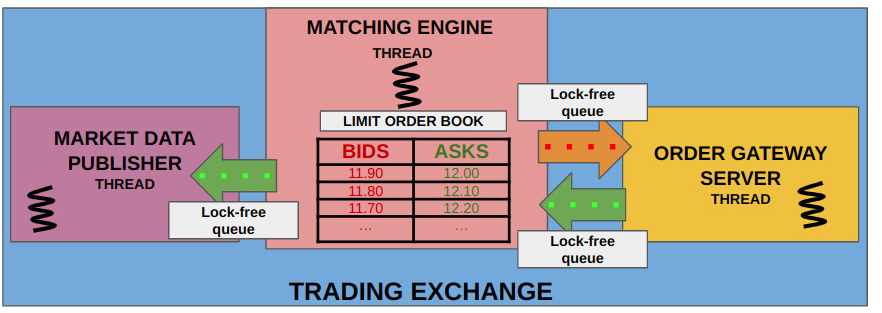
Application Overview :

Bwlow it’s an overview of the architecture. The “Gateway” for now it’s just a UDP communication or FILE reading to pass on the trading data to the matching code.



**Threads :**

The application is using two threads aside from the main thread according to the specifications:

One thread is used for streaming the input data to the main thread. Another thread is used to publish the Ack, Trades and TOB changes from the OrderBook to the console/File.

The module has a ThreadingManager class which has a template createThread method which can be used to create threads for any object.

We also have the thread-safe template queue container implemented in Queue.h. We will use this to transmit data between threads in a safe way.

**Communication :**

The application has a File reading mode. To ensure as little bottleneck as possible the line stream is

being sended to the queue line by line as a string so every line gets processed as soon as it arrives.

Class representation :

The module has an interface (IstreamClient.h) which is implemented in the FileStreamImpl and the possibility for future communication classes. TCP/UDP or others can be easily supported in the future.

**Deserialization :**

Data is being processed line by line and the string is matched by regex validators. If we have a valid match we extract the data by spliting the string of data by ',' and initialize a BookEntry object to be used for orderBook update.

**Communication between Threads :**

So we have in total 3 threads : Main Thread, Input Streaming Thread and Publishing Thread

1) In order to synchronize the data transfer between threads we use the thread-safe blocking queue mechanism explained above which waits in a blocking manner until it has data available.

The main thread uses two Queues (one for processingStreams and one for publishing messages) which are provided as references to the other threads.

2) The input streaming thread is responsible for writing to the processingQueue while the main thread reads from it once data is available and proceed to perform the OrderBook updates.

3) The main thread then is responsible for writing to the publishingQueue once an update is available and the Publishing Thread will proceed to publish the messages by reading the publishingQueue updates.

**OrderBook design :**

As we can have multiple orderBooks by Symbol I chose an unordered map of strings and OrderBooks. [<string, OrderBook\*>]. This way we have an OrderBook object for every Symbol which can be accesed in constant time.

The OrderBook is designed to have bids and asks as multimaps of price and BookEntries which is the BookEntry type [<price, BookEntry>]. The reason for this is that we can have multiple

order entries at the same price so we need to store multiple values for the same map key.

The time complexity in handling insert/erase book entries is O(log n). If we deal with top of book changes the time is O(1) as the multimap is ordered and we can check the price of the topOfBook orders in constant time.

Cancel of orders is of O(n) time complexity.

Space complexity is O(n \* m) where m is the number of OrderBooks and n is the avg of the entries for each OrderBook

**Bids and asks are updated as :**

For cancel : iterate through bids and asks, remove existing orders and send Ack

For new orders : If the new order price matches (>= buy / <= sell) then top of book price or it's a market order we match directly with top of book orders

(if existing, if not we add them or cancel them if market orders) and if there is left quantity we go further and match with the new top of book order.

For flush command we clear the bids and asks.

**Logger :**

Singleton Logger class with options to log to console or file.

**Publisher :**

Simple class which takes the messages received in the publishing queue and pass them to the Logger log method.