CLSA

2017 Sep 12

Question 1 – System design for market data distribution system

Consider a market data distributor which receives live market data from the exchange (with a daily-open snapshot and a sequence of delta-update afterwards, for all instruments) and sends to a set of clients (can be algos or live data displays). Once a new client login, it registers a set of stocks that it wants to observe, market data distributor then sends it the latest snapshot and all subsequent delta-updates. Besides there are multiple clients, who login at different time of the day, thus a client may request delta-updates starting from 09:35, while another may request delta-updates starting from 09:40.

Single thread solution

Let's assume that delta-updates rate from the exchange is slow. We can do everything with single thread, the main thread keeps running an io-service, which reads from one exchange socket and multiple client sockets asynchronously. Besides, there is one stock-snapshot map plus one stock-observer map. Callback for client socket async-read is simple, as the only possible message from client is logon request, observer map is updated and all requested snapshots are sent. Callback for exchange socket async-read is to forward the new tick to all registered clients, and update the snapshot of corresponding stock with the delta change.

When tick rate is slow, all incoming ticks can be processed instantly (accumulated to the corresponding orderbook), there is no need to construct a buffer (or lockless ring buffer) for unprocessed ticks. However, when delta update rate from the exchange becomes extremely fast it is possible that *(1)* there are new delta ticks coming in during snapshot-updating and *(2)* there are new delta ticks coming in during the transmission of snapshot to clients, therefore we need a multithreading design (it is better to be lockless) with low latency synchronization mechanism such as polling, using producer consumer model (or even the *LMAX disruptor* pattern) as shown below.

Low latency solution

Suppose there are 4 threads (each can be replaced by a thread pool if that thread becomes a bottleneck for the system) :

- exhange-thread — which async-reads from exchange
- builder-thread — which runs orderbook-builder
- transmit-thread — which runs tick-transmitter
- client-thread — which async-reads from multiple clients

Here are the required data structures :

- tick-buffer — one for each stock, it is a *SPMC* lockless ring buffer of delta-ticks
- orderbook-builder — one for each stock, it is a *SPMC* lockless ring buffer of snapshot with size equals to 2
- tick-transmitter — one for each stock, it is a client map `std::map<client, latest_transmitted_tick>`

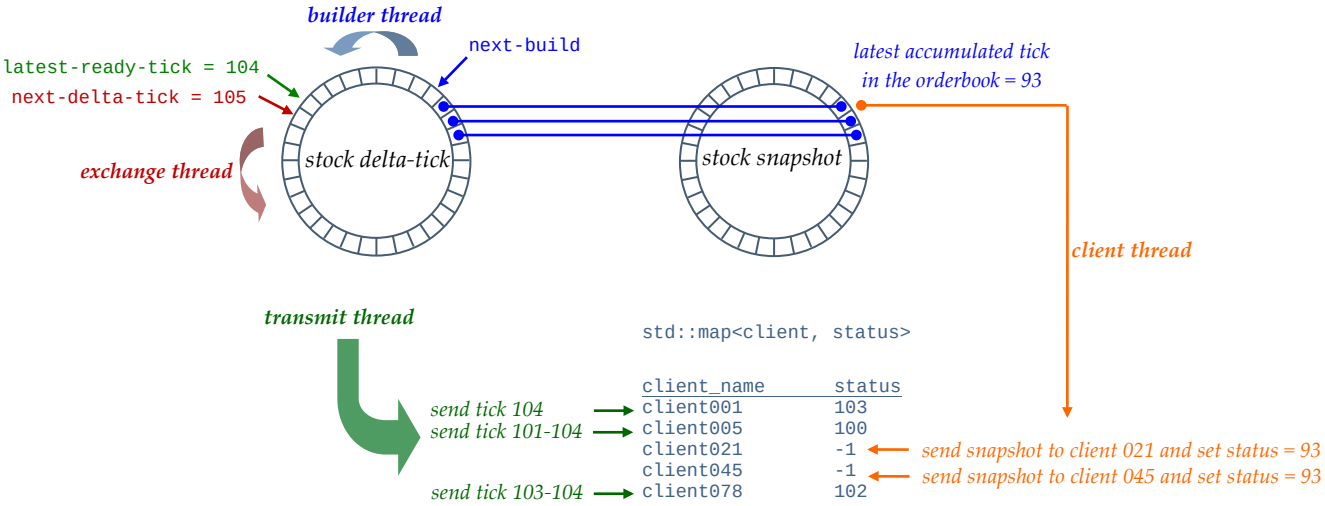
Exchange thread reads asynchronously from the exchange socket and pushes delta-ticks into tick buffer (which is lockless), its first read for each stock (from the exchange) must be a snapshot which is used to initialize the orderbook-builder, thus this thread has to deal with both tick-buffer and orderbook-builder, yet it does not need to communicate with any client.

Orderbook-builder contains a *SPMC* lockless ring buffer of snapshots, it is a specialized ring buffer, as it always consumes the latest-built-snapshot, hence it has a smaller size than the tick-buffer. Builder-thread is the producer of this specialized ring buffer, while client-thread is the consumer. Builder-thread loops through each tick-buffer by polling, whenever there are unprocessed delta-ticks, it accumulates to the cooresponding orderbook-builder, and that's it.

Tick-transmitter contains a map of target clients, together with client-status, client status is simply an integer denoting the latest transmitted tick to a client, it is negative (say *-1*) if initial snapshot transmission is not done. This client-map should be protected by a read-write-lock, which allows concurrent read but exclusive write, as it is modified by client-thread, as clients keep logging in *(insert)* and out *(erase)*. Transmit-thread loops through each tick-buffer, tick-transmitter by polling, send all un-transmitted ticks to clients having non-negative status (clients having negative status is waiting for snapshots instead of delta-ticks). For example, for a particular stock, this is what the transmit-thread does :

```
auto latest_tick = tick_buffer.lates_tick();
for(auto x : tick_transmitter.client_map)
{
      if (x.second >= 0)
      {
            for(int n=x.second; n!=latest_tick; ++n) tcp_send(x.first, tick_buffer[n]);
            x.second++; // What happen if client logout here? Replace read-write-lock by mutex?
      }
}
```

Client thread reads asynchronously from client socket, when there are clients logging in and out, it updates the client-map inside tick-transmitter, transmits the latest snapshot to login-clients, then updates client-status, thus the client-thread is a consumer of the specialized ring buffer inside orderbook-builder, yet client-thread does not transmit delta-ticks to clients. For each tick-buffer, there is one producer (exchange-thread) and two consumers (builder-thread, which runs orderbook-builder, and transmit-thread, which runs tick-transmitter). All consumer indices are contained inside tick-buffer, so don't place it inside consumer classes. Here is a diagram illustrating the design (for one stock only) :



Question 2 – Realtime update the median of a sequence of integers

Given a live-stream of integers (or doubles), find the runtime median of the integer stream, i.e. we keep median updated as integers are entered into the algorithm. Here is my answer : construct a binary tree, such that all nodes in LHS subtree are smaller than the root, while all nodes in RHS subtree are greater than the root, runtime median can be obtained as the root in O(1) if the tree is completed (filled in all non-bottom layers). Therefore, the question becomes finding median in a balanced-but-incomplete binary tree, answer to this question is to augment each node in the tree with a number denoting the size of subtree starting from the node, this number is updated during node insertion / deletion / rotation. As a result, runtime median can be found by looking for a node having subtree size equals to tree size divided by 2, using an O(logN) tree transverse from the root. Interviewer threw me another question : what happens when there are duplicated integers? If we adopt either always-insert-duplicate-to-LHS-subtree or always-insert-duplicate-to-RHS-subtree convention, then it may end up with a non-balanced tree, if we adopt insert-duplicate-randomly approach, then it may violate binary search tree property (some nodes in LHS subtree may be greater than nodes in RHS subtree). A proper solution to duplicates is to augment each node with the number of duplicates, this approach makes insertion /deletion of duplicates very easy.

Please review the following operations of a binary tree :

- node insertion
- node deletion
- rotation

Alternative solution

A smarter solution to runtime-median is to make use of two heaps instead of one binary tree. This approach maintains a runtime median, a max-heap storing all numbers smaller than the runtime median and a min-heap storing all numbers greater than the runtime median. Suppose max-heap and min-heap are equally-sized, let's see how the two heaps can be kept balanced as new numbers are pushed (or popped). When new numbers enters, insert it into max-heap if it is smaller than the runtime median, insert it into min-heap otherwise, if the two heaps are not balanced (differ in size by more than one), they can be balanced by popping from the larger heap, replacing the runtime median, pushing the old median into the smaller heap. Runtime median can then be read in O(1).

Another alternative solution

If the numbers are integers, we can further simplify the algorithm by histogram. Each bin in the histogram is augmented with a number indicating the order of the first entry of the bin. There is a median-pointer, pointing to the bin containing the median, this pointer is updated (by shifting forward or backward) whenever new numbers are pushed (or popped).