**Simulator Architeture**

**The goals of the simulator:**

1. quickly simulating trading strategies over tick data (second by second data, or similar intraday data)

2. accurately simulate fill/miss of limit orders and market orders

3. quickly generating synchronized research data for other research purposes

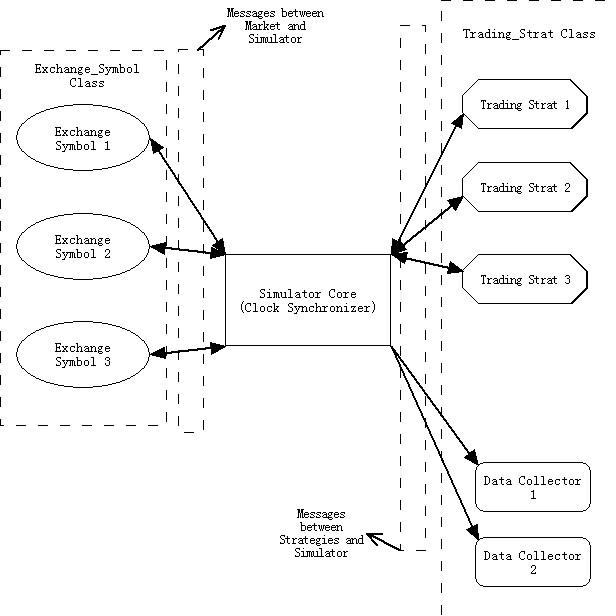
**The design considerations of the simulator**

1. it needs to be fast, (thus in C++)

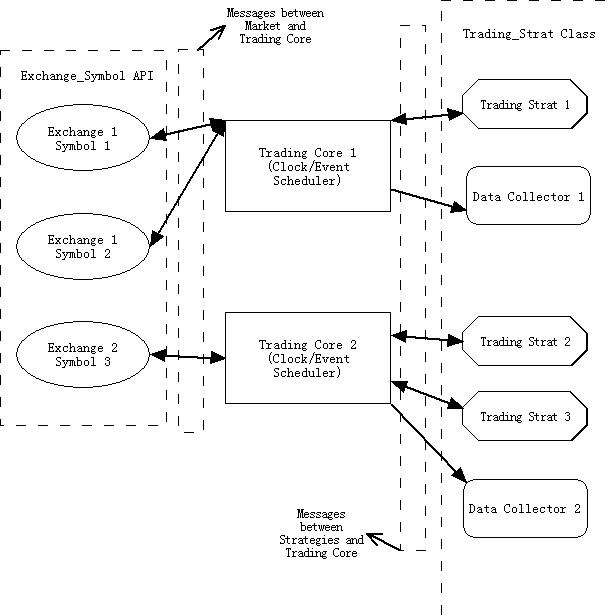
2. it needs to be flexible, (flexible latency assumptions, flexible matching assumptions)

3. it needs to be reusable, (majority of the code should be applicable to production trading)

**The simulator design chart**



**The production trading design chart** (hypothesized, need more detail on production api, vendor architecture etc.)



**The simulator components explained**

There are three major components of the simulator: Exchange\_Symbol abstract class, Simulator Core, Trading\_Strat abstract class.

The Trading\_Strat class has the following functionalities:

1. it listens to messages from Simulator Core (market update, fill messages)

2. it starts computation whenever the Simulator Core flushes the queue of messages to the strategy

3. it sends back messages to simulator Core (order messages, cancel messages)

4. it keeps an internal state of order book for the orders it has sent out.

5. it has the capability to log the messages and computations

6. it calculates current pnl/positions and has the capability to log the pnl/position numbers in file

7. it can be used to collect data (for example a subclass that just received book updates and don’t send out messages).

8. the class should be an abstract class implementing some common features of a strategy; each actual trading strategy will be a sub-class of it.

The Exchange\_Symbol class has the following functionalities:

1. it loads market data at initilization

2. it listens to messages from Simulator Core (order messages, cancel messages)

3. it starts updating market and simulate matching events whenever the Simulator Core flushes the queue of messages to the exchange\_symbol object (based on the timestamps of markets and timestamps of messages)

4. it stops updating market if the next time stamp in data is greater than current timestamp in Simulator Core.

5. it sends back messages to simulator Core (book updates, fill information, canceled information)

6. it keeps an internal state of order book for the orders it has received.

7. the class should be an abstract class implementing some common features of an exchange\_symbol behavior, each actual exchange\_symbol will be a sub-class of it (implementing various fill assumptions and data loading methods, for example)

The Simulator Core has the following functionalities:

1. it runs an internal (pre-scheduled) clock

2. it keeps a list of strategy objects

3. it keeps a list of exchange\_symbol objects

4. as the clock move on to each tick, it flushes messages to the strategy objects, get new messages from the strategy objects, flushes the new messages to the exchange\_symbol objects, and get new messages from the exchange\_symbol objects.

5. it adds latency numbers to each message

The Messages objects/classes

1. each message should have the basic information (exchange, symbol, a sequence number)

2. Message types: from exchange\_symbol to strategy: Fills, Canceled, Book Updates

3. Message types: from strategy to exchange\_symbol: LimitOrder, MarketOrder, Cancel

4. Message should be time stamped, latency number will be added to each message to simulate latency scenarios.

The OrderBook class:

1. it is a utility class which will be used in the strategy class and the exchange\_symbol class

2. it keeps track of the internal queue-spot of limit orders of the same price

**The production components explained**

Most of the production components are similar to the simulation chart.

1. the trading core is replacing the simulator core.

2. the trading core will more likely to be one trading core per each exchange

3. the exchange\_symbol class will be replaced by exchange API

4. the messages between exchange API and trading core will be replaced by interpreter of internal message types to API calls

5. depending on whether exchange API is pull-based or push-based, the clock/event scheduler in the trading core will need to be re-implemented from simulator core. If pull-based, the change is likely to be minor. if push-based, we can still use the simulator core scheduler but an event based scheduler would likely to be better.