<https://www.reddit.com/r/quant/comments/kmdnty/open_source_hft_market_maker_bot_for_crypto/>

Avellaneda-Stoikov is a good model

Here's a structured list of questions based on your design:

1. **Market Data & Order Request Listener:**
   * How will you handle market data and order requests that overlap between historical and live data streams? Will they need different processing methods?
   * Ans – the request.h will have its specific classifications according what the request is, and then it will be sent thru its specific topics on solace, topics that the event listners are subscribed to. For example livedatareqlistner will listen to marketdata/request/live and respond to only those, and handle them in synch or asynch (in this case asynch). The event listener will then route those requests accordingly, so they will choose the adapter and process the request, so yes, the event listeners will of have different processing methods just like what the OMS is doing which will also need an event listener.
   * Do you plan to implement different data parsing rules at the adapter level for historical vs. live data, or should the parser be adaptive to both?
   * Ans – yes so currently I have a separate parser class, but I will migrate this to be on the adapter level instead, as each adapter will need their own parsing rules, the current parser class is tuned for alpaca adapter, which will be migrated to inside the adapter and will be deleted later.
2. **Event Listener & Parallel Processing:**
   * Could you clarify how you'll balance threading for the historical and live event listeners? Will they each maintain independent thread pools?
   * Both eventlistners need to be separated since they are processed differently, synch and asynch so yes they have to handle their thread pools independatly, but inside the same docker container.
   * Would you require a configurable thread pool size or priority for different tasks, especially for parallel processing of strategies?
   * Ans – I think Kubernetes can handle this, or advise me on anything I shld consider
3. **DTOs for Standardization:**
   * You mentioned creating a request.h DTO for standardization. Would you like a similar structure for responses, or do you prefer parsing directly within each class?
   * Ans – we should probably make a response DTO, but the parsing will remain within the individual adapter. So the data will be parsed, placed into the response DTO and sent to the strategy
4. **Order Management System (OMS):**
   * Will the OMS manage a specific order lifecycle, and what states do you envision (e.g., Submitted, Partially Filled, Completed, Rejected)?
   * – we can decide this later, but good job considering now
   * Are there specific risk checks you’d like in place before order execution? Should these be customizable?
   * Ans - Yes, we might make a risk builder class, which will be static methods
   * How would you like the system to handle rejected orders? Should they trigger any specific notifications or retries?
   * Ans – don’t try any retries, we can decide this later
5. **Strategy Class & Execution:**
   * Will each strategy run independently, or would you want to support strategies sharing resources or signals?
   * Ans – yes we need sharing of resources since we will do some internal trade crossing so we need to keep track of the portfolio and signals across the strategies, which the oms or the strategy class can do or we can make a separate docker package for it (because then we can implement a kill switch in case any docker container goes down)
   * How would you like to handle strategy-specific state? Should state data be stored within each strategy, or would you prefer a centralized cache (Redis)?
   * Ans - advise me on which wld be better, since I will be handling large volumes high frequency market data , and if two strategies are using the same asset data, then there is a duplication if there are state caches instead of centralized.
   * Should the system support strategy priority or latency requirements, especially in cases where certain strategies require faster execution?
   * Ans – this can be decided later
6. **Caching & Database Integration:**
   * Could you clarify the purpose of Redis here? Do you envision it as a temporary in-memory cache, and InfluxDB as the long-term storage?
   * Ans – what u stated is what I want
   * Do you need real-time access to the cache data for strategy decisions, or is Redis simply for periodic storage before moving to InfluxDB?
   * Ans – I need it for real time access to the strartegies, since I will have moving avereges and other rolling indicators
   * Do you have specific requirements for schema or structure within InfluxDB?
   * Ans – haven’t made anything yet
7. **Solace Messaging Layer:**
   * What type of message formats and protocols does Solace require? Would you need JSON encoding, or are there other serialization standards?
   * Ans – I wanna focus on json, we can explore later
   * For Solace connections, should each Docker container open a new session, or should multiple containers share sessions to optimize resource use?
   * Ans – I want one session per docker package running
8. **Logging & Monitoring:**
   * What level of logging granularity do you require? Should the system log each data request and response, or only specific events?
   * Ans – it shld log data and order requests, live data streaming start and end, running portfolio pnl risk etc.
   * Would you prefer logging in a centralized logging system, or should each component/container maintain its own logs?
   * Ans - centralized logging, which will monitor all connections going through solace
9. **Error Handling and Failover:**
   * How will you handle failed connections, especially in cases of network or server outages?
   * Ans assuming Kubernetes can be setup accordingly

**AI prompt for context**

I am developing a high-frequency trading (HFT) algo trading software in C++, intended to operate within a containerized architecture, with each module independently deployed in Docker containers. I’m building this to support live and historical data processing, order management, real-time strategy execution, and centralized logging. Here are the specifications, design goals, and areas that may require additional guidance. All relevant code is included.

**1. Core Components and Structure**

* **Generic Adapter**:
  + An abstract base class designed to support various adapter implementations.
  + Each adapter will independently handle market data inflow or order execution (broker communication), or both, with the flexibility to manage different broker APIs.
  + Each adapter processes and parses data directly within its own scope (parser logic is specific to the adapter, e.g., AlpacaAdapter).
* **Event Listeners (Market Data & Order Requests)**:
  + Two primary event listeners, each running in separate threads but within the same Docker container:
    - **Live Event Listener**: Listens asynchronously on the marketdata/request/live Solace topic.
    - **Historical Event Listener**: Processes requests synchronously on the marketdata/request/historical Solace topic.
  + Both listeners utilize unique thread pools managed independently within the Docker container.
  + Each listener routes requests via Solace to the appropriate adapter, processes the data according to request type, and sends responses back in JSON format.
* **Order Management System (OMS)**:
  + OMS is responsible for managing orders throughout their lifecycle, from submission to execution, including risk checks.
  + Orders are standardized using a request.h DTO, with an optional response DTO under consideration.
  + It uses adapters to route orders based on broker and manages state across all orders, providing lifecycle tracking (Submitted, Partially Filled, Completed, Rejected, etc.).
  + A static RiskBuilder class is planned for performing configurable risk checks before executing orders.
  + Orders that fail risk checks are logged but are not retried; error handling here remains under consideration.
* **Strategy Class**:
  + The Strategy class standardizes how trading strategies are built, enabling them to:
    - Request live data asynchronously and historical data synchronously.
    - Place orders to different brokers through the OMS.
    - Run independently on separate threads, leveraging the ability to share resources.
  + Strategies will manage their own state but can access shared resources (for cross-strategy signals and portfolio tracking) with the option for a centralized cache solution (like Redis).
  + Each strategy caches its own data temporarily in Redis and periodically writes to a long-term storage database (e.g., InfluxDB), approximately every 4 hours.
  + Builder classes, containing pricing algorithms and technical indicators, are directly accessible within strategies to allow rapid integration.
* **Solace Messaging Layer**:
  + Solace is used to facilitate inter-module communication across Docker containers.
  + Each Docker container has its own independent Solace session, initialized at startup.
  + All communications use JSON for consistency, with potential for further serialization formats in the future.
  + Solace maintains topics for marketdata/request/live, marketdata/request/historical, and specific order topics for the OMS to listen for and process requests.

**2. Logging, Monitoring, and Error Handling**

* **Centralized Logging**:
  + Centralized logging is used for tracking all incoming and outgoing data requests, live streaming data events, order activities, and OMS actions like PnL, risk, and capital usage.
  + Monitoring all Solace connections and logging connections, activities, and errors through Kubernetes is assumed.
  + All logs are forwarded to a centralized logging system to maintain a single source of truth.
* **Error Handling & Failover**:
  + For failed connections or outages, Kubernetes is assumed to manage failover and scaling.
  + Connection failures in adapters or the Solace session are handled gracefully, ensuring minimal impact on running strategies.
  + Future consideration for enhanced error handling and failover mechanisms within Solace sessions and adapter connections is open.

**3. Performance & Resource Management**

* **Concurrency & Parallel Processing**:
  + Both historical and live event listeners manage their own thread pools for handling requests.
  + Strategy threads run in parallel, managed via Kubernetes, which can handle scaling and prioritization requirements.
  + High priority may be given to strategies requiring low latency or high-frequency data updates, though this is open to further definition.
* **Cache and Database Integration**:
  + Redis is employed as a real-time, in-memory cache accessible by strategies for rapid access to time-series data (moving averages, rolling indicators, etc.).
  + Redis serves as a temporary data store before batch updates to InfluxDB every 4 hours.
  + InfluxDB will store all historical and processed data but currently has no defined schema; design suggestions are welcome for handling high-frequency market data storage.

**4. Outstanding Considerations & Questions**

* **Resource Sharing for Strategies**:
  + Strategies may need to share signals and data, especially when performing internal trade crossing.
  + Advice on whether to centralize state data in Redis versus individual state caches within each strategy is needed.
* **Schema & Structure of InfluxDB**:
  + InfluxDB's schema for high-frequency data storage, including handling historical data and calculations, remains undefined.
* **Risk Management & OMS Handling**:
  + Lifecycle states for orders in the OMS (Submitted, Filled, Rejected, etc.) are under consideration, along with error handling protocols.
  + The static RiskBuilder class is intended for risk checks, but customizable risk handling may be required.
* **Performance Optimization & Threading**:
  + Thread pool configurations may be managed by Kubernetes, but optimal settings for high-volume data handling are open for exploration

**My personal prompt**

i will explain my current design that i want to achieve and provide the code i have until now, make a list of doubts to be cleared, which i will answer. dont only focus on the examples i give, imagine yourself as a solution architect and think of more things i wld require. i am building a hft algo trading software in c++ the code so far is uploaded i have a generic adapter, which is an abstract class to future adapters, which can either handle market data inflow or order execution( broker) or both, and it has been made dynamic enuf for the adapters. i intend to make a market data/order request listener and run this package as its own docker container. i need to run historical and live eventlistners and run them in parallel in sep threads. i also need to make a DTO like order.h as stated in the code to standardise the market request, and have them be attached to all classes that need it (example request.h). parsing of data (marketdataparser) has to be done at the adapter level before being sent. next, i have the order management system (OMS), which has the order specific commands and picks which adapter to use as seen in the code. it uses the order.h DTO to standarise the order format which can be read accross the system. i also need an event listener here, and i need to run this whole package as a seperate docker container too. i will need to do some order lifecycle management, and some basic stats analysis like pnl, currently open trades, risk, etc next is the strategy class. this class is for standardising the format of how strategies are written. so future strategies like for example a pairs trading strategy will be called PTS and they will inherit Strategy class and request for live and historical data, which will be in asynch and synch connections respectively, and be able to send orders to available brokers which is in asynch. now i shld be able to run these strategies that inherit the strartegy class in parellel on seperate threads. each of these strategies will have their own cache (or advise if a central cache is recommended) which will then dump its cache contents into a db (im thinking influxdb) every 4 hrs or so, which i intend to use redis for (the chaching). the strategy class will also have the request.h DTO connected, along with some builder classes full of business logic like pricing algos, technical indicators, which are immeditely available to the strategies. these should also run as a docker package. finally there is the solace layer that connects everything to each other. it is given by the solacelib file, and each docker container will initiate its own session with the running solace server which i will activate, by using the methods in the solace lib as a constructor. i will provide a list of connections to be made by each package in the future. so here is an example flow. PTS requests for historical data. the request is crafted using the request.h DTO format as is heard by the intended event listener thru a topic in solace. it then chooses the appropriate adapter and send the request, recieves the parsed data and sends it back thru solace. this part is synch. then it asks for some live data. the request is sent thru solace and heard by the appropriate listener. it checks if the live data isnt already being published on any topic. if it is, it returns the appropriate topic, if not, it initiates a live websocket connection, as demonstrated in the alpaca adapter code provided, and starts publishing the data thru a newly formed topic, and returns that topic name. this process is asynch. then the strategy uses some of the buikder metods directly provided, then starts sending orders thru solace using request.h and order.h and then the oms starts executing or blocking the orders based on a series of risk criteria. this has to be in low latency and asynchronously. all of these sections shld log what they are doing, so for example the market data package logs the incoming requests, executed or not, the data itself thats coming thru like when it started and ended, all orders and executions times, all decisions made by strategies and running pnl and risk taken and capital used etc

for /r %f in (\*.cpp \*.h) do type "%f" >> combined\_files.txt