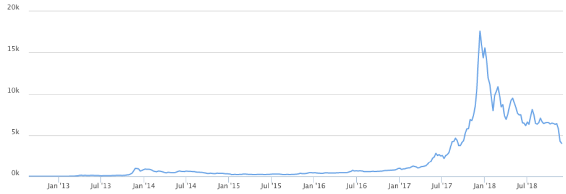
Excel@FIT 2019
http://excel.fit.vutbr.cz

Distributed system for algorithmic trading

Michal Hornický



Abstract
The success of cryptocurrencies like Bitcoin has created many new opportunities. One of them came somewhere around the year 2012-2013, in a form of an online cryptocurrency exchange. Since then, many new online exchanges were created. These exchanges provide unprecedented ease of use and access to everyone, contrasting existing financial exchanges. Day-trading*on these exchanges is easy, and has a large potential because of the extreme volatility of these new markets. This paper outlines the design and implementation of a distributed system, that would facilitate this task. The goals, which include ease of use for new users, scalability for large number of users, and customization for advanced users, combined with problem domain pose interesting requirements, which influenced the design and implementation.

Keywords: Automated trading — Distributed systems — Rust

Supplementary Material: Github Repository

*xhorni14@fit.vutbr.cz, *Faculty of Information Technology, Brno University of Technology*

1. Introduction

[Motivation] The innovations in financial sector, mainly cryptocurrencies like Bitcoin, have created new opportunities. One of them, is the arrival of multiple online exchanges, that focus on cryptocurrency trading . These exchanges have very low barrier to entry, and can be used for easy day trading. Goal of this project is creation of a website, that would allow automatization of day trading on these exchanges.

[Problem definition] In order to effectively support multiple users, the designed system must be able to seamlessly scale according to computing load. It must must allow its users to easily create multiple trading strategies, and execute trades on cryptocurrency exchanges.

[Existing solutions] Most attempts to fail at one or more of the requirements. The older solutions are mostly command-line applications that require complicated installations, and hardware that must be continuously managed. One of these is the **Gekko**[1] trading bot. Main drawback of this solution is the use of JavaScript as the implementation language, and the requirement of Node.js.

Other cloud based solutions, that are more closely related to our approach are usually overly complex, requiring user to write complex strategies that must decide not only when to execute trades, but specifics of these trades. One of these systems is **CryptoTrader**[2], that uses CoffeeScript as a language for implementing user strategies. This system supports multiple assets within one strategy, making them extremely expressive. However, the drawback of this approach is the increased complexity.

[Our solution] our solution aims to surpass other automated trading systems in several aspects. Thanks to the decision to implement the system as a web ap-

[1]https://gekko.wizb.it/
[2]https://cryptotrader.org/

plication, we remove all local software requirements, making the system very approachable. Thanks to the distributed architecture, the system will have the necessary degree of scalability.

[Contributions] Implemented system is built upon scalable architecture that utilizes cloud environment, is able to scale from single to thousands of users seamlessly, It also allows sub-second latency between receiving of new financial information and possible execution of actual trades on real exchange. Implemented system currently utilizes only one exchange, however support of additional exchanges should be extremely easy.

2. Theoretical background

In order to understand automated trading systems, we must first understand how, modern exchanges operate. The core concept of an exchange is the price discovery mechanism. In short, this means, that the exchange does not determine the price of an asset, but rather the price is "discovered" by interactions of individual actors on the exchange. In simplistic terms, this corresponds to supply-demand market mechanism. When the supply of an asset is larger than demand, the price falls, and when the demand rises, the price rises accordingly.

2.1 Automated trading

Today, most of the trading performed even on conventional exchanges is done by automated systems. Origins of these systems can be traced to the 1980s, but probably the biggest milestone was when IBM in 2001[1] experimented with automated trading, and implemented system consistently outperformed even professional traders.

2.2 High frequency trading

Modern incarnation of high-end automated trading systems is called High Frequency Trading(HFT). These systems are commonly co-located with the exchanges, aiming for lowest possible latency between receiving financial data, and execution of market orders. We can divide them into several groups based on the decision process used for creating market orders.

We will focus on **Tick-data market making** strategies. These strategies that utilize periodic information the about price of an asset in order to determine short and long term trends of this price. Based on the short and long term trends these strategies forecast the price into the future.

The benefit of this approach is mainly simplicity. These strategies do not have to rely on complex data describing real world events, that might influence price of an asset(eg. company mergers), and they do not have to perform actual trades explicitly.

2.3 Computing environment

While strategies outlined earlier are easy to implement, they require non-trivial amount of computing power. Coupled with the the need to support multiple users, the requirements for computing power needed to run this system grow.

In order to provide this amount of computing power, we decided to design and implement the system using distributed architecture. This means, that the system is written in way, that allows its individual components to operate separately, and be deployed on different machines. To achieve this goal, we have chosen to use Cloud computing approach[3] on the deployment side. And utilize Actor Model as the core paradigm on implementation side.

2.4 Actor model

Actor model is a conceptual model of describing concurrent computation[2]. Each actor can: Create new actors, send messages, modify its state and decide how to respond to received messages. Primary constraint of this model is the restriction of modifying application state. Each actor can modify its local state however it wants, but can only affect other actors by sending messages.

2.5 Rust & Actix

Due to the choice of the Actor model as a core paradigm, the choice of possible implementation languages was limited. The chosen language would have support this programming model (either implicitly, or through the use of a library). Other considerations included were the runtime overhead, safety, ease of integration with other technologies. Evaluated languages and frameworks include: C# with Akka.NET, Erlang, Java with Akka, and Rust with Actix.

Rust with the Actix library was chosen mainly due to extremely low overhead of this programming language (not requiring a VM), ease of integration with other technologies (LUA), and due the authors personal interest in this language, and corresponding library.

3. Designed system

Probably the biggest obstacle to the implementation of the system was its distributed nature. On the implementation side, this meant the use of Actor model, as

[3] https://en.wikipedia.org/wiki/Cloud_computing

a core architectural paradigm. The use of the Actix library has simplified many challenges with the use of this computing paradigm, but it also came with some drawbacks. The library allows seamless use of actors within single or multi-threaded environments, supporting use of single or multiple concurrent threads, but it does not contain an implementation of primitives that would allow actors to communicate between processes or even different machines.

3.1 Communication

Therefore, part of this project was the design, and implementation of this capability. We have designed and implemented the **actix-zmq** library, that provides actors for communication over the ZeroMQ networking technology, and **actix-comm** library that provides abstractions for implementing simple Request-Reply services, Publish-subscribe pipelines, and other supplementary components (eg. Load balancing broker for services). The **actix-comm** library builds on top of **actix-zmq**, and both of them should be usable in other projects, and will be published as separate libraries, that should enrich already rich ecosystem around the **Actix** library.

3.2 Deployment

Since we are using Cloud environment as our primary deployment target, this side of the system also had to be adapted. We decided to use **Kubernetes** as a primary tool for managing our deployments.

Kubernetes is an orchestration tool, used for automated deployment and management of distributed systems running in the cloud environment. Kubernetes defines a set of primitives, which are used to describe a distributed system. The kubernetes runtime then dynamically modifies state of the system, to conform to described model. The kubernetes runtime runs on a Cluster. A cluster is comprised of multiple virtual machines(Nodes), and can dynamically scale number of used nodes.

3.3 Actual system

The actual system is then designed as a set of loosely coupled components. Each component is comprised of several kubernetes Pods, managed by a Deployment, and exposed by a Service. Within each pod, there might be multiple containers, but most of them only use single one.

Here are components that that describe our system in simplest terms

- Exchange - provides interface to a specific exchange, currently only the Bitfinex exchange,
- Core - Receives updates from exchanges, Decides when to evaluate strategies against this data, and forwards trading decisions to individual exchanges.
- Eval - Evaluates strategies using multiple load-balanced workers
- Web - Provides web interface for user interaction
- Storage - Stores financial and user data.

4. Implementation

As mentioned earlier, the implementation was performed using the Rust language on top of Actix actors as a basic architectural blocks. It is currently divided into 2 executables, The `web` executable houses the user interface implemented using actix-web as a back-end, and LitElement[4] based web application as front-end.

The second executable is the `trader` application. This is implemented as a command line application, that contains the implementations of several different components, and should be split into separate executable for each component in the future.

4.1 Data flow

The whole system is best described by the type of data it consumes, and how this data flows throughout it. Primary data sources are individual exchanges, and the web application. The web application only communicates with the database, and thus is not that interesting in this aspect.

However, the exchanges are more interesting. Most cryptocurrency exchanges provide REST API used for executing trades, and WebSocket endpoint, that is used for providing latest financial data. For each exchange supported by the system there is a dedicated component, that serves as an adapter to this exchange. Main purpose of an exchange adapter is translation system requests into a specific exchange API requests, and forwarding the updates received over WebSocket to core system component.

The individual exchange adapters each connect using **PUB** ZeroMQ socket to core service. This forms a Fan-In topology, that would be difficult to implement using other technologies.

The data flowing from exchanges is in the form of per–minute OHLC[5] data. This is then processed by the core component, which removes duplicate entries, computes data points for different time-scales, and publishes them along with received updates. During this step, the data is also stored into persistent stor-

[4] https://lit-element.polymer-project.org/
[5] https://en.wikipedia.org/wiki/Open-high-low-close_chart

age, which currently takes the form of a PostgreSQL database.

The decision actor in core component periodically loads the information about assignment of strategies to individual assets. Whenever it receives new OHLC data it determines which strategies should be evaluated, and sends this information to the evaluation component, which is implemented as a load-balancing broker, with multiple workers.

Whenever an evaluation worker receives an evaluation request, it retrieves the strategy text, and historical data from the database. It then creates a new Lua VM, configures it (eg. disabling file access), and provides a suite of analytical functions, that can be used by individual strategies. Then it loads the strategy script into this VM, and executes it.

```
// Simple moving average
local sma = ta.sma(10)
// Exponential moving average
local em = ta.ema(10)

// Short term > long term
if sma() > ema() then
    return "short"
else
    return "long"
end
```

Figure 1. Example strategy

The output of the strategy is the desired market position - ”long” or ”short”, the former denoting ownership of target asset and the latter denoting the ownership of the exchange currency, eg. US Dollars.

This work does not focus upon the individual strategies, or methodologies behind them, it only provides basic building blocks for creating them. One of future enhancements might the support of more advanced types of strategies.

Then upon receiving the results of strategy evaluation, the core component checks whether there is trading account information associated with the asset. If there is, it then sends a request to an exchange adapter, which then might check the current market position of the user, and possibly execute one or more trades, ensuring that the requested market position is achieved on this trading account.

5. Conclusions

[Paper Summary] This paper outlined the conceptual idea behind the project, the issues encountered and how they influenced the design and an actual implementation of the system. The system is implemented as a distributed application, with focus on scalability, and is accessible using a web application, satisfying the usability requirements.

[Highlights of Results] The implemented system currently supports single exchange, and over 200 different asset pairs. Each of these asset pairs can have a single LUA strategy, and single trading account associated with it. The system supports executing large number of strategies, with sub-second latency between updates from an exchange, and execution of trades on these exchange. Compared to command-line application solutions, our system can support arbitrary number of strategies, with possible future improvement of tracking individual strategy performance. Compared to other cloud based solutions, our system provides extremely easy strategy implementations

[Paper Contributions] Achievement of these goals was mainly possible due to distributed approach. However, this approach brought its own set of complications, which required solutions. These solutions were implemented in support libraries `actix-zmq` and `actix-comm`, that should be useful in other projects with similar goals.

[Future Work] While functional, the system lacks several pieces of advanced functionality (eg. More complex strategies or testing strategies on historical data). The system should be also extended, to support multiple exchanges. In addition to extending the actual system, the support libraries mentioned earlier are also good targets for future development.

Acknowledgements

I would like to thank my supervisor RNDr. Marek Rychlý Ph.D. for his help, and valuable advice regarding this project provided during frequent meetings.

References

[1] Gerald Tesauro and Rajarshi Das. High-performance bidding agents for the continuous double auction. In *Proceedings of the 3rd ACM Conference on Electronic Commerce*, EC ’01, pages 206–209, New York, NY, USA, 2001. ACM.

[2] Carl Hewitt. Actor model for discretionary, adaptive concurrency. *CoRR*, abs/1008.1459, 2010.