About TraderRun

TraderRun introduction
1CDOI is a small proprietary trading firm. It has a desk that engaged in low latency trading of warrants and CBBC in HK market. Warrants and CBBCs are option and barrier option with spot HSI as underlying. There are market makers which provide liquidity in both bid side and ask side, based on Black Scholes model or advanced models like Heston stochastic volatility model or Merton jump model. As a counterparty, we try to hit the bid and ask order only (i.e. fill or kill), we do not speculate on trends of HSI.

2What we do is to predict the bid and ask quote of the market makers, through various regression techniques and pattern recognition on observed derivative prices (warrant) and underlying prices (HSI, HHI, and other stocks). Algorithms used include regression, Kalman filtering, support vector regression. The objective is to predict next warrant prices faster than the market makers do, on given new underlying tick. The overall latency is about 40us (20us in trading program and the rest in broker supply system), which is short enough to beat most market makers.

3We trade around 200 warrants and CBBCs everyday, underlyings include HSI(HSIF) HHI(HHIF) and stocks. TraderRun generates around 50K orders per day, offset all positions by day end, account for 5%-10% of warrant turnover in HK. The average trading profit is about 1-2 million HKD per day during 2011-2013.

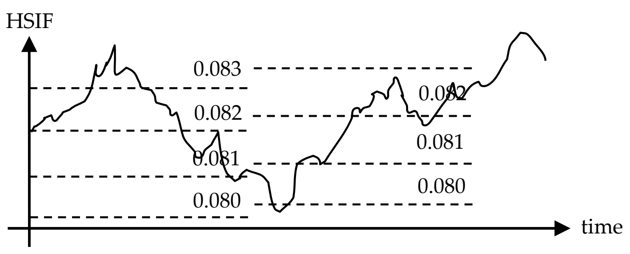
How does TraderRun generate alpha?
1Assume that TraderRun predicts market maker price with 100% accuracy, how can it generate profit without speculating a trend on HSIF? Regardless of how HSIF fluctuates, each change in market maker's bid quote for a HSIF warrant can be considered as a sequence of Bernoulli trials, with probability of going up/down one price level time-dependent.

warrant price changes against time : - - - + + + - - + - + + - - + - - - - + - + + + + - - - + - - - + - -

TradeRun generates :
- buy signal (of warrant) when it predicts a rise + when it is currently – (that is a flip from – to +)
- sell signal (of warrant) when it predicts a fall – when it is currently + (that is a flip from + to –)
- no signal otherwise (there is no flip, which may be + following +, or – following –)

Suppose TraderRun foresees a flip from + to – right after *N* consecutive +, then it can earn *N*-1 price levels by running the above trading logic (minus one is here to account for the bid-ask spread, as TraderRun buys from best ask and sell to best bid). Therefore, the profit of one TraderRun round-trip-trade depends on the number of consecutive +, which follows the geometric distribution. Besides, sequence of alternative +–+–+–+– makes zero profit while wastes transaction cost.

2 $\vec{p}_{opt} = \arg\max_{\vec{p}} h(f(HSIF_{t\in training-example}, \vec{p}), wrnt_{t\in training-example})$ where *f* is regression model, *h* is objective function



- every warrant tick triggers calibration
- every underlying tick triggers prediction
- regression model should handle outlier and regime-switching, such as RanSac and EM-algo

3The drawback of TraderRun is in two folds. Firstly it generates zero profit on encountering a sequence of +-+-+- which results in transaction cost that eats up profits (approximately, one price level profit is consumed through 40 round trips), secondly, warrant is a monopoly market, there is one market maker per warrant, as a result, it can play any tricks to max its profit, such as increasing bid ask spread, shrink in price when open interest increases, decrease in quote size etc.

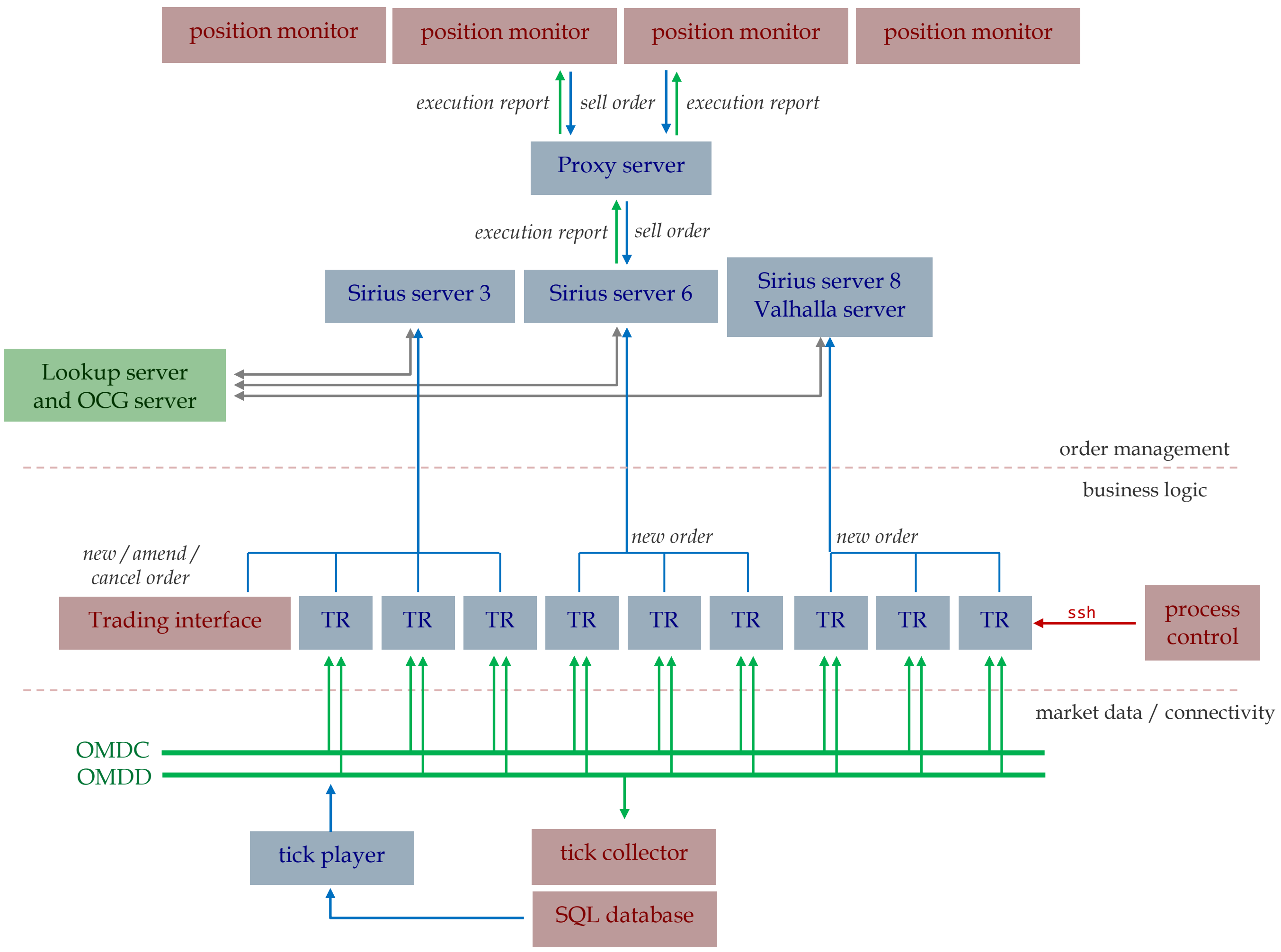
Trading System
There are 8 people in the team, including two traders, three research analysts, one infra-structure engineer specialized in machines, network router & switch, and database admin, one frontend GUI C# developer, one algorithmic C++ developer which is me. Software tools developed by the team are (blue items are my development) :

TraderRun	*(main trading program)*
Sirius	*(broker supply system)*
Valhalla	*(for prediction and direct order placement, run in same machine as Sirius)*
process control	*(for launching programs using ssh)*
position monitor	*(for monitoring orders and position, connected to Sirius)*
trading interface	*(for manual trading)*
tick collector	*(for storing to database)*
tick player	*(for loading from database, and playing tick by tick like a datafeed)*

My contribution is the development of application layer softwares in C++11-14 using STL, boost, involving TCP socket IO, database IO, datafeed parsing, multi-threading, lockfree techniques, maths modelling, numerical methods, linear algebra and regression, order generation and position monitoring.

Various instances forming a DAG

- blue boxes are my development
- green boxes are 3rd party development
- red boxes are my colleagues development

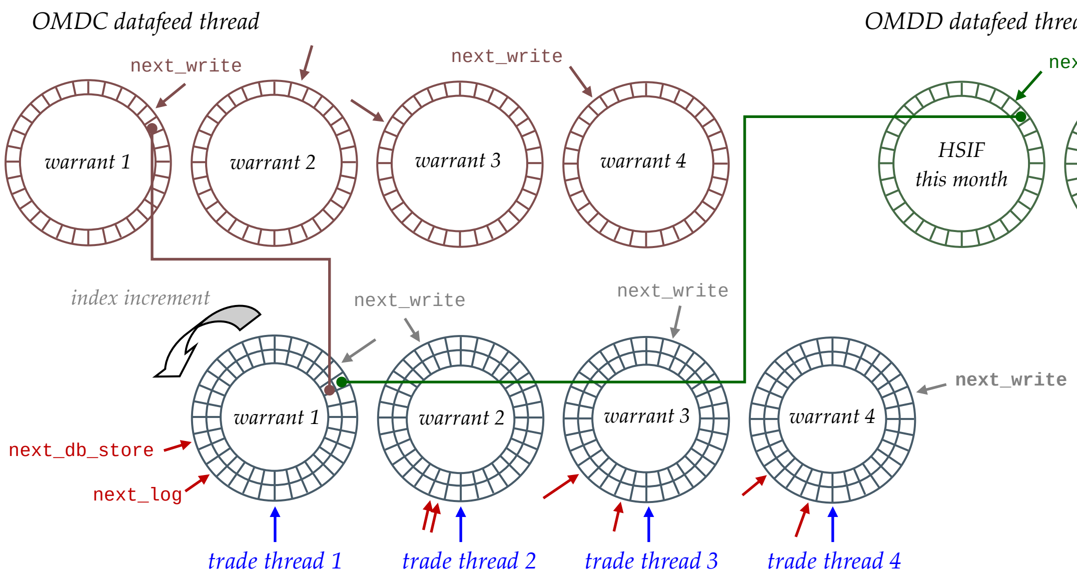


Trading program

Concurrency model in TraderRun is basically producer-consumer model, which is implemented using bipartite graph of lockfree ring buffers. Consider a TraderRun executable that trades *N* warrants, there are *2+N+M* threads in total, two are datafeed threads for *OMDC* and *OMDD* respectively, *N* are trading threads (one for each instrument) and *M* for database read-write logging etc. These threads coordinate via the bipartite graph of ring buffers, one of the partites contains *SPMC* (single producer multiple consumer) lockfree ring buffers, known as datafeed buffers (in which, each entry is a snapshot of orderbook or transaction tick), whereas another partite contains *DPMC* (double producer multiple consumer) lockfree ring buffers, known as trading buffers (in which, each entry is an index-pair pointing to one warrant-datafeed buffer and one HSIF-datafeed buffer). There are *N+K* datafeed buffers and *N* trading buffers, where *K* usually equals to two (one for this month *HSIF* and one for next month *HSIF*). The second partite is considered to be an observer of the first partite, its function is to synchronize warrant data with *HSIF* data. This is very similar to *LMAX disruptor* pattern.

Coordination among various threads :

- *OMDC* datafeed thread async-read from socket and populates *N* warrant-datafeed buffers and trading buffers
- *OMDD* datafeed thread async-read from socket and populates *K HSIF*-datafeed buffers and trading buffers
- each trading thread keeps polling trading buffer for unprocessed tick-pair and proceed to calibration or prediction



```
template<typename T> ringbuf
{
       void push(const T& x) // run by single thread
       {
              values[next_write.load(memory_order_acquire) % 65536] = x;
              next_write.add(+1, memory_order_release);
       }

       T values[65536];
       alignas(64) std::atomic<unsigned long> next_write = 0;
};

template<typename T> reader
{
       reader(ringbuf<T>& buf_) : buf(buf_) {}
       const T& read() const // run by single thread
       {
              while(true)
              {
                     if (next_read.load(memory_order_acquire) < buf.next_write.load(memory_order_acquire))
                     {
                            auto this_read = next_read.fetch_add(+1, memory_order_release);
                            return buf.values[this_read % 65536];
                     }
              }
       }

       ringbuf<T>& buf;
       alignas(64) std::atomic<unsigned long> next_read = 0;
};
```

Trading program - low latency techniques

The system is colocated to HKEX, connecting to :

- *Orion market datafeed for derivative OMDD, PRS*
- *Orion market datafeed for securities OMDD, MDS, EBroker*

The system has 20 machines, each has one Intel cpu, 3.7GHz Xeon, 6 cores, 4 hyperthreads (i.e. 24 threads), each machine trades 10 HSI/HHI warants or CBBCs. The overall latency from the time a triggering tick is received, to the time a buy or sell order is placed is around 40us, measured using TCP dump and wireshark. Here is a rough breakdown :

- *10-20us for datafeed parsing*
- *3-5us for prediction algo*
- *10us for broker supply system*
- *other layers in network stack*

The key to low latency is simple design (labels from *low-latency.doc*) :

`E1.` colocation of data and process
`E2.` linux tuning, such as power management and memory management setting
`D4.` TCP with Nagle algo disabled
`D5.` TCP with async function called
`C1.` lockless ring buffer (single write multi readers) atomic variable and relaxed memory order
`C2.` setting thread affinity, priority and isolation from scheduler
`B1.` cache friendly code :

- ring buffer implemented with contigous vector
- ring buffer implemented with POD tick, which support fast copy by `memcpy`

`B2.` cache line sharing :

- zero padding of tick data
- `alignas(64)` of `next_read` and `next_write`

`B3.` branch prediction :

- remove unnecessary branch
- avoid virtual function, replace by compile time polymorphism

`A1.` object copying

- no redundant copying (socket-read and parse directly to ring buffer)
- avoid `new` operator, smart pointer and node based container

`A2.` price level calculation using integer
`A4.` no file IO, no socket IO, no mutex in trading path (do it in a separate thread)