How to measure and reduce latency in FIX Antenna HFT

- Overview
- Latency measurement
    - Message sending
    - Message receiving
    - Message passing through different HFT Antenna components
- Affinity control
- Thread safety
- Latency reduction

Overview

This article is intended to describe the measurement of the latency of the message passing through the different HFT Antenna components, sending and receiving of the FIX message and ways to reduce the latency occurred.

Latency measurement

Message sending

After the message is sent e.g. by invoking the **session.put()** method, the **msg.getNanosTimestamp()** method is called to retrieve the sending timestamp. This timestamp is taken right before sending the message buffer to the socket.

If the thread is trying to send the message into the socket while another thread is doing the same, HFT Antenna will put the data in the internal in-memory queue. Messages that are getting into the queue will not have timestamps assigned. To filter those out, reset the timestamp before calling the**put()** method using **msg.setNanosTimestamp(0).**

If the message gets queued, the **session->put(&msg)** method will return much quicker as it will not invoke the socket's send call (which takes about 5-10 usec with standard linux sockets and 2-3 usec with kernel bypass like openonload).

This feature can be disabled by session or message settings.

Message receiving

The method **msg.getNanosTimestamp()** is used to get a message arrival timestamp. This timestamp is taken while getting the new data from socket and right before starting to parse it. The **get_nanosec()** function from HFT FA API is used to get timestamps and measure latencies.

Message passing through different HFT Antenna components

It is analogous to measuring synchronous **send()** calls.

- Timestamp 1 - is gotten right before the **session.put()** method is called
- Timestamp 2 - is gotten right before the **send()** call on the socket (**msg.getNanosTimestamp()**)
- Timestamp 3 - is gotten right after the **send()** call is completed on the socket layer

The following piece of code can be considered for timestamp taking:

```
bool processMsg(Engine::MsgPipeElem* elem, Parser::LiteFixMessage& msg)
{
.........
              uint64_t t1 = msg.getNanosTimestamp();  // message ingress timestamp
             msg.setNanosTimestamp( 0 );

              session->put( &msg );
             uint64_t t3 = get_nanosec(); // send complete timestamp

             uint64_t t2 = msg.getNanosTimestamp();  // serialization+persistience complete timestamp
                                                   // there could be 0 we set above if the message wasn't sent
to the socket synchronously
```

Affinity control

If the affinity is set explicitly for each socket thread, it is possible to have some control over the socket affinity for acceptor sessions. Two listen sockets and two reading threads are allocated. All inbound connections that end up at a particular listen socket will be served by the same thread. **TCPDispatcher.IncomingConnectionOnloadStackAffinity** enables this behavior, otherwise, the sockets are spread in round-robin fashion among the reading threads.

So here all connections at port 8200 will be served by thread #1 and the connections at port 8201 will served by thread #2:

TCPDispatcher.NumberOfWorkers = 2
TCPDispatcher.IncomingConnectionOnloadStackAffinity = true
ListenPort = 8200, 8201

There’s no similar affinity setting for the outgoing connections, they are always assigned to threads in round-robin way.

It is recommended to allocate 2-3 shared cores for **TASK_EXECUTER, GENERIC, DISPATCHER_THREAD, Logger** (these threads do non-latency critical tasks such as heartbeats, connect/disconnect, FIX resends) and use dedicated cores for **TCP_READER**(s), 1 core per thread. **FILE_CACHE_SYNC** thread works best when assigned to 1 dedicated core on a separate CPU node, while all the other threads reside on another node.
If there is more than one processor socket on the machine, it is better to have the file sync thread run on a separate CPU node .

See this page for the reference.

Thread safety

1. It is threadsafe to send messages from multiple threads to the same session simultaneously. The engine also attempts to gather messages from multiple threads if they happen to call **put()** simultaneously in order to form larger buffers for transmission (e.g. it allows to pack more messages into a single TCP packet).
2. The incoming messages thread safety
   It is guaranteed that for one instance the session engine invokes callbacks of the **Engine::Application** every moment from only one thread. That is why the process method synchronization is not needed except the case when multiple sessions invoke callbacks of the single application.
3. FIX messages thread safety
   Dealing with one instance of the FIX message (reading, writing message fields, and sending message into the session) simultaneously from different threads is not threadsafe.

Latency reduction

1. The usage of the Solarflare OpenOnload is recommended to achieve better latency results. To use it, it is needed to switch a couple of settings in the engine.properties file and run.sh.

   If Onload is used and spinning in Onload is on then EF_poll timeout should be non-zero but less than spinning time.
   E.g. if **EF_POLL_USEC=600000** then **TCPDispatcher.Timeout** should be 500.

   TCPDispatcher.IncomingConnectionOnloadStackAffinity = true.
   If true then all inbound connections that end up at a particular listen socket will be served by the same thread.
   If false sockets are spread in round-robin fashion among the reading threads like for outbound connections.

2. Setting the following parameters is recommended for better performance given that user wants to persist messages:

   a. extraParams.disableTCPBuffer_ = true;

   b. extraParams.useNewTCPDispatcher_ = true; c. extraParams.socketPriority_ = Engine::EVEN_SOCKET_OP_PRIORITY;
   d. extraParams.storageType_ = Engine::persistentCached_storageType;

The following engine.properties setting: **TCPDispatcher.Timeout = 0** (also available via API) suites to enable spinning in the socket reading thread. This is recommended for OS sockets use.

3. If pre-populated messages are used for all the sending there are a couple of tips to promote faster handling of pre-populated messages so that they can be released to the wire as quick as possible.

    1. You can use zero-padded numbers to reserve the space in the message buffer for numeric tags, e.g. **msg->set(34, "000000000")**;
    2. Alternatively, **msg->setTagLength(34, 9)** call can be used, it does the same job as the call above;
    3. Finally, when all tags are pre-populated you can use the call **msg.prepareContinuousBuffer()** which would re-serialize the message buffer internally to make it ready to send;
    4. You can still do some adjustments to the message before sending it out, like changing 38 or other tags (and changing the integer value tags would be the cheapest operation - changing the string value tags that may change the pre-populated tag's length and cause a couple of extra steps before sending).