Threading Model

- Product
- Market Data Nexus
- CEP
- Smart Order Routing
- Strategy Engines
- DARE
- Photon
- System Administration
- Database Access Layer and Persistence
- Threading Model
- High Availability and Failover

THREADING MODEL

Threading Model

The Marketcetera Automated Trading Platform is highly multi-threaded. Marketcetera can be configured to take full advantage of all available CPU resources to improve efficiency when processing large volumes of executions. A running instance may use tens of threads to over one hundred at any given time. Most of these threads are doing tasks unrelated to order and execution processing, like, answering UI requests, checking incoming executions for cumulative quantity errors, managing FIX session queues, etc. These threads are fixed in quantity and nature and cannot be affected by configuration.

There are a group of threads that can be controlled via configuration. These threads belong to two thread pools that are responsible for managing outgoing orders and incoming executions, respectively. The size of the thread pool controls how many simultaneous orders or executions can be processed at once. The higher the number, the more CPU resources can be dedicated to processing orders and executions. Too high a number and the OS spends all its time swapping threads in and out, a state called “thrashing”. The ideal configuration is a function of the number and speed of available cores on the machine, the additional load on the machine, the distribution of executions relative to orders, and other factors. Since executions for a given order must be processed serially, the best possible distribution would have a single execution each for thousands of orders as opposed to thousands of executions for a single order.

The DARE FIX engine receives an execution and hands it to a thread dedicated to processing messages for that FIX session. The session queue determines what order the execution belongs to. If an Order Queue is already dedicated to processing that order, the execution is appended to the collection of executions that Order Queue is currently processing. If an Order Queue is not already dedicated to processing that order, an Order Queue is allocated from the execution thread pool. If an Order Queue is not available because they are all in use, the Session Queue will block until an Order Queue becomes available. After an Order Queue is done processing, the Order Queue remains dedicated to that order for a configurable period. Setting this value to a high value will make executions for a particular order “sticky”, that is, better able to process executions that are separated by a few seconds. Setting this value to a low amount will better optimize throughput of executions from many different orders.

If more executions are available to be processed than there are available Order Queues to process them, a message will be written to the log indicating that throughput could be faster if more processing capability were dedicated to processing executions.