JCAT: The Software Platform for CAT Games

Based on JCAT *0.12*

Version 1.05
May 1, 2009

Contents

List of Figures

1 Introduction

The *Trading Agent Competition (TAC) Market Design Tournament*, also known as the *CAT Tournament*, supports multiple markets, each regulated by a *specialist*, that run in parallel and compete against each other to attract traders and make profit. Each entrant of the CAT Tournament runs a specialist, and sets the market rules so as to win the game according to a certain set of assessment criteria. The game between specialists is also commonly called a CAT *game*.

The open-source JCAT project [17] provides the software platform for CAT games and an API in Java for entrants to build their market mechanisms. This document illustrates the usage of JCAT and provides a tutorial on how to build market mechanisms by extending the parameterized framework implemented in JCAT.

This document has been developed as a result of inputs and discussions from teams at the Graduate School and University Center, and Brooklyn College, the City University of New York (Kai Cai, Jinzhong Niu, Simon Parsons, Elizabeth Sklar), the University of Liverpool (Peter McBurney), the University of Southampton (Enrico Gerding), and the University of Essex (Steve Phelps, formerly at the University of Liverpool). The annual CAT Tournament is run under the auspices of the Trading Agent Competition (TAC), and sponsored by the UK EPSRC Project, "Market Based Control of Complex Computational Systems" (GR/T19742/01), a research project undertaken jointly by the Universities of Birmingham, Liverpool and Southampton, UK. Further information is available from the project web-site:

`http://www.marketbasedcontrol.com/cat/`.

The JCAT software and related documents, including the latest version of this document, are hosted at:

`http://jcat.sourceforge.net/`.

This document should be cited as [15]. We welcome you to use JCAT in your research, on experimental auction mechanism design in particular, and cite [17] in your publications.

NOTE: This document is still undergoing development, and later sections will be completed in due course.

2 Change Log

- Version 1.00 : first version by Jinzhong Niu in November, 2006.
- Version 1.01 : (Jinzhong Niu, 12/07/2006) more details added in various sections.
- Version 1.02 : (Jinzhong Niu, 12/22/2006) inputs from Peter McBurney added.
- Version 1.03 : (Jinzhong Niu, 02/28/2007) explanation on parameter files added in Section 4.7.
- Version 1.04 : (Jinzhong Niu, 06/12/2008) Sections 1, 3, 4.1, and 4.2 updated.
- Version 1.05 : (Jinzhong Niu, 04/30/2008) Sections 1, 3, and 4 extensively updated.

3 CAT games

A CAT game is designed to compare and evaluate multiple market mechanisms in a direct and intuitive manner. JCAT, the software platform for CAT games, extends the single-threading *Java Auction Simulator API (JASA)*,[1] adding support for multiple parallel markets with trading agents moving between them. It has been used to conduct research on computational auction design ([5, 12, 14, 16, 18, 22, 27, 29]) and was successfully used as the game server in the first and second CAT tournaments in 2007[2] and 2008[3] respectively.

A typical CAT game consists of a CAT server and several CAT clients, which may be trading agents or specialists (markets). As illustrated in Figure 1, the CAT server works as a communication hub between CAT clients. A registry component records all game events and validates requests from traders and specialists. Various game report modules are available to process game events, calculate and output values of different measurements for post-game analysis.

A CAT game lasts a certain number of *day*s, each day consists of *round*s, and each round lasts a certain number of *tick*s, or milliseconds. The game clock in the game server fires events to notify clients of opening and closing of each day and round interval.

[1] `http://jasa.sourceforge.net/`.
[2] `http://www.marketbasedcontrol.com/blog/index.php/?p=30`.
[3] `http://www.marketbasedcontrol.com/blog/index.php/?p=70`.

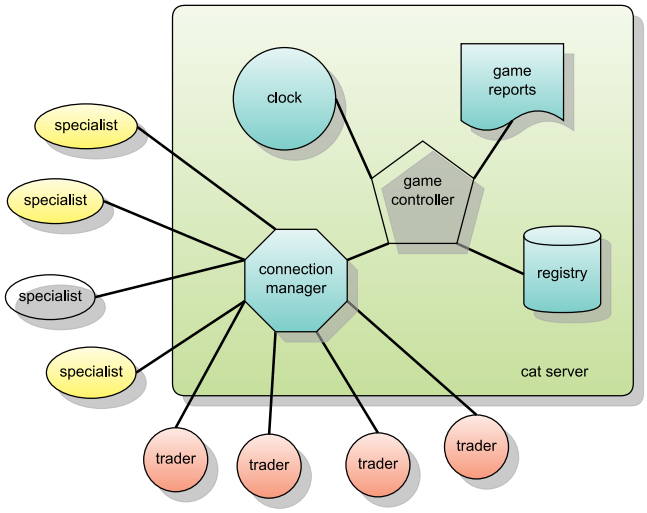

Figure 1: The topology of a CAT game.

Each trading agent is assigned private values for the goods it will trade. For buyers the private value is the most it will pay for a good. For sellers, the private value is the least it will accept for a good. The private values and the number of goods to buy or sell make up the demand and supply of the markets. Private values remain constant during a day, but may change from day to day, depending upon the configuration of the game server.

Each trading agent is endowed with a *trading strategy* and a *market selection strategy*. The first specifies how to make offers, while the second specifies which market to choose to make offers in. Trading strategies provided in JCAT include those that have been extensively researched in the literature and some of them have shown to work well in practice, e.g., ZI-C [11], RE [25], ZIP [7], and GD [10]. A typical class of market selection strategies treats the choice of market as an n-armed bandit problem where daily profits are used as rewards when updating the value function.

Specialists facilitate trade by matching offers and determining the trading price in an exchange market. Each specialist operates its own exchange market and may choose its own auction rules — the aim of the CAT tournaments is to create a

specialist that optimizes a particular set of measures [4]. Specialists may have adaptive strategies such that the policies change during the course of a game in response to market conditions for desired outcomes. JCAT provides a reference implementation of a parameterizable specialist that can be easily configured and extended to use policies regulating different aspects of an auction.

A specialist typically includes components that regulate aspects of its market. The following components were common in entrants to the first CAT competition [16]. *Matching policies* define the set of matching offers in a market at a given time. *Quoting policies* determine the ask quote and bid quote, which respectively specify the upper bound for offers to sell and the lower bound for offers to buy that may be placed in the market at a given time. *Shout accepting policies* judge whether a request by a trader to place an offer in the market should be accepted or rejected. *Clearing conditions* define when to clear the market and execute transactions between matched offers. A *pricing policy* is responsible for determining transaction prices for matched ask-bid pairs. The decision may involve only the prices of the matched offers, or more information including market quotes. *Charging policies* determine the charges a specialist imposes on a trading day. A specialist can set its fees, or *price list*, which are charged to traders and other specialists who wish to use the services provided by the specialist. Each specialist is free to set the level of the charges for registration with a specialist, for making an offer, for completing a transaction, and so on.

JCAT is written in Java, and adopts a client/server scheme for high flexibility and scalability. Its socket-based communication and the use of a plain-text message language (CATP) permit clients to be written in virtually any popular programming language, and CAT games can run across the Internet as in the 2007 and 2008 CAT tournaments.

The JCAT class library provides an extensible framework in which new auction rules can be easily implemented and coupled with other policies. A variety of learning algorithms have been included to support adaptive strategies. JCAT has a user-friendly interface based on Java Swing to monitor an on-going game as shown in Figure 2, and also supports a PHP interface allowing results to be displayed on the Web.

The introduction in this section is brief. For more information on CAT games and the CATP communication protocol, please refer to [4] and [19] respectively.

4 Usages

This section explains how to build, configure, and run JCAT.

Figure 2: Game console.

4.1 Prerequisites

The compilation and execution of JCAT require:

- Sun's JDK 5 (i.e., Java 1.5) or later [28], and
- Apache Ant version 1.6.2 or later [1].

In addition, JCAT uses the following third-party libraries:

- Repast 3.1 [24] (only COLT [6], JUnit [9], and log4j [3] needed),
- Commons Collections 3.2 [2],
- JFree's JFreeChart 1.0.2 [21] and JCommon 1.0.5 [20], and
- GNU Trove 1.4 [8].

These libraries have been included in the JCAT distribution, so you do not need to download and install these packages yourself.[4]

[4] JCAT also includes the parameter-related classes from ECJ [13] with slight modification.

4.2 Installation

JCAT is available for download at:

`http://jcat.sourceforge.net/`

There are:

- `jcat-`*`version`*`.tar.gz`

  This is a tar ball containing all the source code, generated class files, generated API HTML documents, third-party libraries, configuration files, and other resource files. Unpack it in a directory, you are ready to launch JCAT as long as Sun's Java VM and Apache Ant are already installed. Please refer to Section 4.5 on how to run JCAT.

- `jcat-src-`*`version`*`.tar.gz`

  This is a tar ball containing all the files in `jcat-`*`version`*`.tar.gz` except the class files, API HTML documents, and other files that are generated from the source. You must build it before running JCAT. Please refer to Section 4.4 on how to build JCAT.

- `jcat-all-`*`version`*`.jar`

  This is an executable jar file containing all the class files, resource files, and configuration files in JCAT as well as those required third-party libraries. When you system is properly configured, you may start a demonstration of JCAT by double-clicking this jar file, or if you fails, from the command console as to be described in Section 4.5. The demonstration will show you a graphical game console similar to the one in Figure 2. The jar file can be built by running `ant applet` in the root directory after either `jcat-`*`version`*`.tar.gz` or `jcat-src-`*`version`*`.tar.gz` is unpacked. Please see Section 4.4 for more information.

4.3 Directory structure

After you download and unpack the JCAT tar ball file, you will find the `jcat` directory, which we will refer to as `$jcat_root` in later sections. `$jcat_root` has the following subdirectories:

- `src`

  JCAT source code.

- `test`

  unit test code based on JUnit.

- `lib`

  third-party library jar files. Please see Section 4.1 for detailed information on these libraries.

- `doc`

  LaTeX documents on the specification of CAT protocol [19], and this document itself.

- `params`

  files used to configure CAT games.

- `resources`

  image files used for JCAT graphical interface.

4.4 Building

JCAT ships with a `$jcat_root/build.xml`, which is typical for a Java software project based on Apache Ant. The `$jcat_root/build.xml` file basically tells Ant how to perform certain actions, including building, executing, and packing up JCAT.

- To build JCAT, the most common way is to run

  ```
  $ ant jar
  ```

  A `jcat.jar` will be generated in the `$jcat_root` directory. This file contains all the classes of JCAT and only these classes. Now you are ready to run JCAT. Unless you are interested in building JCAT in other ways or additional parts of JCAT, you may skip the rest of this section for now and jump to the next section to see how to run JCAT.

- To build a 'super' jar file like the jar file described in Section 4.2 that includes all the JCAT and third-party library files, edit `$jcat_root/params/cat.params` first to make sure that you have the configuration you want (see Section 4.7 for more details), and then run

  ```
  $ ant applet
  ```

An executable `jcat-all.jar` as well as an `index.html` file will be created in the `$jcat_root/applet` directory. Later on, you may either execute the jar file directly if your system allows, or open the `index.html` file in your browser and run JCAT as an applet. This applet aims to demonstrate the features of JCAT.

- Similar to the `jcat-all.jar` file above, you may build an everything-included jar file for your specialist, which facilitates the execution of your specialist client when the client runs separately from the game server, i.e., on different hosts or as different processes. Because there is no graphical interface associated with the client program, you should NOT try to double-click and run it as you can do with the `jcat-all.jar` file above. Instead you run it from a command console as to be described in Section 4.5. To build the jar file for your specialist, edit `$jcat_root/params/cat.params` first to make sure that you have the configuration you want for your specialist (see Section 4.7 for more details), and then run

  ```
  $ ant specialist
  ```

  A `specialist.jar` file will be created in the `$jcat_root` directory. This `specialist.jar` file uses a main class different from the one used by the `jcat-all.jar` file. The former starts specialists only, whereas the latter starts both the game server and clients specified in the configuration file.

- All JCAT classes are coded in a way that complies with the `javadoc` standard. A whole set of JCAT API pages can be generated automatically from the JCAT source classes, by running:

  ```
  $ ant doc
  ```

  If everything goes fine, you will be able to find the the JCAT API pages in `$jcat_root/doc/api`. The API pages of the latest JCAT package is also available online at

  `http://jcat.sourceforge.net/api/`

- There are corresponding commands to clean up respectively the generated files through the above commands, including:

```
$ ant cleanOutput

$ ant cleanApplet

$ ant cleanSpecialist

$ ant cleanDoc
```

These commands will respectively delete the `$jcat_root/jcat.jar` file, the `$jcat_root/applet` directory, the `$jcat_root/specialist.jar` file, and the `$jcat_root/doc/api` directory.

NOTE: Sometimes Ant may prompt that it fails to delete a jar file. One of the reasons is that the Java class path includes the current directory, i.e., the `classpath` environment variable looking like "`.;<other paths>`" if you are using Windows or "`.:<other paths>`" if you are using some Unix, and the jar files in the current directory are somehow locked up. After you remove the "`.`" part from the `classpath`, Ant may get back to work normally.

4.5 Running

After JCAT is downloaded and built, it is ready to run.

4.5.1 Running with Ant

The basic way to run JCAT is to run the following command in the `$jcat_root` directory using Apache Ant:

```
$ ant run
```

This runs a sample CAT game that includes trading agents and specialists together with the game server in a single process and with the graphical game console enabled. The configuration of the game is defined in the `$jcat_root/params/cat.params` file. If you want to use a different configuration, you may either change the `cat.params` file, or explicitly specify another configuration file. In the latter case, execute the command:

```
$ ant run -Dparams=params/another.params
```

The `params` property for Ant takes the value of `params/another.params`, which is the file you use to configure your CAT game. By default, the `params` property has the value of `params/cat.params`, the default configuration file.

4.5.2 Running without Ant

Alternatively, if you have built `$jcat_root/applet/jcat-all.jar`, you may double-click it, or run the following command in the `$jcat_root/applet/` directory:

```
$ java -jar jcat-all.jar
```

This automatically runs JCAT with the default configuration file `params/cat.params`, which has been built into the `jcat-all.jar` file, equivalent to the command `ant run`. You may also run JCAT as an applet by opening the `$jcat_root/applet/index.html` file in a browser. The advantage of running JCAT with the `jcat-all.jar` file is that you do not need Apache Ant. This is desirable in particular when you need to demonstrate JCAT on another computer where Ant may not be available. You may also choose to use a different configuration file that may or may not within the `jcat-all.jar` file. For example:

```
$ java -jar jcat-all.jar params/another.params
```

NOTE: When a configuration file is given in a relative path as shown in the above examples, it will be searched for first relative to the root of `jcat-all.jar` inside the jar file, and if not found, then relative to the directory where `jcat-all.jar` resides. Be aware of the situation that an external file you intend to use happen to have a 'twin brother' inside the jar file.

4.5.3 Running CAT game server and clients separately

All the above methods launch a complete CAT game, including a CAT game server, a set of competing specialists, and a set of trading agents. In certain scenarios, you may need to run the game server and the clients separately, e.g., during a CAT tournament.

- To run a CAT server only, run

  ```
  $ ant server
  ```

- To run a CAT server and a set of trading agents only, run

  ```
  $ ant servertraders
  ```

  This is typically when the CAT tournament organizers run the CAT game server and provide the trading agents themselves.

- To start a set of specialists only, run

```
$ ant markets
```

- To start a set of trading agents only, run

```
$ ant traders
```

Again, additional "`-Dparams=params/cat.params`" can be added in these commands when you choose to use a configuration file other than the default `params/cat.params`.

NOTE: When you run a CAT game with the game server and clients separately, make sure that the configuration files used by the game server and the separated clients all use the socket-based method to communicate with each other since the alternative methods do NOT support inter-process communication. More information about this can be found in Section 4.8.1.

4.6 Packing up

Commands are provided in JCAT to prepare a JCAT package that can be easily distributed or in the purpose of archives.

- To pack up a customized JCAT of your own, including all the source files and excluding those automatically generated contents, run

```
$ ant srctar
```

  This is how the `jcat-src-`*`version`*`.tar.gz` file described in Section 4.2 is built.

- To pack up a customized JCAT of your own that is executable without further compilation, including all the source files as well we the automatically generated JCAT API pages and the `$jcat_root/jcat.jar` file, run

```
$ ant tar
```

  This is how the `jcat-`*`version`*`.tar.gz` file described in Section 4.2 is built.

- To pack up a specialist client so as to make it easier to run it or distribute it for others to use, you may simply use the `ant specialist` command we described in Section 4.4 to build an everything-included jar file. Entrants of CAT tournaments are required to release their specialists in the TAC agent repository at

`http://www.sics.se/tac/showagents.php`.

The `ant specialist` command is an ideal way for entrants to prepare and release their specialist programs.

4.7 Parameterizable configuration mechanism

The parameter-based configuration mechanism adopted by JCAT is originally implemented in ECJ [13]. This section helps you to understand this mechanism and setup your own configuration if needed.

JCAT, like ECJ, relies heavily on parameterizable configuration files for nearly every conceivable parameter setting. It even relies on parameter files to determine which classes to use in different places. This means that understanding parameters and parameter files is crucial to using JCAT. Some text of this section is copied from the documentation of ECJ[5] with minor editions.

4.7.1 Parameters

JCAT's parameters are written one to a line in Java property-list style. They are in the following format:

```
parametername = value
```

Whitespace is stripped. Parameter values may contain internal whitespace but parameter names may not. Blank lines and lines beginning with a `#` are ignored. Parameter names and values are case-sensitive.

Parameter values are interpreted as one of five data types, depending on the parameter:

- *String*

  A string value consists of everything after the equals sign through the end of line, trimmed of whitespace.

- *Class name*

  A class name value is parsed as a string and must be the full absolute class name, e.g., `java.util.Hashtable`.

- *Path name*

  A path name value is parsed as a string. It can be either absolute (as in `/foo/bar`) or relative (as in `foo/bar` or `../../foo/bar`). If it is relative,

[5] `http://www.cs.gmu.edu/~eclab/projects/ecj/docs/parameters.html`.

it is interpreted relative to the directory of the parameter file in which the parameter is defined. However, if a relative pathname is prefixed with a `$` (as in `$foo/bar` or `$../../foo/bar`), it is assumed relative with respect to the directory in which the Java process was started.

- *Integer and floating-point number*

  A number value must be in the Java-standard numerical form.

- *Boolean*

  A boolean value is of the form:

  ```
  parametername = true
  parametername = false
  ```

  If a parameter is declared but is not one of these two values, it is assumed to take a *default* value, which varies depending on the parameter. For example, if the default value for `myparameter` is `true`, then:

  ```
  myparameter = gobbledygook
  myparameter =
  ```

  both signify that `myparameter` is to be set to `true`.

4.7.2 Parameter files

JCAT reads parameters from a hierarchical set of parameter files, which each ends with the extension "`.params`". When you start up JCAT, you specify a parameter file as such:

```
ant run -Dparams=myParameterFile
```

The `myParameterFile` file is the starting point of the hierarchy of parameter files, and like other parameter files, may have multiple parents which define additional parameters. A parameter file specifies that it has a parent with a special parameter:

```
parent.n = parentFile
```

where `n` indicates that the parent is the `n`th parent. `n` starts at 0 and increases. The parent files must be assigned with consecutive parameter names starting with `parent.0`. For example:

```
parent.0 = ../../myFirstParent.params
parent.1 = ../../../mySecondParent.params
parent.2 = ../foo/bar/myThirdParent.params
```

4.7.3 Precedence

Parameters may also be defined on the command line with the `-p` options, which may appear multiple times. No space may appear between the parameter name, `=`, and value. For example:

```
ant run -Dparams="params/cat.params -p cat.server.daylen=20"
```

If you have two parameters with the same name, here are the rules guiding which ones take precedence:

- Programmatically-set parameters override all others.
- Next come command-line parameters.
- Later command-line parameters override earlier ones.
- Next come parameters in the specified parameter file.
- Within a file, later parameters override earlier ones.
- Child parameter files' parameters override their parents' or ancestors' parameters.
- If a file has parents `parent.m` and `parent.n`, and `m` is less than `n`, then parameters derived from `parent.m` and its ancestors will be overridden by parameters derived from `parent.n` and its ancestors.

4.7.4 JCAT parameter style

Since numerous objects read parameters from the parameter files, JCAT organizes its parameter name space hierarchically using "`.`" to separate elements in parameter names. All parameters used by a class share identical prefixes. The longest one is called the *parameter base* for the class. For example,

```
cat.infrastructure.server = mbc.liv.ac.uk
cat.infrastructure.port = 9090
```

are two parameters used by the `edu.cuny.cat.comm.SocketBasedInfrastructureImpl` class to specify the address and port of the game server. The parameter base of this class is `cat.infrastructure`, which is passed on to `SocketBasedInfrastructureImpl` when it is instantiated. `SocketBasedInfrastructureImpl` is responsible for appending its own parts, i.e., `server` and `port`, to the base to construct the full names of its parameters. As nested classes add their parts to bases and pass on new bases

to further nested classes, you can imagine that this forms a hierarchical name space for parameters. All JCAT parameter names start with `cat.` except for the parameters that are contained in JCAT configuration files but used by other packages, e.g. those parameters used by log4j (see Section 4.8.8 for more details). A class has no control on which base it uses, but when it finds out that a parameter constructed from the base is not defined, it may turn to construct the parameter from its *default parameter base*. You may consider that the default, or *static*, parameter base provides a way to define default values for the parameters, whereas the *dynamic* base that is passed on provides a way to customize the default values when needed. For example:

```
socket_based_infrastructure.server = localhost
socket_based_infrastructure.port = 9090
```

gives the default address and port of the CAT server as `localhost` and `9090`. When the default values are sufficiently good, you do not need to customize them with the dynamic parameters. If the default values of parameters of classes are put in a certain parameter file, you may simply to refer to the file with the `parent.n` parameters.[6] This would greatly ease the task of configuration, making it flexible and less error-prone.

The JCAT API documentation, which is available online at

`http://jcat.sourceforge.net/api/`

provides detailed information on what parameters each parameterizable class uses and how the class is configured. Each page for a class contains two tables, if applicable, which give information about parameters and the default parameter base for that class. For example, the page for `SocketBasedInfrastructureImpl`[7] shows:

Parameters

base`.server` (the domain name or IP address of the CAT game server)
string (default: localhost)

base`.port` (the port number the CAT game server will be listening to)
int (default: 9090)

[6] See `$jcat_root/params/modules/general.params` for an example.

[7] `http://jcat.sourceforge.net/api/edu/cuny/cat/comm/SocketBasedInfrastructureImpl.html`.

Default Base

`socket_based_infrastructure`

NOTE: The default values shown in the above table are those built in the class, rather than those static ones defined with `socket_based_infrastructure.server` and `socket_based_infrastructure.port` in parameter files. Typically the values of dynamic parameters override the values of static parameters, and the values of static parameters override the built-in default values.

4.8 Configuring a CAT game

This section describes how to configure JCAT to run CAT games. Each of the following sections will discuss one aspect of the configuration.

4.8.1 Communication infrastructure

The communication between the CAT server and clients can be supported through different *infrastructures*. All these communication infrastructures provide a way for the server and clients to transfer messages between each other. An analogy is that one can surf the Web through a DSL-based broadband service or through a cable-based broadband service.

There are three communication infrastructures implemented in JCAT:

- *socket-based*

  The CAT server and clients communicate through socket. The CAT server listens on a particular port, 9090 by default, allows clients to connect in before a game starts. This infrastructure is used when the game server and clients run separately on different hosts, as in a CAT tournament, or in different processes.[8] For example, the following snippet defines a socket-based infrastructure with both the game server and clients running on the local host:

```
cat.infrastructure = \
  edu.cuny.cat.comm.SocketBasedInfrastructureImpl
cat.infrastructure.server = localhost
cat.infrastructure.port = 9090
```

[8] Technically speaking, you may run the game server and clients within a single, multi-threading process. However in this case the other infrastructures may incur lower overhead and bring additional benefits.

- *event-based*

  The CAT server and clients all run in a single, multi-threading process, and each as a separate thread, communicate with each other through Java event-dispatching mechanism. For example, the following snippet defines a event-based infrastructure:

```
cat.infrastructure = \
  edu.cuny.cat.comm.EventBasedInfrastructureImpl
```

- *method call-based*

  The CAT server and clients all run in a single, single-threading process. Transferring messages is implemented by method invocations. For example, the following snippet defines a event-based infrastructure:

```
cat.infrastructure = \
  edu.cuny.cat.comm.CallBasedInfrastructureImpl
```

The first two infrastructures involve multiple threads or processes, and the game server and clients communicate in an *asynchronous* manner, whereas the method call-based infrastructure provides a *synchronous* alternative. When you use JCAT to conduct research experiments yourself, the call-based infrastructure is the fastest way to run simulations and guarantees replicatable results. However the socket-based infrastructure is the only choice when you run the game server and clients separately and has better scalability when you run a large number of clients (e.g., several thousand or more).

NOTE: When you try to run the game server and clients separately, make sure you use the socket-based infrastructure in your configuration file. The default `$jcat_root/params/cat.params` uses a call-based infrastructure.

4.8.2 Game duration

The `edu.cuny.cat.server.GameClock` controls the timing issues of a CAT game. The following configuration snippet tells `GameClock` to run 20 consecutive CAT games in a row. Each game lasts 20 days, each day including 10 1000-millisecond rounds. The game initialization period at the beginning of each game, between `GAMESTARTING` and `GAMESTARTED` events, lasts 100 milliseconds, and the day initialization period at the beginning of each trading day, between `DAYOPENING` and `DAYOPENED` events, lasts 10 milliseconds. After each game finishes, there is a game break of 100 milliseconds, and no day break or round break is defined. .

```
cat.server.iterations = 20

cat.server.gamelen = 20
cat.server.daylen = 10
cat.server.roundlen = 1000

cat.server.gameinit = 100
cat.server.dayinit = 10

cat.server.gamebreak = 100
cat.server.daybreak = 0
cat.server.roundbreak = 0
```

When you are using JCAT to conduct research experiments The selection of the parameter values should be cautious.

- Running a CAT game multiple iterations is helpful to collect more reliable statistical data.
- The game length and the day length should be long enough for adaptive game clients to learn, both across days and within each day, and for the game dynamics to converge.
- When an asynchronous communication infrastructure is used, the more clients are involved in a game, the longer a round should be, so as to give clients enough time to trade.
- The game break, the day break, and the round break are mainly used to leave enough time for the game server to do logging and house keeping when an asynchronous communication infrastructure is used. For instance, if you create a game report that does intensive data processing work after each day closes, you should define a relatively long day break.

NOTE: If the call-based synchronous communication infrastructure is used, the round length and the various break lengths do NOT have any effect since everything runs in a single thread and there is no deadline to care about.

4.8.3 Specialists

```
################################################################
# specialists

cat.specialist.n = 2
```

```
cat.specialist.0 = edu.cuny.cat.MarketClient
cat.specialist.0.n = 1
cat.specialist.0.type = specialist
cat.specialist.0.id = M1
cat.specialist.0.auctioneer = edu.cuny.cat.market.GenericDoubleAuctioneer
cat.specialist.0.auctioneer.pricing = \
  edu.cuny.cat.market.DiscriminatoryPricingPolicy
cat.specialist.0.auctioneer.clearing = \
  edu.cuny.cat.market.ProbabilisticClearingCondition
cat.specialist.0.auctioneer.accepting = \
  edu.cuny.cat.market.QuoteBeatingAcceptingPolicy
cat.specialist.0.auctioneer.charging = \
  edu.cuny.cat.market.FixedChargingPolicy
cat.specialist.0.auctioneer.charging.shout = 1
cat.specialist.0.auctioneer.charging.transaction = 1
cat.specialist.0.auctioneer.charging.information = 1
cat.specialist.0.auctioneer.charging.registration = 1
cat.specialist.0.auctioneer.charging.profit = 1

cat.specialist.1 = edu.cuny.cat.MarketClient
cat.specialist.1.n = 1
cat.specialist.1.type = specialist
cat.specialist.1.id = M2
cat.specialist.1.auctioneer = edu.cuny.cat.market.GenericDoubleAuctioneer
cat.specialist.1.auctioneer.pricing = \
  edu.cuny.cat.market.DiscriminatoryPricingPolicy
cat.specialist.1.auctioneer.clearing = \
  edu.cuny.cat.market.ProbabilisticClearingCondition
cat.specialist.1.auctioneer.accepting = \
  edu.cuny.cat.market.QuoteBeatingAcceptingPolicy
cat.specialist.1.auctioneer.charging = \
  edu.cuny.cat.market.FixedChargingPolicy
cat.specialist.1.auctioneer.charging.shout = 2
cat.specialist.1.auctioneer.charging.transaction = 2
cat.specialist.1.auctioneer.charging.information = 2
cat.specialist.1.auctioneer.charging.registration = 2
cat.specialist.1.auctioneer.charging.profit = 0.2
```

There are 2 specialists, each implemented as a `MarketClient`. `MarketClient`, another child class of `GameClient`, deals with CATP communication issues and each instance of this class has an auctioneer to process shouts and transactions. The `cat.specialist.`*n*`.id` parameter is used to specify explicitly the ID of specialist. Typically the ID of a game client is allocated dynamically by the game server. However for the sake of logging data analysis, a certain client needs to be identifiable. Here two specialists are pre-named to be `M1` and `M2`.

`GenericDoubleAuctioneer` is a framework of auctioneer supporting customization on pricing policy, market clearing rule, shout accepting policy, and charging policy. Thus `..auctioneer.pricing`, `..auctioneer.clearing`, `..auctioneer.accepting`, and `..auctioneer.charging` each requires a class name as its value, defining what rules or policies to be used in the auctioneer.

The above 2 auctioneers both are customized: to make a discriminatory transaction price at the middle point between prices of the matching ask and bid; to clear the market whenever a new shout arrives (as in a continuous double auction); to accept shouts only when they can beat the current market quotes (as NYSE does); and to charge at a fixed rate (though the 2 specialists charge at different rates).

4.8.4 Traders

The following configuration segment defines trading agents. The parameter base for trading agents is `cat.agent`.

```
############################################################
# trading agents

cat.agent.n = 2

cat.agent.0 = edu.cuny.cat.TraderClient
cat.agent.0.n = 50
cat.agent.0.type = buyer
cat.agent.0.isseller = false
cat.agent.0.strategy = edu.cuny.cat.trader.strategy.GDStrategy
cat.agent.0.marketselectionstrategy = \
  edu.cuny.cat.trader.marketselection.StimuliResponseMarketSelectionStrategy
cat.agent.0.marketselectionstrategy.learner = \
  edu.cuny.ai.learning.EpsilonGreedyLearner
cat.agent.0.marketselectionstrategy.learner.epsilon = 0.1

cat.agent.1 = edu.cuny.cat.TraderClient
cat.agent.1.n = 50
cat.agent.1.type = seller
cat.agent.1.isseller = true
cat.agent.1.strategy = edu.cuny.cat.trader.strategy.GDStrategy
cat.agent.1.marketselectionstrategy = \
  edu.cuny.cat.trader.marketselection.StimuliResponseMarketSelectionStrategy
cat.agent.1.marketselectionstrategy.learner = \
  edu.cuny.ai.learning.EpsilonGreedyLearner
cat.agent.1.marketselectionstrategy.learner.epsilon = 0.1
```

The trading agent population consists of two sub-populations, the first for buyers and the second for sellers. If you need heterogeneous buyers and/or sellers, you

have to define more sub-populations, each of which is a group of trading agents with identical properties.

Either of the above trader sub-populations has 50 agents and each agent is implemented as an instance of `edu.cuny.cat.TraderClient`. `TraderClient`, inheriting `edu.cuny.cat.GameClient`, reads `cat.agent.`*n*`.type` parameter, whose value is to be used in the `Type` field header for CATP `CHECKIN` requests. The value of this parameter must be a string starting with `seller` or `buyer` for a trader. The prefix helps the game server to identify the type of a trader. Additional suffix can be added to distinguish different buyer sub-populations or seller sub-populations. For example, there may be configured 2 buyer sub-populations respectively with type `buyer_zic` and `buyer_gd`, standing for buyers with ZI-C trading strategy and buyers with GD trading strategy. The following snippet shows this configuration:

```
cat.agent.n = 8

cat.agent.0 = edu.cuny.cat.TraderClient
cat.agent.0.n = 10
cat.agent.0.type = buyer_zic
cat.agent.0.isseller = false
cat.agent.0.strategy = \
  edu.cuny.cat.trader.strategy.RandomConstraintStrategy
...

cat.agent.1 = edu.cuny.cat.TraderClient
cat.agent.1.n = 10
cat.agent.1.type = buyer_gd
cat.agent.1.isseller = false
cat.agent.1.strategy = edu.cuny.cat.trader.strategy.GDStrategy
...
```

For a CAT tournament game, the `cat.agent.`*n*`.type` is also used to provide *security tokens*, which are known only by the tournament organizers and used by the game server to allow only authorized clients to connect. This

Each `TraderClient` instance itself deals with CATP communication issues and uses `TradingAgent` to manage trading and market selection logic. The `cat.agent.1.isseller`, `cat.agent.1.strategy`, and `cat.agent.1.marketselectionstrategy` parameters are all processed in `AbstractTradingAgent`, the super class of `TradingAgent`. The above trading agents all use GD strategy for trading and a stimuli-response style strategy with ϵ-greedy learning for market selection.

4.8.5 Registry

```
################################################################
# registry
```

```
cat.server.registry = edu.cuny.cat.registry.SimpleRegistry
```

A registry component in JCAT records all activities in games. Here `SimpleRegistry`, implementing `Registry` interface, is used to maintain information in memory. In contrast, for example, it is possible to have a registry posting all activity information on a web site for real-time browsing.

4.8.6 Valuation policy

```
############################################################
# valuer

cat.server.valuer = edu.cuny.cat.valuation.RandomValuer
cat.server.valuer.minvalue = 50
cat.server.valuer.maxvalue = 150
```

`cat.server.valuer` defines the valuation policy to determine the demand and supply schedules for trading agents. Here a `RandomValuer` is used, which draws valuations for traders from a uniform distribution from 50 through 150.

4.8.7 Game reports

```
############################################################
# game report

cat.server.report = edu.cuny.cat.stat.CombiGameReport
cat.server.report.n = 4
cat.server.report.0 = edu.cuny.cat.stat.ProfitReport
cat.server.report.1 = edu.cuny.cat.stat.TraderDistributionReport
cat.server.report.2 = edu.cuny.cat.stat.FeeReport
cat.server.report.3 = edu.cuny.cat.stat.ReportVariableWriterReport
cat.server.report.3.filename = log.csv
cat.server.report.3.var.n = 3
cat.server.report.3.var.0 = <specialist>.profit
cat.server.report.3.var.1 = <specialist>.trader
cat.server.report.3.var.2 = <specialist>.<fee>
```

Data collection in a CAT game is necessary for behavior analysis and is done typically in various game reports. A `GameReport` may define *report variable*s to denote certain values and report them to `ReportVariableBoard`, where later on report values may be retrieved by virtually any component in the current JCAT

process[9]. For example, with this configuration example, `ProfitReport` will define `M0.profit` and `M1.profit` to report daily profits of the two specialists.

The above sets up 4 game reports, `ProfitReport` tracking profits of specialists, `TraderDistributionReport` the number of traders registered with specialists over days, `FeeReport` the charges of specialists over days, and `ReportVariableWriterReport` that outputs the daily profits of specialists, the number of traders registered daily to specialists, and the various daily charges of specialists to a CSV file, `log.csv`. Note that report variable names specified for `cat.server.report.3.var.`*n* may use templates in the format of `<..>` to refer to multiple report variables: `<specialist>` and `<trader>` match respectively any specialist and any trader; and `<fee>` matches any of `FEE.REGISTRATION`, `FEE.INFORMATION`, `FEE.SHOUT`, `FEE.TRANSACTION`, and `FEE.PROFIT`. Thus, the above 3 variable names actually request recording the values of the following report variables:

```
M1.profit      M2.profit     M1.trader       M2.trader
M1.FEE.REGISTRATION    M2.FEE.REGISTRATION
M1.FEE.INFORMATION     M2.FEE.INFORMATION
M1.FEE.SHOUT           M2.FEE.SHOUT
M1.FEE.TRANSACTION     M2.FEE.TRANSACTION
M1.FEE.PROFIT          M2.FEE.PROFIT
```

4.8.8 Logging

```
################################################################
# log4j configuration - substitute INFO for DEBUG to turn on debugging output

log4j.rootLogger=INFO, stdout

log4j.appender.stdout=org.apache.log4j.ConsoleAppender
log4j.appender.stdout.layout=org.apache.log4j.PatternLayout
log4j.appender.stdout.layout.ConversionPattern=%m%n
```

This segment is for log4j configuration. Please refer to log4j documentation for more details.

NOTE: The configuration section for log4j must appear in the first level parameter file rather than any parent parameter file. This is because log4j does NOT support hierarchical parameter files.

[9]Theoretically, any component in JCAT can also define a report variable and report it; however this should be avoided unless for the sake of debugging.

4.8.9 Random number generation

```
#############################################################
# prng

#  Use the 32bit version of the Mersenne Twister algorithm
cat.prng = edu.cuny.prng.MT32

#  with the following PRNG seed
cat.seed = 4523
```

`edu.cuny.prng.MT32`[10] is used to generate random numbers with 4523 as the seed.

4.8.10 Game console

```
#########################################################
# game console

cat.server.console = true

cat.server.console.homeurl = http://agents.sci.brooklyn.cuny.edu/
cat.server.console.icon = icon.gif


####################################
# Images configuration

images.url=resources/images/


####################################
# frame

cat.server.console.width = 1100
cat.server.console.height = 750


####################################
# overview

cat.server.console.overview.title = General View
cat.server.console.overview.width = 400
cat.server.console.overview.height = 300
```

[10] The 32-bit Mersenne Twister pseudo random number generation algorithm developed by Makoto Matsumoto and Takuji Nishimura in 1997.

```
cat.server.console.overview.orientation = y
cat.server.console.overview.panel.n = 2

cat.server.console.overview.panel.0.panel.n = 2
cat.server.console.overview.panel.0.panel.0.orientation = y
cat.server.console.overview.panel.0.panel.0.panel.n = 2
cat.server.console.overview.panel.0.panel.0.panel.0 = \
  edu.cuny.cat.ui.PlayerLabelPanel
cat.server.console.overview.panel.0.panel.0.panel.1 = \
  edu.cuny.cat.ui.SpecialistGridPanel
cat.server.console.overview.panel.0.panel.1 = \
  edu.cuny.cat.ui.ClockPanel
cat.server.console.overview.panel.0.panel.1.angle = 260
cat.server.console.overview.panel.0.panel.1.width = 150
cat.server.console.overview.panel.0.panel.1.height = 150

cat.server.console.overview.panel.1.panel.n = 2
cat.server.console.overview.panel.1.panel.0 = \
  edu.cuny.cat.ui.ProfitPlotPanel
cat.server.console.overview.panel.1.panel.1 = \
  edu.cuny.cat.ui.CumulativeTraderDistributionPanel
cat.server.console.overview.panel.1.panel.1.legend = true


####################################
# menus

cat.server.console.menu.n = 1

cat.server.console.menu.0 = ?
cat.server.console.menu.0.text = Help
cat.server.console.menu.0.param = ^H^
cat.server.console.menu.0.icon = Help.gif
cat.server.console.menu.0.item.n = 2

cat.server.console.menu.0.item.0.name = menu.help.about
cat.server.console.menu.0.item.0.text = About
cat.server.console.menu.0.item.0.param = ^A^~A~
cat.server.console.menu.0.item.0.icon = About.gif
cat.server.console.menu.0.item.1.name = menu.help.home
cat.server.console.menu.0.item.1.text = Home
cat.server.console.menu.0.item.1.param = ^H^~H~
cat.server.console.menu.0.item.1.icon = Home.gif


####################################
# buttons

cat.server.console.button.start.text = Start
```

```
cat.server.console.button.start.icon = VCRPlay.gif

cat.server.console.button.pause.text = Pause
cat.server.console.button.pause.icon = VCRPause.gif

cat.server.console.button.resume.text = Resume
cat.server.console.button.resume.icon = VCRForward.gif

cat.server.console.button.exit.text = Exit
cat.server.console.button.exit.icon = VCRStop.gif


####################################
# About Dialog

cat.server.console.aboutdialog.title = About Dialog
cat.server.console.aboutdialog.text = JCAT - \
  The Java Implementation of TAC/CAT Market Design Competition Platform
cat.server.console.aboutdialog.width = 300
cat.server.console.aboutdialog.height = 200
cat.server.console.aboutdialog.icon = About.gif
```

This segment is for the graphical CAT game console setup. The menus, various widgets, and the resources used by them are all configurable. Figure 2 shows an example of the game console. For more details on game console configuration, please refer to the class documents for `GameConsole`, `GameView`, `ViewPanel`, and their subclasses.

5 Design and implementation

5.1 Package organization

Figure 3

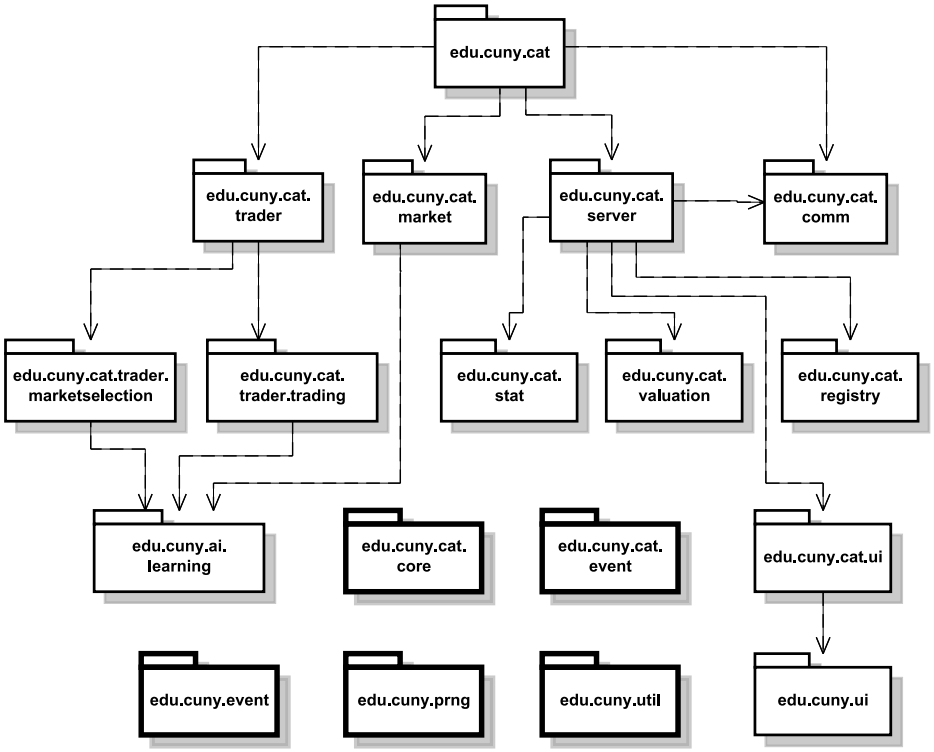

Figure 3: Packages in JCAT.

5.1.1 `edu.cuny.cat`

5.1.2 `edu.cuny.cat.comm`

5.1.3 `edu.cuny.cat.trader`

5.1.4 `edu.cuny.cat.trader.marketselection`

5.1.5 `edu.cuny.cat.trader.trading`

5.1.6 `edu.cuny.cat.market`

5.1.7 `edu.cuny.cat.server`

5.1.8 `edu.cuny.cat.stat`

5.1.9 `edu.cuny.cat.valuation`

5.1.10 `edu.cuny.cat.registry`

5.1.11 `edu.cuny.cat.ui`

5.1.12 `edu.cuny.cat.core`

5.1.13 `edu.cuny.cat.event`

5.1.14 `edu.cuny.event`

5.1.15 `edu.cuny.ai.learning`

5.1.16 `edu.cuny.prng`

5.1.17 `edu.cuny.util`

5.2 Event dispatching

Figure 4

5.3 Registry

5.4 Session management

5.4.1 `CatpProactiveSession`

5.4.2 `CatpReactiveSession`

5.5 Game reports

A collection of game reports, implementing `edu.cuny.cat.event.AuctionEventListener`, may be setup to listen to auction events broadcast by the CAT server.

- `ProfitReport`
- `TraderDistributionReport`
- `FeeReport`
- `ShoutReport`
- `SQLReport`

5.5.1 Report variables

- `ReportVariableBoard`
- `ReportVariableWriterReport`

5.5.2 `ProfitReport`

5.5.3 `TraderDistributionReport`

5.5.4 `FeeReport`

5.5.5 `SQLReport`

5.6 Game console

Figure 2

5.6.1 `GameView`

5.6.2 `ViewPanel`

5.7 Market mechanism facilities

5.7.1 General double auctioneer

5.7.2 Pricing policy

5.7.3 Clearing condition

5.7.4 Shout accepting policy

5.7.5 Charging policy

5.8 Trading agents

5.8.1 Trading strategy

- TT

- ZI-C [11]
- ZIP [7]
- GD [10]
- Kaplan [26]
- RE [25]
- PVT [23]

5.8.2 Market selection strategy

- random
- ϵ-greedy
- softmax

5.9 Learning algorithms

Acknowledgements

We would like to thank developing teams of the following projects:

- JASA
- ECJ
- Repast (COLT, JUnit, and log4j)
- JFreeChart and JCommon

References

[1] Apache Software Foundation. Apache Ant — a Java-based Build Tool. `http://ant.apache.org/`.

[2] Apache Software Foundation. Commons Collections. `http://commons.apache.org/collections/`.

[3] Apache Software Foundation. log4j — A Reliable, Fast, and Flexible Logging Framwork for Java. `http://logging.apache.org/log4j/docs/index.html`.

[4] Kai Cai, Enrico Gerding, Peter McBurney, Jinzhong Niu, Simon Parsons, and Steve Phelps. Overview of CAT: A market design competition. Technical Report ULCS-09-005, Department of Computer Science, University of Liverpool, Liverpool, UK, 2009. Version 2.0.

[5] Kai Cai, Jinzhong Niu, and Simon Parsons. On the economic effects of competition between double auction markets. In *Proceedings of the Tenth Workshop on Agent-Mediated Electronic Commerce (AMEC X)*, Estoril, Portugal, May 2008.

[6] CERN - European Organization for Nuclear Research. COLT — A Set of Open Source Libraries for High Performance Scientific and Technical Computing in Java. `http://dsd.lbl.gov/~hoschek/colt/`.

[7] Dave Cliff and Janet Bruten. Minimal-intelligence agents for bargaining behaviours in market-based environments. Technical report, Hewlett-Packard Research Laboratories, Bristol, England, 1997.

[8] Eric D. Friedman and Rob Eden. GNU Trove — High Performance Collections for Java. `http://trove4j.sourceforge.net/`.

[9] Erich Gamma and Kent Beck. JUnit — A Regression Testing Framework. `http://www.junit.org/`.

[10] Steven Gjerstad and John Dickhaut. Price formation in double auctions. *Games and Economic Behavior*, 22:1–29, 1998.

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

[12] Mark L. Gruman and Manjunath Narayana. Applications of classifying bidding strategies for the CAT Tournament. In W. Ketter, A. Rogers, N. Sadeh, and W. Walsh, editors, *Proceedings of the International Trading Agent Design and Analysis Workshop (TADA 2008)*, Chicago, IL, USA, 2008.

[13] Sean Luke, Liviu Panait, Zbigniew Skolicki, Jeff Bassett, Robert Hubley, and Alexander Chircop. ECJ – A Java-based Evolutionary Computation and Genetic Programming Research System. `http://cs.gmu.edu/~eclab/projects/ecj/`.

[14] Jinzhong Niu, Kai Cai, Peter McBurney, and Simon Parsons. An analysis of entries in the first TAC market design competition. In *Proceedings of the*

IEEE/WIC/ACM International Conference on Intelligent Agent Technology, Sydney, Australia, December 2008.

[15] Jinzhong Niu, Kai Cai, Peter McBurney, and Simon Parsons. JCAT: The software platform for CAT games. Technical report, Department of Computer Science, The Graduate School and University Center, The City University of New York, New York, USA, 2009. Version 2.0.

[16] Jinzhong Niu, Kai Cai, Simon Parsons, Enrico Gerding, and Peter McBurney. Characterizing effective auction mechanisms: Insights from the 2007 TAC Mechanism Design Competition. In Padgham, Parkes, Müller, and Parsons, editors, *Proceedings of the Seventh International Conference on Autonomous Agents and Multiagent Systems*, pages 1079–1086, Estoril, Portugal, May 2008.

[17] Jinzhong Niu, Kai Cai, Simon Parsons, Enrico Gerding, Peter McBurney, Thierry Moyaux, Steve Phelps, and David Shield. JCAT: A platform for the TAC Market Design Competition. In Padgham, Parkes, Müller, and Parsons, editors, *Proceedings of the Seventh International Conference on Autonomous Agents and Multiagent Systems*, pages 1649–1650, Estoril, Portugal, May 2008. Demo Paper.

[18] Jinzhong Niu, Kai Cai, Simon Parsons, and Elizabeth Sklar. Some preliminary results on competition between markets for automated traders. In *Proceedings of AAAI-07 Workshop on Trading Agent Design and Analysis (TADA-07)*, Vancouver, Canada, July 2007.

[19] Jinzhong Niu, Albert Mmoloke, Peter McBurney, and Simon Parsons. CATP: A communication protocol for CAT games. Technical report, Department of Computer Science, The Graduate School and University Center, The City University of New York, New York, USA, 2009. Version 2.0.

[20] Object Refinery Limited. JCommon — A Java common utility library. `http://www.jfree.org/jcommon/`.

[21] Object Refinery Limited. JFreeChart — A Java chart library. `http://www.jfree.org/jfreechart/`.

[22] Ana Petric, Vedran Podobnik, Andrej Grguric, and Marijan Zemljic. Designing an effective e-market: an overview of the CAT agent. In *Proceedings of AAAI-08 Workshop on Trading Agent Design and Analysis (TADA-08)*, Chicago, IL, USA, 2008.

[23] Chris Preist and Maarten van Tol. Adaptive agents in a persistent shout double auction. In *Proceedings of the 1st International Conference on Information and Computation Economies*, pages 11–18. ACM Press, 1998.

[24] Repast — Recursive Porus Agent Simulation Toolkit. `http://repast.sourceforge.net/`.

[25] Alvin E. Roth and Ido Erev. Learning in extensive-form games: Experimental data and simple dynamic models in the intermediate term. *Games and Economic Behavior*, 8:164–212, 1995.

[26] John Rust, John H. Miller, and Richard G. Palmer. Behaviour of trading automata in a computerized double auction market. In Daniel Friedman and John Rust, editors, *The Double Auction Market: Institutions, Theories and Evidence*, Santa Fe Institute Studies in the Sciences of Complexity, chapter 6, pages 155–199. Perseus Publishing, Cambridge, MA, 1993.

[27] Jung-woo Sohn, Sooyeon Lee, and Tracy Mullen. Impact of misalignment of trading agent strategy under a multiple market. In S. Das and M. Ostrovsky, editors, *Proceedings of the First Conference on Auctions, Market Mechanisms and Their Applications (AMMA-2009)*, Boston, MA, USA, 2009. *Forthcoming*.

[28] Sun Microsystems, Inc. `http://java.sun.com/`.

[29] Perukrishnen Vytelingum, Ioannis A. Vetsikas, Bing Shi, and Nicholas R. Jennings. IAMwildCAT: The winning strategy for the TAC Market Design Competition. In *Proceedings of 18th European Conference on Artificial Intelligence*, pages 428–434, Patras, Greece, 2008.

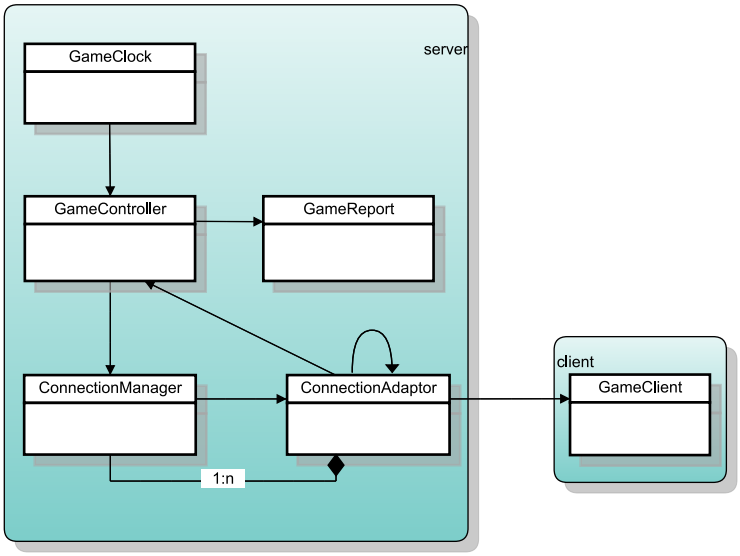

Figure 4: Event dispatching.