Development of an Electronic Broker System with Rules Engine using C++ and Python

Palimar Rao
University of Pennsylvania

August 31, 2014

Abstract

I present a project where a fully functional electronic broker system or electronic sell side is developed using C++. In addition to this, a custom rules engine is developed using Python to encode proprietary trading strategies on the sell-side. FIX (*Financial Information eXchange*) 4.2 is the protocol implemented by the broker. The broker can trade a wide variety of securities, including options, bonds, futures and convertibles. The rules engine accepts rules written in a natural language, and applies it to orders that are sent to the electronic broker system. This is done with the help of a custom tokenizer, parser and template engine. The interface to this broker system is command line and an internal webportal with a Django backend. The electronic broker can connect to and accept orders from multiple buy side firms and manages individual orders with the help of a custom built priority queue with a heap backend.

1 Introduction

A broker or a sell side in a firm is responsible for taking in orders for various securities from buy sides of other firms. These orders are either internalized, where the order is satisfied from the portfolio of the broker itself, or sent in smaller quantities to an exchange such as the NYSE or the CME, or a market maker [1] itself. Since Citadel is an options market maker, electronic brokers that can internalize incoming orders from various buy side firms without human intervention can be used to reduce delays and errors while filling orders.

Market makers are firm which buy and sell a particular security, thus making a 'market' for that particular commodity. While market makers internalize, they need to sell securities from their position in a manner that increases profit and generates 'alpha' [2]. This is by itself a type of trading strategy that heavily depends on automatic execution of algorithms accord-

ing to a model of a portfolio manager. The electronic broker and the rules engine were developed to meet this need.

The electronic broker connects to trading software on the buy side such as EOMS (Electronic Order Management System) and receives orders from the system. These orders are run through the rules engine, and, in the vent of no rules present, through a static profile unique to each buy side. Based on the rules executed and the static profile, an order can be fully filled, partially filled, rejected, canceled, busted or modified. These options need to be provided by the broker as per the FIX specification. Once the order execution is decided, an execution report is sent back to the buy side, and verification is performed to ensure that the positions of both the buy side and sell side have reflected the order execution.

Such an electronic broker can simultaneously connect to many electronic buy sides; with the upper limit being around 1000 buy sides. Each buy side can have as many as 100 orders open with the electronic broker at the same time. Finally, an order can be divided into an arbitrary number of fills. An order is divided into fills and executed fill by fill so as to not negatively affect the movement of the market before the order is fully executed. The number of fills is arbitrary and can be one of the variables in a certain model.

2 Overview

When an order is routed to the Electronic Broker, it passes through a number of modules. These modules are independent of the security type. Each module is responsible for ensuring the correct implementation of the specification and operate independently of each other. Therefore any of these modules can be swapped with newer ones without affecting order flow. There are six module in total:

1. **FIX Server**
2. **Execution Engine**
3. **Profile Loader**
4. **Rules Engine**
5. **Messenger**
6. **Priority Queue**

Figure 1 is a representation of how the modules connect with each and what data flows from one module to another.

3 FIX Server

The FIX server is a server that implements the FIX protocol. The server is a daemon that communicates with the sell side via IP sockets. On the opening of a communication stream with a sell side though a pre-defined port, an acknowledgement of the open connection is sent to the sell side firm. A new execution engine is spawned for the sell side on a new thread, and it is

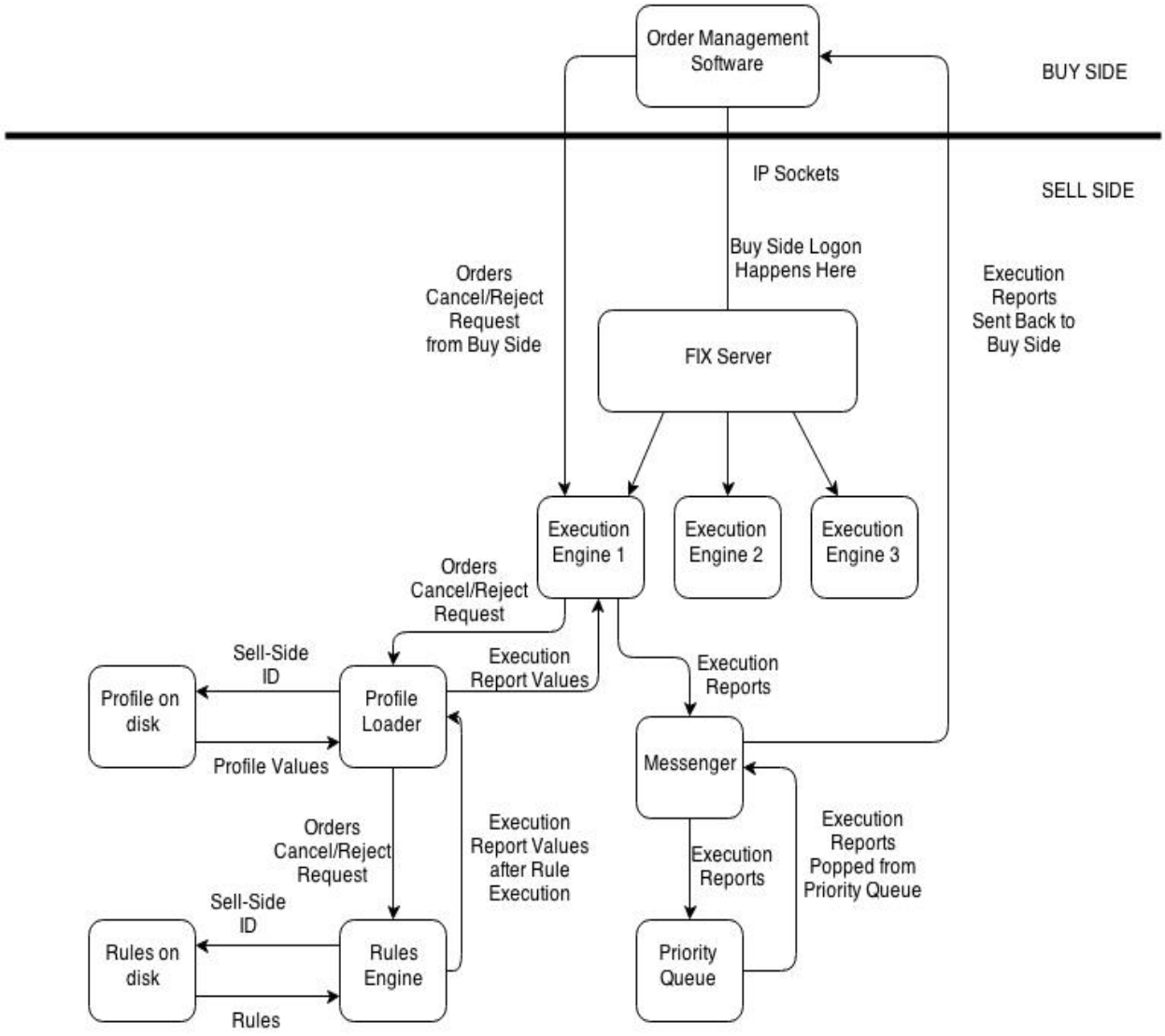

Figure 1: Electronic Broker Architecture

kept alive until the sell-side logs out or the communication link is severed.

4 Execution Engine

The execution engine receives the message from the buy side and processes it. A message can be of three types: buy order request, modify order request and cancel order request. When an order is received, a check is performed to ensure that the security and substantial quantity exists in the position of the portfolio manager to execute that order. After that, the order is sent to a profile loader and the rules engine to calculate values such as execution type, fill time, fill quantity, delay and others. Once these values are calculated, an execution report in accordance with the FIX protocol is constructed and sent to the messenger module with a certain time at which the fill is sent to the buy side. The time is milliseconds past the Unix epoch. In the case of a modify or a cancel request checks are performed to ensure that such a request is feasible. For instance, a fully filled order cannot be canceled. In case of accepting the request, a domain specific message is sent to the messenger module to modify or cancel the order in question. As each buy side is associated with its own execution engine, there is no risk of exposing the position of one firm to another by incorrectly re-routing orders.

5 Profile Loader

The profile loader read a profile unique to a buy side. This human readable profile outlines various values such as rejection rate, cancel rate, fill rate, delays before executing and acknowledging an order, fill limit and more. The order is first sent to the rules engine. In case no rule is executed, these static values are returned to the execution engine. In the case a rule is executed, then the values returned by the rules engine is sent back to the execution engine.

6 Rules Engine

The rules engine is the most complex module in the project, as well as the most critical to correctly define and execute various trading strategies. The rules engine takes in rules which belong to a natural language, as in, the rules are valid statement in English. A rule is of the following form: (Group1, var_{11}, var_{12}, ...) if {Field} satisfies some condition or|and ... else (Group2, var_{21}, var_{22}, ...). This means that if a certain field in the incoming order satisfies a certain set of conditions, then the members of Group1 must assume the values var_{11}, var_{12}, ... else the members of Group2 assume the values var_{21}, var_{22}, ... These members are the same fields found in a static profile (fill time, fill rate, etc.) An example of a rule would be: "(SetFillQty, 200) if Symbol == "MSFT" else (SetFillQty, 100)". In this case, if the order is for Microsoft shares, then the number of fills is set to 200, else

the number of fills is set to 100. Other features such as partial group application and setting variables in rules dynamically from incoming orders are implemented as well. The rules engine has three separate elements to implement its functionality: Tokenizer, Parser and Template Engine.

6.1 Tokenizer

The tokenizer uses a stack to extract tokens from the rules such as group and field names. In case of incorrect group and field names, the rule is silently rejected.

6.2 Parser

The parser has multiple purposes. It uses a regular expression to ensure that the rule is structurally correct, that is, the regular expression denotes the grammar that describes the regular language that all valid rules must belong to. It then builds a dictionary, where the keys are the members of the group returned by the tokenizer, and the values are the ones specified in the rules (var_{11}, var_{12}, ...)

6.3 Template Engine

The template engine uses the dictionary data structure built by the parser and substitutes the values into the rules. After this substitution, the rules become valid python code. This code is then run dynamically using Python's *exec*() command. After the code is run, each member's value in the group is updated to the variables present in the rule. Finally, the members' values are returned by the rules engine to the profile loader, and a -1 value is sent if the value did not change from the execution of the previous rule. This is used to signify that the executed rule did not change the value of the member.

7 Messenger

The messenger module has two functions. It receives an execution report from the execution engine module and sends the execution report to the buy side. When an execution report is sent to the message, the time at which the report is must be sent to the buy side is also included. The messenger then places this execution report in the priority queue module, which is ordered by time. In a separate thread, a function polls the priority queue with a granularity of 25 milliseconds. This function then pops off all the execution reports in the priority queue with a time less than the current time and hands it back to the messenger, as these execution reports are due for sending. The messenger then sends these execution reports to the sell side.

8 Priority Queue

The custom priority queue is written in C++ and is optimized specifically for execution reports. It can contain over a million execution reports at once and has its

own data structure to keep track of the last fill time of an order, canceled fills, modified orders and busted fills. The priority queue never operates at full capacity as each execution engine controls its own priority queue. It runs as two separate threads. One thread is dedicated for priority queue operations such as $push()$, $pop()$ and $find()$, and the other thread updates the data structures within the priority queue. Complexity for $push(), pop()$ and $find()$ is $log\ n$ as is required.

9 Results and Evaluation

The electronic broker and the rules engine were stress tested to emulate real world conditions. The following tables list the various computation times of certain calculations done by the modules. These values were measured using the Linux *time* command.

Table 1: Evaluating FIX Server

Operation(N)	N	Time(ms)
Connect to N Buy Sides	1	< 1
	100	63
	1000	645
Receive N orders simultaneously	1	< 1
	100	18
	1000	194

Table 1 data shows the time taken by the server to accept new connections and receive orders from buy sides. At thousands of new connections, the delay is still less than a second and less than one-fifth of a second with a throughput of 1000 orders. These results are satisfactory, given number of connections is less than 200 at any given time.

Table 2: Evaluating Execution Engine

Operation(N)	N	Time(ms)
Calculate Execution Reports for N orders	1	< 1
	100	< 1
	1000	2

Execution engine compiles the execution report in a very fast manner. Profile loader and rules engine delay is not included.

Table 3: Evaluating Profile Loader

Operation(N)	N	Time(ms)
Fetch static profile for N orders	1	< 1
	100	45
	1000	530

There is some delay as profile loader fetches the reports form disk.

Table 4: Evaluating Rules Engine

Operation(N)	N	Time(ms)
Executing 5 rules for N orders	1	2
	100	230
	1000	2490

Clearly the rules engine is the bottleneck. This is because of two main reasons. For every order a multiple number of rules affecting a multiple number of variables can be executed which quickly makes the complexity of the rules engine's *execute*() method non-linear. Secondly, the rules engine is written in Python, and is therefore not able to fully get the benefit of threading due to GIL (*Global Interpreter Lock*).

Table 5: Evaluating Messenger

Operation(N)	N	Time(ms)
Pushing N Execution Reports into the PQ	1	< 1
	100	< 1
	1000	5
Sending N Execution Reports to buy side	1	< 1
	100	45
	1000	505

Sending execution reports suffers a delay as data is pushed out via sockets. The delay associated with the priority queue is not included.

Table 6: Evaluating the Priority Queue

Operation(N)	N	Time(ms)
Pushing N Execution Reports into the PQ	1	< 1
	100	< 1
	1000	35
Popping N Execution Reports from the PQ	1	< 1
	100	< 1
	1000	29

The priority queue does not add a significant delay when execution reports are in the low thousands. However there can be as much as a thousand execution reports per order as an order can be subdivided into multiple fills. This could significantly affect latency, especially when popping execution reports off of the priority queue, as an execution report needs to be sent at the exact time when it is popped off the priority queue.

10 Conclusions

In this paper, an electronic broker was built from scratch in C++. A rules engine to complement the electronic broker and provide support for implementing trading strategies was written in Python. Heavy testing of the electronic broker was undertaken and it performed according to expectations is normal use cases and adequately in heavy use cases. That being said, there is room for improvement. The rules engine can be re-implemented using the Rete Algorithm [3] to decrease latency. It can be re-implemented in C++ as the Rete algorithm does not require dynamic execution of code. Furth more, some of the modules which do not do a lot of complex work such as the Messenger module can be integrated with other modules to reduce overall complexity.

This system can be a lot of help, whether it be for a huge market maker or for an individual investor looking to directly connect to and trade with sell side firms. It is im-

portant to note that in any of these cases, such a system must be robust and error-free as the potential losses due to software malfunction in electronic trading can be very high, just as the profits achievable by automating trading has virtually no limit.

References

[1] Albert J. Menkveld, High Frequency Trading and the New-Market Makers, 2013

[2] Raj Bector, The Hidden Alpha in Equity Trading, 2013

[3] Charles L. Forgy, Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem, 1981