An oTree-based Flexible Architecture for Financial Market Experiments*

Eric M. Aldrich†
Department of Economics
University of California, Santa Cruz

Hasan Ali Demirci‡
Department of Economics
University of California, Santa Cruz

Kristian López Vargas§
Department of Economics
University of California, Santa Cruz

March 18, 2019

Abstract

This document presents an architecture for experiments in finance. The architecture builds on oTree, a modern platform for behavioral experiments, allowing for sophisticated economic environments, market institutions, and trader strategies. The system supports both continuous- and discrete-time markets, and allows for communication latencies at time resolutions of 10-20 milliseconds. Such precise communication latencies facilitate the experimental study of high-frequency trading. The architecture also modularizes its main components, which makes the system flexible, portable, and scalable.

Keywords: Market Design, Experimental Finance, Algorithmic Trading, Laboratory Experiments, Economic Software.

*The European Research Council has provided funds for this research under the European Union's Horizon 2020 research and innovation programme (grant agreement No 741409). The authors thank James Pettit (academia.edu) for helping design the high-level architecture and for providing technical consulting and support at multiple stages of the project, Darrel Hoy and David Malec for developing the remote exchanges used in the application, and Dan Friedman, Axel Ockenfels, and Peter Cramton for invaluable feedback.

†Email: ealdrich@ucsc.edu.
‡Email: hdemirci@ucsc.edu.
§Email: kristian@ucsc.edu.

1 Introduction

Modern financial markets are evolving quickly. The complexity and speed at which they operate continues to increase as telecommunication technology improves, computing power increases, and algorithmic trading becomes prevalent. Agents' strategies are becoming more sophisticated and new market formats are emerging. In this context, first-order questions of financial market design are becoming more relevant, and new experimental tools are necessary to answer those questions.

In this paper we describe a software architecture that extends a relatively new experimental platform, called oTree, to allow for sophisticated financial environments and market formats to be studied experimentally. The markets emulated in the lab environment consist of several components of a system where oTree is the master, unifying element (and referred to as *experiment server*). Our architecture allows the experimenter to work with arbitrary market engines/mechanisms (including external exchanges, if desired) and to implement experiments in continuous-time environments. Importantly, our architecture is appropriate for environments where communication latencies are an important element of the environment (see application below).

With this architecture, a wide variety of experiments can be implemented under the oTree framework. As in standard oTree, subjects advance through pages as the session proceeds. However, within trading-day pages, they are presented with a dynamically interactive page which allows participation in an evolving financial market. The architecture contributes to existing experimental software by providing proper solutions for efficient data logging in continuous-time experiments.

In the application (see below), the architecture extends oTree with dynamic user interfaces and integrates it with external financial market exchanges. The dynamic interface enables complex decision spaces and continuous communication between participants and experiment server, *without submitting the page*. Participants are able to trade by submitting messages to the financial exchange using graphical components on the interface. Further, the architecture allows the experiment to define a set of trading algorithms for participants. Communication protocols follow real world standards utilized by the industry in communication between exchanges and traders, which allows for highly parameterized orders and prepares the architecture for large scale experimentation. Market state changes or conditions can be manipulated by sending signals to the oTree software, which are evaluated to trigger events. These events, along with any relevant data, are logged in an ordered and hierarchical fashion.

Exchange formats, messaging protocols, trader actions and trading algorithms are

extensible and easily replicable in our framework. These building blocks allow for modeling of a wide array of evolving market environments.

Our architecture facilitates experimental research in modern financial market environments. A similar approach can be taken for experiments in other areas of economics and other disciplines where interactive user interfaces and continuous-time settings are required along with complex decision spaces. Most components can be replaced with or connected to other pre-existing programs which reduces duplication of efforts in the development of experimental environments.

2 The State of Software Solutions

Although a variety of software solutions for behavioral experiments exists, zTree (Fischbacher, 1999, 2007) currently has the largest market share, due to its simplicity and reliability. zTree, however, was designed using a software paradigm of the late 1990s. It is a client-server Windows application optimized to run on a local network of a laboratory with a relatively small number of participant stations. Its emphasis on configurability naturally limits flexibility and is an obstacle for implementing richer environments, continuous-time decisions and interactions, sophisticated graphical user-interfaces, and for handling sub-second events. Furthermore, zTree is not built to support collaboration in software development. These features create barriers for extensibility, causes duplication of efforts, and limits scalability. Although some experimental software developed in the last fifteen years address a few of these issues, most solutions are closed source and limited to configuration menus.

The same is true for most current solutions that are not based on oTree. There are several highly customizable market platforms, but they are either closed source (e.g., Flex-E-Markets (Asparouhova et al., 2016)), or are restricted by the capabilities of zTree (e.g., GIMS (Palan, 2015)). There are also some solutions that allow for continuous-time games, but they do not exploit current capabilities of web browsers (e.g., ConG (Pettit et al., 2014)).

oTree (Chen et al., 2016) is an open source behavioural experimental platform, that embraces modern software development practices. The source code is in Python, a widely adopted high-level programming language, and it is created as an instance of a popular web framework called Django. In oTree, the experimenter builds the logic in a similar way that web developers build database-oriented websites. This has many advantages: (1) there are many open-source libraries that can be used in developing extensions, (2) thousands of online threads provide assistance during the development

process, and (3) oTree supports websocket technology, which allows persistent connections critical in taking advantage of modern web browser capabilities. These characteristics distinguish oTree from other experimental software platforms, making it ideal to build an extensible framework.

In a *standard* oTree experiment, data is generated as subjects make choices by interacting with web page forms and submitting choices to the web server. The content on the page is static and deployed at page load. Choices are submitted and evaluated on a page submission and stored in a relational database system. From the experimenter's perspective, this architecture allows choices at a discrete set of submission times. Although the experimenter has some control over time by implementing page timeouts, collection of continuous time choices and interactions cannot be achieved. One option for approximating continuous time games is to shorten the timeouts. But this approach is insufficient. Experiments where page submission occurs every one or two seconds are not feasible because the architecture is optimized for static websites and comes with substantial load and processing times.

Fortunately, oTree supports websocket technology (Fette and Melnikov, 2011).[1] The websocket protocol creates a continuous connection between two parties, and message exchange can be initiated by either party in an asynchronous paradigm – messages can be sent at any time during the course of a connection. This enables efficient interactions between the experiment server and clients as the interaction does not require a page submit or load, offering lower latencies. The experimenter can therefore simulate continuous time by serving a web page with dynamic content and persistent client/server connections.

Some technical details about oTree are critical in understanding our chosen architecture. oTree runs on a stateless server[2], causing it to revert to its original state after processing a user (subject) message. This imposes the requirement that all data be collected in storage and later retrieved (possibly in part) to process each request. In line with current web development practices, oTree uses a relational database management system (*rdbms*) to store data. The suggested choice is Postgres, a common database system that is known for its reliability. However, for applications that require read and write speeds at millisecond resolution, the developer faces a trade-off between relational databases and in-memory data storage. In the context of studying modern financial markets where sub-second events and reactions are common, this limitation becomes

[1] Network sockets are simply ends of networks. A machine connects to another machine's socket to send data.

[2] A web server architecture that keeps no client state in memory between requests. That is, the only input is the content of the message and the server does not have ex ante knowledge of the request.

important.

In recent years, several oTree extenions have appeared, allowing for real-time interactions (e.g., BA2, 2019; BA3, 2019). We defer a discussion of these solutions, as well as a comparison to ours, to Section 5.

3 The Architecture

The objective of our architecture is to create an experimental financial market that is flexible, portable, and scalable. The software is designed to accommodate the messaging load and data volume of a laboratory with approximately 50 participants, in an environment that allows for high-frequency trading at millisecond resolution. Distributed systems would be required in settings with hundreds or thousands of participants that trade in networks larger than a lab. The codebase is written and organized to create such a scalable application. Furthermore, to facilitate future development, all programming libraries used by the software are open source and their basic documentation is publicly available.

We summarize the architecture by tracing the flow of information during a trading session. Participants (clients) interact with oTree through a user interface, sending their choices (messages) over a websocket connection. Messages are processed by oTree, translated into a binary protocol, and sent to the (remote) exchange server over another (socket) connection between the exchange and the oTree software.

Results of the user input are recorded as experiment data by oTree and records are inserted in the relational database, using computing resources that are distinct from the experiment server's, avoiding potential performance issues from shared resources. Simultaneous with user interactions, a separate process generates exogenous events that are fed into the environment. These exogenous events contain information about the nature and state of the financial market (e.g. noise trader arrivals and the evolution of the asset value). This process also utilizes a websocket connection with oTree.

Figure 1 diagrams the architecture in more detail. Specifically, the experiment software sits at the center of the architecture, converts subjects' inputs to trader actions, and coordinates communication with the exchange server. The conventional experiment flow, where participants advance towards the end page is preserved, and essential data models are added in new modules, extending the oTree framework. These elements together allow oTree to be adapted for continuous-time trading sessions on a single page, at each round.

One of the main features of this architecture is the extension of standard oTree to

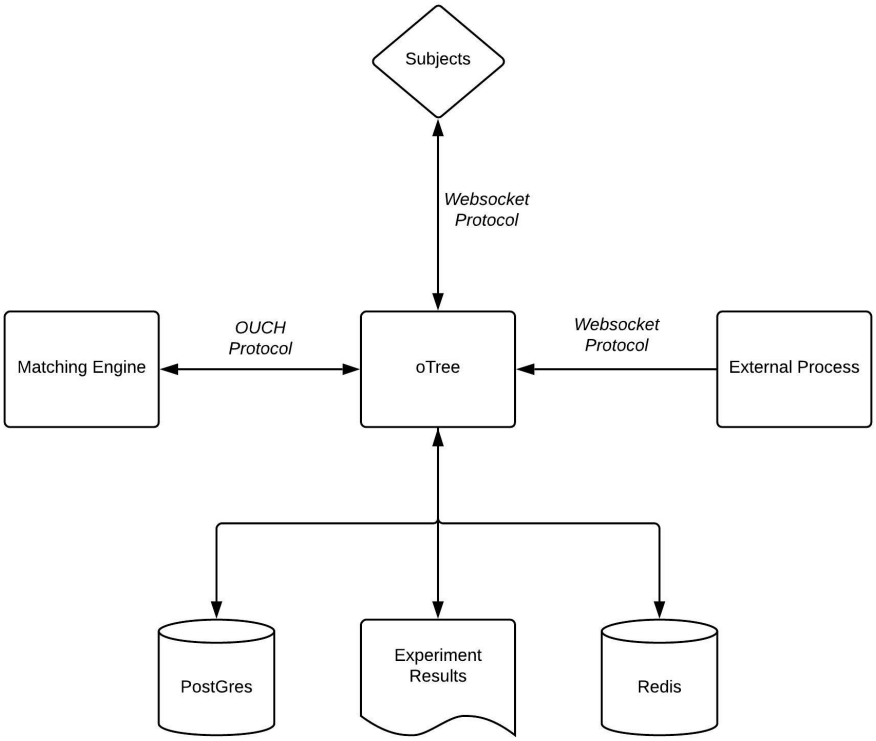

Figure 1: Information flow between main components of the architecture. Double headed arrows imply two-way messaging and single headed arrows specify one-way communication and its direction. oTree receives signals from the external processes (right arrow) and writes experiment data during the session (bottom arrow). Protocol messages are sent to and received from the matching engine (exchange, left arrow). Subjects interact with the user interface, inputting decisions and receiving market updates over the same connection (top arrow).

incorporate an evolving trading page and the ability to create a low-latency and reliable communication with a remote financial exchange. To achieve this last feature we use an in-memory database to satisfy the need for fast data storage and retrieval.

Decision spaces allow automated agents to act on players' behalf with fast-paced trading behavior. This leads to applications that are intensive in data messaging among components and sophisticated in data logging.

The trading page is composed of elements that convey updates about the market and traders' states as the underlying market evolves. The content of this page is determined by the specific experimental design, with a broad range of possibilities. The elements related to human traders' choices are managed by JavaScript code run in the participant's browser, handling the messages received from the experiment software over a websocket connection. These messages include information about updates in market conditions or changes of state of other traders. The same connection also car-

ries participants input in JSON format[3] to the experiment server, which, if necessary, hosts an agent (e.g., a trading algorithm) that acts on the trader's behalf. This agent processes and translates the trader's choices and, if necessary, triggers trading actions. Two-way messaging and near immediate updates of the user interface, occurring repeatedly over a trading session, allow the experimenter to approximate a continuously evolving financial market. For subjects participating as traders, the environment is perceived as a fully continuous-time interaction since communication latencies and the graphical representation are 10 to 100 times faster than the minimum frame rates that human visual perception regards as animated or continuous.

The interactive page which hosts the trading session is initiated and terminated by a signal sent from the experiment server. Thus, trading starts and finishes at the same time for all participants in the session. [4]

We use the same 'Player' and 'Group' data models as standard oTree, where each subject (group of players) is associated with a 'Player' ('Group') object which maintains logic to read and alter the player state in the relational database.

We have also added a 'Trader' class to allow the system to accommodate algorithmic trading strategies. There are as many 'Trader' sub types as available trading strategies in the environment; that is, a separate 'Trader' model per trading algorithm. When a 'Player' activates one algorithm on the user interface, the corresponding trader class is linked to this 'Player' until the player changes its strategy or the trading session ends. When a relevant market event takes place, it is the 'Trader' object that is notified and takes direct action accordingly. The 'Trader' class manages in-session player state and experimenters are able to design and implement arbitrary algorithms using the 'Trader' class.

3.1 Installation and Codebase Map

Installing our software assumes that the following has been previously installed: (1) Python 3.6, (2) Google Chrome, and (3) Redis and Postgres databases. A running experiment consists of several servers (or services) running at the same time and talking to each other. A step-by-step tutorial on how to install and run a test is provided in Appendix A.

The main files in the codebase of the architecture are those related to: (1) exper-

[3] JSON (Java Script Object Notation) is a standard format used for transferring data between two programs using readable key-value pairs

[4] Participants simply see an inactive trading page before and after a session is active. An example of the mechanics of the user interface will be detailed in the implementation section below.

iment configuration, (2) output, (3) trading algorithms, (4) market environment, and (5) communication between the oTree server and a remote market engine. A short description of the main files that constitute our architecture and their location in the codebase can be found in Appendix B.

3.2 Financial Exchange

The financial exchange is referred to as the exchange server. The responsibilities of an exchange in an electronic financial market can be broken down into *matching* and *messaging*. Matching involves maintaining an order book and implementing the rules of order matching. Messaging involves two-way communication with subscribers (*inbound* messages carry market participants' orders and *outbound* messages convey the exchange's responses to those orders). This communication is standardized using messaging protocols that also create the possibility for highly parameterized orders.

We use the same abstraction to separate the functionality of the financial exchange in our architecture. However, the experimental exchange has additional functions to accommodate particular features of our architecture. For example, in order to separate trading sessions, the exchange receives a signal for session start and end, which triggers a reset of the order book.

Also, since multiple groups (subsets of agents that comprise a distinct market) can be present in a session, we allow several financial exchanges to operate in parallel and independently. This design also allows for the development of a market where a single group has access to several exchanges at once. To accommodate such possibilities, we implement a system of IDs that encode the network address of each exchange, as well as (group, exchange) ID pairs. These IDs are specified in the experiment configuration files (discussed below). Most of the messaging during a session occurs between the financial exchanges and oTree. The communication between an exchange and experiment server is implemented using Transmission Control Protocol (TCP) (Postel, 1981).

A TCP connection is opened at session start and persists until the trading session ends, using the server's 'event loop' protocol (Django Software Foundation, 2018). This by-passes the default Django application logic on the experiment server, which avoids overhead induced by using websockets.

In contrast, low-latency communication between the experiment server and subject browsers is handled by websockets. Subject messages are subsequently encoded in the OUCH protocol (a binary messaging specification optimized for bandwidth and latency, used by Nasdaq) by the oTree server and sent to the exchange.

The *financial exchange* is a separate application that provides the functionality

explained above, along with support for the connection type to talk to the *experiment server* over a preset protocol. The number of exchanges and arrangement of group-exchange pairs is arbitrary. A set of open source financial exchanges are available in the LEEPS Lab GitHub public web page (LEEPS Lab, 2019).

3.3 Economic Environments and Exogenous Events

Exogenous processes are key components of experiments as they characterize specific instances of the economic environment. Examples of exogenous events are asset value changes, signal realization, noise trader arrivals, changes in trader inventories, etc. These events trigger specific responses and actions by traders' algorithms or market exchanges.

Our architecture can incorporate exogenous processes in different manners. Since oTree supports *http* and *websocket* connections, the messages from such processes can be sent over a network or tied to a live feed. Alternatively, oTree can read an parse files that contain predetermined signal values.[5] In the application in Section 4, oTree reads exogenous events from CSV files. In such cases, where exogenous events are encoded in text files, it is possible to specify the files in the configuration file (see below) and allow oTree to manage the process.

In all cases, the experimenter can automate the delivery of exogenous events so that they start and stop with the trading session. This can be done using function calls in the application code. Once the process is launched, messaging between the external processes and oTree is one-way (from external process to experiment server).

3.4 Experiment Data

In the typical flow of an oTree experiment, data is generated by users filling forms and submitting the resulting web pages. However, in richer settings (e.g., continuous-time financial markets) most data of interest is generated on a single, interactive page. The amount of data is significantly larger and more granular than in a typical experiment, as it provides context for each recorded event. As a reference, an application built with this architecture in the LEEPS Lab, produces an average of 14,000 data rows per trading period for a group of six traders. The amount of data might vary widely depending on the environment of the experiment.

In order to efficiently represent and store the data, we developed a JSON logging

[5] In future releases, we plan to add controls in the oTree experimenter dashboard (admin panel) to trigger exogenous processes manually.

module with a hierarchical structure encoding entries based on event types. This data format can be generalized and adopted by other experiments. Listing 1 illustrates an example of the data structure for an execution message that is received from the exchange.

Listing 1: A JSON formatted order execution record. Each key-value pair specifies a field of interest to the experimenter. A value can be a set of key-value pairs, allowing for nested structures. The hierarchical structure provides easy look-ups and organization for subsequent analysis.

```
{
    'time in session':  00:02.00,
    'type': 'execution',
    'source': 'exchange',
    'group': 1,
    'context': {
        'player id': 1,
        'order token' : '0A1',
        'price': 100,
        'timestamp': 62380942868000
    }
}
```

The experimenter chooses a set of interactions to capture from the market by making logging module calls in the application logic and providing context in the parameters. These logs are briefly kept in memory and are written to the primary database by a separate process.

Information about traders' state (e.g. current profit, trader role, order information) and the aggregate state for a group of traders is often very important to the experiment and the experimenters. This data is frequently updated throughout the session, as trader states evolve. Although this data can be regarded as transitory, each state forms the basis of the remaining experimental data and provides context for oTree to interpret and process incoming messages. oTree must read this information from a database following the stateless server architecture mentioned above. While this data is small in size, it is to be read and modified extensively throughout a session, possibly with sub-second frequencies. To handle these data, we chose to use Redis, an open-source, in-memory data structure store (Sanfilippo, 2019).

Standard oTree experiments use PostgreSQL, an open-source relational database

management system.[6] Although Postgres is optimized for reading and writing from disk and can keep frequently accessed data in memory, it suffers from large overhead related to the encoding of in-memory data structures to disk writable form (Kleppmann, 2017). This results in read and write latencies of several milliseconds. In the context of environments where controlling latencies is an essential part of the experiment (such as in the study of high-frequency trading), this overhead becomes an important bottleneck – the application performance is slowed down by components that do not have enough resources.

Redis has the advantage of storing data in-memory and offers read and write times of less than half of a millisecond (as tested in one of our applications that required keeping all traders' information).

3.5 Session Configuration

A custom session configuration in oTree is defined in a Python dictionary, whose representation is identical to a JSON formatted string, and is called by a settings module in the application. A configuration for a financial market experiment may include many parameters to tune the market and our architecture allows for a large set of configuration files in a single environment. In order to separate programming from configuration, and to accommodate articulated configurations, we developed a module for reading and validating configuration files. This module accepts YAML[7] formatted configuration files and maps parameters in the files to oTree's session configuration. At server start, configuration files located in a designated directory are read and made available on the session's configuration page. This approach enables experimenters to arrange configurations by creating simple files (see Listing 2, for an example).

Listing 2: YAML configuration file. each key-value pair refers to a parameter that configures the session.

```
rket_parameters:
  initial_price: 10
  exchange_format: 'CDA'
ssion_parameters:
  trade_duration: 90
  number_of_players-per-group: 5
```

[6]Database management systems manipulate, store and retrieve data. In relational databases, data is organized into relations (SQL tables), based on a model suggested by Codd (1970).

[7]YAML is a data serialization standard that is readable and easy of edit, and whose structure is defined by indentation.

4 An Implementation

This section describes how our architecture implements Aldrich and López Vargas (2018) and more general experiments. The experiment of Aldrich and López Vargas (2018) was implemented with software that inspired the architecture we present in this paper. One of the important differences is that their software is not based on oTree. As discussed above, using oTree as the core of the experiment server has many development and deployment advantages.

A version of our oTree-based architecture that replicates Aldrich and López Vargas (2018) is available on Github (LEEPS Lab, 2018), and was created specifically as a companion to this paper. We suggest, however, that interested researchers clone (or download) and work with the more current version of our software (LEEPS Lab, 2019), as it has been optimized and made more stable.

Although our framework can implement more general experiments, we now describe how it replicates the environment in Aldrich and López Vargas (2018). That paper compares two financial market formats, the Continuous Double Auction (CDA) and the Frequent Batch Auction (FBA), in the presence of high-frequency trading. The design is based on the theoretical model presented in Budish et al. (2015), where a single asset is traded on a single exchange. Traders submit commitments to buy or sell the asset in the form of limit orders and market orders. Two exogenous processes generate incentives to trade in this environment: changes in the publicly-observed fundamental value (which follows a Poisson jump process) and the arrival of market orders from automated investors (noise traders) at random times. Purchases (sales) are simultaneously liquidated (purchased back) at the fundamental value, allowing traders to book profits or losses instantaneously.

Human traders choose to stay out of the market or two enter the market using one of two algorithms: (1) *market maker*, or (2) *snipers*. Their choices can be revised continuously throughout the trading session. If the market maker role is chosen, the trader must also specify a *spread* around the asset value at which they post bids to buy and offers to sell.[8] Market makers earn profit when the exchange matches them with a liquidity taking counterpart (mostly noise traders/investors). Snipers attempt to exploit temporarily mispriced limit orders by transacting with them at the time of a jump in the asset value, before a maker's algorithm is able to update its orders. Finally,

[8]Buys and sells are, by construction, placed symmetrically below and above the value, respectively.

market makers and snipers also decide whether to subscribe (for a pre-specified flow cost) to a technology that reduces messaging latency to the financial exchange.

The experiment was designed such that each group is associated with a single financial exchange. That is, each group of players forms a market that operates over several distinct trading periods (i.e. matching is fixed). Each exchange server maintains an order book for its associated group during a trading period. Although market formats in Aldrich and López Vargas (2018) are only varied over multi-period experimental sessions, this is not a restriction in the oTree-based architecture, where different market formats can be deployed in a single session.

Subjects' engage in trading through the user interface (UI) containing multiple real-time graphical components conveying relevant information on the state of the market. Trader controls are also graphic-oriented, as it can be seen in Figure 2. Buttons and other position-aware components accept and channel subject input to the experiment server. Participants are able to change roles or their spread (for market makers) as well as subscribe/unsubscribe to fast-communication technology at any moment during the trading session.

When a participant chooses a trading algorithm, a specific oTree agent (called 'Trader') is activated. The trading logic for different algorithms is inserted in the oTree application (see Section 3). When a triggering event occurs, this agent creates the appropriate buy or sell order and, using the associated binary protocol, sends it to the exchange server. The exchange processes the messages, makes necessary updates to the order book, sends back confirmation messages, and potentially sends execution messages (if the actions result is an immediate transaction).

Similarly, the exogenous processes that alter the market state – the fundamental value and the arrival of noise investors – are embedded in the experiment server. The financial exchange is a separate application that communicates to the experiment server using protocols and channels that are described in Section 3.

The user interface (trading page) presents four streams of data, each over a separate interactive component on the interface. These components update as the market evolves. Every component can be seen as a node on the page, managed and defined with JavaScript code running in the subject's browser. Figure 2 displays an example of the user interface with an exchange running the CDA format.

The information box placed at the top of the UI presents information related to the market and user status and a real-time graph on the left shows the evolution of individual profits. Subjects choose roles by clicking buttons located in the 'choice box' on the right. The 'spread graph' informs market makers about their current spread using

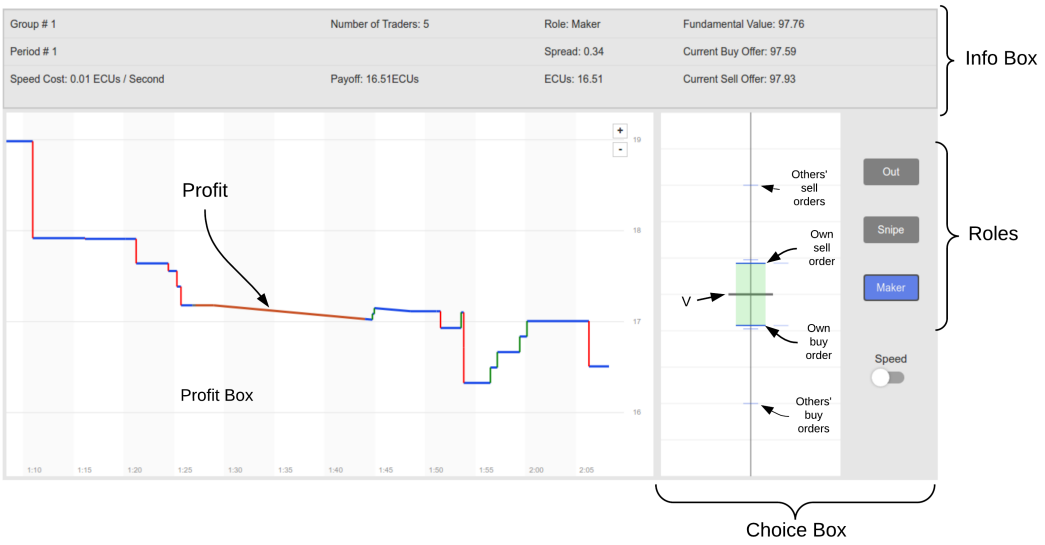

Figure 2: Traders' user interface in Aldrich and López Vargas (2018). The information box (top) provides information about the market state, such as the asset value and number of different types of active traders, as well as information about the individual player, such as current bid and offer. Subjects' adjust choices anytime during the session, via the choice box (bottom right): buttons labeled as *maker*, *sniper* change the user role and, hence, the underlying trader model, and a click on the spread graph adjusts the *maker's* spread according to vertical position of the input. The *Out* button sends a signal to cancel any active orders and deactivates trading algorithms. The box (bottom left) displays subjects' accumulated profit at each moment of time.

horizontal ticks, which they can update at any time by clicking on the spread graph. The 'spread graph' is the core of the UI as it displays order book information. Further, it displays asset value jumps and investor arrivals in an animated fashion. Specifically, the oTree experiment server translates messages from the exogenous processes and sends them over a websocket connection to the browser. The browser dispatches the messages to the corresponding UI and, in the case of the spread graph, depicts the associated event.

User input is parsed and passed to the experiment server, which triggers all processes that correspond to a subject's choice. For example, if a subject chooses to re-price existing orders by clicking on the spread graph, the browser passes this information to the experiment server, which updates the subject's associated trading algorithm, and subsequently sends appropriate order-update messages to the exchange. When the exchange sends order confirmations, the experiment server broadcasts this information to all traders' browsers, resulting in the display of such information on all spread graphs.

Trading algorithms differ among trading roles (maker, sniper) and exchange formats (FBA, CDA). Accordingly, four different 'Trader' classes exist for each role and market format. Upon receiving input that requires a trading action, the correct trader logic is activated. For example, when a user clicks the 'maker' button in the 'choice box', the experiment server receives a message from the browser detailing the user input. This message is then routed to a 'Trader' class that corresponds to 'maker' for the given exchange format. That trader class encodes two 'enter order' messages using the OUCH protocol, the inbound message specification used by Nasdaq. Finally, the OUCH messages are sent to the financial exchange.

One instance of an exchange is initiated for each group prior to the experiment and resets the order book at the beginning of each trading period. Every exchange stores detailed logs of the order book, which can be also be treated as experiment data and an instrument for error checking. We use the same exchange servers as Aldrich and López Vargas (2018), which were developed under a separate software project and are meant to be adapted for different experiments[9]. The exchanges, by design, are anaware of the experiment server or any other part of the system and are only responsible for (1) processing messages in OUCH protocol and (2) maintaining and matching orders. This design results in exchanges that are general, robust and portable. The exchange software is open source and can be adapted for experiments with different underlying environments. Connections with exchange servers can be established by address (host

[9]These python applications can be found as a sub-repo of the LEEPS Lab project (LEEPS Lab, 2019). The initial contributors are Darrel Hoy (TREMOR Inc.) and David Malec (University of Maryland and Cramton Associates LLC).

and port), offering flexibility to talk to exchanges not hosted on a admin workstation or lab server, as long as they are open to internet.

Exogenous processes, such as investor arrivals and changes in asset values, are initiated by external processes that transmit messages to experiment server. For each trading round, the experimenter must provide configuration CSV files containing realizations of the exogenous processes for each group. At session start, all files relevant to the session are read and merged into a dataset indexed by event time. A message to the experiment server that represents an exogenous event carries the details of the event: the buy/sell limit price for an investor arrival and the direction and magnitude for a change in the asset value. Upon receipt of the event, the experiment server triggers conditional actions: investor arrivals result in market orders (on behalf of the investor) that oTree stages for transmission to the exchange and asset value changes are passed to subjects' trading algorithms, which potentially lead to further actions on behalf of individual traders.

5 Discussion

Several options extending oTree to allow for real-time interactions have appeared in recent years. BA2 (2019); BA3 (2019) are two examples. In this section we provide a broad comparison of these alternatives with ours.

BA2 (2019) illustrates how websockets can be used to extend oTree for real-time interaction among players and implements a basic double auction market. The paper describes the real-time oTree extension in detail and provides a useful explanation of Django Channels for non-experts. Similarly, BA3 (2019) details the advantages of Django Channels in oTree-based experiments. That paper illustrates how its oTree extension works in three commonly-used and important applications and the presentation of the architecture (*consumers* and *routing* files) is particularly useful for researchers that want to use this approach. A valuable contribution of this extension is that it contains a general, configurable real effort task application, which can be used in a wide range of experiment applications.[10] Overall, the objectives of these applications are similar: both extend oTree with websockets in order to design traditional (relatively simple) real-time interaction experiments.[11]

[10] As in our software, these two software alternatives are open-source and therefore suitable to be reused and extended by other researchers.

[11] A pre-oTree, browser-based experiment platform that allowed for real-time interactions was provided by the Redwood Framework, developed at the LEEPS Lab between 2013 and 2017 (LEEPS Lab, 2013-2017).

Although our software also allows for interaction in real-time, the architecture is not designed to accommodate standard experiments. Our purpose is to design a framework for nearly arbitrary financial market experiments. In particular, we are fundamentally concerned with controlling communication latencies, allowing complex strategy spaces, and the possibility of algorithmic trading. These priorities call for the modularization of our software components. Accordingly, our software consists of a network of components in which human traders interact with a market engine, typically using trader algorithms.

Our architecture is extensible and scalable, and individual components can be modified to include additional realism. For example, the market engines in our implementation closely emulate real-word financial exchanges and therefore can be re-used in nearly any realistic financial experiment. This modularization also allows us to create experiment environments that are intensive in data and messaging.[12]

Another difference with the two papers mentioned above relies on the link between data/message intensity and constraints induced by the version of Django Channels that oTree uses (version 0.17.3). To our knowledge, using Django Channels to send messages between the oTree server and clients' browsers (using JavaScript libraries) takes between 10-200 milliseconds. In messaging intense experiments, e.g. Aldrich and López Vargas (2018), such a constraint may not provide satisfactory control over communication latencies.

Whether our software or either solution provided by BA2 (2019) and BA3 (2019) is most appropriate to implement a specific financial experiment will depend on the nature of the project in mind. Solutions in BA2 (2019) and BA3 (2019), we believe, are suitable for market games that require real-time interactions but are not highly complex and do not necessitate tight control over communication latency. Our architecture, in contrast, is suitable for studying more complex financial markets where controlling communication latency and allowing for complex strategies (presumably algorithmic trading) are key elements of the environment.

[12]Each implementation can vary its approach to handle data and data stores when necessary.

References

Aldrich, E. M. and López Vargas, K. (2018), "Experiments in High-Frequency Trading: Testing the Frequent Batch Auction," .

Asparouhova, E., Bossaerts, P., and Nielsen, J. (2016), "Flex-E-Markets," (accessed 2018-04-15).

BA2, B. (2019), "Real Time Interactions in oTree using Django Channels: Auctions and Real Effort Tasks," *manuscript*.

BA3, B. (2019), "oTree: Implementing Websockets to Allow for Real-Time Interactions - A Continuous Double Auction as First Application," *manuscript*.

Budish, E., Cramton, P., and Shim, J. (2015), "The high-frequency trading arms race: Frequent batch auctions as a market design response," *The Quarterly Journal of Economics*, 130, 1547–1621.

Chen, D. L., Schonger, M., and Wickens, C. (2016), "oTree An open-source platform for laboratory, online, and field experiments," *Journal of Behavioral and Experimental Finance*, 9, 88–97.

Codd, E. F. (1970), "A Relational Model Of Data for Large Shared Data Banks," *Communications of the ACM*, 13, 377—-387.

Django Software Foundation (2018), "Daphne," https://github.com/django/daphne, (accessed 2019-01-15).

Fette, I. and Melnikov, A. (2011), "The WebSocket Protocol," RFC 6455, RFC Editor, http://www.rfc-editor.org/rfc/rfc6455.txt.

Fischbacher, U. (1999), *Z-Tree 1.1. 0: Zurich Toolbox for Readymade Economic Experiments: Experimenter's Manual*, Institute for Empirical Research in Economics, University of Zurich.

— (2007), "z-Tree: Zurich toolbox for ready-made economic experiments," *Experimental economics*, 10, 171–178.

Kleppmann, M. (2017), *Designing data-intensive applications: The big ideas behind reliable, scalable, and maintainable systems*, " O'Reilly Media, Inc.".

LEEPS Lab (2013-2017), "Source Code and Documentation for LEEPS Lab Redwood Software," https://github.com/Leeps-Lab/RedwoodFramework.

— (2018), "Source Code for LEEPS' oTree-based High Frequency Trading Experiments (as of November 2018)," https://github.com/Leeps-Lab/high_frequency_trading/tree/version_as_of_cologne_pilot".

— (2019), "Source Code for LEEPS' oTree-based High Frequency Trading Experiments," https://github.com/Leeps-Lab/high_frequency_trading.

Palan, S. (2015), "GIMS—Software for asset market experiments," *Journal of behavioral and experimental finance*, 5, 1–14.

Pettit, J., Friedman, D., Kephart, C., and Oprea, R. (2014), "Software for continuous game experiments," *Experimental Economics*, 17, 631–648.

Postel, J. (1981), "Transmission Control Protocol," STD 7, RFC Editor, http://www.rfc-editor.org/rfc/rfc793.txt.

Sanfilippo, S. (2019), "Redis," https://redis.io/, https://github.com/antirez/redis, (accessed 2018-10-15).

Appendices

A Setup Procedure

In this appendix we provide two alternatives to run a demo of our software. The first (and simpler) uses a Linux virtual machine. The second option is the full installation of our software and all software dependencies.

A.1 Virtual Machine via Vagrant

1. Download and install VirtualBox (https://www.virtualbox.org/).

   - Note: VirtualBox installation may fail for Mac OS 10.14 Mojave. If so, open System Preferences after the failure and navigate to 'Security & Privacy'. Under the 'General' tab, select the lock icon and provide an administrator password in order to make changes. Finally, select 'Allow' next to the message "System software from developer 'Oracle America, Inc' was blocked from loading" and repeat the installation.

2. Download and install Vagrant (https://www.vagrantup.com/).

3. Download the file `vagrant.zip` from our project repo (https://github.com/Leeps-Lab/high_frequency_trading/tree/master/vagrant) and extract its contents in a directory.

4. At the command line, navigate to the directory with the extracted contents of `'vagrant.zip'` and issue the command

   ```
   # vagrant up
   ```

   Vagrant will download and configure everything you need to run the demo program (this could take few minutes).

5. Open a Chrome browser and navigate to address http://192.168.33.10:8000.

A.2 Full Installation

A.2.1 Prerequisites

The instructions below work for Linux and Mac OS operating systems. In principle, it should possible to install our software on Windows, but it has not been tested. In addition, the instructions assume the reader has installed the following open-source software:

1. Python 3.6
2. pip3: Python3's package manager
3. Redis (with command line interface): an open source, in-memory data structure store (https://redis.io).
4. PostgreSQL: an open source object-relational database system (https://www.postgresql.org)
5. Google Chrome browser.

A.2.2 Step-by-step Installation and Demo Run

1. Open four command-line shells/terminals.
2. Create a virtual environment. You will install a slightly modified version of oTree in this new environment. A virtual environment will keep this version separate from the oTree version you might be already using. If you have a version of oTree installed in your computer and do not use a virtual environment to do this step, you will overwrite your current oTree installation.

   In terminal 1, make sure to have virtualenv installed by checking the version.

   ```
   # virtualenv --version
   ```

   The version for `virtualenv` should be printed on console. If it does not, then run:

   ```
   # pip3 install virtualenv
   ```

Then run:

```
# mkdir otree_hft_env
# virtualenv -p python3.6 otree_hft_env
```

3. Activate the virtual environment (still in Terminal 1).

   For mac and linux

   ```
   # source otree_hft_env/bin/activate
   ```

4. Clone the software repository, navigate to the cloned directory, and install dependencies (Terminal 1).

   ```
   # git clone https://github.com/Leeps-Lab/high_frequency_trading.git
   # cd high_frequency_trading
   # pip3 install -r requirements.txt
   ```

5. In Terminal 2, navigate to the sub-directory `exchange_server` of the cloned repository and download updates for the exchange server.

   ```
   # cd exchange_server
   # git submodule init
   # git submodule update
   ```

   Note: the exchange server has its own repository and, for convenience, our main software repository includes the exchange server libraries as sub-repositories.

6. Still in Terminal 2, install dependencies and run a CDA exchange instance.

   ```
   # pip install -r requirements.txt
   # python3 run_exchange_server.py --host 0.0.0.0 --port 9001
       --debug --mechanism cda
   ```

Three time-stamped lines, similar to those below, should be printed to the screen.

```
[14:45:00.803] Using selector: KqueueSelector
[14:45:00.803] DEBUG [root.\_init\_:35] Initializing exchange
[14:45:00.803] INFO [root.register_listener:112] added listener 0
```

7. Return to Terminal 1, reset the database and copy static files.

```
# otree resetdb
# otree collectstatic
```

8. In Terminal 1, launch the oTree server.

```
# otree runhftserver
```

9. In Terminal 3, determine if Redis is running:

```
# redis-cli ping
```

If '`PONG`' is printed to the screen, Redis is running. If not, issue the command:

```
# redis-server
```

10. In Terminal 4, navigate to the virtual environment directory, '`otree_hft_env`', and repeat Step 3 (activate the virtual environment). Navigate to repository directory, '`high_frequency_trading`', and start the following background process.

```
# cd high_frequency_trading
# otree run_huey
```

11. Open your Chrome browser and navigate to `localhost: 8000`. Select 'demo session' and follow the on-screen instructions to launch clients' (traders') screens as tabs in the same browser.

An up-to-date version of this tutorial can be found in the 'readme.rst' file of the main repository of this project (LEEPS Lab, 2019).

B Codebase Main Files

Below, we provide a list of the main files used in our architecture, their function, and their location in the codebase. These files correspond to those in our software repository at the time of this writing and may change in the future. The version of the repository associated with this paper can be accessed at https://github.com/Leeps-Lab/high_frequency_trading/tree/JBEF.

1. **Configuration files:** The main experiment configuration files are placed in `session_config /session_configs` folder. These configuration files are in YAML format and contain the parameters for the session (market format, market sizes, input files paths for exogenous processes, etc). YAML is used because it allows for human-readable configuration files that are highly self explanatory. Paths provided in this file are relative to the root directory.

2. **Output:** Database models for the experiment data and other outputting procedures are contained in the file `'output.py'` (located in the root of the oTree project). Experiment results are downloaded in JSON format with a `.txt` extension into the `results` sub-directory (located in the root of the oTree project). The results files are tagged with the date, a session identifier, and a market identifier.

3. **Trading Algorithms:** The trading algorithms or bots are in the file `'trader.py'` which handles trading on behalf of the human participants (located in the root of the oTree project).

4. **Market environment:** Several of the key components of the market environment – e.g., data models and code that handles the interaction with the exchange server – is contained in the `'market.py'` file (located in the root of the oTree project).

5. **Communication with exchange server:** The code that handles communication between oTree and the exchange (e.g., opening a TCP socket with a remote server) is contained in the file `'exchange.py'` (located in the root of the oTree project).

6. **Communication protocol with exchange:** All translation procedures for OUCH messages between the exchange and oTree are in files `'translator.py'` (located in the root of the oTree project) and `'OuchServer'` (located the repository sub-directory `'exchange_server'`).