The instability of market-making algorithms

An agent-based simulation in miranda.

Elias Court

MEng Computer Science

Submission year: 2013

Supervisor: Dr. Christopher D. Clack

ABSTRACT

The Hot Potato Effect (HPE) is a type of financial market instability that is thought to arise from complex non-linear interactions between market-making algorithms. The main goal of this project was to recreate and observe the behaviour of the HPE by constructing an agent-based simulation, and to construct possible base sets of circumstances for the HPE to occur as well as to evaluate the efficacy of existing and proposed stabilisation mechanisms. In order to model the HPE, a generic agent-based simulator harness was created (which can be reused by other researchers for other experiments), together with a number of agents to support this specific HPE experiment - these included a faithful recreation of the CME E-Mini exchange and numerous other agents such as High Frequency Trader (HFT) market-makers and noise traders.

Table of Contents

1 Introduction

1.1 2010 flash crash and the HPE

On May 6th 2010 the E-Mini S&P 500 equity index futures market experienced a rapid drop of more than 5% in price followed by a rapid rebound with massive repercussions to closely related markets. Separate analyses of the events have been unable to agree on a single "trigger" of the events but do agree that one of the contributing factors was a prevailing negative sentiment. Analysis by Andrei Kirilenko et al.[1] uncovered that at the heart of the crash there was an unusual amount of "Hot Potato" trading amongst High Frequency Trader (HFT) Market Makers (MMs), this is to say that HFT MMs were appearing to buy and sell large numbers of contracts which would normally indicate a large change in inventory.

'We find that compared to the three days prior to May 6, there was an unusually high level of HFT "hot potato" trading volume — due to repeated buying and selling of contracts accompanied by a relatively small change in net position. The hot potato effect was especially pronounced between 13:45:13 and 13:45:27 CT, when HFTs traded over 27,000 contracts, which accounted for approximately 49% of the total trading volume, while their net position changed by only about 200 contracts.' [2]

1.1.1 Hot potato effect

During the flash crash HFT MMs were issuing large sell market orders[3] to offload inventory that they had just purchased, this is uncommon behaviour for HFT MMs as they are normally known to issue limit orders[4] during standard operation therefore seeming to indicate that they switched into a "panic state". Other HFT MMs were purchasing the inventory being panic sold only to panic shortly thereafter and repeat the behaviour, this is the hot potato effect.

The hot potato effect is very difficult to observe in laboratory conditions; if one is to analyse and test a simple, sound, HFT market-making algorithm on its own it is incapable of behaving in such a manner. In fact, analysis leads us to believe that even once introduced into a system as long as the algorithm behaves as intended it should never panic and thus a hot potato event shouldn't be possible. There is, however, no guarantee that in the presence of other elements of a system that any given algorithm will behave as intended. We know that the hot potato effect is possible because we've

---

[1] (Kirilenko, et al. 2011)

[2] (Kirilenko, et al. 2011)

[3] Market order: An order to immediately buy or sell a number of contracts at the best available prices.

[4] Limit order: An order to buy or sell a number of contracts at a set price or better.

seen it occur before in the 2010 Flash Crash and it was attributed to the HFT MMs. By drawing parallels between this scenario and concurrent programming we choose to treat the HFT MMs as concurrent agents and model and analyse their interactions. In doing this we can look for the kinds of conditions needed to achieve the HPE. We discover that if we're to introduce slight, realistic changes to the system such as a small delay in information or forced compliance quotes (see Section 5.2) then the HPE becomes entirely possible and observable in simulation.

1.2 Aim of the project, and objectives

Aim: To try to discover the factors that determine whether or not a system of market-making agents will exhibit instability. The measurable objectives (O1 – O5) associated with this aim are as follows:

- (O1) Build an agent-based simulator and a number of different agents needed for our experiments.
- (O2) Test the simulator first through unit testing each agent then validate by ensuring stability in systems we know are stable.
- (O3) Show that the hot potato effect can be recreated and observed in simulation.
- (O4) Investigate the effects of changing factors such as information delay and compliance quotes, which we believe to cause the hot potato effect.
- (O5) Investigate whether proposed protection measures prevent or dampen instability and if so how effective they are.

1.3 Contributions of this project

This project will contribute the following:

1. A generic agent based simulator written in Miranda with user documentation, which will be used by other students to run experiments.
2. An Implementation of a simulated market to recreate the hot potato effect.
    a. An exchange agent that realistically mimics the E-Mini exchange and its current market stabilisation measures.
    b. A high frequency trading (HFT) market-making agent that mimics the behaviour of HFT MMs during the Flash Crash of 2010.
    c. A noise agent that mimics the effect of many traders buying & selling in the market.
3. An explanation of why the HPE is difficult to achieve and a demonstration of how it can be achieved if information delays or compliance orders are present.
4. An assessment of the efficacy of proposed E-Mini market stabilisation mechanisms in terms of preventing the HPE.
5. A statistical analysis of the results of the experiments.

2 Background

2.1 Different types of trading algorithms

Algorithmic trading has been used since the late nineties/early noughties and employs a multitude of strategies such as volume-weighted average price (VWAP[5]) and time-weighted average price (TWAP[6]) algorithms; both of which aim to minimise the market impact of their activities and obtain an average price for the securities[7] that they wish to buy or sell.

We, however, are particularly concerned with a certain branch of these algorithms called high frequency trading (HFT[8]) algorithms, particularly those that employ market-making strategies.

High frequency trading is the high-speed trading of securities[6] using state-of-the-art computer algorithms and hardware; this enables parties to trade in and out of positions on a microsecond basis and take advantage of small fleeting price changes. *"Fractions of a penny accumulate fast to produce significantly positive results at the end of every day."*[6]

2.1.1 Market-makers

Market-making is a strategy in which the trading party does not take any particular position, that is to say buying or selling exclusively, but rather both. MMs typically place buy-side orders at (or just above) the best bid (highest price at which someone is willing to buy) or sell-side orders at (or just below) the best offer (lowest price at which someone is willing to sell), these typically do not overlap (because if they did a trade would occur), this "gap" between prices is called the spread.

Market-makers make a profit on this spread by selling securities at a price higher than that at which they bought them or equally by buying securities at a price lower than that at which they sold them. HFT MM algorithms are typically inventory driven and so base their order size and price on the amount of inventory currently held. Furthermore, these parties typically have very little long-term interest in the underlying securities being traded; this is because HFT MMs tend to have low capitalisation making it in their best interest to make profits primarily on the short-term bid-ask spread and return to a zero inventory at the end of the trading day.

---

[5] (INVESTOPEDIA US, 2013)

[6] (Palmliden, 2013)

[7] Security: A certificate attesting credit, the ownership of stocks or bonds, or the right to ownership connected with tradable derivatives. (Oxford Dictionary)

[8] (Aldridge, 2010)

HFT MMs typically use limit orders to control their inventory and modify these limit orders or issue new ones as and when necessary. For example, if an HFT MM wants to increase its inventory it will either cancel or reduce the size of existing offers it has placed and either increase the size of existing bids or issue additional ones.

It will use this mechanism to continuously buy and sell throughout the day making money off of the spread. However, should there be a lack of counter-parties also placing orders near the spread an HFT MM might accumulate a large inventory and exceed its inventory limits.

Upper and lower inventory limits are defined to defend the HFT against veering too far from a zero position, making it very difficult to return to one at the end of the trading day. Usually HFT MMs are able to use limit orders to stay within these limits but should one be exceeded the HFT MM will switch into a state of inventory panic. At this point it will start using market orders (see Section 2.3) to trade contracts at any given price in a desperate attempt to return to safe operating conditions.

2.2 Agent based simulation

Agent based simulation[9] (Alternatively agent-based modelling or ABM) is a type of simulation in which the interactions of self-governing agents are simulated for the purpose of evaluating their effects when viewed as an entire system rather than a single entity in a system. It is usually employed with the goal of re-creating or predicting the appearance of complex phenomena in a system, in our case the HPE.

Agent based simulation is in our case preferable to using a mathematical model of differential equations. The mathematical models for relatively large systems such as the one we wish to investigate can be quite complex and as such are hard to synthesise, whereas with agent based modelling each agent can be easily defined and far less complex yet still enabling us to create and inspect the phenomena that arise.

Agent based models comprise agents with simple rules – this approach makes them easier to understand, validate, test and debug than other approaches such as analytical models using differential equations.

2.3 Limit order book

A limit order book is *"A list of all limit orders[10] for a certain security that were placed by members of the public. The limit order book contains orders that have not yet been filled. The orders, however, are not public; only the bookkeeper has access to the details*

---

[9] (Getchell, 2008)

[10] Limit order - "*An order placed with a brokerage to buy or sell a set number of shares at a specified price or better.*" (INVESTOPEDIA US)

of most orders. Market makers and specialists have access only to the highest and lowest orders in order to facilitate trade." [11]

Modern day limit order books are automated and instead of having a bookkeeper to monitor and uncross the book they have a matching engine. The matching engine works by regularly checking to ensure that the best bid[12] and best offer[13] are not "crossed", which simply means that the best offer is lower than the best bid. Should they be crossed the matching engine matches the two orders with each other causing a trade, this occurs at the price of the oldest of the two orders, any remaining stock from the larger of the two orders remains on the book.

Market orders may also be placed; these are simple sell or buy orders for a specified size that must be fulfilled immediately regardless of price. If an incoming sell market order is for a size larger than the depth[14] of the tick[15] containing the best bids (or vice-versa for an incoming buy market order and the best offers) then the matching engine will walk down the book's ticks (or up for a buy market order) until the entire market order has been fulfilled or that side of the book is out of liquidity[16]. This can occur as a result of low liquidity or simply because the market order is very large.

2.4 Related work

2.4.1 Agent based simulation of financial markets

Interest in using agent-based simulation for economic analysis has skyrocketed and plenty of research in the area has already been accomplished. In 2006 Fredrik Nilsson & Vince Darley published *"On complex adaptive systems and agent-based modelling for improving decision-making in manufacturing and logistics settings"* (2006) in which they research the efficacy of using ABM in aiding the decision making process at a packaging company in the United Kingdom.

They found that by using ABM they could construct accurate "what-if" scenarios of the dynamic interaction among several business functions as well as effectively including and paying attention to numerous aspects of financial systems thus providing a better understanding of and explanations for certain phenomena which arise in that setting.

---

[11] (Farlex, Inc., 2012)

[12] Best bid – Oldest, highest-priced buy-side order on the limit order book

[13] Best offer – Oldest, lowest-priced sell-side order on the limit order book

[14] Depth – The total volume of orders.

[15] Tick – Discrete price points on the limit order book on which orders may be placed.

[16] Liquidity – "The ability to convert an asset to cash quickly. Also known as "marketability." (INVESTOPEDIA US)

Blake LeBaron has also done a lot of work in assessing the suitability of ABM in the world of economics; one of his papers *"Agent-based Financial Markets: Matching Stylized Facts With Style"* (2004) summarises the benefits of using ABM and argues that they are preferential to single representative agent models in finance.

2.4.2 Hot potato effect

Very little research into the hot potato effect has been done aside from that by Kirilenko, A. et al. in their paper entitled *"The Flash Crash: The Impact of High Frequency: Trading on an Electronic Market"* (2011).

However, even in Kirilenko's paper very little attention was given to the HPE except for noting its existence, defining/describing it and giving some conjecture as to what may cause it.

Kirilenko noticed that during the events of the flash crash HFT MMs were observed to have been issuing very large and aggressive sell market orders to get rid of inventory that they had only just purchased. For an HFT MM using market orders is very unusual, they normally tend to issue bids close to the best bid and offers close to the best offer in order to make a profit off of the bid-offer spread[17].

This behaviour indicates that the HFT MMs were switching to a panic state when they accumulated too large a position (See section 2.1.1). During this period of time the behaviour was noticed to have been occurring repeatedly as other HFT MMs purchased the contracts being panic sold only to repeat the same behaviour moments later themselves. This cyclic behaviour is the hot potato effect.

Despite the lack of research on the hot potato effect it has been mentioned in numerous papers and reports, all analysing the May Flash Crash or other Mini Flash Crash events such as *"High Frequency Trading and Mini Flash Crashes"* (2012) by Golub, A. et al. and *"The impact of High-Frequency Trading on Markets"* (2011) by Zhang, F. & Powell, S. B.

[17] Bid-offer spread: This is the difference between the highest priced bid and the lowest priced offer on a limit order book (See section 2.3).

3 Simulator Analysis & Design

We've decided that an agent-based simulator would be the best mode of simulation for our purposes. This would allow us to define numerous simple agents with the goal of looking at the large scale effects on the system as well as looking at a single agent alone and the system's effects on it. This also allows for easy debugging because we can just look at a single agent and the messages it receives in a particular step and ensure it sends out the desired responses to those inputs according to its rules.

If any single agent is behaving unexpectedly (and it isn't a bug) then we can also monitor it on its own through the messages it receives and sends out and try to determine the actions that lead to it exhibiting the behaviour.

Agent-based simulators also allow us to look at the system as a whole at each time step and thus easily identify repetitions in states.

In this section I outline the requirements of the simulator and analyse them using the MoSCoW method. I then describe at a high level the logic necessary for meeting each of these requirements.

3.1 Simulator requirements

Here are the requirements for the proposed agent-based simulator:

- **Must have:**
    - The simulator should operate in discrete time steps so that we can easily see the actions and reactions of each agent by time step.
    - The simulator should be agent based so that we can keep the agent definitions simple while being able to observe the higher-level effects and to keep debugging simple.
    - The exchange (limit order book) should (i) be an agent in our simulation; (ii) be realistic, so that we can validate the efficacy of proposed prevention measures and (iii) be designed to closely mimic the E-mini exchange involved in the May 6 2010 Flash Crash because this was the exchange where the HPE was observed.
    - The simulator harness must be as generic as possible to allow the user the maximum amount of freedom possible to specify the simulation to be done. However, there are some basic features that must be present; the simulator must be able to facilitate the interactions of agents and display it to us in an easily digestible manner- messages would seem to be the obvious choice.
    - We wish to be able to view all messages that are sent by all agents throughout the entire duration of a simulation for debugging and analytical purposes.

- **Should have:**
    - We wish for others to be able to reuse the simulator and thus want to keep it as generic as possible and so should be structured as (i) an immutable simulator harness and (ii) user defined agents, with a well specified interface between them to which the user must conform when coding the agents.
    - Agents will communicate solely through messages that are channelled by the simulator harness, the harness will also randomise the order in which these messages are then received by agents in order to prevent any phenomena that may, or conversely may not, arise due to simulator bias.
    - The simulator will be implemented in Miranda because development time is much shorter than that of other languages such as C++ or Java. This is because almost all of the errors Miranda throws are from the strong static type checks, which make runtime errors less probable.
    - We may also want some information about the simulation at each timestep that is dependent on all the previous steps of the simulation but recreating this by working through the messages being transferred one by one is tedious. Furthermore, some information from the simulations may be better presented in csv format so that we can easily plot graphs and manipulate the output data in various ways, therefore we want the simulator to have some facility for accepting additions to this data file, which would be "snapshots" of the current state of the simulation, and printing them out in CSV format. It seems the easiest way to do this would be to take the same approach as with broadcasts and define a special message type that is treated differently to "normal" messages.
- **Could have:**
    - We wish for agents to be able to send out these messages to large groups of agents; these messages would perhaps contain less sensitive information such as general market statistics, thus we wish to define an additional case of messages known as broadcasts which our simulator will treat differently in that instead of sending it to one agent it will send it to a group of them. This also means we somehow need to accommodate a notion of "agent/broadcast groups".

3.2 Simulator framework design

Figure 1 is a diagram of the structure of the simulator that illustrates the recursive nature of the system.

The simulator is driven by a main "harness" element that you can see in Figure 1 labelled "sim". Its purpose is to place an initial element labelled "startsimstate" in the list labelled "allstates" to allow for the circular dependency visible in Figure 1 before initiating the loop. At each lap of the loop (a timestep) the simulator harness outputs to files all of the messages exchanged at that particular timestep.

This simulator design is an extension of an initial design by Christopher D. Clack in 2011.

3.2.1 Style of programming in Miranda using infinite streams

We've decided to use infinite streams of information as our method of channelling messages from agents to the harness, and agent-unique redacted sim states[18] from the harness to each agent. This means that all the harness or the agents ever have to do to find the input for their next iteration is look at the next element in this stream. It's important to note that the simulator creates its states by combining and modifying the sets of messages received from the agents in the previous time step and these agents create the sets of messages they send out by looking at the state received from the simulator. This of course means there must be a base case, which is a default state (startsimstate) that all agents receive upon the simulation starting.

Below is a summary of how this scheme works.

1. The harness checks for messages from the agents, to output them and to forward them.
2. This causes the agents to then create the messages that the harness is looking for.
3. In order to do this they must operate on the simstate from the previous time step.
   Note: In time step 0 they receive a default state.
4. The harness then moves on to the messages from the next time step. When the agents now check for messages from the previous time step they find that they now exist.
5. Repeat step 1-4 until the end of the simulation.

[18] These "redacted" states from harness to agents are modified to contain only the information that the recipient agent is allowed to see.

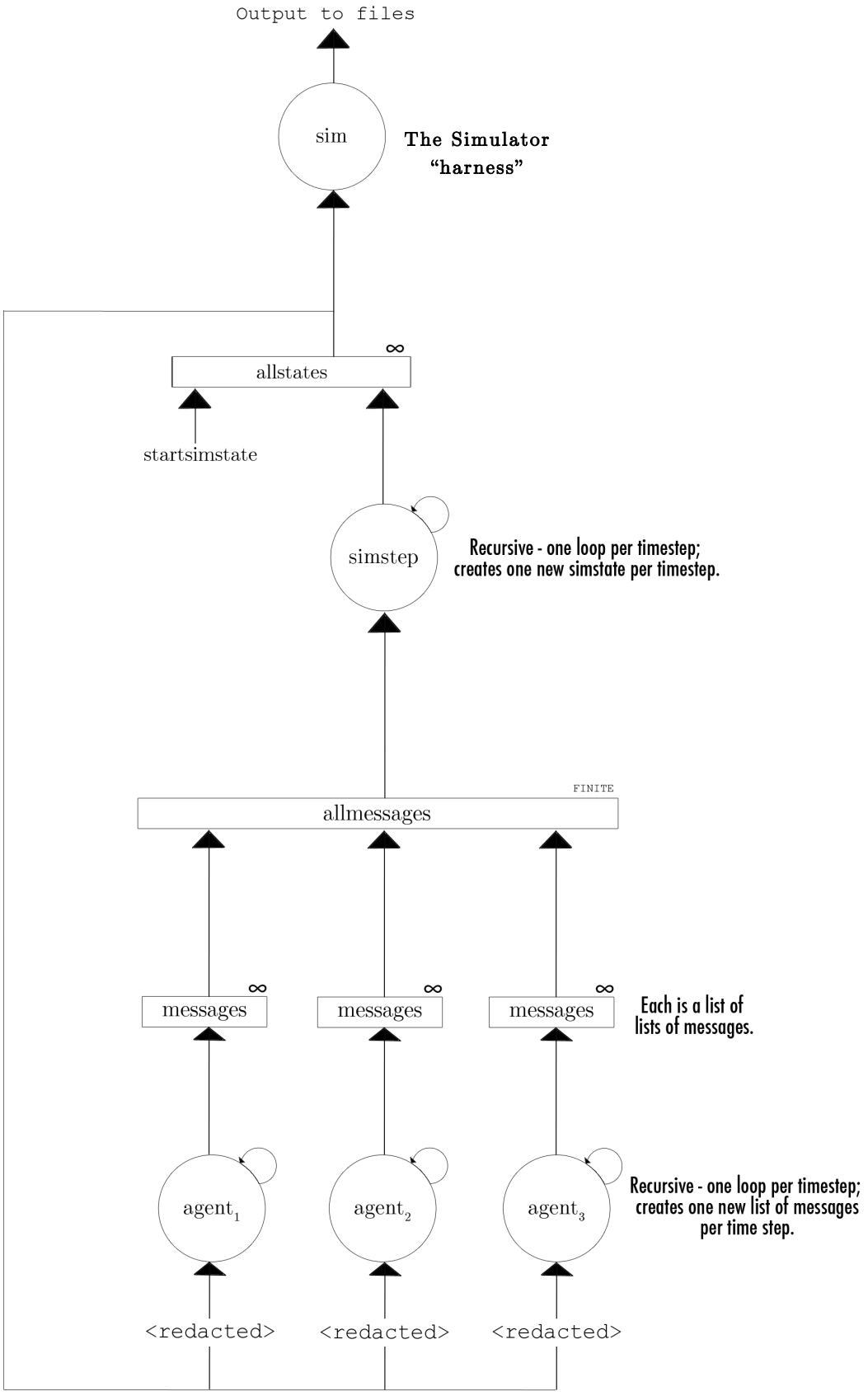


Figure 3-1 – showing the main components of the system

<u>Important note</u>: The simulator and agents must only ever look at the first element of the input stream. This is a design feature that requires some user discipline.

I will explain the reasoning for this through example.

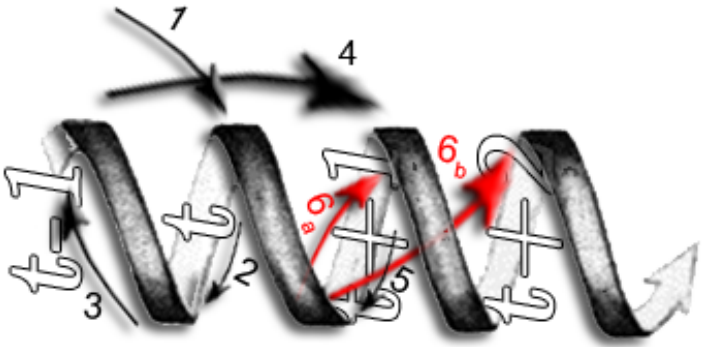


Figure 2 – In this diagram the troughs represent collections of agent messages for a timestep (an item of allmessages) and the peaks represent a single simstate (an element of allstates). In this diagram the peak at t-1 has already been generated. Time is flowing in the direction of the helix (left to right).

Each element of allstates is generated by processing every agent's output stream. The very first item (at t=0) in allstates is the only exception as this must be predefined. All other elements in the stream are computed, and this computation depends on the outputs of the agents. This computation is captured by arrows 1 through 4 in Figure 2 (where the t-1th timestep's simstate, an element of allstates, has already been computed): a demand for the tth timestep's simstate is made by the agents (arrow 1) which requires all agents' outgoing messages from timestep t-1 to have been computed (arrow 2) which in turn requires that the simstate for t-1 must have been computed (arrow 3); because this simstate already exists the agents can create their outgoing messages, the harness can create the tth timestep's simstate and program flow can continue to the next timestep (arrow 4). From the simulator's perspective, at time t, the outputs from the agents are used to calculate the element in the agents' input streams at time t+1. From an agent's perspective, at each time step it consumes the head of the current input stream and computes the next item in its output stream; the agent then calls itself recursively on the tail of the stream – at each time step the agent therefore sees as its current input stream the tail of the input stream seen at the previous time step, and the head element of that tail will depend in part on the value just output by the agent. Thus, if an agent attempts to inspect any element in the current input stream other than the first element, there will be a circular dependency on the agent's output that cannot be resolved and the agent will hang.

Consider if an agent at some time step t were to look at the third element of the input list (essentially looking at t+2 – arrow 6b in Figure 2). For this element to exist the agents would have needed to send out the messages for timestep t+2 but for in order to do so they would require the state at t+1, which depends on the values being generated in this iteration (at time t) and so the dependencies would form a cycle and the program would hang.

Each iteration through the simulator harness and consequently each index in the harness' list of states represents a single time step. Agents are only permitted to interact with each other through this harness and therefore each unmodified sim state is a faithful depiction of the entire system's global state. Any messages sent by an agent on any given time step are not delivered to the recipient until the next time step.

3.2.2 Generic handling for user defined agents

We wish to create a simulator that can work with user-defined agents and so we need to create a type synonym that the user defined agents must conform to – this is so that the harness can call the agents with full knowledge of the parameters they expect and the types they return.
The agent type should take in an infinite stream of filtered system states (as outlined in Section 3.2.1) as well as a list of arguments.
The agents must also be able to maintain a private internal state so that they can act according to that as well as the external state, this internal state must also be able conform to the users needs and therefore we allow the user to define them.
The agent should return an infinite stream of messages & broadcasts to the simulator. At each time step multiple messages (of type msg_t) can be sent, so the output is of type [[msg_t]].
Below we propose a type synonym agent_t for user-defined agents:

```
agent_t      == agentstate_t -> [arg_t] -> [(num, [msg_t],
                  [msg_t])] -> num -> [[msg_t]]
```

An `agentstate_t` is an instance of the agent's internal state – this is user defined. The list of `arg_t` is a list of key, value pairs that will modify the runtime behaviour of both agents and the simulator harness (see Section 4.1). The next element infinite list of redacted simulator states, where the initial num (number) inside each inner element is the timestep, the first list of messages (type msg_t – see Section 3.2.3) is the list of incoming messages at that time step and the second list is the broadcasts (see Section 3.2.4) for that agent.

3.2.3 Generic handling for user defined messages

The simulator must also operate with user-defined messages without problem and we need to define a type that the harness can handle and the user can define. In order for the harness to properly sort and forward messages it must:

- Have access to the recipient ID and as such every single message case must support this functionality.
- It would also seem logical to contain a sender ID so that recipients can check whom they received the message from.

- Each user-defined message must conform to a message abstype that enforces that any message case permits at the very least the two recipient and sender id getter functions outlined in the coming code snippet to be called on it as well as a function that converts it to a string (for output purposes). Below shows the first three methods for the msg_t abstype, which may be extended as more functions are required for messages.

```
>abstype msg_t
>with
>    msg_getid :: msg_t -> num
>    msg_getfromid :: msg_t -> num
>    showmsg_t :: msg_t -> [char]
```

- We also wish to have a number of messages for the harness itself such as data messages, debug messages (that do nothing but print to the message log) and broadcasts, which we address in the next subchapters.
- We wish to be able to send a message to a group of agents instead of a single agent in some cases.
    - We wish to maintain the same level of customisation as with messages however we want to introduce this new functionality at the harness level (labelled "The Sim harness" in Figure 3-1).
    - To do this we define a message case that the harness treats differently called a broadcast message. A broadcast message contains (as well as recipient group and sender identification) a broadcast that has the following extendable abstype that enforces that each case be convertible to string and defines a single constructor and getter method.

```
>abstype broadcast_t
>with
>    showbroadcast_t :: broadcast_t -> [char]
>    broadcast_getnumlist :: broadcast_t -> [num]
>    broadcast_numlist :: [num] -> broadcast_t
```

    - We then have these recipient IDs correspond to groups of agents instead of singular agents and so must implement some mechanism for allowing agents to enrol in these groups.

3.3 Design of specific agents

In addition to creating the harness to drive the simulation we also need to design the agents specific to the simulations we wish to run. It is clear that we need HFT MM agents to instigate and propagate the hot potato effect (Section 3.3.1). We need an exchange agent that is a faithful recreation of the exchange involved in the flash crash (Section 3.3.2) – this has never been done before and is to investigate whether or not the protection measures currently in place, as well as proposed protection measures, can prevent the hot potato effect. Finally, we also desire a noise agent to recreate the statistical effects of many parties participating in the market (Section 3.3.3).

3.3.1 The HFT MM agent

We choose a very simple design in which HFTs place limit orders of the maximum size possible that won't send them into a panic. The HFT MMs operate under the assumption that each limit order will rest on the book for only a single timestep before expiring and thus issue new orders at each timestep.

3.3.1.1 Use of limit orders on both sides of the book

As market-makers our HFTs will place orders aimed at making returns on the bid-ask spread, however they are completely disinterested in long term returns and wish to maintain a net inventory of as close to zero as possible. This means that our HFTs will be primarily inventory driven and will use limit orders on both sides of the book to control their inventory whenever necessary.

3.3.1.2 Sizes of orders

The HFT MM will determine its bid order size and offer order size using the two functions shown in Figure 3-2. As an HFT MM amasses more inventory moving towards its upper limit[19] (UL) it will place smaller bids and larger offers, as shown by the bid function sloping towards 0 as we approach the upper limit and the offer function sloping towards 2LL. Note: Though the y values of the red (offer) line are negative order sizes cannot be negative – we take the absolute value of these y values.

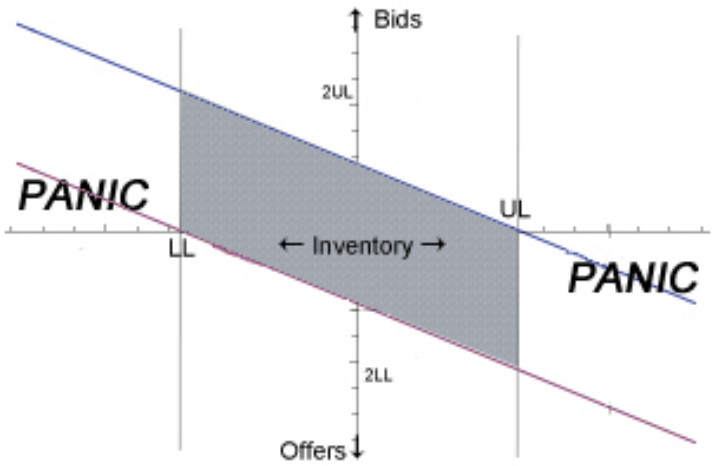


Figure 3-2 – Diagram with order size functions for offers (2UL to 0) and bids (0 to 2LL).

Here are the two formulae that define the plots in Figure 3-2, the functions calculate the maximum order size possible that if fulfilled would not exceed inventory soft limits:

```
>   totbidsize = max [0, UL - 1 - inventory]
>   totoffersize = max [0, inventory - LL - 1]
```

If the total active limit orders on the limit order book stay in the shaded area the HFT cannot panic. If, however, they are in panic then these orders are issued as market orders of size UL or LL respectively and therefore have no pre-set price.

3.3.1.3 Prices of limit orders

We price our orders using a function of the best bid, best offer, current inventory and soft limit size (this is the upper limit). As an HFT gets closer and closer to its panic zones it has to make its corrective orders more and more attractive so we have to reflect this in our pricing.

---

[19] UL and LL are the upper and lower inventory soft limits the HFT operates within, if it exceeds either then it "panics" and uses market orders to attempt to get its inventory back within the limits.

We chose to use the following formula for our pricing:

```
>baseoffset = -(((bo - bb) - 1)*(inventory / softlimit))
```

This offset is then added to the best offer and best bid to calculate the offer price and bid price respectively. I would like to remind the reader that the inventory can be negative and so when it is negative the entire baseoffset term is a positive value, which when added to the best bid creates a higher, more attractive bid price, reflecting the fact we wish to buy, and a higher, less attractive offer price, reflecting the fact we do not wish to sell and vice versa for positive inventories.

In addition to this our HFT MM agents should be aware of the price banding constraints enforced at the exchange and select their prices to conform to this. When adding our offsets we first ensure that the best bid and best offer are each within the price band (± 12 of the last traded price), if they are not then we add the offset to the closest band instead. We then apply the strict banding to ensure the price is within the last traded price ± 12, rounding to the nearest band if the price is outside it.

3.3.1.4 Advanced order management

In reality HFTs often split their orders up into many smaller orders with prices around their base price in an attempt to misdirect other traders. This is because competing companies use their own proprietary trading algorithms with the aim of outcompeting, mimicking or taking advantage of competitor algorithms and so algorithm owners do not want competitors to be able to backwards engineer their algorithm.

After the base order size and price has been calculated the order is split up into a 10 smaller orders. Each of these sub-orders has a unique price drawn from a normal distribution of prices with a mean of the base price and a maximum spread of ± 3 from the mean.

As a result of this change in price the new price may lie outside of area enforced by the price bands, thus we reapply the price banding. In order to prevent an excess of orders being placed at the band itself at this stage we “reflect” prices around the closest band instead of rounding them. E.g. an upper band of 3 and an initial price of 3.3 would result of a corrected price of 2.7.

3.3.2 The exchange agent

We design our exchange to mimic the E-mini market exchange as closely as possible to later assess the efficacy of the existing and proposed safeguards. The E-mini exchange supports many order types. However, some of them can be effectively recreated using a combination of simpler order types and some of them we aren't interested in using in our simulation so they won't be included. The types we need follow[20]:

- Limit order
  - Good until date
  - Good until cancelled
- Market order
  - Fill or kill
  - Fill and kill

3.3.2.1 The generic management of bids and offers on the order book

We will need a data structure for maintaining the orders on the book. We choose to use two ordered lists of lists; one for buy side orders and one for sell side orders. Each list in the encompassing list represents a single tick on the book; in order to save memory we only have a tick present if there are orders present at that tick. We have the buy side list in decreasing order of tick prices and the sell side list in increasing order of tick prices to save on complexity when searching for the best bid/best offer respectively.

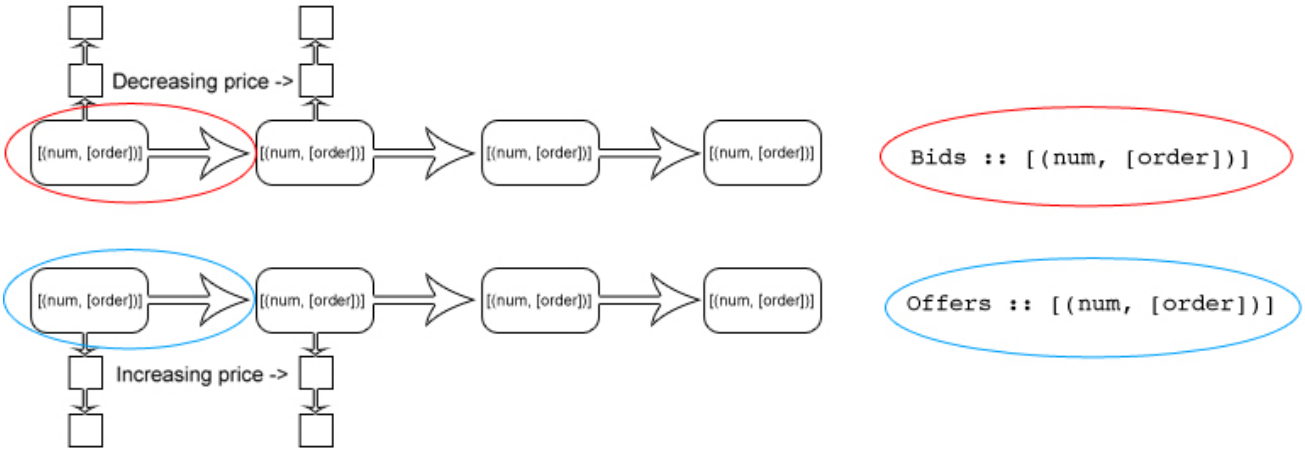


Figure 3-3 – Diagram showing the two data structures used to maintain orders on the books.

Each element of the encompassing list (the larger boxes in Figure 3-3) is a 2-tuple containing the price of the tick and a list of orders placed at that tick (the smaller boxes coming out of the larger ones in Figure 3-3).

[20] (CME Group Inc.)

Searching for the best bid or offer will be as simple as taking the leading element of the list. This simplifies order matching, checking if the book is crossed etc. because the majority of activity happens near the best bid and best offer, which are the heads of the lists. The offers data structure has a reverse price-order to the bids data structure for this precise reason, the best offer will be the lowest one on the book with higher offers being less attractive.

3.3.2.2 Matching market orders

Market orders are different to limit orders in that they have no specified price; instead they are matched against the leading orders of the opposite side of the book in series for the entire requested size.

Our exchange agent matches market orders by "walking" along the data structure for the opposite side of the book to the incoming market order, first within a tick until all of the orders are matched and the tick is made empty, then along to the next tick and so on. It continues to do this until it has matched the market order against enough counter-orders to satisfy the entire requested order size. If there are no longer any orders to match the market order against then the remaining order is rejected.

3.3.2.3 Uncrossing the book

On the arrival and addition of an order the exchange checks to see whether the book is now crossed (the best offer is at a price lower than the best bid), if this is the case then it matches the best offer against the best bid at the price of whichever order was on the book first and reduces their respective order sizes or removes the order entirely if the order size is reduced to zero. The book is then rechecked to ensure that it is no longer crossed, if it still is then the process is repeated.

3.3.2.4 Safeguards at the exchange[21]

In this section we detail the pre-existing and proposed safeguards to be implemented at the exchange that we wish to check for efficacy.

Price banding

Orders placed on the book must be within +/- 48 ticks of the last traded price, this is to prevent orders being placed on the book at extreme prices that could cause sudden jumps in traded price if liquidity is depleted.

Max order sizes

Orders of any type cannot exceed size 2000; this is to prevent any one trader from having a colossal impact on the market with their order.

---

[21] (CME Group Inc.)

Stop spike logic

If an order reaches the exchange and it is executed (either for being a market order or for being crossed) and this causes the price to jump by +/- 600 ticks from the last traded price then that trade is rejected and trading is stopped for 100 timesteps.

The E-Mini stop spike logic pauses trading for 4 seconds. Although there is no direct correspondence between real-world seconds and simulator timesteps, we choose to pause trading for 100 timesteps (this value can be systematically varied in experiments to determine whether market benefits are sensitive to its precise value).

During this pause market orders cannot be submitted and limit orders can but won't be uncrossed. Note that this can only be triggered by market orders because of the price banding applied to limit orders.

Maximum allowed stock on book

Finally, no one trader can have more than +/- 100,000 net futures on the book, this is to prevent a single trader from massively skewing the market in one direction.

Minimum resting times (proposed)

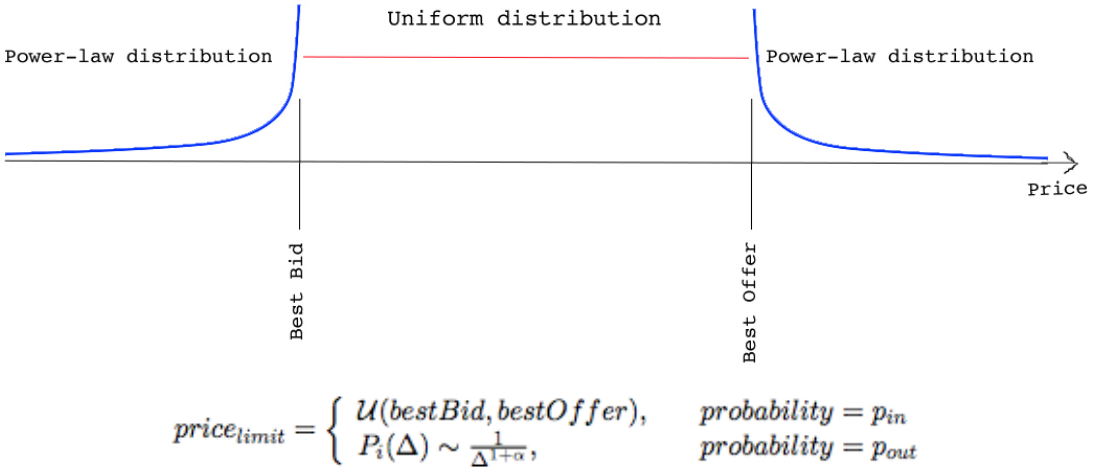
Orders must remain on the exchange without being cancelled for a specified minimum time; this is to counteract HFTs placing and cancelling orders en masse at very high speeds and overwhelming the market.

3.3.3 Noise Agent

We wish to simulate trading in a realistic environment and thus want a noise agent that mimics the statistical effect of a large number of traders participating in the market. It will follow a simple design which takes a number of parameters to modify its behaviour: probability of submitting a buyside order (50%), probability of submitting a limit order (35%), probability of submitting a market order (15%), probability of submitting a cancellation order (50%), probability that a limit order is within the spread (35%)[22].

The noise agent will operate by drawing three uniformly distributed random numbers to determine the type of action it will undertake at any particular moment according to the probabilities it is using. Prices inside the spread will be drawn from a uniform distribution between the best bid and best offer, prices outside the spread will be calculated by drawing an "offset" from a power law distribution which will be added (or subtracted) to offers (or from bids) before applying price banding (as detailed in Section 3.3.1.4).

[22] (Airimitoaie, 2012) – Section 4.4.1 Noise Traiders



$$price_{limit} = \begin{cases} \mathcal{U}(bestBid, bestOffer), & probability = p_{in} \\ P_i(\Delta) \sim \frac{1}{\Delta^{1+\alpha}}, & probability = p_{out} \end{cases}$$

Figure 3-4 – The formula for pricing limit orders and accompanying probability graph.[23] Δ is the offset from the spread, p_{in} is the probability of being inside the spread and p_{out} is the probability of submitting outside of the spread.

Market order sizes will be the size of the opposite side of the book at the best price; limit order sizes will be drawn from a log-normal distribution.

$$orderVolume = \begin{cases} [v(bestBid)|v(bestOffer)] & \text{for market orders} \\ P_i(\log v) \sim \mathcal{N}((\mu, \sigma)) & \text{for limit orders} \end{cases}$$

Figure 3-5 – The formula for calculating order size.[24] v(o) is the volume of the order o and N is the normal distribution with mean μ and standard deviation σ.

[23] (Airimitoaie, 2012)

[24] (Airimitoaie, 2012)

4 Simulator Implementation

The simulator can be separated into three key components: The harness, the agents and the messages. The harness enables the agents to communicate using these messages. In this chapter I will detail how the harness does this, how one would implement runtime modifications to the simulator behaviour and how one is able to define his or her own messages and agents.

4.1 Modifying Simulator Behaviour

We may want to modify certain elements of how the simulator harness and agents behave at runtime, for example turning randomisation of messages (see Section 4.2.2) on or off and having a particular agent be more inclined to buy or sell; runtime arguments (args) seem the best way to accomplish this. One motivation for this is so that we can automate the execution of successive simulations with different parameters and store them to different files. As mentioned, some of these args will be for the simulator harness itself, these will be parameters such as the location in which to output trace and data files or whether or not to randomise messages. The simulator harness will also pass down the entire list of args to each agent and each agent will then be responsible for determining which args are relevant to it and act upon them.

Arguments are of the type arg_t (defined below) and are passed to the simulator at the start of a simulation. Both the simulator and the agents to have access to the entire list of arguments, which may vary each time the simulator is started, and the args allow for different pre-coded harness and agent behaviours to be selected when the simulator is started.

Data of type arg_t is either the value EmptyArg (a place holder in case an arg_t is required but there is no appropriate data to use) or the value Arg – the latter contains data of the type (str,num) and has a number of methods predefined for it.

```
arg_t ::= EmptyArg | Arg (str,num)
```

The predefined methods are:

- arg_getstr when passed an arg_t will return its string element (its “key”).
- arg_getnum when passed an arg_t will return its num element (its “value”).
- arg_findval when passed a key string and a list of arg_t will search through the list for the arg with the string that matches the key and return the num element, if it cannot find a matching key it will return -1.
- arg_findstr does the same as findval but this time the key should be a num instead of a string and it matches against the num elements eventually returning the corresponding string if it finds a match and the empty string otherwise

Agents can then use these four functions to find arguments by string or num key inside the argument list, which is passed to them by the harness.

No two args should share the same key because the find functions will only return the first match. Args are almost always searched exclusively by string key and searching by value is only expected for just one[25] "magic" number, this must obviously not share it's num key with any possible num value in the other args.

4.1.1 Modifying the simulator harness behaviour

There are two[26] args that modify the behaviour of the simulator:

- (Arg (String "Randomise", x))
    - This makes the simulator harness nondeterministic in the order in which it passes messages around. (See Section 4.2.2)
    - If the randomise arg is not used the sim is deterministic. The value x can take any number but is typically zero – it's value is simply ignored.
- (Arg (String prefix, 9989793425))
    - This "prefix" string corresponds to where the simulator should write out the data and message logs.
    - E.g. if prefix is "~/home/test" then the message log will be printed to "~/home/test-trace" and the data will be printed to "~/home/test-data.csv".
    - The number 9989793425 is a unique key used to locate this argument by the simulator; as such no other args should contain this number.

[25] This mechanism should be changed if any more "magic" numbers are used because it will become difficult to ensure that they do not share the same "magic" number with a num value.

[26] In reality there is currently at least one additional arg responsible of the control of an automated statistical suite that is integrated into the simulator but this is not ready for this release.

4.2 The simulator harness

The simulator harness is a function called sim with the following type:

```
>sim :: num -> [arg_t] -> [(agent_t, [num])] -> [sys_message]
>sim steps args agents
```

The arguments to sim are (i) the number of steps the simulation should run for, (ii) a list of arguments (of type arg_t) to the simulator, and (iii) a list of tuples of agents participating in the simulation and their respective lists of broadcast subscriptions (numerical ids). The broadcast subscriptions of an agent are assigned statically at the start of a simulation and cannot be changed during the simulator run (see Section 4.3.3 for more details of broadcast messages). The function returns a list of sys_messages - these are the print-to-file commands.

The simulator operates by invoking each agent function (each of which is recursive) with four arguments: (i) a default emptyagentstate (which is internally modified by each agent to their own state type), (ii) the entire list of args, (iii) a potentially infinite list of redacted simulator states, and (iv) the agent's numeric id. Each of these agents then returns a potentially infinite list of lists of messages. The potentially-infinite output from each agent has one item per timestep, and at each timestep an agent can send zero or more messages, so each item itself is a (finite, but of varying length) list. The list consisting of all these lists will from here on be known as allmessages – it is a finite list, its length being the number of agents participating, of potentially-infinite lists of finite lists of messages:

```
>allmessages = map f (zip2 [1..] agents)
>              where
>              f (id,(a, brcs))
>               = a emptyagentstate args (map (g id) allstates) id
>                 where
>                 g x st
                   = (sim_gettime st,
>                     sim_getmymessages st x,
>                     concat (map (sim_getmybroadcasts st) brcs))
```

The redacted states are obtained by mapping a redact function (above called `g`) over the list allstates; this is done once per agent as the redactions are agent specific.

Allstates is primed with a default state, startsimstate, which provides a starting point for the recursion – in diagram 4-1 this recursion would be traversing through the loop in an anti-clockwise direction starting at allstates:

```
>allstates = startsimstate : simstates
>simstates = simstep steps 1 args startsimstate allmessages myrands
```

The agents evaluate the list of redacted states in sequence to create their messages at each timestep to send out, this is used to create the next element of allstates through a single iteration of simstep.

Simstep is a recursive function that returns a potentially infinite list of simulator states; it is called with allmessages as one of its arguments. On each call of simstep it takes the single leading list from each item in allmessages and passes them to sim_updatestate along with a "clean state", an empty simstate with skeleton structures in place for the messages and broadcasts, which sorts them into a list of lists by destination before randomising each list.

Sim_updatestate state does this recursively by working on a single element from the list passed to it by simstep until that list is empty. Each element in that list is a list of messages from a single agent, it splits these into broadcasts and messages before recursing over each agent id filtering out the messages and broadcasts addressed to it and appending them to the existing lists of messages and broadcasts to that agent.

Finally it funnels the return data into a simstate (which is exactly that list and some additional information such as time). It then creates an output list containing this result as the head of the list and the tail of the list being a recursive call to simstep.

Simstep decrements 1 from the number of timesteps to run for on each recursive call, terminating at 0 (not shown in presented code fragment) and thus ending the simulation.

As you can see in Figure 4.1 there exists a cyclic dependency between allmessages and allstates that is satisfied by the loop through simstep, which creates the next simstate, and the next recursive call of each agent function to generate the next set of messages that simstep uses.

Thus it is important that simstep and every agent function only ever look at the leading element in their respective infinite lists otherwise the program will hang because this cyclic dependency is broken.[27]

```
>simstep n t args st msgs myrnds
>  = newstate : (simstep (n-1) (t+1) args newstate
>                        (map tl msgs) (drop t myrnds))
>newstate = sim_updatestate t args cleanst (map hd msgs) myrnds
```

Simstep can terminate in three ways: (i) if steps = 0, (ii) if agentmessages = [ ], and (iii) if any agent terminates (i.e. if its list of messages becomes [ ]).

4.2.1 Simulator output

While running the simulation, at each time step we write out to file all the data collected from that time step. This is to minimize the amount of memory required by the simulator. If we wanted to analyse the data on the fly we would have to retain all the previous states while the simulation was going on. In certain simulations this could result in us running out of memory because of the sheer number of messages that are always being sent.

Thus, the solution is to wait until the simulation has finished and then pass the list of simstates to a tracer function that prints the messages to the trace or data files depending on type. Afterwards, we may read in the data required from file for our required statistical analysis before outputting the results to another file.

Multiple runs of the simulator (for statistical tests) are controlled by another function that calls sim many times, each run being given a different file prefix for storing results. The same function will then, after all simulator runs have finished, call another function to read in from all these files and run statistical tests. All of these processes could therefore potentially operate within a single Miranda program.

[27] This recursive technique is based on a template by Clack.

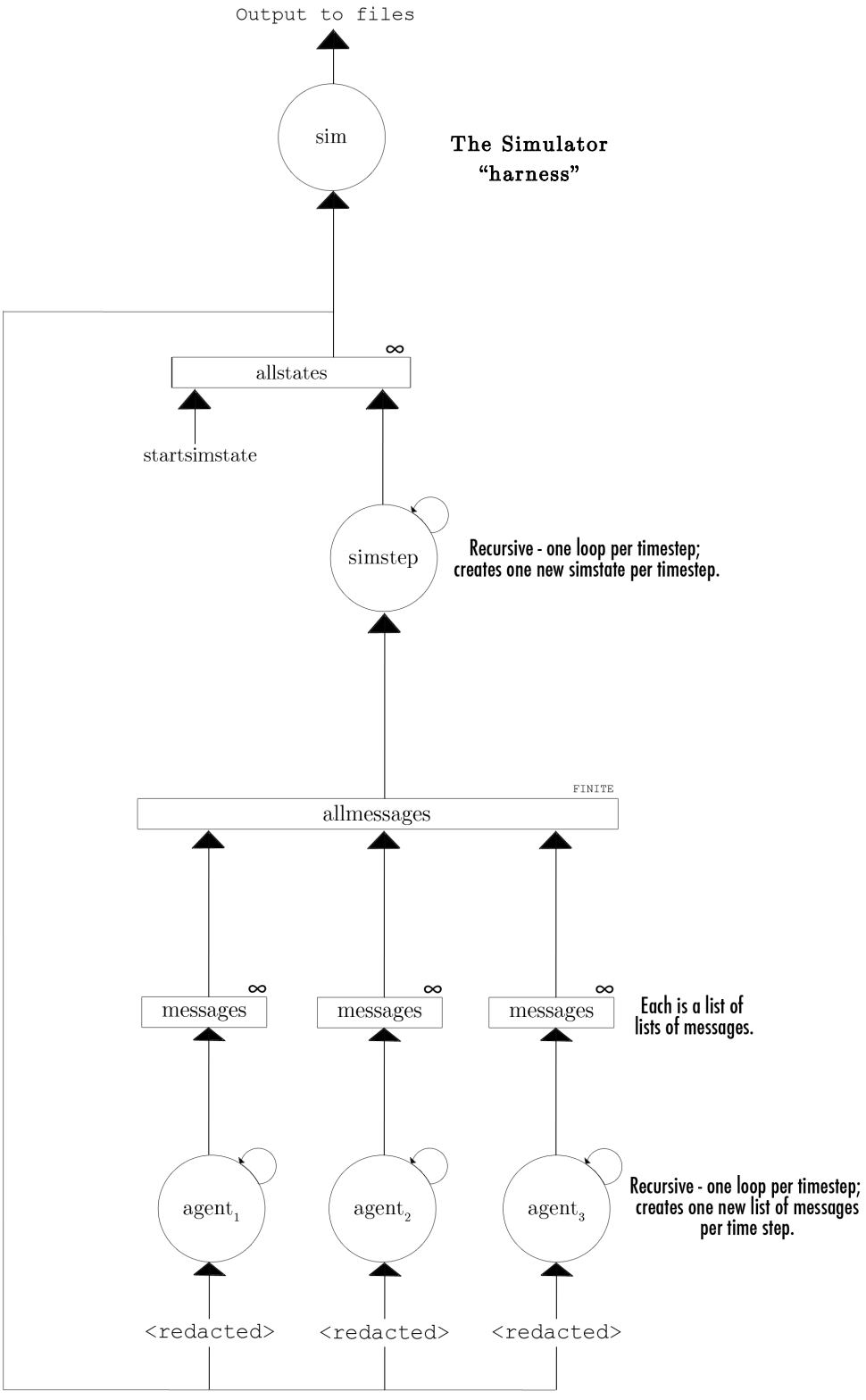


Figure 4-1 – Diagram of the flow of information within the simulator.

4.2.2 Randomisation of message arrival

Without a specific mechanism to achieve the random processing of messages, the simulator harness would be entirely deterministic; it would process and forward the messages from each agent in the same order at every time step.

This is a problem because it would provide a preferential bias to some agents and therefore we would not be sure whether certain observed behaviours were due to the agents themselves or due to the bias and so we decide to randomise these messages.

To avoid bias, the simulator harness therefore ensures that the messages arriving at each agent at each timestep are made available to that agent in random order. A subset of pseudorandom numbers (obtained from HotBits[28]) is used to create a list of (message, num) tuples that are sorted in increasing order before being turned back into a list of messages. However we noticed upon successive calls to sim (for separate simulator runs) that we were using the same pseudorandom list for each call and thus getting the exact same run. We then thought about using the nth element of the pseudorandom list to index into it at the start of the simulation, in effect creating an entirely new pseudorandom list, but that element used as an index would always be the same. We extended this idea to using an external system seed to find the element to index into, which we decided to be the system time. However we noticed that simulations were still coming out exactly the same on successive runs and this was due to Miranda "memorising" the return value of the external system time call because the arguments are always identical despite the return value changing. We ultimately decided to have this seed provided by the user at the beginning of each run in the form of an argument.

Internal to any one simulation we refresh the list of random numbers by dropping used numbers off the front of the list.

[28] http://*www.fourmilab.ch/hotbits/*

4.3 Messages

Messages are the sole method of communication between agents and the harness and they must all be of type msg_t (an abstype).

4.3.1 The message type

The msg_t message abstype employs the mymessage_t algebraic type that is defined as follows:

```
mymessage_t ::= Hiaton | Message (num,num) [arg_t]
                |… more message types …| Examplemessage (num,num)
```

Every message (except Hiaton) must contain a two-tuple of numbers; these correspond to the sender ("from") and recipient ("to") numeric ids.

The abstype msg_t contains constructor methods for each case in mymessage_t (and a new constructor should be defined for msg_t if a new message variant is added to mymessage_t). The Hiaton is a message that corresponds to "no message" – this should be used when an agent wants to convey explicitly that it has nothing to send.

4.3.2 msg_t functions

There are a number of pre-existing functions in msg_t and other functions are easily added for user message types, these are the following.

- msg_getid :: msg_t -> num
  - Example case: msg_getid (Examplemessage (from, to)) = to
- msg_getfromid :: msg_t -> num
  - Example case: msg_getid (Examplemessage (from, to)) = from
- showmsg_t :: msg_t -> [char]
  - Example case: showmsg_t (Examplemessage (from, to)) = "This is an Examplemessage"

4.3.3 Special case: Broadcasts

There is a special message variant (Broadcastmessage (num,num) broadcast_t) where broadcast_t is an abstype for a broadcast message.

Similarly to mymessage_t users are able to extend broadcast_t as they see fit.

The simulator passes the broadcasts down to each agent in a separate list to the standard messages.

4.3.4 Special case: Data messages

Data messages are a particular case of mymessage_t and are treated differently from other messages in that instead of being printed out to the message log they are printed out into a csv file. The simulator adds newline characters after each timestep and agents are expected to send their column headers at the start of a simulation; the data messages

sent in subsequent time steps must be actual data, with a one-to-one correspondence of data values to the initial headers. For example:

```
>tracemsg
>  = [(datamessage (myid,0) ("HFT" ++ (show myid)
>                             ++ "i,"))], if time = 0
>  = [(datamessage (myid,0) ((show invent) ++ ","))], otherwise
```

For an HFT with id 4 this would output the column header “HFT4i” on the very first timestep of a simulation and its inventory on every subsequent timestep. Because the simulator harness knows not to randomise the order of these types of messages they will always be printed in a fixed order and thus contain the correct column values.

4.3.5 Orders & Order Messages

Order messages have the sole purpose of carrying orders from sender to recipient, an order is a type containing a price, size, time, fractional time, trader id, order type and unique identifier with setters and getters defined on each value as well as a constructor. The fractional times in these objects are set by the exchange and are to specify in which order the exchange received the messages in any one time step. Order types can be of the following types:

- Order types can be one of many
  - Bid limitorder_t
  - Offer limitorder_t
  - Sell marketorder_t
  - Buy marketorder_t
  - None
  - Abort
- Where marketorder_t and limitorder_t can be the following.
  - marketorder_t ::= Orkill | Andkill
    - Orkill – Fulfill the entire market order or none at all.
    - Andkill – Fulfill as much of the order as possible and return a rejection for the rest.
  - limitorder_t ::= Goodtillcancelled | Goodtilldate num
    - Goodtillcancelled – Order remains on the book until explicitly cancelled
    - Goodtilldate num – Order remains on the book until the time specified by num or explicitly cancelled.

The unique identifier in an order is an id that coupled with a trader id forms a unique key for any single order; this can be used to cancel orders at the exchange. Unique identifiers alone are only unique to a single trader.

Order messages have the same getters and setters as others (set receiver, set destination, get destination, get sender) as well as a function that fetches the associated order.

4.3.6 Trade Messages

Trade messages are defined as (Trademessage (num,num) order order) and have the following constructor:

```
>trademessage :: (num,num) -> order -> order -> msg_t
```

These are messages containing two orders and are sent to each participant of a trade when two orders are matched at the exchange. The orders are modified from their original state if necessary to reflect the size and price at which the orders were executed (this is done by the exchange).

As usual there are getters and setters defined on these messages. The exchange agent implemented for my experiments ensures that the first order in the message is the recipient's order and the second one is the order the recipient traded with. However, the simulator does not enforce this – in general, the recipient should check the trader id of each order to determine which is the recipient's order.

4.3.7 Ack messages

Ackmessages are defined as (Ackmessage (num,num) num order [char]) and have the constructor:

```
>ackmessage :: (num, num) -> num -> order -> [char] -> msg_t
```

These are used to acknowledge the receipt of a message by the exchange.

The first tuple corresponds to the from and to ids, the following num is the ack code, the third argument is the order being acknowledged and finally there is a string describing the type of acknowledgment that has been made – this is printed out in the trace file alongside the message so that we know which type of ack it was without having to look up the code.

There are many codes corresponding to different acks, these are listed below.

- 0 accept,
- 1 order too large,
- 2 no more liquidity,
- 3 outside sliding window of acceptable prices,
- 4 too many contracts on book,
- 5 order cancelled,
- 6 book is spiked,
- 7 minimum resting time not obeyed.

The usual getters and setters are defined on these messages as well as a specific one to retrieve the ack code.

4.3.8 Cancel messages

Cancel messages are defined as (Cancelmessage (num,num) (num,num) and have the constructor:

```
>cancelmessage :: (num,num) -> (num,num) -> msg_t
```

These are sent to the exchange to cancel an order. They consist of a (from, to) pair followed by a unique (Trader ID, UID) pair to identify an order and they have the usual getters and setters defined on them as well as one to fetch the unique pair.

4.4 Agents

Agents generally consist of two major components:

- The agent wrapper, which is recursive and both consumes and generates infinite lists. The wrapper also updates the local state in the recursive call at each timestep.
- The underlying logic, which is called once at each timestep, is not recursive and neither consumes nor generates infinite lists.

The concept behind this way of constructing agents is that one can focus purely on developing the logic for an agent that takes any argument, performs any number of operations and returns any results in any desired format.

The wrapper is then constructed to act as a medium for transferring and marshalling information between the harness and the underlying logic.

4.4.1 Logic wrappers

The wrapper recursively digests & filters information (messages, broadcasts) coming in from the harness as well as from the previous state of the agent. It then passes this information to the underlying logic. Finally, it converts whatever is returned from the underlying logic into formats compatible with the harness. E.g. Messages, broadcasts, agent states.

These wrappers must conform to the following type specification:

```
agent_t == agentstate_t -> [arg_t] -> [(num, [msg_t], [msg_t])]
           -> num -> [[msg_t]]
```

Below I describe the arguments an agent_t should receive and what it should return in more detail:

- Agents take an agentstate_t as their first argument, initially this will be an emptyagentstate from the harness and it is the wrapper's responsibility to detect the case where the agent state is empty and reset it to be an appropriate state for the agent, and from then on to use the correct wrapper. Agent states are discussed in more detail in Section 0.
- A list of [arg_t] from the simulator harness. These must be checked to see whether any of the arguments should affect the way that the agent behaves (this could potentially be done either in the wrapper or in the logic).

- An infinite list of redacted simulator states: [(num, [msg_t], [msg_t])]
    - This is a list of snapshots of the simulator state at each timestep tailored for the particular agent, all that is available in each state is what the agent is permitted to access.
    - The num is the current simulator time.
    - The first list of messages is the list of messages sent to this agent in the previous timestep (destined to be received this timestep).
    - The second list is a list of broadcast messages to the broadcast groups to which this agent is (statically) subscribed.
- A num corresponding to the agent's unique ID as assigned by the simulator. This is used to tag the agent's outgoing messages.
- Finally, the agent wrapper must return an infinite list of lists of messages that correspond to a list of outgoing messages for each time step of the simulation.

As mentioned the simulator passes in an infinite list of tuples (redacted simulator states) to an agent function and expects an infinite list of lists of messages in return. The wrapper typically manages these infinite lists, stripping the head element from the incoming list and passing it to the logic, and using the finite result returned by the logic to create the list of outgoing messages for this timestep which is consed onto the recursive call to the wrapper (thereby creating an infinite output list). In this way, the logic does not have to manage infinite lists.

The wrapper also typically deals with retrieving arguments relevant to the underlying logic and passing them to it, as well as sometimes retrieving arguments for itself. As a result of the foregoing, there are many behaviours common amongst most/all wrappers for all agents.

4.4.2 Agent states

Each agent holds local state (which is updated by passing an altered version as an argument to its recursive call). This local state is private to the agent and is never seen by the simulator harness and is not output to file unless an agent specifically outputs a data message containing its local state.

An agent state is defined by the following algebraic type, which contains a number of custom-defined agent states for different agents, each of which requires different local state:

```
>agentstate_t ::= Agentstate (num, num, num, lob)
>                 | Emptyagentstate
>                 | Exchstate lob
>                 | Nicemimestate nice_mime_lob
>                 | Traderstate (num, num, num, sentiment,
>                                 num, [order -> order])
>                 | Newagstate ([num], [order], sentiment,
>                               [order -> order], [num], [num])
```

As mentioned previously, each agent starts with an Emptyagentstate and it is the responsibility of the agent wrapper to detect that input case. The wrapper then calls itself again (immediately, without incurring a time step) with the local state reset to be an appropriate value (such as Newagstate, for example).

Precise details of agent states will be explained as each agent type is discussed later in this section.

4.4.3 Example trading agents

Two very common trading agents are a Fundamental Trader (which might be either a Fundametal Buyer or a Fundamental Seller) and a Noise Trader (which is a statistical proxy for a large number of smaller traders in the market). We have implemented both of these for our experiments. In each case we describe below the wrapper, the state and the logic for the trader type.

4.4.3.1 The Fundamental Trader

Wrapper: The Fundamental Trader's wrapper looks for an argument specifying whether any particular agent instance should behave as a buyer or seller - it passes this to the Logic as a Boolean value.

Local state: The Fundamental Trader, like many other trading agents), uses Newagstate to store its internal state. This holds the following data:

- ***[num]:*** a list of inventory counts (an inventory count is the number of contracts an agent holds or owes on any one timestep);
- ***[order]:*** the list of orders that have been sent out previously (traders use this if they later wish to cancel those orders);
- ***sentiment***: this is an assessment of the current market sentiment (e.g. taking values such as Calm, Toxic or Ramp)
- ***[order -> order]:*** a list of partial applications of the function order_setuid. Each time an order is going to be sent out the head of this list is applied to it to give it a unique ID before being removed from the list; this ensures all orders have unique IDs;
- ***[num]:*** an infinite list of random numbers (though it may be empty for other agents if they don't require random numbers);
- ***[num]:*** a list of numbers for general use by an agent (each agent uses this in a different way - for our Fundamental Trader it is empty)

Logic: The Fundamental Traders have a goal inventory they wish to accrue over a period of time; after this time period passes their inventory resets to zero. Their strategies become more and more aggressive as they approach their deadlines. They place their orders as follows:

1. A base price is calculated: this is simply the (target invent/period * (current time MODULO period)) + 1
2. We calculate a multiplier for this base size. This reflects our desire to order more if prices are favourable.
    a. This is simply 1 if they aren't.
    b. If prices are favourable, i.e. the underlying value of an item is higher than the best offer if we're buying and vice versa for selling then we multiply the difference between these two values by a predefined constant "booster".
3. The product of the multiplier and base size is the new size, we then ensure this new size is less than the max order size, if it isn't then it becomes the max order size, then we check that the remaining stock we wish to purchase/sell is not

smaller than this size, if it is we just place an order for the remaining stock we wish to buy/sell.

4. We now calculate the price:
    a. We simply use the underlying value if it is within the exchange enforced price bands and the current prices are favourable.
    b. Otherwise buyers place orders at underlying value – 2 and sellers at underlying value + 2. (Still respecting price banding of course)

4.4.3.2 The Noise Trader

Wrapper: The Noise Trader has a wrapper that is almost identical to the Fundamental Trader. The differences are in the treatment of random numbers (the Noise Trader wrapper will detect a "randseed" argument to modify its list of random numbers) and Gaussians (the wrapper passes a different list of Gaussian numbers each time it calls the logic), and the detection of Buyer/Seller status (which isn't required for a Noise Trader).

Local state: The Noise Trader uses Newagstate, just like the Fundamental Trader.

Logic: The underlying logic for the Noise Trader is the series of equations outlined previously in Section 3.3.3 and repeated below:

The noise agent decides its action for each timestep as follows:

1. Picks a random number from a uniform distribution between 0 and 1 and decides to submit a buy side order if this number is less than the probability of submitting a buy side order, sell side otherwise.
2. Picks another uniformly distributed random number, if it is below the probability of submitting a cancellation order then a cancellation is submitted, otherwise if it is below probability of submitting a cancellation order plus probability of submitting a limit order then a limit order is submitted, otherwise a market order is submitted.
3. Pick another uniformly distributed random number:
    a. If it is below the probability that a limit order is within the spread then pick a price from a uniform distribution between the best bid and best offer.
    b. Otherwise, take the best offer if the order is sell side and add a value drawn from a power law distribution. Vice versa for buy side orders.
4. Finally, if the order is a market order then the size is equivalent to the depth of the top of the book on the opposite side. Otherwise the size is simply a value drawn from a log normal distribution.

4.4.4 The HFT Market-Maker

We wanted to create a primitive inventory driven HFT that would just try to purchase or sell as much as possible without breaking any of it's internal rules – e.g. surpassing inventory limits while at the same time mimicking realistic HFT behaviours such as order splitting/foaming.

4.4.4.1 The HFT MM wrapper

The HFT wrapper is very similar to that of the Fundamental Trader, except that it handles more arguments. HFT MM agents are able to detect and act upon a number of arguments passed to the simulator, as follows:

- Enable order foaming.
    - If the argument `(Arg (String "UseGaussians", y))` is detected then order foaming is enabled as described in Section 3.3.1.4.
- Set soft limit (default 2700).
    - If the argument `(Arg (String "SoftLimit", y))` is present then the soft limit (see Section 3.3.1.2 for details) is set to y, otherwise it defaults to 2700.
- Set hard limit (default 20000).
    - If the argument `(Arg (String "SoftLimit", y))` is present then the soft limit (see Section 3.3.1.2 for details) is set to y, otherwise it defaults to 2700.
- Market sentiment: this is an assessment of the current market sentiment (e.g. taking values such as Calm, Toxic or Ramp)
    - This is simply another parameter used to assist in determining the foam offsets.

4.4.4.2 The HFT MM Local State

The HFT MM uses Newagstate, just like the Fundamental Trader.

4.4.4.3 HFT MM Logic

Determining initial order prices (before scattering)

First we calculate a base price for the buys and offers. As an HFT gets closer and closer to its panic zones it has to make its corrective orders more and more attractive. However, at the same time the HFT wants to maximise profits and mimimise losses. Thus we calculate an offset, which we add to the best bids and best offers respectively to get their base prices. The offset proposed is shown below:

(((bestoffer - bestbid) - 1)*(inventory / softlimit)) * -1

As an HFT's inventory approaches the panic zone on either side it initially slowly makes the opposite side's orders more attractive and the offending side's less attractive. As it progresses closer and closer it does this more violently until eventually using market orders instead.

Finally we make sure these prices are within the price bands enforced by the exchange.

Determining total order sizes

The HFTs quite simply put in the maximum offers and bids they can each turn to bring their inventory up to but not over their respective soft limits, as explained in Section 3.3.1.2.

Making the order tuples

Given the base prices and the total sizes we now generate a set of offers and bids that sum up to each total size. We do this by dividing each total size into the number of orders we wish to place each turn.

We then proceed with order foaming if it is enabled. To do this we generate as many random numbers as there are orders between +- the maximum desired spread (e.g. 3) and generate an order at baseprice + random number. This is so that we have many small orders within and around the spread rather than a few large orders by each HFT – designers of these algorithms often do this to obscure the underlying logic and prevent reverse engineering of their algorithm.

If any one of the prices is outside the price band we reflect it about the price band. E.g. price band 1.7 and order of 1.78 would then become an order of 1.62.[29]

Should it panic?

If the HFT has an inventory that exceeds the respective soft limits it places market orders for the full magnitude of the soft limit in an attempt to return to a net zero position.

Order Validation

The function (markettuples) ensures that no single order is larger than the maximum order size.

---

[29] In practice, foam mostly has an effect where spreads are large and may have no effect at all when spreads are small and prices are rounded to the nearest tick.

4.4.5 The Exchange agent

Our exchange agent accurately mimics the behaviour of the Chicago Mercantile Exchange's E-Mini market (often abbreviated to CME E-Mini). Our agent is called "nice_mime" (an anagram of CME EMini).

The exchange agent takes its old state, a list of orders, its own ID and a list of cancellations and returns a new state and a list of outgoing messages

It does this by first pruning the book for expired entries and cancelled ones, and then it places the new limit orders on the book, uncrosses the book and finally attempts to execute all the market orders it received as well.

4.4.5.1 The Exchange Wrapper

The exchange agent's wrapper initialises and maintains the exchange's local state as well as looking for an argument that specifies what the minimum resting time should be, otherwise it defaults to a minimum resting time of 0, which it then passes onto the exchange logic. It will detect the current market sentiment and use this to initialise the internal state. It also extracts the orders from order messages and the cancellation tuples from cancellation messages before passing them to the underlying exchange logic as two separate lists.

4.4.5.2 Local state

Our exchange agent has it's own local state called Nicemimestate, which holds additional data of type nice_mime_lob. The definition for nice_mime_lob is given below:

```
>nice_mime_lob ::= Nice_mime_lob sentiment listoftotals
>                  nubids nuoffers num ticksize
>                  lasttradedprice stats
>listoftotals == [num]
>stats == [num]
>nubids == [tick]
>nuoffers == [tick]
>ticksize == num
>tick == (num, [order])
>lasttradedprice == num
```

As before, sentiment corresponds to the market sentiment for the current simulation.

Listoftotals is a list of each trader's total number of contracts on the book indexed by trader id.

Stats is a list of numbers corresponding to statistics generated and updated by the exchange on each iteration. The list itself is some 30+ elements in size and can be found in the code.

Nubids is a list of ticks containing the bids on the book and Nuoffers is a list of ticks containing the offers on the book.[30]

A tick is simply a tuple of a price and a list of messages.

Ticksize corresponds to the constant ticksize assigned to the simulation. This is hard coded (`nice_mime_emptylob`).[31]

Lasttradedprice is the last price at which a trade was executed.

4.4.5.3 Exchange Logic

The following steps describe how the input is processed in order to generate the exchange output:

1. First and foremost, the exchange agent checks to see if the old Exchange state is still spiked[32]. (nice_mime_lob_isspiked)
    a. If it is spiked then it decrements the remaining spike time and sets a Boolean flag (isspiked) to indicate so, it then skips out steps 4 & 5 below, so that no trades are done.
    b. Otherwise nothing happens.
2. The state being worked on is now pruned for expired entries and entries that are to be cancelled. (nice_mime_lob_prune)
    a. It does this by going through both the bid and offer list of ticks holding the orders on the book entry by entry checking that they both haven't expired and are not present in the list of cancellations otherwise it removes them from the book.
    b. While doing this any time an order is removed from the data structure a cancellation acknowledgement message is created. The set of these is named (cmsgs)
3. The newly modified (see below) incoming orders are now appended to the data structure. (nice_mime_lob_appendorders is a function which takes an instance of the order book and a list of orders to append)
    a. The incoming orders receive a "fractional timestamp" before being appended to the book. This fractional timestamp is in addition to its original timestamp

---

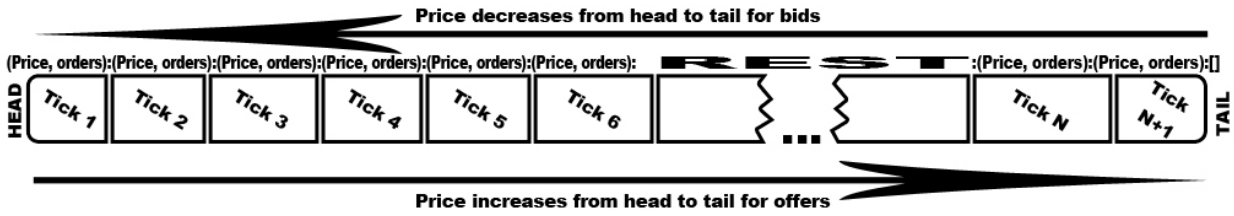
[30] See bullet 3.b in Section 4.5.3 Implementing the exchange for more information on ticks.

[31] N.B. In real life, the tick size can vary for different regions of the order book.

[32] See 3.2.3.1.1.4 Stop spike logic for more information.

and serves to fix problems that arise from large numbers of messages seemingly arriving at the "exact same time" as a result of our discrete simulation. More on that later.

b. Before I get into the details of what happens at this stage I'd like to describe the data structure used to hold the orders in memory.



There are two structurally identical sorted lists (bids offers), one for bids and one for orders. Their only difference is that the bid list is in descending order and the offer list is in ascending order. Each list contains "ticks"; a tick is a price at which orders can be placed, these are discrete prices placed at specified intervals, as well as a list of orders at that tick. The tick exists in the structure only if there is at least one order present in that tick.

i. Take the order at the front of the list of orders.

ii. Recursively call (nice_mime_lob_appendorders) with the rest of the orders.

1. The base case is when the order list is [] and it just returns the limit order book it received with no additional messages.

iii. Round the order to the nearest tick. For bids we round down and for offers we round up.

iv. Append order to the book returned from the recursive call in the respective list (buyside or sellside) and tick then generate a successful ack message unless one of the following criteria are met, in which case do not append and create unsuccessful ack message with the corresponding ack code.

1. The order is set to expire before (time + minimum resting time[33]). Ackcode: 7

2. The order exceeds the maximum size for a single order. Ackcode: 1

[33] See 3.2.3.1.1.3 Minimum resting times (proposed) for more information.

3. The order causes the party who made it to exceed their total number of contracts allowed on the book. Ackcode: 4
4. The order was not within LTP +/- priceband. Ackcode: 3

4. The book is now uncrossed. (nice_mime_lob_uncross_book)
    a. If there are no bids, no offers or the book is not crossed then just return the state received.
    b. Otherwise, take the best bid and execute it (nice_mime_lob_execute_order) against the offers.
        i. This takes an order and executes trades between it and the leading orders on the opposite side.
        ii. It updates/removes orders as it executes trades and creates the relevant trade messages (trademessage).
        iii. The price traded at is the price specified by the older of the two messages. (Hence the fractional times mentioned earlier)
        iv. Many specific cases... (Presented in order of precedence)
            1. If the order is a buy or sell and the order has been exhausted then just return the lob & return emptyorder.
            2. If you've run out of liquidity on a side and you're dealing with a limit order just return.
                a. There is nothing left to uncross.
            3. If you've run out of liquidity on a side and you're dealing with an X and kill market order (see 3.4.3 for description of order types) then return reject for the remaining quantity.
            4. If the order is a bid and no liquidity remains on the first sell side level then return the lob without the exhausted level & remaining order – the caller function then checks if the book is still crossed and calls nice_mime_lob_uncross_book if it is.
                a. This differs from the behaviour in 5 & 7 because when matching limit orders against limit orders we care that the orders being executed have overlapping prices.
            5. If the order is a (Buy Andkill) and no liquidity remains on the first sell side level then continue iterating without the exhausted level.
                a. Here we continue iterating because we are not interested in the prices of the orders we are considering matching against.
            6. If the order is an offer and no liquidity remains on the first buy side level then return the lob without the exhausted level & remaining order.

        a. Same as 4a.
    7. If the order is a (Sell Andkill) and no liquidity remains on the first sell buy level then continue iterating without the exhausted level.
        a. Same as 5a.
    8. If the order has been exhausted and it's not a (Buy Andkill) or a (Sell Andkill) then return the lob with it removed.
    9. If the old order is non-zero then match! (Recall price banding from nice_mime_lob_checkorderallowed ensures that SSL will never be triggered)

v. Many small details (helper functions for dealing with the data structure, keeping tabs on the number of contracts on the book etc.) that can be seen in the code.

5. Execute the market orders against the book.
    a. Before executing a market order we check whether or not it would cause the new last traded price to differ from the old by more than the spike trigger distance, we do this by pseudo matching the order against the book.
        i. If it doesn't trigger the spike protection logic, continue.
        ii. If it does, do not execute the market order, cancel the rest of the market orders queued for execution (& generate their acks) and enable SSL protection.
    b. Execute the market order using nice_mime_lob_execute_order as described earlier.
6. Update the exchange statistics held internally to reflect the new changes using (nice_mime_lob_updatestats), these statistics will be later sent out as a broadcast to all agents.
    a. Please refer to code for details, some 33 values are maintained.
7. Finally, generate a trace message containing these statistics, append it to all other messages generated throughout this iteration
8. Messages & broadcast go to the simulator, new lob state goes to the next iteration of nice_mime.

4.5 Testing

4.5.1 Test plan

Testing was simply done by running agents alone (or with a simple probe agent) and analysing the message report and data output ensuring that it is as expected. Once each agent was validated on its own we would check several agents together in stable conditions and make sure the data being output reflected that. For the exchange, which was by far the most complicated agent in this project, I engineered a series of tests for the underlying logic alone.

4.5.2 Test results

4.5.2.1 Compile-time testing

Miranda is strongly statically typed and thus the majority of errors are type errors that appear at compile time or are otherwise syntactic errors. Miranda checks all uses of functions for type correctness (it also checks their function bodies) – the type it checks for can either be inferred or explicitly declared by the programmer.

Runtime errors are very rare but can appear on occasion – the most common being taking the head of an empty list. Debugging at this point can be very difficult as the error message returned tells you nothing about where the error took place, it simply states that you tried to take the head of an empty list.

To combat this I wrote my own wrapper for the head function that takes an additional string argument – this should be a string denoting where in the code the call is taking place. This wrapper function then operates as head usually does if the list is non-empty. If it isn't then it throws an error displaying the string it took as input.

4.5.2.2 Unit testing

While developing new functions I would test them by executing them on their own with predetermined input arguments. Any functions called by the function being tested would be set to return a default value for the duration of the test. I would then check that the various return values of the function correspond to what I expected from the function. I would also test boundary cases to check for any runtime errors.

4.5.2.3 Link testing

For the exchange agent (the most complex agent in this simulation) I developed an automated test suite to check all possible execution paths for correctness. The results of which are displayed in Appendix 2.

4.5.2.4 Integration testing

I ran the system with a small number of each agent type in Calm market conditions[34] and checked that the data coming out from the simulator reflected this.

E.g. stable price, inventories staying within limits, agents trading with each other.

Details are provided in Appendix 2.

[34] "Calm" is one value that the sentiment can take.
Sentiment: this is an assessment of the current market sentiment (e.g. taking values such as Calm, Toxic or Ramp)

5 Experiments

In this chapter we demonstrate that the hot potato effect is indeed replicable and determine some of the contributory factors to a hot potato event. We then evaluate the efficiency of existing and proposed market measures as well as look for statistical indications that their preventive and remedial effects on hot potato events are significant.

5.1 Recreating the hot potato event

Once we had a reliable simulator in place we set ourselves the task of recreating the hot potato effect so that we could analyse it.

In order to do this we would need to get the HFT MMs to panic in a coupled manner so that non-panicking HFT MMs would place limit orders that the panicking HFT's market orders would hit, sending the other HFT MMs into panic and bringing the panicking HFT MMs out of panic. The phenomenon of this happening cyclically amongst a number of HFT MMs is the hot potato effect.

In addition, one panicking HFT MM may not be enough to send others into panic; it could quite easily offload its excess inventory over multiple HFT MMs safely (i.e. without sending any of the other HFT MMs into panic) and so we may need to get multiple HFT MMs to panic.

This proved to be difficult; in normal conditions HFT MMs use limit orders to manage their inventory giving them complete control over it thus never intentionally surpassing their limits, which they must do to panic offload inventory.

When running simulations in standard conditions the HFTs would oscillate well within their limits and none would panic as can be seen in Figure 5-1.

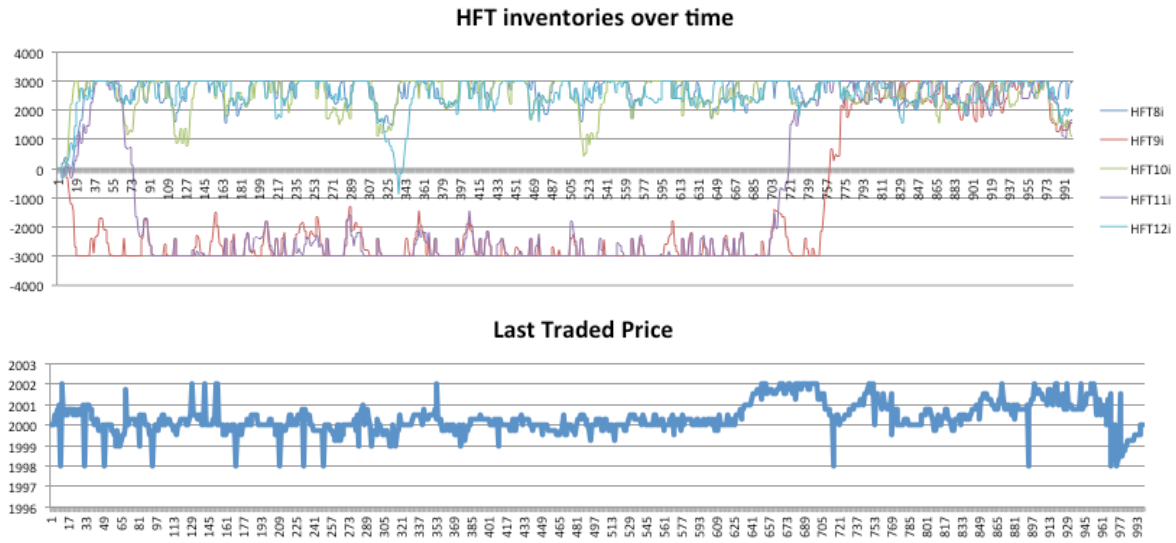


Figure 5-1 – Results of a simulation run for 1000 timesteps with 5 HFTs with soft limits of ± 3000, 1 noise trader, 3 fundamental sellers and 3 fundamental buyers. The upper chart is of the HFT inventories over time and the lower chart is of the last traded price over time.

Thus we faced two problems:

1. Getting an HFT MM's inventory close to its inventory limit.
2. Getting it to surpass this limit.

We could easily address the first problem by having some sort of anti-HFT probe agent that always met the HFT MMs orders on one side of the book only.

However, we discovered that even if we were to remove the fundamental sellers, buyers and noise traders and replace them with a such a probe, designed to place a lot of pressure on one side of the market forcing the HFTs to their limits, they would still put themselves into an equilibrium where none of them were in panic – as seen in Figure 5-2.

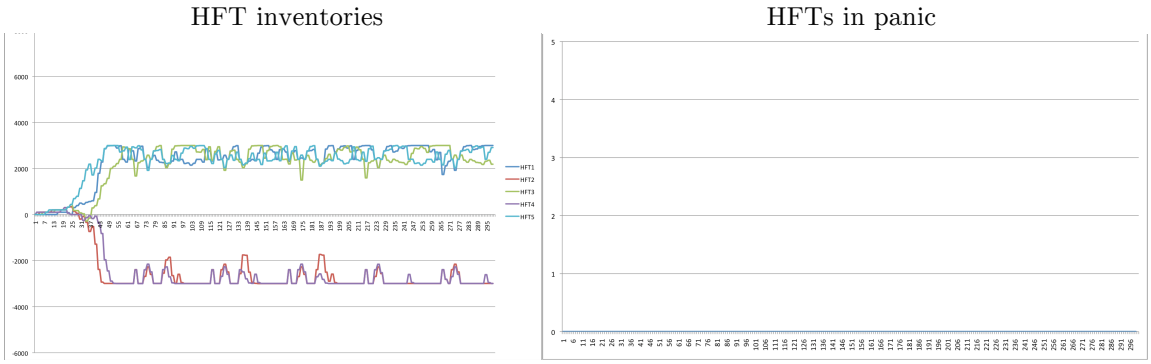


Figure 5-2 – Results of a simulation run for 300 timesteps with 5 HFTs with soft limits of ± 3000 and a probe trader that issues sell orders of size 100 for each of the first 40 timesteps. The graph on the left is of each HFT's inventory (y axis) over time (x axis). The graph on the right is the total number of HFTs in panic (y axis) on any one timestep (x axis).

We found ourselves at a standstill – we knew the HPE was possible (because it had been observed during the May 6th 2010 Flash Crash) but had yet been able to recreate it in simulation.

We needed to address the second problem of getting them to surpass their limits however this was a lot harder; we couldn't do this while HFT MMs were operating in standard conditions.

We needed to either force the HFT MMs to behave in a certain manner or lull the HFT MMs into believing they had complete control over their inventories and were operating normally when they actually weren't.

5.2 Compliance orders

We observed that HFT MM inventories would reach their soft limits where they would remain for a short period of time before moving away again; it was at this point we considered implementing compliance quotes.

Compliance quotes are orders placed well off a stock's market price when a trader does not want to trade but has to conform to its requirements as a market-maker to maintain a two-sided quotation for certain money saving concessions or otherwise.

Compliance orders are widely used in practice for this purpose and thus their addition makes our HFT MM algorithm more realistic.

In addition, compliance quotes represented a significant amount of the trades that were executed against on the flash crash of May 6th.[35]

5.2.1 Extending the implementation

If we are yet to panic but close enough to the soft limit for our order sizes to round down to 0 we normally do not place an order.

However, if the exchange enforces compliance orders of a minimum size (or the HFT decides to place compliance orders and they are permitted by the exchange) then we place a compliance order with a predefined compliance quote size which is passed to the simulator as one of its arguments.

We achieve this in the simulator simply through the HFT MM's detection of an additional two arguments to the simulator:

```
(Arg (String "StubsEnabled", x1))
(Arg (String "Stubsize", x2))
```

where x1 can be any numerical value with no change to the argument's effects (x1's value is ignored) and x2 is the size of these compliance quotes.

In the absence of the Stubsize argument there is a default compliance size of 1.

---

[35] (SECURITIES AND EXCHANGE COMMISSION 2010)

5.2.2 Results

After this functionality was added we ran an experiment similar to that of Figure 5-2 - the results of which are visible in Figure 5-3.

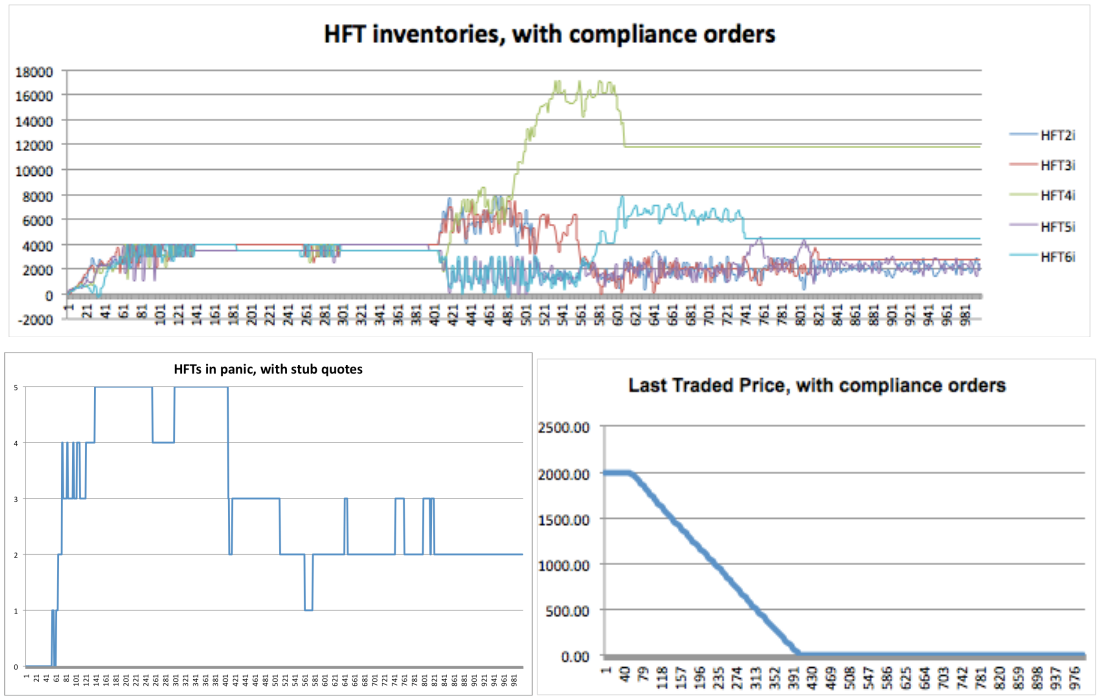


Figure 5-3 - Results of a simulation run for 1000 timesteps with 5 HFTs with soft limits of ± 3000 and compliance order sizes of 1000 as well as a probe trader that issues sell orders of size 500 for each of the first 500 timesteps. The chart on the top shows HFT inventories with compliance quotes enabled over time, the chart on the lower left shows the total number of HFTs in panic over time and the chart on the lower right shows the last traded price over time.

As demonstrated in Figure 5-3 we were able to recreate the hot potato effect using compliance quotes; however, these compliance sizes were unrealistically large and we had to provide even more sell-side pressure than before from the probe trader.

5.3 Delays

We then considered what would happen if we were to introduce a small, entirely realistic information delay to the system so that the HFT MMs were acting on out-dated information.

Such delays are inherent to trading in the real markets because of technological and financial limitations and are most prevalent in times of high activity.

In addition, if an HFT issues a single order every 100µs but confirmations from the exchange take approximately 6ms to return after an order is sent then information is always delayed in practice. However, even this is a conservative estimate to how long it

takes an HFT to send out a single order – with modern bespoke hardware such as the Raptor Raw[36] orders can be sent out as quickly as every 1.5µs.

The Nanex report on the flash crash mentions delays of up to 24 seconds during the crash.[37]

In such a case the HFT MMs would be forced to act without having up to date information on the orders they've placed on the book or the trades they may have participated in.

5.3.1 Extending the implementation

All of the results presented prior to this subsection were with zero delay - achieved by placing an additional constraint on the system that agents only be allowed to trade on even timesteps thus allowing them to receive up to date information on prior orders before acting.

Therefore implementing a delay of 1 was a relatively simple task; we just allowed agents to send orders on all timesteps. To explain the built-in delay consider the following steps:

1. T=0 trader sends order to exchange.
2. T=1 exchange receives order, matches it and sends trade notification to trader.
3. T=2 trader receives notification and acts on it.

If the agent were to send an order at T=1, it would have to make an assumption regarding whether the order sent at T=0 resulted in a trade, or not. Therefore, the agent would not have full information about its inventory.

Increasing delays even further was rather simple; we just extended the affected agents by modifying the agent wrapper to maintain a queue of lists of messages they received with an initial (n-1) empty entries at the head of the queue, n corresponding to the desired delay.

We set this delay simply by passing the following argument to the simulator at runtime:

```
(Arg (String "Delay", y))
```

where y corresponds to the desired delay.

[36] (FUSION SYSTEMS 2012)

[37] (NANEX 2010)

5.3.2 Results

We implemented this functionality and tested again with the same parameters to the experiment outlined in Figure 5-2 with the addition of a 1-timestep delay.

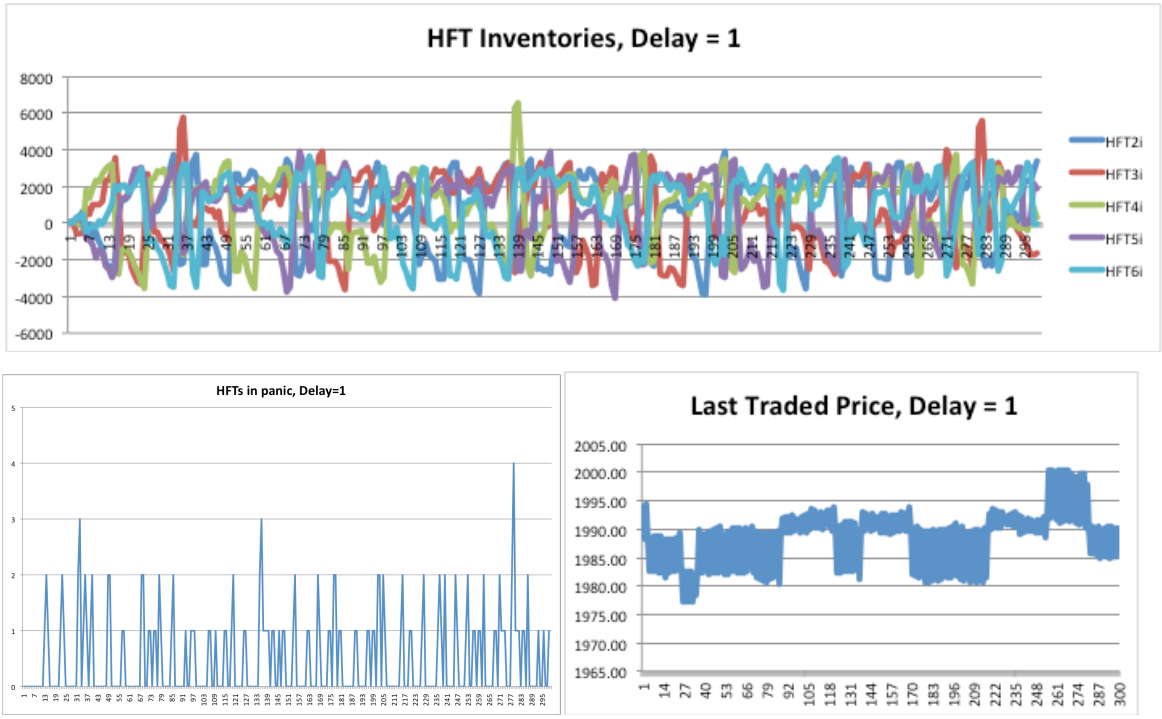


Figure 5-4 - Results of a simulation run for 300 timesteps with 5 HFTs with soft limits of ± 3000 and a probe trader that issues sell orders of size 100 for each of the first 40 timesteps. The chart on the top shows HFT inventories over time with a delay of 1, the chart on the lower left shows total HFTs in panic over time and the chart on the lower right shows the last traded price over time.

As can be seen in Figure 5-4 we were able to recreate the hot potato effect with dramatically less sell-side pressure from the probe trader and in absence of compliance quotes.

In addition, the HFT inventories were far more volatile than that of the compliance-quote-induced HPE, and panic inventories were both positive and negative. It can also be seen that although prices are not crashing, they are highly volatile and unstable.

This price instability and deceptive volume of trading would create problems for other trading algorithms, as observed in the flash crash.

6 Analysis of Market Protection Measures

Once we recreated the hot potato effect the next step was to evaluate the efficacy of both the market protection measures in place and certain proposed market protection measures in either preventing or stopping the hot potato effect.

There are a number of protection measures we had implemented and enabled throughout the experiments run in the previous subchapters and these are: Price banding, order limits & position limits (See Section 3.3.2.4 for details).

These however, unlike the measures in the following subsections, are voluntarily adhered to by the HFT MMs and as shown in Sections 5.2 and 5.3 they were still participants in a hot potato event therefore showing that the protection measures have no preventive effects on the HPE.

6.1 Circuit breakers (stop spike logic)

As detailed in Section 3.3.2.4 exchanges have circuit breakers, such as the stop spike logic, in place that are designed to cease trading when the markets become unstable.

During the flash crash of 2010 it is thought that this circuit breaker was what stopped the hot potato effect when the circuit breaker was triggered and halted all trading.

The previous experiments in which the hot potato effect was apparent were all run with this protection measure in place, but the circuit breaker was never triggered.

In order for the circuit breaker to be triggered the price needs to jump a total of 600 ticks as a result of a single market order – limit orders cannot trigger this logic because their trading prices are already limited by the 48 tick price banding (See Section 3.3.2.4 for details) enforced by the exchange.

This means that an order would have to be on the book lingering at a price 600 ticks away before being hit by a market order that is placed during a period of little/no liquidity (other than the lingering quote itself).

However, this rarely happened in these experiments because (i) the HFT MMs only placed transient limit orders that were quickly cancelled, and (ii) the limit orders placed by fundamental and noise traders were consumed by the HFT MMs before the price could move significantly.

6.2 Minimum resting times (proposed)

On the 8th of December 2010 the European Commission of Financial Services Policy and Financial Markets published a review of the Markets in Financial Instruments Directive (MiFID) in which they made the following proposal:

"Amendment 2.3. f: Automated trading and related issues:

Market operators would be required to ensure that orders would rest on an order book for a minimum period before being cancelled. Alternatively they would be required to ensure that the ratio of orders to transactions executed by any given participant would not exceed a specified level. In either case, further specification would be needed on the specific period or level." [38]

Furthermore, in September 2012 members of the European Parliament voted that all orders should be valid for a minimum resting time (MRT) of at least 500 milliseconds.[39] This effectively limits the ability for an HFT MM to cancel a limit order if the market moves suddenly, regardless of how low the latency of their market access is. With such a limit in place, HFT MMs might respond by placing limit orders at more conservative prices (thereby widening the spread on the order book), and/or might place several smaller limit orders spread across time (thereby reducing the risk of being unable to cancel any one order).

In order to test the efficacy of MRTs we needed some indication of how many timesteps there were in a millisecond. As of late 2012 one of the best available data networks boasts a round trip time of less than 8.5 milliseconds[40] between CME Group, Nasdaq OMX, BGC and ELX Futures.

If we compare this with our 2-timestep minimum round trip time then a single timestep would correspond to 4.25 milliseconds – meaning our HFTs would not be allowed to cancel their orders for an MRT of ≈118 timesteps.

We had to make some modifications to our implementation in order to enforce minimum resting times.

---

[38] (EUROPEAN COMMISSION 2010)

[39] (EUROPEAN PARLIAMENT 2012)

[40] (Stafford 2012)

To enforce them at the exchange we made two modifications:

1. nice_mime_lob_appendorders

We added the constraint that any incoming gooduntildate orders have a date specified that is later than the currentTime + MRT, else we reject.

2. nice_mime_lob_prune

We added the constraint that any cancellations attempted on an order earlier than orderArrivalTime + MRT are rejected.

For the purposes of these experiments we also had to modify the existing HFT agents; they would not have coped well with the MRT enforcement because they are acting under the assumption their orders expire in the requested time, one timestep, which of course they would not if an MRT larger than 1 was enforced.

The new HFT MM agents instead send in orders that are due to expire as soon as the MRT is met. They also keep track of the trade confirmations and cancellations they receive as well as the valid contracts they still have on the book to determine whether or not they are in a position to issue a new order.

Furthermore, we decided against running experiments with the full 118 timestep MRT because this would severely restrict the amount of activity our HFT MM agents could engage in within the constraints of a 1000 timestep simulation and extending the length of these simulations would make it infeasible to run the number of simulations necessary for statistical analysis.

Instead we ran an exploratory series of 30 independent simulations for MRTs of 0, 5, 10, 15, 20 and 25 respectively. We needed a way of quantifying how prevalent the hot potato was in a simulation and how unstable the HFTs in the simulation were in order to compare the stabilising effects of MRTs. We decided to calculate a "panic integral" for each independent simulation; this is simply the sum of the total number of HFT MMs in panic on a timestep over the course of a simulation.

Here are the median panic integrals for each series of tests:

Series	MRT = 0	MRT = 5	MRT = 10	MRT = 15	MRT = 20	MRT = 25
Median	770	332	243	264	170	142

We then analysed the medians of each distribution using the Mann-Whitney U test to see whether one distribution tends to have panic integrals larger than the other, this is the alternate hypothesis (h1); the null hypothesis (h0) is that the distributions are identical. We chose this test because the distributions were not necessarily normal in nature.

Series 1	MRT = 5	MRT = 10	MRT = 15	MRT = 20
Series 2	MRT = 10	MRT = 15	MRT = 20	MRT = 25
Favoured hypothesis	h1	h0	h1	h0
p-value[41]	3.9283e-04	0.9630	5.2261e-06	0.1234

These findings led us to think that implementing a minimum resting time may reduce the instability of a system that occurs as a result of the HPE – this can be inferred both from the lower medians for MRTs 5 and 10 when compared to 0 and the very low p values they have.

Furthermore, the amount of trading done in panic between HFTs (defined as the total number of successful trades between any two HFTs where one order was a Buy or Sell) seems to decrease as the MRT increases, which we would anticipate, however there's a conspicuous spike both when initially introducing MRTs at all and at the introduction of MRT 25 (Figure 6-1).

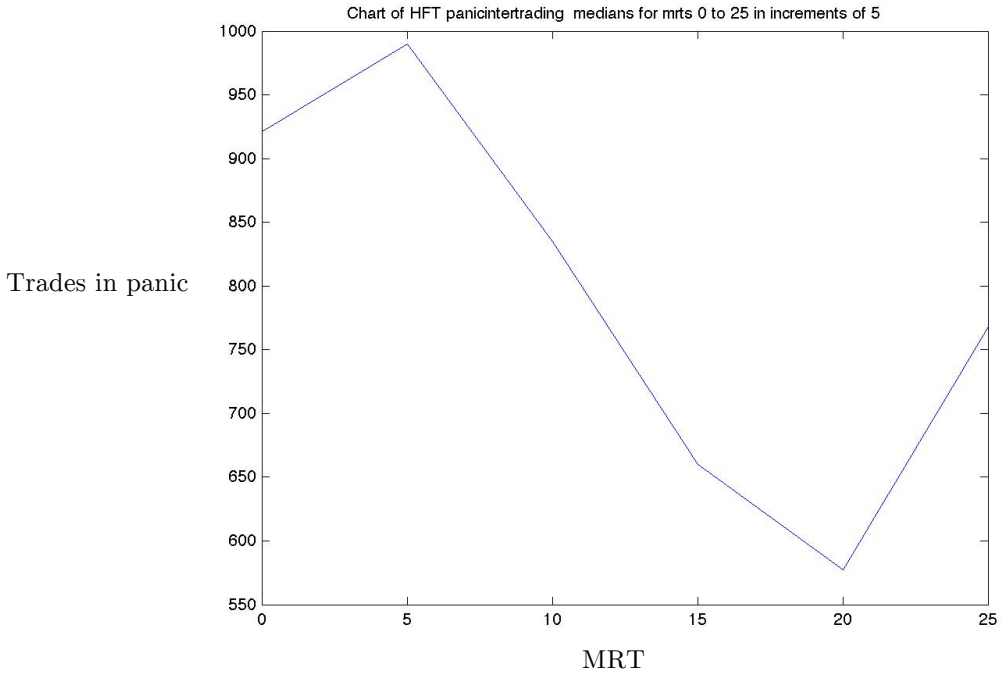


Figure 6-1 Diagram showing median values for the distributions of the total trading done in panic between HFTs.

[41] The p-value is the probability of observing a test statistic as or more extreme than the observed value under the null hypothesis.

The results of these simulations gave us some indication of what we should be looking but were not conclusive or consistent enough to allow us to draw any real, concrete conclusions.

Instead, we chose to run 40 independent simulations for the MRTs 0, 5, 10, 15, 20, 25 and 50 to assess the effects of increasing this MRT time on the time taken to the onset of the first HPE, which we could hopefully extrapolate up to the proposed MRT time.

We formally defined a HPE in these experiments as being a time period of which the start is identified by 3 timesteps in a row with at least one HFT in panic and the end of which is identified by a period of at least 2 timesteps in a row in which no single HFT is panicking.

To be a HPE they must also meet the constraint that there is a trading loop between a subset of the HFTs within the defined period.

On preliminary analysis of the data we realised that the outliers in each data set were extreme enough to severely skew the mean and thus instead decided to analyse the sample medians and modes. (The outliers can be seen in Figure 6-2)

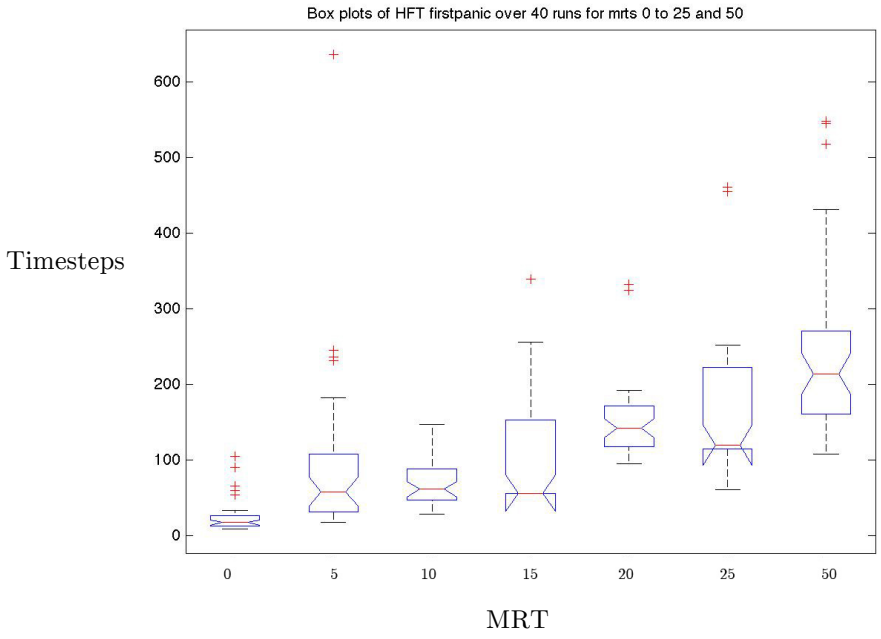


Figure 6-2 – Diagram showing box plots for the distributions of the time taken to the onset of the first HPE over 40 simulations for increasing MRTs. Outliers are indicated by red plus symbols. The central mark of each box is the median. The edges of the box are the first and third quartiles respectively. The whiskers extend to the most extreme data points that are not considered outliers.

In Figure 6-3 we can see the effects the outliers in each series have on their means.

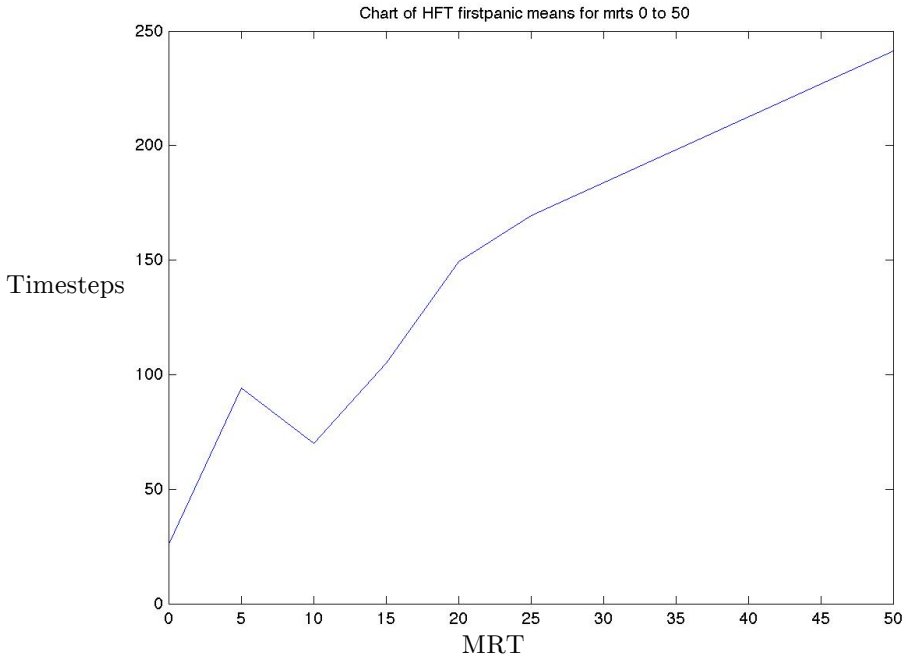


Figure 6-3 - Diagram showing mean values for the distributions of the time taken to the onset of the first HPE over 40 simulations for increasing MRTs. As you can see the curve is non-monotonically increasing.

Figure 6-4 shows the sample modes for each of the distributions. We see that the sample modes increase with the MRTs to provide an overall monotonic sigmoid curve with a steep gradient between MRTs 0 and 20 that begins to decrease after that.

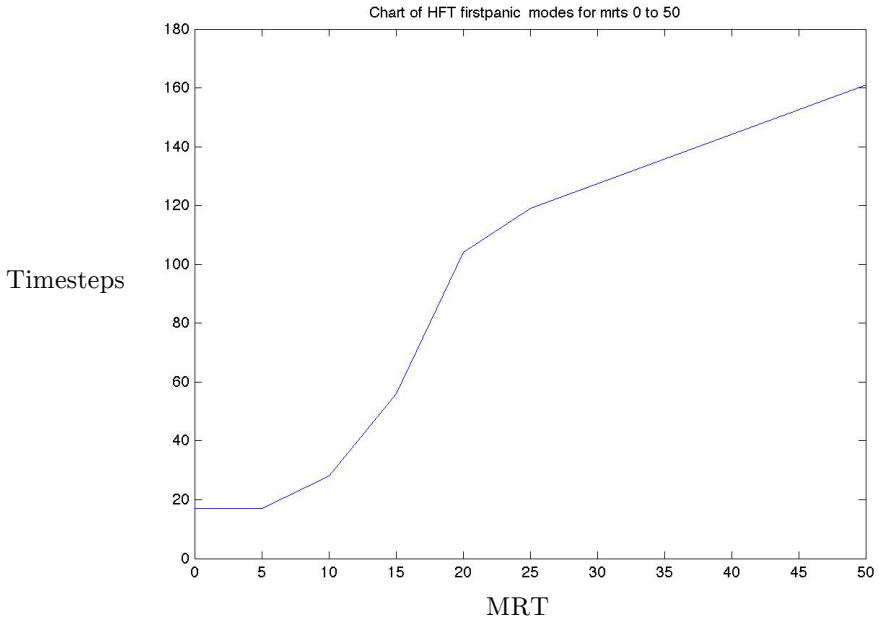


Figure 6-4 - Diagram showing mode values for the distributions of the time taken to the onset of the first HPE over 40 simulations for increasing MRTs.

We analysed the change of the medians of the distributions using the Mann-Whitney U test[42] on pairs of contiguous data sets – these were our findings:

Series 1	MRT = 0	MRT = 5	MRT = 10	MRT = 15	MRT = 20	MRT = 25
Series 2	MRT = 5	MRT = 10	MRT = 15	MRT = 20	MRT = 25	MRT = 50
Favoured hypothesis	h1	h0	h0	h1	h0	h1
p-value[43]	1.4970e-07	0.6129	0.0594	6.2086e-05	0.4864	6.8844e-04

Given that certain comparisons did not suggest that the distribution values tend to increase with MRT increases we decided to start comparing non-contiguous pairs with gaps:

Series 1	MRT = 0	MRT = 5	MRT = 5	MRT = 15
Series 2	MRT = 10	MRT = 15	MRT = 20	MRT = 25
Favoured hypothesis	h1	h0	h1	h1
p-value[43]	3.9206e-10	0.1747	1.8725e-06	1.3454e-04

The large increase of the mode between MRT 15 and 20 as well as the results of its U test lead us to believe that the probability of the onset of a HPE is highly sensitive to changes in the region between MRT 15 and MRT 20 and this is supported by the statistical analysis between MRT 15 and MRT 20.

The change between MRT 0 and MRT 5 also seems to have great effect.

However, the changes in mode between MRT 5 and MRT 15 as well as past MRT 20 are far more gradual; therefore, if large MRTs are considered undesirable for other reasons (e.g. because they reduce a trader's ability to avoid exposure to sudden changes in price), then this would indicate that it might not be of great worth to have MRTs larger than 20.

---

[42] Where the null hypothesis (h0) is that the two distributions are identical and the alternate hypothesis (h1) is that one tends to have larger values than the other.

[43] The p-value is the probability of observing a test statistic as or more extreme than the observed value under the null hypothesis.

7 Summary and Conclusion

In the course of this project I designed and implemented an agent-based simulator in Miranda. I also researched for, designed and implemented a number of agents specific to my investigation into the hot potato instability in the financial markets.

One of these agents was the "niCe_miME" agent to mimic the CME E-mini futures exchange, which is to my knowledge the first time an accurate copy of the exchange was used in experimentation.

I then extensively tested all of the aforementioned agents using a number of different testing paradigms.

After running a series of control simulations that exhibited stability, I extended the implementation to support both information delays and compliance orders – both believed to be important factors in instability.

I then ran a number of suites of experiments to determine whether hot-potato instability can be induced in a system that was previously stable. This included both some initial experimentation exploring the effects of varying parameters and rigorously designed experiments for which tests for statistical significance could be applied.

7.1 All objectives met successfully

At the outset of this project I outlined a number of objectives, all of which have been met.

- (O1) I have built a generic agent based simulator as well as the agents I needed for my particular experiments.
- (O2) I thoroughly tested and validated their implementations.
- (O3) I successfully recreated the Hot Potato Effect in simulation.
- (O4) I investigated the effects of compliance quotes and delays and determined that the bigger the compliance quote or delay the more violent the instability.
- (O5) I also determined that none of the proposed or pre-existing protection measures prevent hot potato instability and that minimum resting times are the only measure that dampen it.

In meeting these objectives I have achieved my aim by discovering two factors that seem to be major determinants in whether or not a system of market-making agents display instability – compliance quotes and delays.

7.2 Recap of contributions

- I have developed a generic agent based simulator that can be reused.
- I have developed an accurate copy of the CME E-mini exchange, which is where the 2010 flash crash is believed to have began, this can also be reused with the agent-based simulator.
- I have packaged these with a number of other agents which may be of wider use and have been constantly maintaining and extending them for use by other students in their experiments.
- I have recreated the hot potato effect in simulation.
- I have determined that the existing market protection measures do nothing to dampen the hot potato effect.
- I have determined that the existing and proposed protection measures do nothing to prevent the hot potato effect.

7.3 Discussion

Price banding doesn't prevent hot potato because price changes are gradual.

Price banding prevents orders being placed on the book at more than ± 48 ticks of the last traded price. We've observed that during hot potato effects the last traded price changes in a gradual fashion of less than $\Delta 48$ ticks per timestep.

Therefore we are led to believe that it does not prevent or dampen the hot potato effect.

Stop spike logic doesn't prevent hot potato because price changes are not dramatic.

Stop spike orders also so not prevent or dampen the hot potato effect – this is because it is never triggered. In order for stop spike logic to be triggered there must be a massive instantaneous jump in price that can only occur as a result of a single market order being placed at a moment of low liquidity and thus being traded against a cascade of limit orders. This doesn't happen during a hot potato effect because other HFT MMs are placing liquidity on the book for the panicking ones to offload to.

Max order sizes don't prevent hot potato because orders can be split.

A maximum order size does not prevent an HFT MM from offloading whatever quantities it wishes during moments of panic – it simply partitions its desired order size into smaller market orders of the maximum order size.

E.g. if it desires to sell 6000 futures and the maximum order size is 2000 then it can just issue 3 sells of 2000.

Maximum allowed stock on book doesn't prevent hot potato because stock is constantly changing hands between HFTs.

The maximum allowed stock has no effect because the HFTs are persistently trading with one another leaving little or no orders on the book. In addition, these maximums are very high making it quite difficult to reach the maximum.

Minimum resting times

We've observed that minimum resting times have a damping effect on the HPE. The graph of the minimum resting time vs. damping effect has a sigmoid curve indicating that there is a central optimal region with the largest effect for the smallest constraint.

This implies that an MRT can be chosen in this optimal region minimising the restrictions placed on an HFT MM to react to rapid price changes.

Although there is no absolute link between a simulated timestep and real time, I have previously argued (in Section 6.2) that one timestep might correspond to roughly 4.25 milliseconds for very fast HFT MMs, in which case an "optimal" value for MRT would be roughly 85 milliseconds (and the 500 milliseconds recommended by the European Parliament[44] is overkill).

7.4 Conclusion

We were able to create a simple HFT MM algorithm that behaves properly in "normal" conditions and observe that once we placed it in a realistic testing setting with other HFT MMs and market participants and introduced an information delay or small constraints like compliance orders it became unstable and a hot potato interaction appeared between these agents. We have also determined that all but one of the current and proposed protection measures do nothing to prevent or end the hot potato effect.

We also succeeded in creating an agent-based simulator that can be used in future research, and has already been used by other researchers.

[44] (EUROPEAN PARLIAMENT, 2012)

References

Airimitoaie, A. (2012). *Interference and Interaction between Trading Algorithms.* University College London, Computer Science, London.

Aldridge, I. (2010). *What is High Frequency Trading, After All?* Retrieved 2013 from Huffington Post: http://www.huffingtonpost.com/irene-aldridge/what-is-high-frequency-tr_b_639203.html

CME Group Inc. (2013). *CME Rulebook*. Retrieved 2013 from CME Group: http://www.cmegroup.com/rulebook/CME/

CME Group Inc. (2013). *GCC Product Reference Sheet*. Retrieved 2013 from CME group: http://www.cmegroup.com/confluence/display/EPICSANDBOX/GCC+Product+Reference+Sheet

EUROPEAN COMMISSION. (2010). *REVIEW OF THE MARKETS IN FINANCIAL INSTRUMENTS DIRECTIVE (MIFID).* Retrieved 2013 from EUROPEAN COMMISSION: http://ec.europa.eu/clima/consultations/0013/consultation_paper_en.pdf

EUROPEAN PARLIAMENT. (2012). *Financial trading rules: Economic Affairs Committee MEPs outline reform plan.* Retrieved 2013 from EUROPEAN PARLIAMENT: http://www.europarl.europa.eu/pdfs/news/expert/infopress/20120924IPR52148/20120924IPR52148_en.pdf

Farlex, Inc. (2012). *Farlex Financial Dictionary*. Retrieved 2013 from http://financial-dictionary.thefreedictionary.com/

FUSION SYSTEMS. (2012). *Introducing RAPTOR.* Retrieved 2013 from RAPTOR: Guaranteed Ultra Low Latency Direct Market Access for APAC and EMEA: http://www.ulldma.com/products

Getchell, A. (2008, June 8). *Agent-based Modeling.* Retrieved 2013 from University of California, Davis: http://insecure.ucdavis.edu/Members/adam/physics/Agent-based_Modeling.pdf

Golub, A., Keane, J., & Poon, S.-H. (2012). *High Frequency Trading and Mini Flash Crashes*. Retrieved 2013 from http://arxiv.org/pdf/1211.6667.pdf

Investopedia US. (2013). *Definition of 'Price Discovery'.* Retrieved 2013 from INVESTOPEDIA: http://www.investopedia.com/terms/p/pricediscovery.asp

INVESTOPEDIA US. (2013). *Definition of 'Volume Weighted Average Price - VWAP'.* Retrieved 2013 from INVESTOPEDIA: http://www.investopedia.com/terms/v/vwap.asp#axzz2NG3YWOS0

Kirilenko, A., Samadi, M., Kyle, A. S., & Tuzun, T. (2011). *The flash crash: The impact of high frequency trading on an electronic market.* Retrieved 2013 from http://www.q-group.org/pdf/Kyle-paper-Q-group_flash_crash_01182011_small.pdf

LeBaron, B. (2006). Agent-based financial markets: Matching stylized facts with style. In D. Colander, *Post Walrasian Macroeconomics: Beyond the DSGE Model* (pp. 221-235). Cambridge University Press.

Mathiason, N., Fitzgibbon, W., & Ross, A. K. (2012). *Britain opposes MEPs seeking ban on high-frequency trading.* Retrieved 2013 from The guardian: http://www.guardian.co.uk/business/2012/sep/16/meps-ban-high-frequency-trading

NANEX. (2010). *Nanex Flash Crash Summary Report.* Retrieved 2013 from NANEX: http://www.nanex.net/FlashCrashFinal/FlashCrashSummary.html

Nilsson, F., & Darley, V. (2006). On complex adaptive systems and agent-based modelling for improving decision-making in manufacturing and logistics settings: Experiences from a packaging company. *International Journal of Operations & Production Management , 26* (12), 1351 - 1373.

Palmliden, F. (2013). *Time-Weighted Average Price (TWAP): A New Approach.* Retrieved 2013 from Tradestation: http://www.tradestation.com/education/labs/analysis-concepts/time-weighted-average-price

SECURITIES AND EXCHANGE COMMISSION. (2010). *SEC Approves New Rules Prohibiting Market Maker Stub Quotes.* Retrieved 2013 from SECURITIES AND EXCHANGE COMMISSION: http://www.sec.gov/news/press/2010/2010-216.htm

Stafford, P. (2012). *TechBeat: Microwaving high-speed trading.* Retrieved 2013 from Financial Times: http://www.ft.com/cms/s/0/647fc00c-db31-11e1-a33a-00144feab49a.html

Zhang, F., & Powell, S. B. (2011). The impact of high-frequency trading on markets. *CFA Magazine , 22* (2), 10-11.

Appendices

1 User and System Manual

This appendix provides details about how to use the simulator as well as providing all information necessary to make changes to the simulator and recompile it.

Before being able to use this simulator one must install Miranda, installation instructions and downloads can be found through the following link:
http://miranda.org.uk/

Should you need an introduction to programming in Miranda and using the Miranda system I strongly recommend reading Programming with Miranda[45].

There are three key components of the simulator: The harness, the agents and the messages. The harness enables the agents to communicate using these messages.

In this chapter I will explain how to define your own messages and agents and ultimately how to use the simulator.

1.1 Arguments to the simulator harness and agents

Arguments are of the type arg_t and are passed to the simulator at the initialisation of a simulation. The simulator then passes these down to all agents. Both the simulator and agents have access to the entire list and can search through it. These args allow the agents to have access to information that varies each time the simulator is called, and for different pre-coded agent behaviour to be selected when the simulator is called.

Data of type arg_t is either the value EmptyArg or the value Arg – the latter also has additional data with type (str,num) and has a number of methods predefined for it.

- arg_getstr when passed an arg_t will return its string element.
- arg_getnum when passed an arg_t will return its num element.
- arg_findval when passed a key and a list of arg_t will search through the list for the arg with the num that matches the key and return the string element, if it cannot find a matching key it will return -1.
- arg_findstr does the same as findval but this time the key should be a num instead of a string and it matches against the num elements eventually returning the corresponding string if it finds a match and the empty string otherwise

[45] Clack, C and Myers, C and Poon, E (1995) Programming with Miranda. Prentice Hall International

After you pass arguments to the simulator these four functions can be used by the agents to find the arguments they are interested in the argument list which is passed down to them by the simulator.

For example: You want an agent to behave in one manner or another.

We define an arg as follows: Arg (String "Behave!", x) where we set x to be 1 or 0.

The agent you've defined can later use arg_getnum "Behave!" on the arglist to retrieve x.

Alternatively, if you want to fetch a string instead of a num you can use a unique identifying num and search for that.

IMPORTANT NOTES: No two args should share the same key as the find functions will only return the first match. This means when setting a key (string or num) you must make sure that no other arg can share that!!! A key CAN NOT share a possible value with ANY other key OR value. That means if you want to use the number 6 as a unique identifier no other arg can contain 6 as its num element, key or otherwise.

1.1.1 Default args

There are a number of args standard to the simulator, here they are as well as their purposes:

- (Arg (String "Randomise", y))
    - This makes the simulator harness nondeterministic in the order in which is passes messages around.
- (Arg (String prefix, 9989793425))
    - This "prefix" string corresponds to where the simulator should write out the data and message logs.
    - E.g. if prefix is "~/home/test" then the message log will be printed to "~/home/test-trace" and the data will be printed to "~/home/test-data.csv".
    - The strange number is a "magic" number used to locate this argument by the simulator.
    - It is of THE UTMOST IMPORTANCE that none of your other args feature this number as key or value.

1.2 Messages (Messages.m)

Messages are how the agents communicate to each other.

There are a number predefined but you may also define your own.

First I will explain how to define a new message that must extend the type msg_t and be able to deal with it's functions, then I will tell you about the predefined messages and broadcasts which are a special type of message.

All of the messages exchanged will be printed out at the end of a simulation either to the *prefix*-trace file (default behaviour) or the *prefix*–data.csv file (if the message was a data message – see Section 4.3.4).

1.2.1 Adding a new type of message

You will have noticed in Messages.m there is an algebraic type mymessage_t. You must add your message type to this list.

In general at the very least your message should be able to hold the id of the agent that sent it and the id of the agent it is being sent to.

(See Section 1.4 to find out how agents are assigned unique identifiers)

For example:

mymessage_t ::= Hiaton | Message (num,num) [arg_t] | ... more message types ... | Pointlessmessage (num,num)

Where each num would correspond to one of the ids. Typically the first is the sender and the second is the recipient.

You must then add a function to the abstype msg_t to allow creation of your new message type. In this case you would add the following to abstype msg_t underneath the other entries.

Pointlessmessage :: (num, num) -> msg_t

Which is to say that pointless message is a function that takes a tuple of two nums and returns a msg_t.

Now you have to actually add this function (constructor) to the script. Just underneath the section with the abstype you will find a block of constructors.

Append your constructor to the bottom of that list, in this case it would simply be the following.

pointlessmessage (from, to) = Pointlessmessage (from,to)

Once your constructor is added it is as simple as calling the constructor to create an instance of your message. For example!!!

pointlessmessage (from, to)

Would create a pointlessmessage from the agent with id 1 to the agent with id 2.

1.2.2 Adapting default msg_t functions to your new type and adding new ones

That's it! You can now create your messages using this function and they will be of the type abs_t, however, we've yet to talk about adapting the default functions to this new type you've created and adding new ones!

The following functions are already defined and must be modified to handle your new type.

For each of the following, add a new line in the script to deal with your new type. (I will use Pointlessmessage to demonstrate.)

- msg_getid :: msg_t -> num
    - msg_getid (Pointlessmessage (from, to)) = to
- msg_getfromid :: msg_t -> num
    - msg_getid (Pointlessmessage (from, to)) = from
- showmsg_t :: msg_t -> [char]
    - showmsg_t (Pointlessmessage (from, to)) = "Pointlessmessages don't contain anything worth displaying"
    - Normally your messages would contain some data worth displaying and this is the function that allows you to do that. Whatever you define here is what will be printed in the message log during a simulation.

You can also define your own functions on msg_t by adding it's type signature to the abstype and defining it somewhere in the block of msg_t functions.

To define a a function which tells us whether a message really is a Pointless one we would add the following lines of code.

msg_isPointless :: msg_t -> bool

msg_isPointless (Pointlessmessage (from, to)) = True

Great! But what if our msg_t isn't a Pointlessmessage and we call msg_isPointless? We get the following nasty error!!!

Miranda msg_isPointless hiaton

program error: missing case in definition of msg_isPointless

So we have to deal with the other possible types of msg_t!

If we ACTUALLY want different behaviours for each different type of msg_t then we'll have to define a different case for each one (and an "any"/otherwise pattern matching case which throws an error if we want to be program defensively!!!!) but in this case we just want to say that everything else is definitely not pointless!

msg_isPointless anything = False ||We don't care what anything is but provided it comes below the case for Pointlessmessages we'll get the behavior we want.

We're done!
You've defined a new message and it's functions, congratulations!

1.2.3 Special case: Broadcasts

Broadcasts are a special type of message that is treated differently by the simulator.

Instead of being sent to a single agent defined by its id the message is sent to a group of agents defined by their group id (gid).

These broadcasts are of abstype broadcast_t and are wrapped in wrappers of type (Broadcastmessage x args) which is of the abstype msg_t!!!

Confusing eh?

DON'T WORRY. The simulator passes the broadcasts down to your agents as a separate list to the messages and then it's as simple as mapping the function msg_getbroadcast over the broadcast list to fetch the underlying broadcast types.

In the later sections where I talk about how to define agents and how to call the simulator I will explain both how an agent should deal with these broadcasts and how an agent subscribes itself to a broadcast group.

New broadcast types and messages are added in the same manner as new message types and functions.

There are a few key points to adding a new type.

- Add the constructor's signature to the abstype broadcast_t.
- Add the constructor itself to the list of functions.
- Add the underlying type to the algebraic type mybroadcast_t.
- Modify the other broadcast functions to deal with your new type.
  Similarly when adding a new function:
  - Add the function signature to the abstype broadcast_t.
  - Add the function itself to the list of functions.
  - Ensure your function can deal with the other types of broadcast, be it by ignoring them, doing some particular actions or throwing an error.
    REMEMBER: When using broadcasts the "to" id you give your broadcastmessage no longer corresponds to a single agent but a GROUP of agents.

1.2.4 Special case: Data messages

These messages are created by calling the datamessage function which has the following type:

```
datamessage :: (num, num) -> [char] -> msg_t
```

The function takes a two tuple corresponding to a "from id" and a "to id". This permits data messages to be sent to any agent, but in normal usage the "to id" would be 0 (the id of the harness). The second argument is a string that corresponds to part of a row of a csv file – i.e. it has a trailing comma. The harness treats data messages differently from other messages in that instead of being printed out to the message log they are printed out into a csv file. Newlines are added by the simulator after each timestep.

Standard practice is to send a data message in the very first timestep which contains the column headers; the data messages sent in subsequent time steps will be actual data.

1.3 Agents

In general the agents consist of two parts (which can be collapsed into one but I find this method of writing them is easier to understand):

- The underlying logic
- The wrapper:

  The idea behind this is that the underlying logic is a function that takes any arguments you want, does whatever you intend your agent to do with them and then returns whatever you want it to return.
  This then interfaces with the wrapper, it's the wrapper's job to digest data from the simulator into a format that your underlying logic can deal with and then change whatever your underlying logic returns into a format supported by the simulator.
  Thus here I will explain how agent wrappers should be coded.

You have the choice of either writing your agents directly into agent.m or writing them into a separate file which you then insert into the agents.m file using the following command.

> insert "myagent.m"

1.3.1 Agentstate_t

An agent state is what an agent will use to store any data it wishes internally.

As long as your agent state is part of the algebraic type agentstate_t it will be supported by your agents.

Apart from this constraint your state can be anything.

Here are some example states to give you an idea:

1. Agentstate (num, num, num, lob)
2. Traderstate (num, num, num, sentiment, num, [order -> order])
3. Newagstate ([num], [order], sentiment, [order -> order], [num], [num])
   There is one predefined agent state which is Emptyagentstate and you will see what this is for and how to deal with it in the next section...

1.3.2 Agent_t

An agent is just a function which has to be of a specific type.

agent_t == agentstate_t -> [arg_t] -> [(num, [msg_t], [msg_t])] -> num -> [[msg_t]]

Notice that this is just a type synonym.

It is good practice to say that your_agent :: agent_t just before you define your agent so that if you get the type wrong Miranda will scream at you while compiling but it is entirely possible not to and still have it work provided it is of the same signature.

So, an agent is essentially a function which:

- Takes an agentstate_t as it's first argument, this can be any agentstate_t you've written.
    - NOTE: The first time your agent function is called it will be passed the Emptyagentstate and it is the agent's responsibility to change this to its respective internal state and pass that to its recursive call.
- A list of args, [arg_t], from the simulator harness. Using the functions mentioned earlier in the section on arg_t your agent can find args by key and act on the values within them.
    - E.g. If you wanted an arg to decide how many messages a particular agent should send out per timestep in a certain simulation this is how you would achieve it.
- An infinite list of censored simulator states: [(num, [msg_t], [msg_t])]
    - This is a list of obscured snapshots of the simulator state at each timestep, all that is available in each state is what the agent is allowed to access.
    - The num is the current simulator time.
    - The first list of messages is the list of messages sent to the agent in the previous timestep (destined to be received this timestep).
    - The second list is actually a list of broadcast messages.
        - At this point preprocessing can be easily achieved by map msg_getbroadcast over this list to turn it into a list of broadcast_t.
    - IMPORTANT NOTE: One should only ever look at & act upon the head of this list and pass the tail to the recursive call. Remember, the next element of the list is not generated until ALL agents have sent out messages for the current time step.
        - This is akin to the following paradox: I will look into my future to decide what to do now.
        - The paradox: The future is a consequence of the action you're about to undertake.
        - What this means in Miranda with our agents? THE SIMULATOR HANGS.
- A num corresponding to the agent's unique ID as assigned by the simulator. This should be used to tag your outgoing messages.

- Finally, your function should return an infinite list of lists of messages which corresponds to a list of messages for each timestep of the simulation.
- In actuality your function **must** return a single list of messages consed (using the : operator) to the result of a recursive call to your function with updated arguments as a result of the current call. Your updated arguments will usually be as follows:
    - Your newly updated internal state.
    - The same list of args that were received from the simulator.
        - Can optimise by checking for these just once at the beginning and passing them through in your agentstate instead. Reduces computing because you no longer have to search for them.
    - The tail of the censored infinite list of sim states.
    - The agent ID you were assigned.

      Provided your wrapper follows the aforementioned it can do anything it wants in any format both in itself and in the underlying agent logic, which is why earlier I said it is not completely necessary to split the two. You must just ensure that what you send out is in the form of broadcasts and messages.

      Details for creating messages and extracting data from messages are in the previous chapter on messages.

1.4 Calling the simulator

The simulator is called as follows:

>sim :: num -> [arg_t] -> [(agent_t, [num])] -> num -> [sys_message]

You can either do this explicitly with Sim.m as the working file in Miranda or by again writing a wrapper file which imports Sim.m and defining your own functions that call Sim.m

By importing Sim.m you will have access to all the functions and types you need to make this function call.

The arguments correspond to the following (in the order as displayed above):

1. This is the number of steps the simulation should run for before terminating.
2. This is your list of Args to be passed to the simulator and all agents.
3. This is a list of tuples of the following:
    a. An agent function. (agent_t)
    b. A list of num corresponding to the broadcast groups it should be subscribed to.

   The agent will be assigned a unique id at the start of the simulation and this corresponds to its index in this list counting from 1 because id 0 is assigned to the simulator harness, messages sent to this id (0) will be printed out to the trace file but otherwise ignored.

KEY NOTE: This is how you know the id to use when exchanging messages.
This is a simple example call to Sim

- sim 200 [(Arg (String "Randomise", -1)), (Arg (String "~/Home/slice", 9989793425))] [(myAgent, [0,1,2]), (hisAgent, [0]), (herAgent, [1,2])]
  This would run the following simulation:
- Run for 200 time steps.
- Randomise the order in which the harness passes messages.
- Output the data and trace files to the folder "~/Home/" with the prefix "slice".
- The simulation is to be run with the following 3 agents.
  - myAgent subscribed to broadcast groups 0, 1 and 2.
  - hisAgent subscribed to broadcast group 0.
  - herAgent subscribed to broadcast groups 1 and 2.

1.5 Communicating with the exchange agent!

Provided with the simulator harness is a ~~CME E-mini~~ niCe_miME exchange agent and a number of predefined message types to communicate with it and for it to communicate back to an agent, these are:

- **To the exchange:**
  - ordermessages: These are used to indicate to the exchange that you would like to place an order on the book and here is an example as well as an explanation for its construction:

```
(ordermessage (myid, 1) (order_setuid "ta1" 1
(order_create (ltp - 12) 500 time (FundSeller myid)
(Bid (Goodtilldate time)))))
```

You will notice that embedded in the ordermessage constructor there are two function calls: `order_create` and `order_setuid`, these are used to construct the order held within the ordermessage, aside from this there is only a tuple with a from and to address `(myid, 1)` where 1 is the to address.
`order_create` takes the following arguments: price volume time `traderid ordertype`
Price, volume *(to trade)* and time are all of type `num` and are self-explanatory.

```
>traderid ::= Phantom | Intermediary num | HFT num |
              FundSeller num | FundBuyer num |
              Small num | Opportunistic num |
              SSTaker num | BSTaker num | BSMaker num
              | SSMaker num | Maker num | Taker num
```

These are simply your agent's unique ID (the num) and its "type", these

tags will simplify identifying one order belonging to one agent from another later on.

```
>ordertype ::= Bid limitorder_t | Offer limitorder_t |
Sell marketorder_t | Buy marketorder_t | None | Abort
>marketorder_t ::= Orkill | Andkill
      >limitorder_t ::= Goodtillcancelled |
      Goodtilldate num
```

OK – Next, ordertype, essentially this lets the exchange (and by consequence other agents) know what type of order you're placing.

Market orders – Orkill means trade for the ENTIRE market order volume or don't trade at all, Andkill means trade what you can and return a rejection for the remaining volume.

Limit orders – Goodtillcancelled means leave the order on the book until cancelled, Goodtilldate means leave the order on the book until the timestep specified by num.

Finally, `order_setuid` lets you assign an agent unique order id to an order, this coupled with the agent's id forms a unique id which can later be used to cancel messages. The string that `order_setuid` takes is the agent name and the num is the agent ID.

NOTE: There is a large array of functions associated with the type "order" and these are available for viewing in order.m

There is also a function defined on ordermessages that takes a list of them and returns a list of messages:

- `msg_getorders :: [msg_t] -> [order]`
- `|| returns orders from list of messages, ignores non-ordermessages`

- cancelmessages: These are used to indicate to the exchange that you would like to cancel an order, here's how they're constructed.
`(cancelmessage (myid, toid) (myid, ouid))`

As you can see, they also contain the typical from/to tuple but now they also contain a second tuple which is simply once again the "from id" and the unique order id of the order you wish to cancel.

- **From the exchange:**
  - `ackmessage`: These are sent back from the exchange to indicate the status of an order. It has three methods defined on it: `msg_isack` *(returns true if a message is an* `ackmessage`*)*, `msg_isreject` which returns true if the ackmessage is a rejection and `msg_getackcode` which returns the ackcode associated with the ack. Below is a list of ackcodes:

```
0 accept,
1 order too large,
2 no more liquidity,
3 outside sliding window of acceptable prices,
4 too many contracts on book,
5 order cancelled,
6 book is spiked,
7 minimum resting time not obeyed.
```

  - `trademessage`: These indicate that two orders have been matched and executed against each other. The methods `msg_istrade` *(returns true if a message is a trademessage)* and `msg_gettrade` *(returns a two tuple of orders in which the first element is your order and the second is the one it was matched against)* are defined on it.
  - `The broadcast`: This is slightly trickier to explain away than the others. Once a turn the exchange will send out a broadcast to **broadcast group #0**, this is a numlist broadcast of statistics. You can fetch these from the broadcasts using `broadcast_getnumlist` and the list will contain the following values in order:

1. **bb – Best bid**
2. **bo – Best offer**
3. **bsld – Buy side liquidity**
4. **ssld – Sell side liquidity**
5. **bsls – Buy side levels**
6. **ssls – Sell side levels**
7. **bsd – Buy side depth near top**
8. **ssd – Sell side depth near top**
9. **or – Orders received this timestep**
10. **ltp – Last traded price**

IMPORTANT NOTE: When calling Sim you will have to make sure you add `nice_mime_wrapper` to the list of agents you pass in. Recall: It's position in this list will be it's harness assigned ID and will be the ID you should use to send messages to the exchange.

MOST IMPORTANT NOTE:

If you have any questions PLEASE don't hesitate to get in touch!!!!
(Elias.Court.09@ucl.ac.uk)

2 Test results

2.1 Unit testing

I created an extensive automated suite of tests for nicemime.m as it was the largest of the units in this project. Below is a subset of the final test results (All of which returned "Success."):

```
Test result: Success.          Test result: Success.          Test result: Success.
Test result: Success.          Test result: Success.          Test result: Success.
Test result: Success.          Test result: Success.          Test result: Success.
```

2.2 Integration testing

Once I was confident that all the agents were operating as expected I ran a 300-timestep experiment in calm conditions with 4 HFT MMs, a noise agent and 10 fundamental buyers and sellers (to fuel the market with liquidity).

As expected the last traded price stayed relatively stable as well as the spread after an initial period of frequent crossed books (reminiscent of price discovery[46]). In addition, the HFT MMs frequently traded to both extremes of their inventory as per their design.

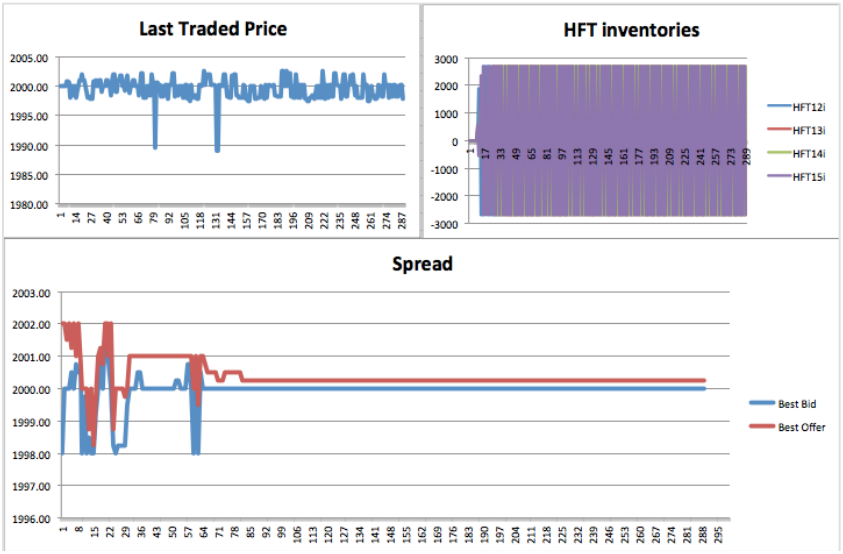


Figure 2-1 – The graph on the top left shows the last traded price over time, the graph on the top right shows the current inventory for each HFT over time and the graph on the bottom shows the best bids and offers over time.

[46] Price discovery – "A method of determining the price for a specific commodity or security through basic supply and demand factors related to the market." (Investopedia US)

3 Project plan

Name: Elias Court

Project title: The instability of trading algorithms

Supervisor's Name: Christopher D. Clack

Aims and objectives

Aim: To model and simulate the interactions between computer trading algorithms and how this can lead to instability in the financial markets. The project will aim to use a functional language to simulate the algorithms.

Objectives:

- Try to determine factors that determine whether or not a system of market making agents will exhibit instability.
- Build an agent-based simulator and a number of different agents needed for our experiments.
- Validate this simulator first through unit testing each agent then by ensuring stability in systems we know are stable.
- Show that this certain instability can be recreated in experimental conditions.
- Investigate the effect of changing the deciding factors (as mentioned earlier) in the system.
- Investigate whether proposed protection measures prevent or dampen instability and if so how effective they are.

Deliverables:

- Design specification for the simulator.
- The simulator itself.
- Documentation for the use of the simulator.
- A series of unit tests validating the agents and the simulator.
- Results obtained and conclusions reached from running experiments.

Work plan

- Project start to end October (4 weeks) - Literature search and review.
    - Learn Miranda.
    - Collate documents describing functionality of agents that need to be implemented.
    - Learn about agent based modeling.
    - Read about 2010 flash crash.
- Mid-October to mid-November (4 weeks) - Analysis and modelling.
    - Try to determine factors that cause instability.
    - How it can be induced.
    - Conditions to maintain it.
    - What are the agents we need to run the necessary simulations?
- November (4 weeks) - System design, coding small test programs and simple prototypes.
    - Design the simulator.
        - Determine the mechanisms required for running the simulations we want e.g. Broadcasts for messages aimed at multiple agents.
    - Turn the documents collected in first 4 weeks into specifications for each agent.
- End November to mid-January (6 weeks) - System implementation.
    - Build the simulator harness.
    - One by one incrementally build the agents unit testing as we go along.
    - Validate the simulator and agents as a whole.
- Mid-January to mid-February (6 weeks) - Experimentation
    - Run experiments investigating the factors that affect instability.
    - Implement protection measures and investigate efficacy.
- Mid-February to end of March (6 weeks) - Work on Final Report

4 Interim report

Name:

Elias Court

Project title:

The instability of trading algorithms

Supervisor:

Christopher D. Clack

Progress made to date:

I have completed implementation of the simulator harness and a set of agents (HFT MM, noise trader, exchange agent, probe agent etc.).

I have also completed compile-time, and unit testing of these.

I have run simulations in what are considered stable conditions to ensure that agents behave as expected in these (link testing) – and the debugging that goes alongside it.

I have completed experiments that show a hot potato effect despite price banding, maximum order sizes and maximum allowed stock market protection measures on the book.

I have designed a number of additional agents for experimentation purposes (lagged HFT MM, spike-probe trader) as well as refactoring existing code to allow for certain conditions to be engineered.

I have run and statistically analysed a number of experiments testing the efficacy of minimum resting times.

I have completed a preliminary draft of the abstract chapter, the introduction chapter and the background chapter. I have also completed a preliminary draft of the design and implementation chapters however these are subject to some change as further requirements reveal themselves during experimentation.

Remaining work to be done:

I have yet to devise and run all the necessary experiments and analyse statistically the data retrieved from these experiments.

As I come up with new experiments the design/implementation requirements grow as a result of needing new agents or additional functionality.

I have yet to complete a preliminary draft of the experiments chapter and the conclusions chapter.

I have yet to write the final draft of any one chapter.

5 Code listing[47]

5.1 Sim.m

```
>|| A Literate script - all lines starting with ">" are code; all other lines are
comments.
>|| A block of code lines must be preceded by and followed by a blank line.
>|| Other comments may be added in code lines using "||" as usual



>%include "./Agents.m"
>%include "./Messages.m"
>%include "./stats.m"
>%export + "./Agents.m" "./Messages.m"


>||%include "./sim-orderlist.m"
>||%include "./sim-lob.m"


>assetreturnscoln = 27


Generic Agent-Based Simulator
=============================
Modifications 2013 by E. Court
Copyright 2012 C.D.Clack
Based on previous 2011 version by C.Clack

This generic simulator support a discrete-time simulation.  The generic framework
supports message-passing between
any number of defined agents.  It is suggested that Miranda's heap size should be set to
the maximum available.

The main simulator
==================
Args are (string,value) pairs with numeric values - these are passed on to simstep,
sim_updatestate and agents
The "agents" parameter to sim is a list of functions, which are applied to their
arguments inside sim
Once applied to their full set of arguments, agents consume the potentially-infinite
stream of simulator system states ("allstates")
and each produces a potentially-infinite stream of lists of messages to be sent to other
agents (according to the target id in the message).
Notice that at each timestep an agent can produce a list of messages - because one agent
might want to simultaneously send messages to several different
other agents.
They do this in a recursive loop - it is important that (i) there is a precomputed first
state for agents to view, (ii) the
agents only ever look at the head of the incoming list of states each time around their
own loop, and (iii) the "simstep" function
only ever looks at the head of the message lists from the agents each time around its own
loop.  Any attempt to lookahead will cause the simulation
to deadlock.
Any message sent to target id "0" is a message for the simulator harness and is used to
update the simulator state - which is
printed to the trace file and is also visible to all other agents.
Each agent has its own private state.
The last arg to sim is the required highest group number for broadcast messages
```

[47] The full code listing is provided on the accompanying disk.

```
>sim :: num -> [arg_t] -> [(agent_t, [num])] -> [sys_message]
>sim steps args agents = ((Tofile (prefix ++ "trace") ("START OF SIMULATION" ++ "\n\n"))
: (tracer allstates prefix)) ++ statstuff
>                                  where
>                                  statstuff = (printstatoutput (statfuncs (prefix ++
"data.csv") assetreturnscoln True) (prefix ++ "-asset_returns-")), if (arg_findval
"autostats" args) ~= -1
>                                            = [], otherwise
>                                  highestbrg = max ((concat (map snd agents)) ++ [0])
||The zero is incase there are no agents subscribed to any broadcast groups.
>                                  myrands = drop (pmh + #agents) randoms, if
(arg_findval "randseed" args) ~= -1
>                                          = drop ((arg_findval "randseed" args) * (pmh
+ #agents)) randoms, otherwise
>                                  pmh = (sum (map code prefix)) ||Poor Man's Hash
(function)'
>                                  prefix = (arg_findstr 9989793425 args) ++ "-", if
(arg_findstr 9989793425 args) ~= ""
>                                         = "", otherwise
>                                  startsimstate = sim_emptystate agents highestbrg
myrands
>                                  allstates = startsimstate : simstates
>                                  simstates = simstep steps 1 args startsimstate
allmessages myrands
>                                  allmessages = map f (zip2 [1..] agents)
>                                                where
>                                                || f is a function that fetches the
messages and broadcasts for a given agent (using sim_getmymessages and
sim_getmybroadcasts)
>                                                || and then applies the agent to (i)
the empty state; (ii) the args (that are sent to the simulator at the start);
>                                                || (iii) an infinite list of lists of
messages and broadcasts, and (iv) the agent id.
>                                                || It fetches the messages trivially
by using the agent's ID to index into the message list
>                                                || (which is sorted by destination ID
for each message).
>                                                || It fetches the broadcasts for an
agent as follows:
>                                                || - each agent is subscribed to a
list of broadcast groups (this is the data called brcs)
>                                                || - for each subscribed broadcast
group ID, that ID is used to index into the broadcast list (which is
>                                                ||   held in the state called st, and
which is sorted into broadcast groups).  The indexing is done by sim_getmybroadcasts.
>                                                || - all of the returned broadcast
lists (for different broadcast group IDs) are concatenated into a single list.
>                                                f (id,(a, brcs)) = a emptyagentstate
args (map (g id) allstates) id
>                                                                   where
>                                                                   g x st =
(sim_gettime st, sim_getmymessages st x, concat (map (sim_getmybroadcasts st) brcs))
>
>                                  simstep n t args st []   myrnds   = []
|| for completeness - should never happen
>                                  simstep 0 t args st msgs myrnds   = []
|| end of simulation steps
>                                  simstep n t args st msgs myrnds   = [], if (member
msgs [])                            || end simulation if any agent ends
>                                  simstep n t args st msgs myrnds   = newstate :
(simstep (n-1) (t+1) args newstate (map tl msgs) (drop t myrnds))
>                                                                     where
>                                                                     newstate =
sim_updatestate t args cleanst (map (cpsafehd "simstep") msgs) myrnds   || t is the
"time" - the simulation step
>                                                                     cleanst =
sim_clean_msg_br st
>

Tracer can take an argument to send its output to specified files. The argument is
detailed below.
Presence of (Arg (String prefix, 9989793425)) indicates files should be written to the
messyoutputfolder folder in the files prefixed by the specified prefix.
```

```
E.g. if prefix is "MyLatestTest" files will be written to folder "MyLatestTest" and the
files will be called "MyLatestTest-data.csv" and "MyLatestTest-trace".
The number 9989793425 acts as a key so that the HFT agent can find the value (the
prefix).

>tracer :: [simstate_t] -> [char] -> [sys_message]
>tracer [] prefix     = (Tofile (prefix ++ "trace") ("END OF SIMULATION\n")):(Closefile
(prefix ++ "data.csv")):(Closefile (prefix ++ "trace")): []
>tracer (x:xs) prefix = [(Tofile (prefix ++ "trace") ((show x) ++ "\n==============\n"))]
++ datatraces ++ (tracer xs prefix)
>                       where
>                       datatraces = [(Tofile (prefix ++ "data.csv")
dtracepostformatting)]
>                       dtracepostformatting = dtracepreformatting ++ "\n", if
dtracepreformatting ~= ""
>                                              = dtracepreformatting, otherwise
>                       dtracepreformatting = ((concat (map msg_disptrace (filter
msg_isdata (sim_getmymessages x 0)))))


The abstract type definition for simulator state
================================================

>abstype simstate_t
>with
>    sim_emptystate :: [(agent_t,[num])] -> num -> [num] -> simstate_t
|| creates an empty message stream for each agent
>    sim_updatestate :: num -> [arg_t] -> simstate_t -> [[msg_t]] -> [num] -> simstate_t
|| time, args, oldst, one msg_t per agent
>    showsimstate_t :: simstate_t -> [char]
>    sim_gettime :: simstate_t -> num
>    sim_getmymessages :: simstate_t -> num -> [msg_t] ||num is id
>    sim_getmybroadcasts :: simstate_t -> num -> [msg_t] ||num is groupid
>    sim_clean_msg_br :: simstate_t -> simstate_t
>    || and so on - we will need access methods
>

The implementation of simulator state
It must implement methods sim_emptystate, sim_updatestate and showsimstate_t

>simstate_t == ([[msg_t]], num, [msg_t], [[msg_t]])                        || list of
messages for each agent - a snapshot at time t - plus time and other info
>sim_emptystate []     hbrg rnds = ([[]], 0, [(debugmessage (0,0) (show (take 30
rnds)))], (map f [0..hbrg]))            || this base case gives the extra list of
messages for agent id which is the sim harness
>                                   where
>                                   f x = []
>sim_emptystate (a:as) hbrg rnds = (([]:x), b, c, br)
>                                   where
>                                   (x,b,c,br) = sim_emptystate as hbrg rnds
>
>sim_clean_msg_br (m, b, c, br) = (emptym, b, c, emptybr)
>                                  where
>                                  emptym = map f m
>                                  emptybr = map f br
>                                  f x = []
>sim_updatestate t args (m, b, c, br) [] myrands     = (reverse (map reverse m2), t,
safehd m "sim_updatestate", reverse br2) ||just copy over broadcast??
>                                                      where
>                                                      m2 = ((map (randomWrap myrands)
(take (#m - 1) m)) ++ (drop (#m - 1) m)), if randomise = True
>                                                         = m, otherwise
>                                                      br2 = (map (randomWrap myrands)
br), if randomise = True
>                                                          = br, otherwise
>                                                      randomise = True, if (arg_findval
"Randomise" args) ~= (-1)
>                                                                = False, otherwise
>sim_updatestate t args (m, b, c, br) (x:xs) myrands
>                                                    = sim_updatestate t args (fm, b, c, fcasts)
xs (drop 72 myrands)
>                                                      where
>                                                      || updatemsgs takes messages from agents
```

```
and puts them in msgqueues for agents (id 0 = for sim)
>                                                 updatemsgs 0 (y:ys) = [(filter
((=0).msg_getid) (filter ((~).msg_isbroadcast) x)) ++ y]
>                                                 updatemsgs n (y:ys) = ((filter
((=n).msg_getid) (filter ((~).msg_isbroadcast) x)) ++ y): (updatemsgs (n-1) ys)
>                                                 updatecasts 0 (y:ys) = [(filter
((=0).msg_getid) (filter msg_isbroadcast x)) ++ y]
>                                                 updatecasts n (y:ys) = ((filter
((=n).msg_getid) (filter msg_isbroadcast x)) ++ y): (updatecasts (n-1) ys)
>                                                 newm =  (updatemsgs ((# m) - 1) ( m))
>                                                 newcasts = (updatecasts ((# br) - 1) (
br)) ||These guys need to know the number of broadcast groups...
>||                                                (fm, fcasts) = (((map (randomWrap
myrands) (take newml newm)) ++ (drop newml newm)),(map (randomWrap myrands) newcasts)),
if randomise = True
>                                                 (fm, fcasts) = (newm, newcasts)
>||                                                newml = (# newm) - 1
>||                                                randomise = True, if (arg_findval
"Randomise" args) ~= (-1)
>||                                                          = False, otherwise
>||
>
>
>showsimstate_t (m,b,c,br) = ", \n\nTime: "++(show b)++"\nMessages to sim: "++(concat
(map (concat.(map show)) m))++", \n\nHarness messages: " ++ (concat (map show c)) ++
"\n\nBroadcasts: "++(show br)++"\n\n\nEND OF STATE\n"
>||showsimstate_t (m,b,c,br) = ", \n\nTime: "++(show b)++"\nMessages to sim: "++(concat
(map (concat.(map show)) m))++", \n\nUnknown: "++(show c)++", \n\nBroadcasts: "++(show
br)++"\n\n\nEND OF STATE\n" OLD UNKNOWN FIX ME
>||Only index 2 of messages and br is populated and for some reason only one message is
retained...
>
>sim_gettime (m, b, c, br) = b
>
>sim_getmymessages (m,b,c,br) idnum = si "sim:1" m idnum
>sim_getmybroadcasts (m,b,c,br) groupnum = si "sim:2" br groupnum


Helper functions
================================================




>randomWrap myrands list
>                = randomize (map ((* coef).(converse (-) 500)) (drop (systemTime len)
myrands)) list
>                  where
>                  coef = 1, if len < 500
>                       = (entier (len / 500)) + 1, otherwise
>                  len = # list

>randomize rands [] = []
>randomize (r:rs) msgs
> =  a : (randomize rs b)
>    where
>    a = msgs ! ((entier r) mod len)
>    len = # msgs
>    b = msgs -- [a]
```

5.2 Messages.m

```
>|| A Literate script - all lines starting with ">" are code; all other lines are
comments.
>|| A block of code lines must be preceded by and followed by a blank line.
>|| Other comments may be added in code lines using "||" as usual


User includes
==============================

>||%include "./sim-orderlist.m"
>||%include "./sim-lob.m"
>%include "./Orders.m"
>||%export + order

Standard type definitions
=========================

>str ::= EmptyString | String [char]
>arg_t ::= EmptyArg | Arg (str,num)
>arg_getstr :: arg_t -> [char]
>arg_getstr (Arg ((String x),y)) = x
>arg_getnum (Arg ((String x),y)) = y
>arg_findval key [] = -1
>arg_findval key ((Arg ((String x),y)):t) =  y, if x = key
>                                        = arg_findval key t, otherwise
>arg_findstr key [] = ""
>arg_findstr key ((Arg ((String x),y)):t) =  x, if y = key
>                                        = arg_findstr key t, otherwise



The message type & broadcast message type
================

>abstype msg_t
>with
>    hiaton :: msg_t    || an empty message
>    msg_getid :: msg_t -> num
>    msg_getfromid :: msg_t -> num
>    showmsg_t :: msg_t -> [char]
>    message :: (num,num) -> [arg_t] -> msg_t
>    ordermessage :: (num,num) -> order -> msg_t
>    cancelmessage :: (num,num) -> (num,num) -> msg_t || First tuple is from/to, second
tuple is tid/ouid
>    trademessage :: (num,num) -> order -> order -> msg_t
>    datamessage :: (num, num) -> [char] -> msg_t
>    debugmessage :: (num,num) -> [char] -> msg_t
>    ackmessage :: (num, num) -> num -> order -> [char] -> msg_t ||0 accept, 1 order too
large, 2 no more liquidity, 3 outside sliding window of acceptable prices, 4 too many
contracts on book, 5 order cancelled, 6 book is spiked, 7 minimum resting time not
obeyed.
>    broadcastmessage :: (num,num) -> broadcast_t -> msg_t
>    msg_isbroadcast :: msg_t -> bool
>    msg_isdata :: msg_t -> bool
>    msg_disptrace :: msg_t -> [char]
>    msg_getbroadcast :: msg_t -> broadcast_t
>    msg_getorders :: [msg_t] -> [order] || returns orders from list of messages, ignores
non ordermessages
>    msg_getnumlistfrombcast :: msg_t -> [num]
>    msg_gettrade :: msg_t -> [order]
>    msg_isack :: msg_t -> bool
>    msg_getackcode :: msg_t -> num
>    msg_istrade :: msg_t -> bool
>    msg_isreject :: msg_t -> bool || Named this isreject instead of isack to avoid
nameclashing and confusion.
>    msg_getcanceltuple :: msg_t -> [(num,num)]
>    msg_getackdorder :: msg_t -> order


Start of constructors.
```

```
====

>message x args  = Message x args
>debugmessage x y = Debugmessage x y
>ordermessage x anOrder = Ordermessage x anOrder
>broadcastmessage x broadcast = Broadcastmessage x broadcast
>cancelmessage fromto idtuple = Cancelmessage fromto idtuple
>trademessage fromto ordera orderb = Trademessage fromto ordera orderb
>datamessage from data = Datamessage from data
>ackmessage fromto ackd x msgg = (Ackmessage fromto ackd x msgg)

====




>msg_t == mymessage_t || Key here in broadcast message is group id. Msg is targetid
followed by (key,value) pairs (and underlying object?)
>mymessage_t ::= Hiaton | Cancelmessage (num,num) (num,num) | Message (num,num) [arg_t] |
Ordermessage (num,num) order | Trademessage (num,num) order order| Broadcastmessage
(num,num) broadcast_t | Datamessage (num,num) [char] | Ackmessage (num,num) num order
[char] | Debugmessage (num,num) [char]
>                 || (from,to) identifiers
>hiaton = Hiaton


Start of msg_t functions
====


>msg_getid Hiaton = 0
>msg_getid (Message (from,to) args) = to
>msg_getid (Ordermessage (from,to) args) = to
>msg_getid (Broadcastmessage (from,to) args) = to
>msg_getid (Trademessage (from,to) ordera orderb) = to
>msg_getid (Datamessage (from, to) data) = to
>msg_getid (Ackmessage (d,e) b c m) = e
>msg_getid (Cancelmessage (f,t) b) = t
>msg_getid (Debugmessage (f,t) m) = t
>msg_getfromid Hiaton = 0
>msg_getfromid (Message (from,to) args) = from
>msg_getfromid (Ordermessage (from,to) args) = from
>msg_getfromid (Broadcastmessage (from,to) args) = from
>msg_getfromid (Trademessage (from,to) ordera orderb) = from
>msg_getfromid (Datamessage (from, to) data) = from
>msg_getfromid (Ackmessage (d,e) b c m) = d
>msg_getfromid (Cancelmessage (f,t) b) = f
>showmsg_t Hiaton = "\nHiaton"
>showmsg_t (Datamessage x data) = ""
>showmsg_t (Cancelmessage x (trader, order)) = "\nMessage from/to " ++ (show x) ++ ":
Trader #" ++ (show trader) ++ " is cancelling order " ++ (show order)
>showmsg_t (Message x args) = "\nMessage from/to "++(show x)++" ["++(concat (map
arg_getstr args)) ++ "]\n", otherwise
>showmsg_t (Ordermessage x args) = "\nMessage from/to "++(show x)++" "++(showorder args)
++ ""
>showmsg_t (Broadcastmessage x args) = "\nBroadcast from/to group "++(show x)++"
["++(show args) ++ "]"
>showmsg_t (Trademessage (from,to) ordera orderb) = "\n==========\n\nTo: " ++ (show to)
++ "\nOrder: \n" ++ (show ordera) ++ "\n\n matched with\n\nOrder:" ++ (show orderb) ++
"\n\n=========="
>showmsg_t (Ackmessage (d,e) b c m) = "\nAckmessage(" ++ (show d) ++ "->" ++ (show e) ++
"): The following order " ++ text ++ (show c)
>                                    where
>                                    text = "was unsuccessful because " ++ m ++ ": \n",
if b ~= 0
>                                         = "was cancelled successfully.", if b = 5
>                                         = "was successful: \n", otherwise
>showmsg_t (Debugmessage ft m) = "\n+=+=+=+=+=+=+=+=+=+=+=+=+=+=\nDebug message:" ++ (show
ft) ++ "\n" ++ m ++ "\n+=+=+=+=+=+=+=+=+=+=+=+=+=+="
>msg_disptrace (Datamessage x data) = data
>msg_disptrace any = ""
```

```
>msg_getnumlistfrombcast (Broadcastmessage a b) = broadcast_getnumlist b
>msg_isbroadcast (Broadcastmessage a b) = True
>msg_isbroadcast any = False
>msg_isdata (Datamessage a b) = True
>msg_isdata any = False
>msg_isack (Ackmessage a b c m) = True
>msg_isack any = False
>msg_getackcode (Ackmessage a b c m) = b
>msg_getackcode any = error "msg_getackcode - calling get ack code on non-ack message."
>msg_getackdorder (Ackmessage a b c m) = c
>msg_getackdorder any = error "msg-getackdorder - Called this on a non-ack message"
>msg_istrade (Trademessage (from,to) ordera orderb) = True
>msg_istrade any = False
>msg_isreject (Ackmessage a b c m) = b ~= 0
>msg_isreject any = error "msg_isreject called on non-ack message."
>msg_getbroadcast (Broadcastmessage a b) = b
>msg_getbroadcast any = error "Make sure you're trying to get the broadcast FROM a
broadcast message first..."
>msg_getorders [] = []
>msg_getorders ((Ordermessage a b) : rest) = b : (msg_getorders rest)
>msg_getorders (any:rest) = msg_getorders rest
>msg_gettrade (Trademessage (from,to) ordera orderb) = [ordera, orderb]
>msg_gettrade any = []
>msg_getcanceltuple (Cancelmessage x b) = [b]
>msg_getcanceltuple any = []


====


map msg_getbroadcast


>
>
>
>abstype broadcast_t
>with
>    showbroadcast_t :: broadcast_t -> [char]
>    broadcast_getnumlist :: broadcast_t -> [num]
>    broadcast_numlist :: [num] -> broadcast_t
>    broadcast_str  :: str -> broadcast_t


The implementation of broadcasts

>broadcast_t == mybroadcast_t
>mybroadcast_t ::= Numlistbroadcast [num] | Strbroadcast str ||  Add your self-defined
broadcasts here
>showbroadcast_t (Numlistbroadcast nums) = "LOBbcast contents: ["++(show nums) ++ "]"
>showbroadcast_t (Strbroadcast string) = show string
>broadcast_getnumlist (Numlistbroadcast lobsum) = lobsum

>broadcast_numlist ob = Numlistbroadcast ob
>broadcast_str string = Strbroadcast string
```

5.3 Agents.m

```
>|| A Literate script - all lines starting with ">" are code; all other lines are
comments.
>|| A block of code lines must be preceded by and followed by a blank line.
>|| Other comments may be added in code lines using "||" as usual

System includes
==============================

>%include "./Messages.m"

User includes
==============================

>||%include "./sim-orderlist.m"
>%include "./sim-lob.m"
>%include "./Traders.m"
>%export + "./sim-lob.m"

>%insert "./nicemime.m"
>%insert "./Hft.m"
>%insert "./LaggedHft.m"
>%insert "./Noiseagent.m"
>%insert "./Fundamentals.m"



The agent and agentstate types
==============================

>agent_t      == agentstate_t -> [arg_t] -> [(num, [msg_t], [msg_t])] -> num -> [[msg_t]]
||Agents take a tuple of (time, messages, broadcasts) last num is id from sim
>agentstate_t ::= Agentstate (num, num, num, lob) | Emptyagentstate | Exchstate lob |
Nicemimestate nice_mime_lob | Traderstate (num, num, num, sentiment, num, [order ->
order]) | Newagstate ([num], [order], sentiment, [order -> order], [num], [num])   ||
traderstate is old_bestbid, old_bestoffer, old_ordernum
>
>emptyagentstate = Emptyagentstate     || we don't yet know what this will or should be


The collection of agents
========================

>nice_mime_lob ::= Nice_mime_lob sentiment listoftotals nubids nuoffers num ticksize
lasttradedprice stats
>listoftotals == [num]
>stats == [num]
>nubids == [tick]
>nuoffers == [tick]
>ticksize == num
>tick == (num, [order])
>lasttradedprice == num



exchagent's ID will be ID 1.

>||price 2000 size 100 bid (price, size, time, traderid, ordertype)
>testagent1 :: agent_t
>testagent1 mystate args []     myid                = []
>testagent1 mystate args ((time, messages, broadcasts) : simstateinfos) myid
>   = (probeorder ++ liquid) : (testagent1 mystate args simstateinfos myid)
>     where
>     liquid = [], if (arg_findval "PeriodicLiquidity" args) = -1
>            = rep 1 (ordermessage (myid, 1) (order_setuid "ta1" 1 (order_create (ltp -
12) 500 time (FundSeller myid) (Bid (Goodtilldate time))))), if time mod 70 = 44
>            = [], otherwise || [(cancelmessage (myid, 1) (myid, 1))], otherwise
>     probeorder = [(ordermessage (myid, 1) (order_setuid "ta1" 1 (order_create 2000
probesize time (FundSeller myid) (Sell Andkill))))], if time <= teststop
>               = [], otherwise
```

```
>     probesize = arg_findval "ProbeSize" args, if (arg_findval "ProbeSize" args) ~= (-1)
|| CDC 17/08/12
>               = 15, otherwise
>     teststop = arg_findval "ProbeStop" args, if (arg_findval "ProbeStop" args) ~= (-1)
>              = 140, otherwise
>     [bb, bo, bsld, ssld, bsls, ssls, bsd, ssd, or, ltp] = [1993, 2007, 500, 500, 1, 1,
500, 500, 2, 2000], if time <= 0 || [bb, bo, bsld, ssld, bsls, ssls, bsd, ssd, or, ltp]
>                                                          = msg_getnumlistfrombcast
(findG0bcast broadcasts), otherwise
>                                                            where
>                                                            findG0bcast [] = error ("No
broadcast found?" ++ (show time))
>                                                            findG0bcast (f:r) = f, if
(msg_isbroadcast f) & ((msg_getid f) = 0)
>                                                                                =
findG0bcast r, otherwise

>spikeagent :: agent_t
>spikeagent mystate args ((time, messages, broadcasts) : simstateinfos) myid
>  = (liquidity ++ curativeliquidity):(spikeagent mystate args simstateinfos myid)
>    where
>    liquidity = rep 20 (ordermessage (myid, 1) (order_setuid "sp1" 1 (order_create (ltp
+ 12) maxorder time (FundSeller myid) (Offer (Goodtillcancelled))))), if time = 0
>              = [], otherwise
>    curativeliquidity = rep 10 (ordermessage (myid, 1) (order_setuid "sp1" 1
(order_create (ltp + 1) maxorder time (FundSeller myid) (Offer (Goodtillcancelled))))) ++
rep 20 (ordermessage (myid, 1) (order_setuid "sp1" 1 (order_create (ltp - 1) maxorder
time (FundSeller myid) (Bid (Goodtillcancelled))))), if findG1bcast broadcasts
>                      = [], otherwise
>
>    findG1bcast [] = False
>    findG1bcast (f:r) = True, if (msg_isbroadcast f) & ((msg_getid f) = 1)
>                      = findG1bcast r, otherwise
>     [bb, bo, bsld, ssld, bsls, ssls, bsd, ssd, ors, ltp] = [1993, 2007, 500, 500, 1, 1,
500, 500, 2, 2000], if time <= 0 || [bb, bo, bsld, ssld, bsls, ssls, bsd, ssd, or, ltp]
>                                                          = msg_getnumlistfrombcast
(findG0bcast broadcasts), otherwise
>                                                            where
>                                                            findG0bcast [] = error ("No
broadcast found?" ++ (show time))
>                                                            findG0bcast (f:r) = f, if
(msg_isbroadcast f) & ((msg_getid f) = 0)
>                                                                                =
findG0bcast r, otherwise

>testagent2 :: agent_t
>testagent2 mystate args []                      myid = []
>testagent2 mystate args ((time, messages, broadcasts) : simstateinfos) myid =
[(ordermessage (myid, 1) (order_setuid "ta2" 3 (order_create 0 100 time (FundBuyer myid)
(Buy Andkill)))), (ordermessage (myid, 1) (order_setuid "ta22" 4 (order_create 1850 100
time (FundBuyer myid) (Bid Goodtillcancelled))))] : (testagent2 mystate args
simstateinfos myid)

>|| THE EXCHANGE AGENT
>exchagent (Emptyagentstate) args ((time, messages, broadcasts) : simstateinfos) myid =
[message (myid,0) [Arg (String (showlob initiallob), 0)], broadcastmessage (myid,0)
(broadcast_numlist (getlobsummary initiallob))] : (exchagent (Exchstate initiallob) args
simstateinfos myid)
>
where
>
initiallob = lob_setsentiment getsent mylob ||primed_emptylob
>
where
>
mylob = emptylob, if (arg_getnum (args!0)) = 0
>
= primed_emptylob, if (arg_getnum (args!0)) = 1
>
= error "exchagent - primary lob not recognised", otherwise
>
getsent = Calm, if (arg_getstr (args!0)) = "Calm"
```

```
>
= Choppy, if (arg_getstr (args!0)) = "Choppy"
>
= Ramp, if (arg_getstr (args!0)) = "Ramp"
>
= Toxic, if (arg_getstr (args!0)) = "Toxic"
>
= error "Unrecognised sentiment.", otherwise
>
>exchagent (Exchstate curlob) args ((time, messages, broadcasts) : simstateinfos) myid
>          = ([datamessage (myid, 0) trdata, message (myid,0) [Arg (String (showlob
nextlob), 0)], broadcastmessage (myid,0) (broadcast_numlist (getlobsummary nextlob))] ++
(tdtomsg trades)) : (exchagent (Exchstate nextlob) args simstateinfos myid), otherwise
>            where
>            tdtomsg [] = []
>            tdtomsg ((o1, o2) : rest) = [(trademessage (myid, order_gettraderidno o1) o1
o2), (trademessage (myid, order_gettraderidno o2) o2 o1)] ++ tdtomsg rest
>            (nextlob, trades, trdata) = (a,lob_gettrades a,lob_gettrace a) ||
(a,b,lob_gettrace a)
>                                        where
>                                        a = foldr match
((lob_increment_time.lob_clear_trades) curlob) orders

>                                    ||   (a,b) = foldr setret
(((lob_increment_time.lob_clear_trades) curlob), []) orders
>                                    ||   setret ordxr (loba, tradesxa) = (lobb,
(lob_gettrades lobb) ++ tradesxa)
>                                    ||                                          where
>                                    ||                                          lobb =
(match ordxr loba)
>                                       orders = (filterorders messages)
>                                                where
>                                                filterorders qx = msg_getorders qx


>traderagent (Emptyagentstate) args any myid = traderagent (Traderstate (0, 0, 0,
getsent, 0, (map (order_setuid "traderagent") [0..]))) args any myid
>                                             where
>                                             getsent = Calm, if (arg_getstr (args!0)) =
"Calm"
>                                                     = Choppy, if (arg_getstr (args!0))
= "Choppy"
>                                                     = Ramp, if (arg_getstr (args!0)) =
"Ramp"
>                                                     = Toxic, if (arg_getstr (args!0))
= "Toxic"
>                                                     = error "Unrecognised sentiment.",
otherwise
>
>traderagent (Traderstate (obb, obo, oon, sent, invent, uids)) args ((time, messages,
broadcasts) : simstateinfos) myid
>     = [] : (traderagent (Traderstate ((xsum!0), (xsum!1), (xsum!8), sent, newinvent,
remaininguids)) args simstateinfos myid), if bequiet = True
>     = (ordstomsg (map fixorder fordxr)) : (traderagent (Traderstate ((xsum!0),
(xsum!1), (xsum!8), sent, newinvent, remaininguids)) args simstateinfos myid), otherwise
>       where
>       bequiet = False, if quiettime = (-1)
>               = True, if quiettime <= time
>               = False, otherwise
>
>       quiettime = (arg_getnum (args!(myid-1)))
>       (remaininguids, fordxr) = traderagent_applyuids uids [ordxr]
>       newinvent = foldr (+) invent (map getmovement mytrades)
>                   where
>                   mytrades = map (cpsafehd "traderagent") (filter (~=[]) (map
msg_gettrade messages))
>                   getmovement o = osize,     if (or [otype = (Bid Goodtillcancelled),
otype = (Buy Andkill)])
>                                 = (- osize) , if (or [otype = (Sell Andkill), otype =
(Offer Goodtillcancelled)])
>                                 = 0, otherwise
>                                   where
>                                   osize = order_getsize o
```

```
>                                   otype = order_gettype o

>
>       ordstomsg o = map (ordermessage (myid, 1)) o
>       ordxr = generic_trader mytraderid randoms gs xsum (obb, obo, oon, sent, time,
newinvent)
>               where
>               mytraderid = (HFT myid), if (arg_getstr (args!(myid-1))) = "HFT" || -2
assuming first trader is at index 2 in agentlist
>                          = (FundBuyer myid), if (arg_getstr (args!(myid-1))) =
"FundBuyer"
>                          = (FundSeller myid), if (arg_getstr (args!(myid-1))) =
"FundSeller"
>                          = (Intermediary myid), if (arg_getstr (args!(myid-1))) =
"Intermediary"
>                          = (Opportunistic myid), if (arg_getstr (args!(myid-1))) =
"Opportunistic"
>                          = (Small myid), if (arg_getstr (args!(myid-1))) = "Small"
>                          = error "traderagent - unrecognised trader category",
otherwise
>       fixorder x = x, if (pr >= (ltp - 12)) & (pr <= (ltp + 12))
>                  = order_setprice (ltp - 12) x, if pr < (ltp - 12)
>                  = order_setprice (ltp + 12) x, if pr > (ltp + 12)
>                    where
>                    pr = order_getprice x
>                    ltp = last xsum
>       xsum = [1993, 2007, 500, 500, 1, 1, 500, 500, 2, 2000], if time <= 0 || rep 9 0
[bb, bo, bsld, ssld, bsls, ssls, bsd, ssd, or]
>       xsum = msg_getnumlistfrombcast (findG0bcast broadcasts), otherwise
>              where
>              findG0bcast [] = error ("No broadcast found?" ++ (show time))
>              findG0bcast (f:r) = f, if (msg_isbroadcast f) & ((msg_getid f) = 0)
>                                = findG0bcast r, otherwise

Generic trader returns a list of list of orders...
My agent needs to pass through a list of list of


Helper functions (for this particular case)
=========================

>      gs = gaussians
>      ||bfilter :: [msg_t] -> [broadcast_t]
>      ||bfilter (() : ys)
>      find x list = realfind x list 0
>                    where
>                    realfind x [] n = error "Couldn't find the item in the list..."
>                    realfind x (f : r) n = n, if f = x
>                    realfind x (f : r) n = realfind x r (n+1), otherwise


>      traderagent_applyuids uids orders = (uids, []), if orders = []
>                                        = (remaininguids, fixedorder : rest), otherwise
>                                          where
>                                          fixedorder = (safehd uids "fixedorder1")
(safehd orders "fixedorder2")
>                                          (remaininguids, rest) = traderagent_applyuids
(tl uids) (tl orders)
>cpsafehd caller x = safehd x caller
>safehd :: [*] -> [char] -> *
>safehd x caller = hd x, if x ~= []
>                = error ("safehd - from " ++ caller), otherwise

>zipfunc x [] = []
>zipfunc [] x = []
>zipfunc (f:rf) (a:ra) = ((f a) : (zipfunc rf ra))

>isOtype typestr or = result
>                     where
>                     result = check (order_gettype or)
>                              where
>                              check (Offer x) = True, if typestr = "Offer"
>                              check (Bid x) = True, if typestr = "Bid"
```

```
>                              check (Sell x) = True, if typestr =  "Sell"
>                              check (Buy x) = True, if typestr = "Buy"
>                              check x = False

>isBuySide order = True, if (or [isOtype "Bid" order, isOtype "Buy" order])
>                 = False, otherwise
>isSellSide order = True, if (or [isOtype "Offer" order, isOtype "Sell" order])
>               = False, otherwise

>si warn list n = list!n, if n < (#list)
>               = error warn, otherwise

>timestep = 0.04 ||This might need to be changed to 0.00425
>stepstoseconds s = s * timestep
>secondstosteps s = s/timestep

>systemTime :: num -> num
>systemTime x
>      = (numval timeinseconds) mod 887
>        where
>        (time, st, ste) = datecall x
>        timeinseconds = time -- "\n"

>datecall x = system ("date '+%s'" ++ (concat (rep (x mod 249) " ")))


>delta onbook wantedonbook = wantedonbook - onbook, if wantedonbook >= onbook
>                          = 0, otherwise

>tan x = (sin x) / (cos x)
>mymax a b = max [a,b]
>mymin a b = min [a,b]
>mysuperor f1 f2 x = or [(f1 x), (f2 x)]

>round x = entier x, if (x - (entier x)) < 0.5
>        = (entier x) + 1, otherwise

>       applyuids uids orders = (uids, []), if orders = []
>                             = (remaininguids, fixedorder : rest), otherwise
>                               where
>                               fixedorder = (safehd uids "fixedorder1") (safehd orders
"fixedorder2")
>                               (remaininguids, rest) = applyuids (tl uids) (tl orders)
```