underlying DBMS provides iCDA with an efficient, scalable, and customizable means of storing the limit order book as well as obtaining quotes, placing bids, and matching offers. In fact, each of these procedures can be expressed in two or fewer simple SQL queries.

As a distributed system, there are a number of daemons comprising the full iCDA service (figure 2); among these are agent servers, market servers, and master servers. The master servers are responsible for load balancing, reliability, as well as the dispatching and termination of jobs; the agent servers are responsible for scheduling and running any number of autonomous trading agents; and the market servers are responsible for accepting bids and asks and enforcing the rules of running auctions. Communication is reliable and facilitated by the robust *remote method invocation* (RMI) system provided as part of the J2EE package.

The general workflow of the system is straightforward: a new job is con-

veyed to a single master server by means of a RESTful web service interface. Once this job has recieved and unmarshalled, the location of the market server corresponding to the requested commodity is identified via a second database look-up; if no mapping is found, a new server is started on a machine selected by a load-balancing module. Once the market server has been initialized, the master server will identify an agent server with a vacant slot– again possibly starting a new server daemon. The master will then request that a new agent thread be created in accordance with the particulars of the job specification. Note that a central, transactional data store is required to describe the system's overall topology so as to facilitate job dispatching and tracking; this requirement should not present any problems or performance obstacles so long as the underlying DBMS can support replication or another form of parallelization (see below for further discussion).

Once the agent and market server have been coupled, the agent is free to execute an arbitrary trading strategy that utilizes the interface exported by the market server as well a comprehensive agent framework. Throughout this process, the master servers ensure the liveness of each agent and market server, restarting daemons as needed to handle incidental faults. Due to the careful design of the agent server and the market server, no critical state is left in volatile memory; correspondingly, arbitrary failures should not typically result in any sort of data corruption. Likewise, the master server stores critical data in a transactional, snapshotted data store so that the consistency of control data will be mantained.

The system is deployed on any number of remote servers by means of a collection of shell scripts. After starting a single master and establishing an initial logical topology, the complete system can be brought up and monitored automatically. With iCDA active, all job requests are channeled through the

master server cluster via a traditional web services API; this affords a great deal of flexibility in the implementation and design of the more traditional web application component.

3 Methodology

3.1 Preparation

An iterative, agile approach has been applied to the development of the iCDA system. Several versions of the software have been produced, incrementally adding functionality and sophistication to the base system. An initial Python-based proof-of-concept implemented a single one-variable CDA entirely in memory. This code was eventually re-written in Java, and split into separate remote modules– a market server and an agent server– via Java remote method invocation technology. With this iteration tested and working, these modules were further extended to support multiple simultaneous markets and agents, respectively.

As testing progressed, it became apparent that bids and asks (collectively "offers") were sufficiently complex to warrant the creation of more intelligent container class. Subsequently, these new classes were endowed with the ability to directly target market segments based on the specific attributes of the traded commodity.

With many of the elements of a reliable distributed system in place, the next iteration entailed the creation of a central master server to coordinate each of the servers as well as the corresponding machines. The master server gradually grew in complexity to the point that multiple, redundant master servers would cooperate so as to reduce risk to this central point of control; additionally, in-memory buffers were introduced via a lock-free, concurrent queue data structure

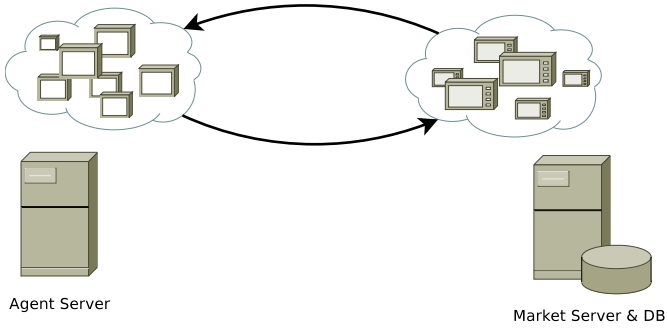

Figure 3: Market servers and agent servers host multiple instances.

so as to further improve support for larger deployments of iCDA.

Each iteration revealed new challenges and design issues. Confronting these required compromises in the overall architecture of the system. Below, each subsystem is described in detail along with the challenges, solutions, and compromises that sourrounded its development.

3.2 Core Servers

3.2.1 Market Server

The *market server* is responsible for hosting multiple, simultaneous markets for any number of distinct commodities. Each server must be associated with a relational database system to store the limit order book, and to facilitate a number of market operations. No critical state is stored in volatile memory; the complete market description is recorded in a central database and the market server operates on this data exclusively through simple SQL queries. Each market instance trades a single commodity, though the particular items traded may vary in various dimensions. Typically, there is a one-to-one mapping between market instances and commodities. However, in the event of heavy load, it is possible for a single commodity to be traded in more than one market.

A single *market manager* serves as the primary access point for a market

server; this manager provides facilities for creating, fetching and destroying individual market instances on a per-commodity basis. Internally, a concurrent map data structure is used to maintain the mapping from commodity to market instance. In the event that a market server should fail with active market instances, the master server will deploy a new market server and re-create all of the affected market instances. The master server will utilize its own reliable record of the network topology to ensure that the newly deployed market server is associated with the appropriate database connection. Finally, each market instance will confirm that the required tables do not currently exist before adding them to the database. As a result, the limit order book is preserved in the event of unexpected market failures.

Each market instance presents a minimal remote interface suitable for use by arbitrary trading agents (figure 3). This interface is synchronous; the user may be assured that the offer has been added to the limit order book if and only if the associated method returns successfully. The market instance relies on the threading of Java RMI as well as the semantics of the synchronized keyword to automatically handle queuing of simultaneous requests. Operations are idempotent and so sporadic failures may be readily handled simply by reissuing any problematic method calls.

The market interface consists of three simple factory methods: one to create a remote bid object, one to create a remote ask object, and a final one to create a remote quote object. These objects and the methods they implement complete the remote market interface. Once a new bid or ask object is returned by the market server, a corresponding record in the limit order book will have been successfully inserted. The agent may subsequently use this bid or ask object freely; both provide methods for updating as well as revoking the underlying offer. Offers may be reclaimed in the event of failure as each record in the

database is associated with a unique job identifier.

Clear events are detected prior to inserting a bid or ask into the market database. A clear is said to occur when a bid exceeds or matches the prices of an ask; furthermore, both the ask and the bid must be compatible, which is determined by evaluating the constraints associated with both offers. Due to the design objective of running on readily available web infrastructure, this test is implemented as a single comprehensive SQL query. If multiple offers match, price is held above all other attributes with seniority used to resolve ties. It is left to the trading agent to determine the tradeoff between price and all other contributing factors. After all, this is the fundamental purpose of any trading agent.

Once the clear has been confirmed, the market manager is notified, causing a remote procedure call to be issued to the first available master server. The master server atomically records the clear event and updates the corresponding jobs, ensuring that the clear is final. This call will return with references to both remote agents; the market will notify the agents that a clear has occurred, typically causing the agent to terminate immediately (in the event of a total clear). Should the agent submit any additional offers to the market prior to notification, they will not be honored; the market server disregards requests made by the client until the transaction has completed. Figure 4 illustrates the complete process.

3.2.2 Agent Server

Markets interact primarily with automated trading agents, pieces of software designed to issue bids and asks in fulfillment of a particular algorithmic trading strategy. The software providing this functionality within iCDA is the *agent server*. There are many different ways to implement an agent, and so iCDA endeavors to impose as few constraints on the agent author as is reasonably

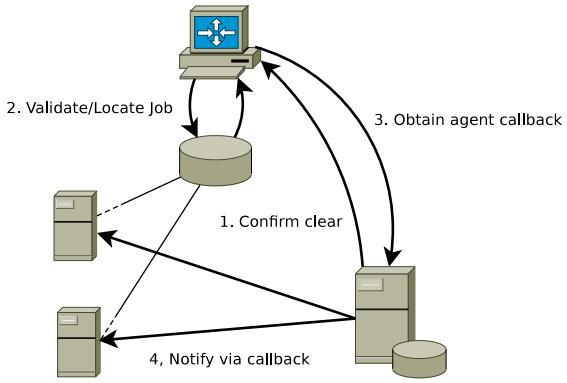

Figure 4: Market clear execution path.

possible. In particular, the only significant requirement of a trading agent is that no critical state be stored in volatile memory. However, as many successful trading strategies generate decisions based on current market conditions alone, this constraint is typically not an obstacle[5]. If permanent storage is required by a particular implementation, that agent is free to explicitly serialize data to a non-volatile store (i.e., a distributed "ambient" database such as Amazon's SimpleDB); each agent is provided a job description containing a unique ID that is suitable for use as a primary key in such a setting.

A single agent manager provides an interface for creating, querying and removing agents. The list of current agents is stored entirely in-memory; should an agent server fail, all jobs handled by that server will flagged as unscheduled. Clearing the market involves a call to a master server which, among other details, provides the market server with the current network location of each involved agent (or an error value, if the agent no longer exists). This information is sufficient to allow agents to be located even after they have been moved. As each agent is implemented as an individual thread, Java's executor service abstraction is utilized to provide fine-grained control and pooling of shared processing resources. Agent code is assumed to be trustworthy; consequently, very

few restrictions are enforced by the manager.

Agent classes descend from a common super-class implementing the scaffolding and core functionality needed to interact with a market server. This class provides a number of useful synchronization primitives relating to the occurrence of critical market events as well as the necessary framework for interacting with the agent's manager. Each individual agent implementation need only define a trade() method; so long as this code does not depend on persistent state, the implementer is free to do as he or she pleases.

Each agent is created by means of a single factory method in the agent superclass; this method dynamically loads the agent from a given file and assigns a new instance to a managed thread. Status information for each agent can be queried asynchronously. An agent is considered to be starting if it has not yet been assigned a thread, canceled if the agent has been explicitly terminated, done if the agent code has returned, or running in all other circumstances.

Every agent instance is constructed with a job description as well as a reference to the agent manager. The job description contains a number of remote objects and entities (discussed below) that allow the agent to interact with remote resources, including the market server. The relationship between agents, markets, and jobs is depicted in figure 5. To get started, an agent might request one or more quotes from the associated market server. Based on this data, an agent may then choose to sleep or to make a bid or an ask.

When an agent submits an offer (or, more accurately, requests that an offer be created on its behalf), the details and constraints pertaining to the offer are written to the market's backing storage and an offer handle returned to the agent. The agent may then idle for some time, utilizing one of several synchronization primitives provided by the agent framework for sleeping until a clear event (or timeout) occurs. Alternatively, the agent might continue to

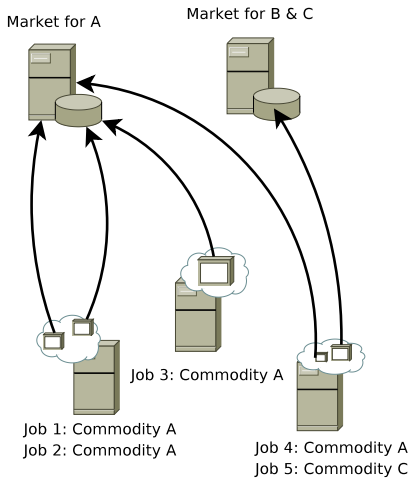

Figure 5: Relationship between agent and market servers.

execute, periodically polling for a clear event using a separate non-blocking interface. At present, jobs encompassing more than one item are not supported by the system: at most one clear event may occur per agent lifetime.

3.2.3 Master Server

iCDA depends on one or more *master servers* to act as a central authority for reliability, load-balancing, and interfacing with the front-end application (figure 6). While this represents a compromise in the overall scalability and robustness of the system, many techniques are utilized to mitigate the side-effects of centralization. As mentioned, master servers may be replicated across many machines; each master maintains a record of other active master servers which it forwards to any corresponding slave servers (viz., agent servers and market servers). As all of this data is maintained using a reliable central database, the system's primary bottleneck is effectively shifted from the master servers to the underlying DBMS; so long as this system provides for distribution and replication, iCDA should be capable of scaling to the levels of traffic characteristic of

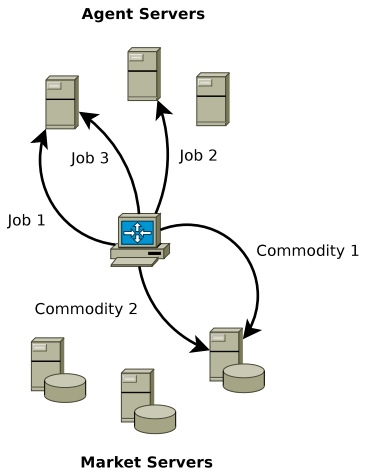

Figure 6: The master server organizes shared state.

popular web applications.

The master servers are responsible for the reliability, organization, and dynamic control of the iCDA system. More specifically, the master servers (1) dispatch requests, allocating resources where necessary, (2) maintain a mapping of abstract cluster topology to lower-level network addresses (typically, host names to be resolved using a separate dynamic DNS system), (3) monitor servers for availability, restoring problematic machines when necessary, and (4) provide reliable job monitoring and reporting. Figure 7 illustrates these objectives.

In fulfillment of the first objective, each master server supports a RESTful interface for handling requests from the user application. Various calls are processed via this interface including job creation, termination, and modification requests. These are encoded and transmitted via HTTP to a master server. Subsequently, the pertinent data is extracted from the request and used to initialize or modify an iCDA job. In the event of modification, the corresponding agent is located using the master database (or an error generated if the job is

invalid) and asynchronously notified of the update. In the event of job creation, a market server corresponding to the associated commodity is located; if the commodity isn't currently hosted, a server with available resources is identified or created and a new market instance assigned to host the commodity; a handle to this market server is then incorporated into the job instance. Finally, an agent server with available resources is identified or created and a new agent instance assigned to process the job.

To implement these steps, it is essential to maintain a reliable representation of the overall system's logical topology. In particular, discovering the location and availability of slave servers (in fulfillment with the master server's second design objective) as well as storing and interrogating the status of running jobs each require queries to a central information repository. This central data store organizes all critical information required to reconstruct the system's organization after catastrophic or partial failure as well as the data required to reliably process jobs. Correspondingly, the performance of an iCDA deployment is fundamentally tied to the performance of the DBMS underlying the central store.

To preserve the availability of the iCDA deployment, in fulfillment of the third design objective, each master server periodically pings slave servers. Any slave that does not respond after a fixed number of tries is assumed to be defunct and is promptly restarted. Effort is taken to guarantee that the restarted slave is assigned the appropriate host name so as to maintain a consistent system-wide naming scheme; further, if the affected daemon is a market server, it is reassociated with the corresponding order book database. Any agents or markets that are unable to contact the affected machine will retry their requests at exponentially longer intervals. If the retry count exceeds a certain value, the call will fail and the request will be discarded.

The final responsibility of the master server is to enable reliable job moni-

toring and reporting. As such, the master server is responsible for ensuring that each and every job is processed exactly once with a minimum of latency; moreover, the master server is responsible for providing accurate and timely feedback to the user application as regards the execution and status of each scheduled job. In fulfillment of this goal, all jobs are journaled to the central data store; this allows iCDA to monitor the physical assignment of jobs to machines and to serve as a strictness-point in validating and authenticating job state transitions. Additionally, the master server is responsible for synchronizing external requests with asynchronous events triggered by the rest of the system (such as market clears).

Should the iCDA system fail for any reason, all jobs can be safely resumed using the records written to the central data store. Once the market servers and agent servers have been restarted and bound to appropriate host names, incomplete jobs are dispatched as if they were new. Recall that this is valid as agent instances do not store state in volatile memory. Each agent instance automatically reclaims ownership of bids associated with the corresponding job ID: the agent instances are provided with a list of active offers at construction time. Finally, the status and result of each job is reported to the user application by means of the monitoring system described below.

3.2.4 Transparency

Providing feedback to the user application is a non-trivial challenge. Ideally, the status of each market and each agent would be reported in approximately real-time. Moreover, this information would be in-depth, incorporating market quotes and activity, clear notifications, and numerous other features. By relaxing these constraints– and utilizing some preexisting database software– iCDA is able to provide most of this functionality.

In particular, as the freshness of data reported to the user application is sec-

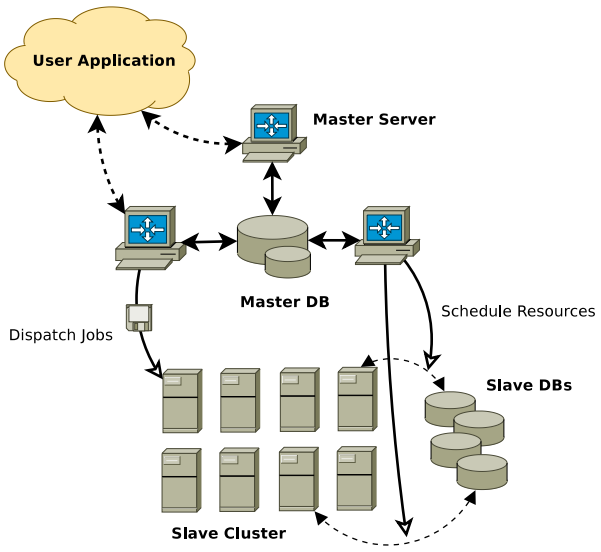

Figure 7: Resource scheduling by the master server.

ondary to accuracy, it is reasonable to return slightly older data to achieve the other design objectives of this subsystem. By leveraging an eventually consistent[3] distributed database platform for data persistence (Eg., Amazon SimpleDB, CouchDB, etc.), the burden of supporting system-wide logging is significantly reduced. Namely, each iCDA daemon inserts minimally structured rows into the distributed database over the course of normal execution; simultaneously, the user application interrogates this store to obtain reasonably fresh system status data.

While some data may be anywhere from seconds to minutes old, this is typically acceptable. Slightly stale market quotes and job status reports ought not cause any significant inconvenience to end-users of the system as most critical market interactions are entirely automated. Those that are not automated are limited to interactions modifying a pre-existing job. For instance, a job might be updated to reflect different utility or be canceled altogether. However, had the job already completed (perhaps due to a market clear), it would no longer

be legally mutable. This scenario is handled by employing the master server to detect the conflict using its own strictly reliable records. In the event of an invalid request, the master server will simply disregard the changes and post a report to the distributed database. Again, so long as this is presented to the end-user in a reasonable way, it should not cause undo inconvenience or surprise.

3.3 Entity Framework

3.3.1 Motivation

iCDA relies heavily on the underlying database to mantain consistency and to perform several essential computations. As a result, it is typically necessary to convert freely from Java objects to database records. While there are many *object relational mappers* readily available to confront this problem, many of these impose a number of performance constraints on the system that would not be acceptable. Additionally, various tasks such as single-query constraints checking are particularly difficult to achieve using these mappers. As a consequence, a custom system was developed to support the storing and querying of iCDA entities.

The entity framework is comprised of *attributes* and *constraints*; an attribute represents a potentially unbound, typed value whereas a constraint represents a rule describing valid assignments to an attribute. Furthermore, each attribute is associated with either a scalar or compound type. Attributes are arranged into collections mapping names to attribute values; similarly, constraints are arranged into collections associating attribute names to any number of constraints of the same type. Any Java object that can be converted to an attribute map is considered an *entity* and any Java object that can be used to generate a constraint map for this entity is referred to as *entity constraints*. The association between an entity and entity constraints is weak: entity constraints need not

specify constraints for all attributes in an entity, and an entity need not posess all attributes constrainted by the same.

The entity and entity constraints interfaces expose a flexible and simple method for processing Java objects. Rather than relying on unsafe and inefficient reflection techniques to inspect a Java object, entities make use of a simple, consistent visitor pattern allowing for interpretation by any number of loosely-coupled visitor implementations. As such, iCDA utilizes an assortment of SQL visitors that are capable of converting entities and entity constraints into efficient database schemas, queries, updates, and insertions. These particular transformations are straightforward and result in highly optimized, carefully crafted SQL.

3.3.2 Concrete entities

Entities are simple Java objects that can be readily converted to attribute maps. As such, these objects may be efficiently introspected and processed by means of the visitor interface described above. Throughout iCDA, several distinct entities are used for market concepts that must interact with a database. These are enumerated below.

Commodity A distinct variety of good. This entity represents a schema for a particular commodity; it fulfills a similar purpose to that of classes in an object oriented programming language. Commodities act as the unit of differentiation between markets; a market may only serve a single commodity at a time. Commodity descriptions are transmitted by the user application to the master server when each job is created; the master server assigns each unique commodity an identification number and generates a corresponding commodity instance.

Item A single traded instance. Whereas a commodity describes the attributes defining a particular type of good, an item represents a discrete, marketable instance of that good. More concretely, an item represents concrete bindings of the attributes associated with a single commodity. Items are extracted from requests to sell from the user application; each item is associated with a single commodity. A single market instance trades in potentially disparate items that are, broadly speaking, the same commodity. Items enable a degree of sub-market granularity. The item abstraction captures variation in a single commodity that is significant enough to influence transaction price, but not so significant as to warrant the creation of a distinct marketplace.

Listing Properties associated with the sale of a single item. User applications typically allow individual items to be sold with one or more caveats describing the nature of the transaction. For instance, the mode of payment supported by the end-user is a property that falls into the domain of the listing entity. Another example property is an item geographical location. In many regards, a listing represents those commodity attributes that are shared by all saleable commodities. Finally, a global listing schema is specified by the user application when iCDA is deployed.

User Metadata concerning a single client (human or otherwise) of the user application. Agents typically utilize metadata relating to the user responsible for a particular job to estimate the quality of particular offers. This allows for the implementation of a *reputation system* within the user application. User data is included in the job description provided by the user application to the master server.

3.3.3 Entity Constraints

Often, it is useful to describe a broad range of items; as an example, it would be cumbersome if not outright unacceptable to require the user application to issue buy requests for concrete item instances. Typically, buyers tolerate a coniderable degree of variation in the item sought. It is essential for a CDA service to provide a means for capturing this variance.

Furthermore, the ability to express constraints on an item is required to support market segmentation by trading agents. Rather than requiring that an agent target a specific listing or, conversely, the market as a whole, entity constraints allow agents to solict quotes for and post offers to a range of items in a marketplace. Correspondingly, each of the concrete entities described above is paired with a matching constraints type.

Constraints are comprised of one or more *conditions* on specific entity attributes, where a single attribute can be associated with at most one condition. The implemented conditions are *inequality*, *equality*, *membership*, and *range*. These provide sufficient flexibility for most applications while being readily adaptable to storage and querying using a relational database system. Individual conditions may be associated with normalized, fractional weights as well as a bias indicating the end-user's preference. These weights are utilized as a means of conveying relative importance whereas bias is used to provide for soft constraints that can be relaxed over time or eliminated outright. The default bias is *required*.

3.3.4 Object relational mapping

Entities and entity constraints are readily transformed by obtaining the underlying attribute and constraint maps and processing these using the *visitor pattern*. This pattern is a common recipe for processing hierarchical data such

as an abstract parse tree and are commonly found in the implementation of compilers and interpreters. A visitor implementing a particular transformation is passed to the tree structure's root node, resulting in the invocation of a type specific method of the visitor object. This method performs some processing and recursively visits the child attributes of the corresponding node. The entity framework utilizes dynamic dispatch to ensure that the method corresponding to the node's type is invoked, simplifying the implementation of new visitor classes.

As an example of the visitor pattern, an attribute map is typically processed by invoking the base *accept* method and passing it an instance of the desired visitor. Each accept method does nothing more than invoke a type specific method in the visitor instance. This *visit* method processes each attribute within the map recursively by invoking the accept method of each with the same visitor instance. If an attribute is compound, each sub-component of this attribute will be visited individually. In this way, every attribute within the map is systematically explored and processed without coupling the transformation with the data being transformed.

Attributes are stored in the database simply by mapping the attribute type to the closest corresponding SQL type. Constraints, on the other hand, are slightly less straightforward. As relational databases typically do not provide a means of storing and querying columnar constraints, the developer is forced to choose between a space-efficient representation and one that can be queried as efficiently as possible. iCDA is designed with the second objective in mind due to the reliance of the market server's clearing mechanism on the querying of entity constraints. Consequently, the greatest advantage is achieved by optimizing for query efficiency.

As a result, every table supporting the use of entity constraints must include

seperate columns for every possible constraint type. Fortunately, the current implement requires the addition of at most four constraint columns. The extra storage required for these columns varies based on the type of the underlying attribute (three of the four constraint columns share this type), but is typically at most 32 bytes per constraint per row. Despite this memory inefficiency, the inclusion of these columns makes it possible to select items satisfying a certain set of constraints as well as to select items that are satisfied by a certain attribute binding in just one, efficient query each. This readily offsets the slightly increased memory requirements of the system.

3.4 Market components

3.4.1 Offer Objects

Agents interact with markets primarily via offer instances, where an offer is either a bid or an ask. Bids represent offers to purchase a commodity at a particular price whereas asks represent offers to sell a commodity at a particular price. Both bids and asks alike implement a common offer interface and can be constrained to particular segments within a broader commodity market.

Offers are constructed remotely via factory methods exported by the market interface. As such, offers are inexorably bound to their parent market and may not be reused or submitted to other markets. Each offer is inserted into the corresponding market's order book if and when it is constructed successfully. Modifications to the offer are achieved by invoking mutator methods on the offer instance. Clear notifications occur asynchronously and do not interact with the corresponding offer objects. Offers that have cleared are removed from the limit order book causing any future modifications made via the corresponding offer instance to result in a market exception. Finally, a single agent is currently constrained to posting a single offer to a market at a given time. Thus, updating

an offer and posting a new offer have the same net effect.

The abstract offer super-class provides a limited interface to the underlying bid or ask. In particular, all offers provide a method for interrogating the type of the offer, the price being offered, as well as the agent that owns the offer. This information is sufficient for broad handling of offer objects; detailed interaction is achieved via the underlying bid and ask interfaces themselves. Throughout the code base, this is achieved chiefly via Java's dynamic dispatch facility.

All offer objects are backed by a single row in the corresponding market server's order book. Bid creation and modification is achieved by executing either an insert or update query on that database. As no agent is itself granted access to this database, the corresponding methods on the offer are executed remotely with stubs on the market server delegating the request to the appropriate, privileged market instance.

Bid Offers Agents wishing to purchase an item at a certain price may utilize the bid interface to do so. Bids extend the general offer interface by including a number of purchaser constraints that can be used to segment the underlying market. In particular, these include constraints on the seller, the listing, and the specific item. Thus, an agent may carefully craft bids that address only those items that satisfy its objectives, for example, by excluding those that are too old or in poor shape.

Ask Offers Asks are congruent to bids and indicate an agent's willingness to sell at a certain price. Aside from differing in semantics, asks and bids respect a disparate set of market constraints (an ask may only place constraints on the associated buyer, whereas bids support a wider array of constraints). In general, asks describe a concrete item for sale while bids provide boundaries on the item to purchase. Each ask includes a description of the particular item for sale, the

listing in the user application, as well as the user selling the item.

3.4.2 Quote Objects

There are two useful types of quotes in the iCDA system. The first, more traditional variety, indicates the highest unmatched bid and lowest unmatched quote (the bid-ask quote) while the second indicates the prices of past transactions (the price quote). As agents are typically stateless, the price quote is useful in that it gives each agent access to historical trading information that is necessary to make informed trading decisions. Meanwhile, the bid-ask quote is useful as it provides agents with insight into current market conditions. Together, these two inputs form the basis upon which agents generate and schedule offers.

Quotes, like offers, may be targeted to particular market segments by means of a simple constraints system. For instance, an agent may solicit a quote for all commodities that are in outstanding condition and that are green. The resulting quote will incorporate only those offers and completed trades that reflect these particular constraints. This feature is needed to support markets with somewhat heterogeneous commodities as is typical of many real-world CDA applications.

The quote abstraction is far simpler than that provided for offers, serving only as a container for data retrieved by the market instance. A single quote consists of two monetary values comprising the bid-ask quote as well as a bounded list of prices and times comprising the price quote. The agent may request as many quotes as are required to implement the corresponding strategy though, to decrease network congestion, agents ought to be written in such a way as minimizes the number of quotes requested. This approach was taken to avoid requiring markets to track participating agents.

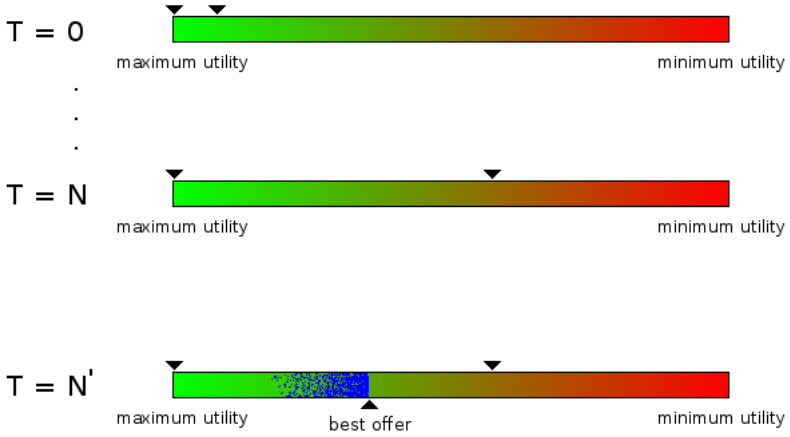

Figure 8: Trading algorithm over time.

3.4.3 Clear Objects

In the event that an offer clears, both agents are notified via remote callback and provided with a clear object. Among other things, this object indicates the final transaction price as well as the particular attributes of the item sold. In the event that the clear was partial, the clear object can be used to track those items that have been bought or sold versus those that have not. Any non-partial clear will result in the immediate cancellation of the corresponding agent.

3.5 Agent Components

3.5.1 Overview

iCDA supports a wide variety of agent designs and strategies. Among these, zero information (ZI) agents are generally the easiest to implement due to the lack of native support for data persistence by agent servers. Typically, these limited intelligence strategies tend to perform suprisingly well despite the sim-

plicity of the underlying strategy: the literature provides numerous examples of naïve agents performing at near market efficiency[2]. This seems to indicate that the continuous double auction structure itself encourages efficient trading irrespective of the rationality or irrationality of the participating agents. Correspondingly, the reference agents bundled with iCDA implement markedly simple strategies that are nonetheless capable of performing quite efficiently.

There are two agents currently included with the system: *BuyAgent* seeks to acquire goods while *SellAgent* seeks to sell goods. Both of these accept a utility function in the form of entity constraints as well as a target time frame. The implementation of each agent is described below.

3.5.2 BuyAgent

The BuyAgent utilizes a stochastic, near zero-intelligence trading strategy to process jobs. Utility is approximated by computing the distance between an offer and the user's ideal configuration. This configuration is represented within the job description as a series of entity constraints including weights and biases. In addition to these constraints, a soft deadline is fixed to convey the relative urgency of the job. Broadly speaking, the agent computes offers by selecting random offers within the currently acceptable range of configurations while simultaneously broadening this range linearly with respect to the proximity of the job deadline (figure 8).

The distribution of the random numbers underlying this search is dependent on current market state; for instance, an inactive market will result in the agent's bids skewing lower whereas active markets will result in the agent selecting bids uniformly. Additionally, if an ask is found within the currently acceptable range, the BuyAgent will sample offers using a gaussian distribution centered around the highest such ask.

In many respects, this strategy resembles reverse simulated annealing. The

BuyAgent is initially inflexible, permitting only minor variations in bid configuration. As time passes, assuming that the market is demonstrably competitive, the agent will gradually tolerate lower and lower utility so as to ensure that the job is fulfilled within the allowable timeframe. Simultaneously, the agent reacts to bid-ask quotes by altering the underlying random source and offer window to ensure that potentially attractive offers are not overlooked.

Constraints with non-required biases are considered disposable; the relative ordering in which constraints are relaxed is determined using the corresponding weights. If a weight is not provided, it is assumed to be maximum. As the deadline draws nearer, those constraints with lower weights are relaxed at a greater rate than those with higher weights. Specifically, each weight is normalized and interpreted as the percentage of time over which the corresponding constraint will be relaxed. Constraints with *continuous-positive* or *continuous-negative* bias will be interpolated in a particular direction throughout this period. Bias may also be *expendable*– possibly in combination with interpolation– causing the constraint to be dropped once the time period has elapsed.

3.5.3 SellAgent

The SellAgent is identical the BuyAgent. However, rather than seeking the lowest price, the SellAgent seeks the highest price.

3.6 Web Interface

A simple web interface has been created to allow for the evaluation and testing of the iCDA system. This interface takes the place of the oft-mentioned user application, providing a means for submitting and monitoring jobs, as well as observing the operation of the iCDA system as a whole. The data reported is fine-grained, including individual offers and recent bid-ask quotes. Additionally,

an interface is provided allowing for the creation of arbitrary commodities, as well as the listing of particular items for sale. Users may issue buy requests for items satisfying simple constraints and observe the behavior of agents as bids and asks are exchanged over time.

In addition to reporting on the individual actions of agents, diagnostic information is also included regarding overall system status. This includes a depiction of the current active topology, machine load and ping metrics, as well as the status of each master server. The overall health of an iCDA deployment can be gauged from the information published on these pages to ensure that the underlying hardware, software, and market configuration is functioning appropriately.

4 Results and Discussion

The iCDA system described in this paper has been successfully implemented and deployed. Thorough unit and integration testing has been performed with a limited degree of load and robustness testing as well. Additional testing remains to be completed to assess and confirm the robustness of the software; issues stemming from network faults, hardware failure, heavy concurrency, and high load must be identified and resolved in order to ensure the reliability of iCDA as a potentially mission critical internet technology.

Overall, the design and architecture of the iCDA system has proven robust and amply flexible. Nonetheless, several small issues have arisen over the course of development. One particularly troublesome case occurs due to the *at most once* semantics provided by Java RMI[4]. For instance, while a remote method may return successfully from the callee's perspective, a network issue may arise while transmitting the return value back to the caller. As a result, rather than receiving the return value, the client will detect an opaque remote exception

without any indication as to whether or not the remote method was invoked. In the presence of database transactions, this becomes a significant issue.

As a remedy, all remote methods have been carefully written to be *idempotent*. Executing a remote method multiple times has no ill effect. In fact, multiple invocations are guaranteed to be operationally equivalent to a single invocation. Clients need only reissue failed remote method invocations until the method completes succesfully. In the event of multiple failures, an exponential backoff strategy is employed so as to limit network congestion; once a certain number of calls have failed, the request is simply aborted.

While this approach successfully achieves the desired degree of reliability, it also significantly weakens the network abstraction that Java RMI strives to mantain. As a result, future iterations of iCDA may very well do away with RMI altogether, replacing it with message passing or another more comprehensive technology.

5 Future Research

Building a system such as iCDA requires a considerable investment of time and effort. As such, there are many features that were originally intended for inclusion in iCDA that could not be implemented in the time allotted. Additionally, a rigorous study of the trading agent's economic efficiency as well as a careful analysis of the system's overall performance remains to be completed.

Perhaps the most important feature missing from iCDA is support for *item quantities*. As the vast majority of markets on the internet incorporate quantity in one embodiment or another, the number of potential applications supported by the system is greatly reduced. That said, iCDA has been carefully designed from the ground up to allow for the rapid introduction of item quantities into the system; in particular, the handling of clear events has been designed to

support the addition of *partial clears*– clears that do not exhaust the supplied or demanded quantity– in the future.

With the introduction of partial clears, it becomes possible for agents to recieve multiple clear events over the course of one job's lifetime. Consequently, the corresponding agent infrastructure will likely need to change both in how jobs are restored in the event of failure as well as how multiple clear events are synchronized across the system. This will constitute the majority of work required to support trading in bulk.

Another avenue worthy of investigation is the introduction of Apache Hibernate as a replacement for the existing entity framework. Hibernate is a sophisticated and well-supported Java persistence suite widely utilized by web application developers. While Hibernate does not appear to confer any advantage to the storing and querying of constraints in a database (which is, in fact, the reason Hibernate was not utilized to begin with), provided some amount of work, it should be possible to introduce these capabilities into the framework via a plugin or external module. Integration with Hibernate will endow iCDA with the ability to efficiently operate upon a wider array of different database management systems as well as to access powerful database processing tools and resources.

Finally, as alluded to earlier, a rigorous study of iCDA's economic properties and efficiency would undoubtedly prove valuable in planning future development of the system. These studies would serve to identify those components within iCDA that would most benefit from further optimization and development. With this information, iCDA can be iteratively improved until a satisfactory degree of economic and computational efficiency is achieved.

6 Conclusion

iCDA is a highly functional web-scale continuous double auction system designed to support a wide range of practical applications. While much remains to be done to fully quantify the real-world performance and economic properties of iCDA, the current system is both stable and fully operative. Personal experience interacting with the system indicates that iCDA is both reliable and remarkably adaptive to different computing environments; in particular, iCDA has been succesfully deployed on systems ranging from a single, low-end laptop to a multi-node Xen cluster provided by Linode.

The desiderata set forth for iCDA earlier in this paper have been satisfied in full. iCDA (1) presents an intuitive RESTful web service interface to the developer; (2) scales easily and seamlessly by means of a distributed master/slave architecture; (3) supports untrustworthy, naïve users via job sanitation and centralized validation; and (4) degrades predictably and gracefully under heavy usage by evenly distributing load across all servers. Indeed, iCDA is a robust and extremely useful tool for building market-based web applications.

References

[1] Albert N. Badre. Shaping web usability: interaction design in context. *Ubiquity*, 2(46):1, 2002.

[2] Dhananjay K. Gode and Shyam Sunder. Allocative efficiency of markets with zero-intelligence traders: Market as a partial substitute for individual rationality. *The Journal of Political Economy*, 101(1):119–137, February 1993.

[3] James Murty. *Programming Amazon Web Services: S3, EC2, SQS, FPS, and SimpleDB*. O'Reilly Media, 2008.

[4] Esmond Pitt and Kathleen McNiff. *Java.rmi: The Remote Method Invocation Guide*. Addison-Wesley Professional, 2001.

[5] Chris Preist. Commodity trading using an agent-based iterated double auction. In *AGENTS '99: Proceedings of the third annual conference on Autonomous Agents*, pages 131–138, New York, NY, USA, 1999. ACM.

[6] Peter R. Wurman, Michael P. Wellman, and William E. Walsh. The michigan internet auctionbot: a configurable auction server for human and software agents. In *AGENTS '98: Proceedings of the second international conference on Autonomous agents*, pages 301–308, New York, NY, USA, 1998. ACM.