CS1951k: Trading Platform

Introduction

TradingPlatform (TP) is a client-server model designed to support the creation of trading environments for autonomous agents. TradingPlatform utilizes KryoNet to mitigate the networking burden in designing trading environments.

One of our goals in developing TP is to enable researchers to create a more consistent -- yet at the same time more diverse and more robust -- array of trading agent games. We are testing out its viability in CS1951k and CS2951z, by using it to create three types of games: various toy prediction market games, an ad exchange game, and a wireless-spectrum, combinatorial auction game.

Trading Environments

A trading environment consists of a set of markets together with a set of agents. A market consists of a mechanism, one or more tradeables (goods, securities, etc.), a current state, and a history, often in the form of a ledger that records transactions. Associated with each agent is some private information, as well as an account, which records their bank balance and holdings.

Note that TP is flexible, in that it does not bind market mechanisms to tradeables. On the contrary, anything tradeable can be bought and sold using any market mechanism. So you can design a market simulation in which agents buy and sell copies of a painting, or you could design a market simulation in which agents trade shares of Google stock.

Market Simulation

A market simulation proceeds as follows:

1. Markets open with the server sending all agents an initial endowment, which consists of an initial bank balance and initial holdings. The server also sends agents their private information, if any, at this time.

2. For each open market, the server then sends all agents a TradeRequest, which describes the current market state: quotes for one-sided markets, and orderBooks for two-sided markets. Whenever an agent wants, it can respond to the TradeRequest by constructing and sending the server a BidBundle (using the methods bid, buy, sell, etc.) for that market.

3. Whenever the server receives a valid BidBundle, the corresponding market handles it. If a trade clears, depending on the market’s rule, the server can send a MarketUpdate (i.e., an updated ledger) to all agents and a BankUpdate to the agents directly involved in the trade; if not, the market updates its current state.
4. The server then sends a new `TradeRequest` to all agents. (It has to send a new request to all agents, because the market state has changed.)

5. The markets can also send agents `GameReports` (e.g., summary statistics) at other times. For example, in a simulation of two sequential second-price auctions, a `GameReports` might be sent between the two.

6. This cycle repeats until all markets close, at which point a final `GameReport` and a closing message is sent to all agents.

### Market Mechanisms

TP was designed to support a variety of market mechanisms, including the following:

1. **One-sided Auctions**: These auctions involve a central auctioneer (server), and decentralized bidders (clients). The auctioneer is offering either to buy or sell (reverse) goods. The four standard one-sided auction designs (1st price, 2nd price, English, and Dutch) are built in. The one-sided auction interface is versatile, and intended to model any one-sided combinatorial auction design.

2. **Two-sided Auctions (i.e., Exchanges)**: These auctions involve a market maker (server), as well as decentralized buyers and sellers (clients). The market maker may be willing to take a loss to facilitate trades. Continuous double auctions (CDAs), call markets, and prediction markets with a logarithmic market scoring rule (LMSR) are built in. The two-sided auction interface is versatile, and intended to model any combinatorial exchange.

3. **Direct Negotiation**: Agents also have the option of negotiating directly with one another (i.e., there is no center/market maker), as in a bargaining context.

### Market Mechanism Interfaces

In addition, market designs are defined by their allocation and payment rules, as well as their information-revelation policies.

1. **One-sided Auction** - all one-sided mechanisms implement the `OneSidedAuction` interface. This interface exposes a method for getting the current market state (a quote), as well as a bid method that constructs a `BidBundle` and sends it to the server. It also exposes a method for getting details about the market’s `Tradeable`. Finally, `OneSidedAuction` exposes a mechanism (allocation and payment rule),

   a. **SealedBid** - the `SealedBid` one-sided auction is a built-in implementation of the `OneSidedAuction` interface. It is parameterized to be 1st price or 2nd price, and forward or reverse.

   b. **OpenOutcry** - the `OpenOutcry` one-sided auction is a built-in implementation of the `OneSidedAuction` interface. It is parameterized to be ascending or descending, and forward or reverse. It is also parameterized by activity rules. One example of an activity rule is: agents must respond to every single `TradeRequest`; without such an activity rule, agents might only respond sporadically.
2. **Two-sided Auction** - all two-sided market mechanisms implement the `TwoSidedAuction` interface. This interface exposes a method for getting the current market state (the `orderBook`), as well as `buy` and `sell` methods that construct a `BidBundle` and send it to the server. It also exposes a method for getting details about the market's `Tradeable`. Finally, `TwoSidedAuction` exposes a mechanism (allocation and payment rule).

   a. **CDA** - a continuous double auction allows agents to post offer prices and quantities of `Tradeables` that they would like to buy or sell. Whenever two agents’ postings match, meaning they satisfy one another’s offers, the mechanism performs the trade, by reallocating dollars and tradeables.

   b. **Call Market** - a call market accepts postings like a CDA but clears periodically, setting a price in each round that clears the highest quantity of `Tradeables` possible. This means that some agents may pay less than they were willing to, or receive more than they requested, if a higher price clears a higher quantity. Orders that do not fully clear during a given round may carry over to the next round, or they may be cancelled.

   c. **Market scoring** - a scoring rule in a forward market is a market maker that charges buyers the difference between the cost of all shares after they transact and the cost before; in a reverse market, the market maker pays sellers this difference. The market clears continuously, and traders do not need a counterparty to transact with, as the market produces the shares being bought and consumes the shares being sold.

   d. **LMSR** - the logarithmic market scoring rule is a particular form of market scoring in which the formula for pricing shares is a logarithmic function of the number of outstanding shares in the market.

3. **Direct Negotiation** -- in a bargaining scenario, in which the center serves merely to facilitate inter-agent direct trading, agents send each other offers via the center/server, who validates all offers before forwarding them to their intended recipients.

   a. The `DirectNegotiation` interface exposes a method for getting the current market state (the current offers), as well as `accept`, `reject`, and `partialAccept` methods that construct a `BidBundle` and send it to the server. It also exposes a method for getting details about the market’s `Tradeable`. Finally, `DirectNegotiation` exposes a matching mechanism, for determining when offers match.

---

**Bid Bundle Interface**

The `BidBundle` interface permits the creation of an arbitrary data structure in which to hold bids (and asks). The built-in `BidBundle` is called `SimpleBidBundle`, and holds one double.

Agents do not need to know what `BidBundle` is used because the auctions expose methods that take in the required information. For example, the `SealedBid` and `OpenOutcry` auctions expose a method called `bid` that takes in a double (and possibly a quantity as well) and
constructs a `BidBundle`. Likewise, the CDA, CallMarket, MarketScoring, and LMSR markets expose methods called `buy` and `sell` that take in doubles (and possibly quantities as well) and construct `BidBundles`.

**Tradeables**

Tradeables are intended to be flexible enough to model any type of good or financial security. Good is a built-in class that implements `ITradeable`. Security is a built-in class that implements persistent financial instruments. Contract is a built-in subclass of Security that implements financial instruments that expire. A market simulation of the stock market would use securities, while a simulation of prediction markets would use contracts.

**Tradeable Interface**

The Tradeable interface is implemented by everything that is bought and sold, including tangible goods, like paintings, as well as securities and contracts.

4. **Good** - Goods specify their market mechanism, a price, a quantity, and an owner (either an an agent or the market mechanism).
5. **Security** - Securities specify their market mechanism, a start time, a quantity, and a price. Securities represent financial commodities, such as publicly-traded stocks.
6. **Contract** - Contracts are securities, with an expiration date, such as those traded in prediction markets. What happens when a contract expires depends on the StateOfTheWorld, which is usually simulated by the server (but could just as well be the outcome of some external event: e.g., a presidential election).

**Agents**

The Agent abstract class is the starting point for implementing an autonomous agent in any environment. It handles all networking and concurrency, and exposes only the methods needed to receive acknowledgement, market updates, account updates, and trading opportunities.

Agents are exposed to client-side versions of markets through MarketWrappers: e.g., a OneSidedWrapper, a TwoSidedWrapper, or a NegotiationWrapper. These wrappers expose the current state of a market, as well as `bid`, `buy`, and `sell` functionality. Agents can then simply call `bid`, `buy`, or `sell` on a Wrapper, and the Wrapper will handle all communication with the server.

To construct an Agent, you should supply the host and port of the server that your agent should connect to. Upon construction, an agent’s private information is initialized to a value received from the server. Likewise, for the agent’s account balance; throughout the simulation, the server provides automatic updates to the agent’s account balance whenever it changes.
Messages

TP has a simple set of messages that agents use to communicate with the server. The Agent superclass is preconfigured to pass messages to the appropriate handlers, so agents do not need to think about message passing. They simply interact with the market objects they receive from the server, which handle the communication for them.

Agents can trade in markets at any time, asynchronously, by invoking the market wrapper’s trade methods:

1. onMechanismUpdate([Mechanism]Wrapper [mechanism]Wrapper) - the server sends a Wrapper exposing a market’s functionality, and containing information about a market that utilizes the [Mechanism] rule. For example, a OneSidedAuction is passed into either onSealedBid or onOpenOutcry; likewise, a TwoSidedAuction is passed into either: onCDA, onCallMarket, onMarketScoring, or onLMSR.

The following messages from the server do not require a response from the Agent.

1. onAcknowledgement(Ack message) - this message notifies an agent that a message was received from them by the server.
2. onBankUpdate(BankUpdate bankUpdate) - whenever an agent’s account balance changes, the server sends a message containing the old Account object and the new Account object.
3. onMarketUpdate(MarketUpdate marketUpdate) - whenever the simulation wants to send out a report summarizing information about one (or several) market(s), it can compile a market report, and send around a market update. The information provided in this report depends on the details of the simulation.

API for Prediction Market Bots

You will be asked to implement onContinuousDoubleAuction(UnitCDAWrapper wrapper). This is a subclass of CDAWrapper that only trades in unit quantities and only accepts integer prices.

UnitCDAWrapper:

1. buy(Agent agent, int price) - submits a buy order for one share at this price
2. sell(Agent agent, int price) - submits a sell order for one share at this price
3. cancel(Agent agent, boolean buy, int price) - cancels an outstanding order that matches buy and price if one exists.
4. getBuyBook() : Map<Double,Double> - returns the current buy side order book that maps prices to quantities. For example, 1:5 means that there are 5 outstanding buy orders at price 1. The prices and quantities will only be integers in this game.
5. `getSellBook()` : Map<Double,Double> - returns the current sell side order book that maps prices to quantities.
6. `getTradeableType` : FullType - returns the type of the tradeable being transacted. For this assignment, it will either be PredictionYes or PredictionNo.
7. `getLedger()` : Ledger - returns the list of Transactions that have occurred while this market is running. This list is anonymized.

**Server**

There is a `Lab{3,4}Server` included with the `TradingPlatform.jar` that will run a market matching the instructions for each of those two respective labs. To start the server, in your `main` method, invoke `new Lab{3,4}Server(<port>)`. You can also run these servers with port 2121 by running `Lab{3,4}Server` from the `TradingPlatform.jar` in Eclipse.

**Agent**

There is a `Lab{3,4}Agent` included with the `TradingPlatform.jar` that will run a market matching the instructions for each of those three respective labs. To start the server, in a `main` method, invoke `new Lab{3,4}Server(<port>)`. You can also run these servers with port 2121 by running `Lab{3,4,5}Server` from the `TradingPlatform.jar` in Eclipse.

In order to run your own agent, simply construct it inside a `main` method and then loop indefinitely (e.g., `while(true) {}`). You can copy and paste this functionality from your lab code, or use the following (which is nearly identical):

```java
public static void main(String[] args) throws AgentCreationException {
    new MyAgent("localhost", 2121);
    while(true) {}
}
```