Order Matching Engine

# Introduction

In electronic trading world, the Order Matching Engines match the buy and sell orders that are submitted by the brokers to electronic trading networks, like NASDAQ.

# System Architecture and Software Design

## Dependency Injection

The 3rd party tool, Unity, developed by Microsoft is used for dependencies management. All of the dependencies and singleton instances are registered in the class **DependencyRegister.cs** within **OrderMatchingEngine.Main**

## Connectivity

The Order Matching Engine opens its gateway using FIX connectivity. QuickFix library is used to manage the FIX protocols as well as the connection infrastructure.

The configuration for the QuickFIX is done in the file FIX.OrderMatchingEngine.cfg within **OrderMatchingEngine.Main**

Default configuration for the port is 5001 for local network. FIX 4.4 is used for this engine.

## **NetworkRequestHandler.cs** is responsible for the FIX gateway.

## Disruptor Pattern

The disruptor pattern is used for the event messaging queue within the engine. Two disruptors for two type of the events, NewOrderEvent and CancelOrderEvent, are created. Therefore the requests for creating new orders and cancel orders will use different ring buffers which related to their event types.

The architecture of the pattern will be multiple producers with a single consumer for each event type. For the case of the NewOrderEvent disruptor, the flow of the events are as follows:

Producers

(FIX)

NewOrder Registrator

NewOrder Matcher

NewOrder Responder

NewOrder Statistic Recorder

FIX Client

## Locking Mechanism

Minimum number of locks are used to minimize the chances of having deadlock as well as to make sure the correctness of the Order Book operations.

For example, in MarketOrderMatchingService.cs

public OrderMatchReport ProcessOrder(Order order, SessionID sessionId)

{

if (order.Quantity <= 0)

return null; // **TODO ErrorReport is needed**

lock (locker)

{

var orderBook = orderBookRepository.GetOrderBook(order);

var report = ExecuteOrderMatching(orderBook, order);

return report;

}

}

The lock is only placed before the data manipulation in the Order Book.

## Order Book Data Structure

Since the Order Book (OB) is the core of the Order Matching Engine, the data structure of the OB has to be carefully designed to give the best performance for its CRUD (Creation, Read, Update, Delete) operations. The data structure of the entities relevant to the Order Book are shown below:

public class Order

{

public long OrderId { get; set; }

public bool IsLimit { get; set; }

public MessageType CommandType { get; set; }

public OrderType OrderType { get; set; }

public Asset Asset { get; set; }

public OrderSide OrderSide { get; set; }

public decimal Price { get; set; }

public decimal Quantity { get; set; }

public decimal OriginalQuantity { get; set; }

public decimal FilledQuantity { get; set; }

public long LastUpdateTime { get; set; }

public string ClOrdId { get; set; }

public TradingAccount Account { get; set; }

public Order NextOrder { get; set; }

public Order PrevOrder { get; set; }

public OrderList OrderList { get; set; }

. . .

}

public class OrderList : IEnumerable<Order>, IEnumerator<Order>

{

public Order HeadOrder { get; private set; }

public Order TailOrder { get; private set; }

public int Length { get; private set; }

public decimal Volume { get; private set; }

private Order last;

. . . .

}

public class OrderTree

{

private readonly SortedDictionary<decimal, OrderList> priceTree =

new SortedDictionary<decimal, OrderList>();

private readonly Dictionary<decimal, OrderList> priceMap =

new Dictionary<decimal, OrderList>();

private readonly Dictionary<long, Order> orderMap =

new Dictionary<long, Order>();

public decimal Volume { get; private set; }

public int OrderSize { get; private set; }

public int Depth { get; private set; }

. . . .

}

public class OrderBook

{

public OrderTree Bids { get; private set; }

public OrderTree Asks { get; private set; }

private long time;

. . .

}

With the above structure, the key operations will be run with good performance for Market and Limit Order:

Add – O(log M) where M is the number of price Limits (which is << N the number of orders)

Cancel – O(1)

Execute – O(1)

GetVolumeAtLimit – O(1)

GetBestBid/Ask – O(1)

# What I did So Far

Due to the time constraint, I managed to

* setup the dependency injection pattern to manage the dependencies of different services and event handlers
* setup the FIX connectivity for the engine’s gateway
* the disruptor for NewOrder events
* the event handlers for the new orders’ operations including the market and limt order matching

The codes are implemented based on SOLID principles to make sure the code base is scalable in the future.

# Future Enhancement

* Add unit tests on all of the features
* Load test on the engine to measure its performance and to find out its limitations
* Implement the event handlers for Cancel/Update Order events
* Add the event sourcing mechanism