BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INFORMATION SYSTEMS
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

DESIGN AND IMPLEMENTATION OF DISTRIBUTED SYSTEM FOR ALGORITHMIC TRADING

NÁVRH A IMPLEMENTACE DISTRIBUOVANÉHO SYSTÉMU PRO ALGORITMICKÉ OBCHODOVÁNÍ

MASTER'S THESIS
DIPLOMOVÁ PRÁCE

AUTHOR Bc. MICHAL HORNICKÝ
AUTOR PRÁCE

SUPERVISOR RNDr. MAREK RYCHLÝ, Ph.D.
VEDOUCÍ PRÁCE

BRNO 2018

Abstract

Innovation in financial markets provides new opportunities. Usage of algorithmic trading is a perfect way to capitalize on them. This thesis deals with design and development of a system that would allow its users to create their own trading strategies and apply them on real financial markets. The emphasis is put on designing a scalable and reliable system using cloud computing technologies.

Abstrakt

Inovácia na finančných trhoch poskytuje nové príležitosti. Algoritmické obchodovoanie je vhodný spôsob využitia týchto príležitostí. Táto práca sa zaoberá návrhom a implementáciou systému, ktorý by dovoľoval svojím uživateľom vytvárať vlastné obchodovacie stratégie, a pomocou nich obchodovať na burzách. Práca kladie dôraz na návrh distribuovaného systému, ktorý bude škálovatelný, pomocou technológií cloud computingu.

Keywords

Trading, Algorithmic trading, Cloud, Distributed system, Rust, Kubernetes,

Klíčová slova

Obchodovanie,Burza,Distribuovaný systém, Rust, Kubernetes

Reference

HORNICKÝ, Michal. *Design and Implementation of Distributed System for Algorithmic Trading.* Brno, 2018. Master's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor RNDr. Marek Rychlý, Ph.D.

Design and Implementation of Distributed System for Algorithmic Trading

Declaration

Hereby I declare that this term project was prepared as an original author's work under the supervision of RNDr. Marek Rychlý, Ph.D. All the relevant information sources, which were used during preparation of this thesis, are properly cited and included in the list of references.

. . . .
Michal Hornický
April 29, 2019

Acknowledgements

I would like to thank RNDr. Marek Rychlý, Ph.D, the supervisor of this thesis, for extremely valuable feedback provided during consultations

Contents

Abstract

Chapter 1

Introduction

Financial markets are complex systems, in which, market players interact with each other to determine price of an asset. Advances in financial technologies, like the advent of blockchain technology, and corresponding proliferation of cryptoccurencies , like Bitcoin[?] have changed nature of trading.

As a result of these advances, financial markets are now more approachable than ever, and thus present a significant opportunity. One example of services that successfully exploit this opportunity are cryptocurrency exchanges. They are a whole new kind of marketplace, that provides several advantages to its users. These exchanges usually provide approachable Web based user interface for everyone and, HTTP/WebSocket API for advanced users.

In order to capitalize on these advances, we must use advanced trading techniques. One of these is algorithmic trading. Basis of algorithmic trading, is utilization of some kind of algorithm, along with market data, in order to determine most profitable actions, that should be performed on the market.

This approach, has several requirements. One of them is large amount of computing power, since used algorithms might be extremely complex. Latency is also a big concern, since this space is extremely competitive, and a party, which is able to perform optimal actions sooner than all other parties, will net a larger profit. Thanks to these requirements, usage of this technique is not easy, or cheap.

However, advances in development and usage of distributed systems, might be an easy solution to these problems. Cloud computing[?] is now more widespread, and easy to use than ever. Thanks to new technologies like docker[1] and kubernetes[?], the creation and management of distributed systems is easy, and systems created with these technologies can be easily secured, are scalable and provide other benefits for developers creating them compared to more monolithic architectures.

1.1 Objectives

This thesis is concerned with creation of a system for algorithmic trading. This system was concieved as a distributed application. Usage of distributed was chosen in order to minimize cost of approach should help with performance requirements, and the difficulty of implementing such complex system. The system should be designed with latest technological advances in mind, and should utilize cloud computing environment.

[1]https://www.docker.com/

The system itself should be extensible and scalable. The extendability requirement deals with ability to integrate new markets, with types of assets, or add new functionality to existing ones. The scalability of the system deals primarily with the system's ability to automatically scale based on amount of users and resulting load on the system.

From users' perspective, the system should be a easy to use web application. The user should be able to define custom algorithms and strategies, and apply them to different markets.

Chapter 2

Current state & existing solutions

Existing solutions for algorithmic trading that are aimed to regular users instead of specialized investment companies have been available for some time. These solutions range from simple command-line applications that connect to single exchange to large distributed deployments with web interface that connect to largest stock exchanges[**?**].

2.1 Examples

We chose to look at a few solutions from different part of this spectrum in order to better understand the requirements that will be placed upon the designed system. All evaluated solutions perform algorithmic trading of cryptocurrencies. While restricting our research to this small part of global markets might affect our findings, the primary market in which the designed system will operate also is a cryuptocurrency market.

2.1.1 Gekko

On the lower end of the spectrum, there is a simple application written in javascript called Gekko[1]. This application is open source and runs on top of the Node.js. The application can import historical data, and use this historical data to backtest[[**Footnote**]] created strategies. The strategies are written in Javascript, with the support of a simple library that contains implementations of financial indicators, that are commonly used with these types of trading strategies.

It also provides simple web interface, but can only connect to one exchange at a time, and only supports one user at a time.

Therefore, it lacks the scalability of a distributed approach.

2.1.2 CryptoTrader

On the higher end of the scale spectrum, we have CryptoTrader[2]. This solution is implemented as a web application, that supports multiple users at the same time. Each user can define multiple strategies, and each strategy can utilize multiple data sources. The strategies are written in language called CoffeeScript, with slightly inconvenient but very powerful API. This system serves as a good benchmark for our system.

[1]https://gekko.wizb.it/

[2]https://cryptotrader.org/

Chapter 3

Theory

This chapter describes theoretical approach to different parts of target system.

3.1 Trading & Exchanges

In order to define algorithmic trading, we must first define what trading is, and how it is performed. Trading is performed on exchanges. Key aspect of exchange trading is the price discovery mechanism. For all assets, traded on an exchange, the price is not dictated by any single party. Instead, the price is „discovered“ by interaction of buyers and sellers. Buyers advertise the highest price they are willing to pay for an asset, and sellers advertise the lowest price they are willing to accept. These 2 prices correspond to basic economic principle of supply and demand. When there are more sellers active on the exchange, the price will fall, since there are isn't enough buyers to buy an asset. This principle also applies in reverse, if there are more buyers active on the market, the price will rise. The maximum price listed by a buyers known as **bid** price, and the minimal price listed by a seller is known as **ask** price

Since these exchanges are dynamic environments, with always fluctuating pressures on either side, the price of an asset, varies over a time. The amount of this variance is called **volatility**.

Historically, the exchanges were physical, mainly used for trading stocks, and were called stock exchanges. They were physical locations , where individual traders met, and traded one asset for another. Primarily, these trades consisted of stocks or goods against money. Another type of trade is when 2 parties trade one currency for another. Exchanges specializing in these types of trades are called **FOREX** (Foreign exchange) markets.

Most recent of exchange types, is the cryptocurrency exchange. These exchanges are almost always purely virtual. All trading is performed via web interface. The main advantage for our purposes is the ease of use of these exchanges, and their modern features. Virtually all of them provide multiple APIs for different purposes. A real-time API for low-latency streaming of updates to clients, and a REST API provided for executing trades on the exchange.

3.1.1 Algorithmic trading

First financial markets with electronic execution and connection to communication networks appeared in late 1980s and 1990s. This allowed some degree of automation, but weren't yet used for fully automated trading. In 2001 a paper published by IBM[**?**], encouraged adoption of algorithmic trading. In this paper, fully automated trading strategies consistently outperformed human counterparts. Since then, the amount of trading performed by automated systems has steadily risen.

As algorithmic trading became more common, new trading strategies started popping up, and an arms race was started. In this arms race, the parties were consistently introducing new, more effective ways of performing trading decisions, and executing resulting trades. The **HFT**(High frequency trading) is a culmination of the automated trading arms race.

High frequency trading

This form of trading is characterized by high turnover and order-to-trade ratios(number of created orders compared to executed trades). It utilizes highly specialized order types, co-location of trading equipment as close as possible to exchange. In 2010, only 2% of US based trading firms specialized in HFT, but these 2% accounted for more than 73% of all trading volume[**?**].

There are four key categories of HFT strategies[**?**]:

- Order-flow based market making - Utilizes data about amount & volume of newly created orders to determine state of the market & then creates orders on a regular basis to capture bid-ask spread
- Tick data based market making - Utilizes tick data (current bid & ask prices) to determine state of the market creates orders on a regular basis to capture bid-ask spread
- Event arbitrage - Utilizes external information, about events that might affect the market to create specialized orders to profit from this event (Company mergers)
- Statistical arbitrage - Utilizes multiple asset classes, to create complex transaction chains, which allow for relatively risk free profit

Our system will mainly support strategies, that would fall into **Tick data market making** category. This is due to simplicity of these strategies, and the fact that the information provided by the cryptocurrency exchanges is best suited for these strategies. However, we should clarify, that this system does not aim to achieve extremely low latencies. The general goal is to achieve latency of one second. This latency is measured between the moment the system receives update from an exchange to a moment at which the system starts executing API calls related to order creation on said exchange.

The strategies will be implemented in a generic way. That means that an individual strategy will not be referencing any particular asset, but will be working with a general representation of financial data within the system. The selection of an asset, to which a particular strategy should be applied will be performed by users. The system will allow application of a strategy to multiple assets.

The users' capability to use multiple strategies on multiple assets, and the multi-user nature of the system will require large degree of scalability in different parts of the system. To satisfy these constraints, it will have to be of distributed nature.

3.2 Distributed systems

Distributed systems are systems, that are comprised of many loosely coupled components. These components might be threads in single process, processes on single computer, or multiple computers connected through shared memory or a network. These components communicate by utilizing shared memory, or by passing messages to one another. Components interact with one another in order to achieve shared goal. Distributed systems have several key properties[**?**]:

- Concurrency - The computation in one component is concurrent with computations performed by other components
- No global clock - There is no single global clock, each component has only local clock
- Independent failures - Failure of one component does not imply failure of other components

We can use these properties to make a very loose definition of what a distributed system is. In order to better understand these types of systems, we will have to analyze several additional properties.

3.2.1 Additional properties

We can analyze whether a distributed system uses homogeneous or heterogeneous components. The systems that only use homogeneous components are commonly used in open environments. Systems like BitTorrent or similar file distribution software are the perfect example.

In order to find an example of heterogeneous system , we don't have to look further than World Wide Web. In this system, we have servers and clients, which are 2 different types of system components.

Another property of distributed system is the communication method. There are 2 primary approaches to communication between 2 components. Message passing or shared memory.

Message passing is used more commonly, since it allows lower degree of coupling between components, allowing them to communicate over any communication channel. We can simulate former approach with the latter and vice versa at the cost of performance. Going further, this thesis will only deal with systems that use message passing as the communication paradigm.

Another aspect of component communication is the communication protocol. This does not mean the underlying technology that is used to send messages, but rather the protocol that determines what messages will be sent, and when.

Components might communicate using simple request - reply based protocol , like HTTP. Or they might communicate using slightly different publish - subscribe model over technologies like ZeroMQ, or message buses like Kafka.

We can also examine the rigidity of the system. The number and types of components can be either dynamic, or static. This property influences the ability of a system to independently resolve local failures (self-healing systems).

Designed system properties

Using these properties, we settled on a model of a distributed system, that utilizes large amount of heterogeneous components. Each of these components communicates primarily using request-reply style of communication utilizing message passing paradigm. This model is called the Actor model.

3.3 Actor model

Actor model is a conceptual model of describing concurrent computation. It treats Actors as primitives of concurrent computation. Each actor can: Create new actors, send messages, modify its state and decide, how to respond to received messages. Primary constraint 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.

Thanks to this property of isolation, there are no necessary locks to ensure memory safety. It originated in 1973, and has been used for understanding distributed computation, and also as a basis of several implementations of concurrent systems.

According to Hewitt[**?**], the actor model is based on physics. This contrasts other computational models, which are most commonly based on mathematical logic, set theory or similar concepts. The primary takeaway from physics , that can be observed in actor model, is taken from quantum physics, and it is the idea of uncertainty. We cannot observe precise state of a whole system, because attempting to do so will affect it, and therefore invalidate measured results.

3.3.1 Alternative models

Actor model is very high level, and shares both goals and properties with other programming paradigms. These include:

OOP

If we consider Smalltalk, and its message passing model of object oriented programming, we observe several common properties.

- Encapsulation - Both actors and objects can only directly manipulate their local state
- Message passing - Both actors and objects can send messages to other actors and objects respectively
- Polymorphism - Both actors and objects can decide, how will they respond to specific message

While these models are similar, Smalltalk was tied to particular implementation, and it did not provide tools for concurrent programming. But the similarity nonetheless still stands, and actor model can be also understood as an extension of OOP paradigm.

Petri nets

Petri nets have been widely used to model concurrent computation. However, while they are extremely well suited for modeling control flow, They can't be used to model data flow in their basic form. Another problem is simultaneous action. While we can easily simulate simultaneous action of removing a marker from one place, performing a transition and placing a marker in output place, in reality, these 3 actions will not be simultaneous.

Communicating sequential processes

While CSP model has similar goals to actor model, there are several ways, in which these 2 models differ. Most important difference is that Actor model is inherently dynamic, while CSP model if based on a fixed number of sequential processes communicating in a fixed topology[?]. Usage of this model therefore is not particularly suited for our purposes, since designed system should be dynamically scalable, which is not possible in CSP model.

3.3.2 Implementations

While the actor model is almost 4 decades old, implementations of concurrent and distributed systems that are based on this model are more common than ever. One of the oldest implementations of the actor model is in the **Erlang** language. This language was originally developed for telecommunications, with the goal put upon high-availability. The language was created in 1986, originally was implemented in Prolog, and thus extremely slow. But in 1995 it gained custom VM (BEAM VM), and Ericcson deployed more than a million lines of Erlang code in production.

But the Erlang is not the only implementation of this paradigm. There are several libraries implementing Actor model in wide variety of languages, ranging from Lisp to C++. Today, one of the most commonly used libraries is Akka[?], which was originally developed for Java and Scala, but was reimplemented using C# for the .NET platform. While Akka is extremely easy to use, extensible, and performant, it is still only implemented for managed languages, which require heavy runtime with tracing garbage collection.

This was one of the aspects , which influenced the decision to look for another library, that would be implemented in non-managed language. The library, we chose is called Actix[?], and is implemented in Rust[?] programming language.

3.4 Rust

Rust is a new programming language developed by Mozilla. It was created as a response to many shortcomings of existing low level languages such as C and C++. While these languages have crucial place in programming landscape, providing the highest performance and degree of control over hardware, they are outdated, unergonomic and unsafe (particularly with respect to concurrency)[[Cite]]. On opposite side of this equation , are managed languages, that are highly ergonomic, and seem to contain most innovation in this space.

Rust language aims to position itself among the low level languages, bringing new and exciting features to this space. It was originally developed by Graydon Hoare[?] while working at Mozilla, and was based on ML. Probably the most important step, was adoption of the language by Mozilla for the purpose of creating new browser engine called Servo[?]. Goal of Servo was to experiment and innovate in the web browser space, without the depending on over 30 years of legacy code, that was Firefox.

3.4.1 Language basics

Basic concepts of the Rust language are very similar to other C based languages. Basic structure is denoted by braces, It has functions, a module system, and other features, that will be uninteresting to intermediate programmer. However, some of the more advanced concepts make this language particularly well suited for large concurrent systems, and will be explored later in this chapter.

Language has 2 primary entities – types, and traits. For the types, language supports product and sum types(structs and enums respectively), and references (which can be mutable or immutable). Traits are more interesting feature. They are similar to interfaces in Java, but their closest analogy would be typeclasses from Haskell. Traits are used to declare a set of constraints, types, constants, and functions, that an implementing type(implementor) must provide. Each implementor, can implement any number of traits.

The language supports all the most common control flow constructs like conditionals and loops. In addition to that, it also supports expressive pattern matching using the `match` expression. However, it does not support the `goto` control flow construct for unrestricted jumps.

3.4.2 Features

The requirements influenced the design of the language in a significant way. It started out as general purpose programming language with functional features, very similar to ML, and due to its use for the implemetatio of the Servo browser, it acquired some features that make it excellent systems programming language. These features are:

- Generic programming based on traits
- Memory model that allows safety without garbage collector
- Primitives to eliminate data races
- Integrated build system and package manager

3.4.3 Generic programming, traits

Generic programming is a paradigm, in which algorithms are written in terms of unspecified types. The types are then specified upon instantiation. These types are called type parameters, or generic types. By using this tool, programmer can write common functions or types only once, and use them with multiple types, thus reducing duplication. This paradigm was pioneered by ML, and is supported in virtually every modern language in one shape or form.

Modern implementations follow 2 primary approaches for typing generic constructs: structural, or protocol based.

Structural generic typing (also called Duck typing) is primarily used in C++. With this approach, the type checking is performed after instantiation of generic construct. This allows for greater flexibility. But virtually all implementations of this approach suffer poor diagnostic messages[**?**].

Protocol(Interface) generic typing is an approach , in which the generic construct itself undergoes type checking, and every instantiation requires minimal amount of additional checks. This requires programmer to describe the required interface of each type parameter explicitly. These requirements take form of Interfaces (Java, C#), Concepts (Future C++), Type classes (Haskell) or Traits(Rust). Then, upon instantiation it is only necessary to check whether each type parameter satisfies specified constraints.

3.4.4 Traits

As described earlier, traits are used to declare interface, that a type must provide. They are used to support other language features, and must be understood in order to effectively use the language. Below is an example of a simple trait.

```
pub trait Ord: Eq + PartialOrd<Self> {
    fn cmp(&self, other: &Self) -> Ordering;
    fn max(self, other: Self) -> Self where Self: Sized {
        if other >= self { other } else { self }
    }
}
```

Listing 3.1: Trait definition

?? Shows definition of an Ord trait that defines complete ordering over implementors type. This trait specifies Additional constraints for implementing types (Also called supertrait constraints). Every type that implements Ord, must also implement Eq, and PartialOrd trait with generic argument of impelementing type. The implementor, must provide implementation for `cmp` method, that takes an one argument of implementors type, and returns an ordering. The trait definition also specifies `max` method for types, that implement Sized trait, and provides default implementation. The Self keyword is used to refer to implementing type in the trait definition. Traits can also be generic, accepting type parameters, such as the PartialOrd trait used earlier.

```
impl Ord for bool {
    fn cmp(&self, other : &bool) -> Ordering {
        if self & !other {
            return Ordering::Greater;
        }
        if self == other {
            return Ordering::Equal;
        }
        return Ordering::Less;
    }
}
```

Listing 3.2: Trait implementation

?? shows simple implementation of Ord trait specified earlier for boolean data type. This sample uses an impl block, to implement a trait for specific type. Impl block can itself be generic, and have to provide implementations for all functions that do not have default implementations specified in trait definitions.

3.4.5 Marker traits

The traits Eq and PartialOrd used earlier are self-explanatory, they denote the availability of equality comparison, and partial order in implementing types. However, the Sized trait might not be so easy to comprehend.

This trait belongs to special category of traits, called marker traits. These include Send , Sync, Sized and several others. The marker traits do not provide any functions and serve, as their name implies, as markers. They are used for marking specific types. The Sized trait marks types, which have their sizes defined at compile time, and is automatically implemented for these types by compiler. The example of unsized type might be [u8] , which is an array of unsigned bytes with unknown length.

The Send and Sync traits are crucial for features supporting safe concurrent programming , and will be explored later in this chapter.

3.4.6 Memory management

Modern programming languages primarily use one of 2 approaches to manage memory. The garbage collection is an approach most commonly used in High-level languages. 2 most common variants of garbage collection are reference counting and tracing garbage collection. Both of these approaches have drawbacks, primary ones being difficulty handling reference cycles for reference counting and necessary program pauses for tracing garbage collection

Second common approach used is called RAII, which stands for „Resource acquisition is initialization“. It is most prominently associated with C++, but is used in D, Ada, and Rust. This approach was originally developed for exception safe resource management in C++[**?**].

RAII is more oriented for management of resources, but if we consider dynamically allocated objects a resource, it serves the same purpose as garbage collection.

The lifetime of a resource is tied to object lifetime. The resource is acquired during creation of the object, and released during destruction. The object can have unconstrained lifetime (Allocated on a heap), or scope constrained lifetime (Allocated on the stack).

In C++ the creation and destruction of object is performed by specific functions(Constructor and destructor). Rust does not support object oriented programming in a classical sense. The data types are created structurally(enumerating all component values).

The destruction of values is performed with the help of a trait system. If a type, implements the Drop trait, it must implement the `drop` method, which has similar semantics to C++ destructor. This method will be invoked when variable of this type goes out of scope, and memory associated with it will then be deallocated

Move semantics

Another important concept taken from C++ is move semantics. Until C++11, the only approach was copy semantics, in which assignment to a variable from another variable would create copy of referenced object. The drawback of this approach, is inability to express a type, that should not be copyable, but should be movable.

The move semantics on the other hand, can express this concept easily. With copy, the assignment to a variable, invalidates the old variable. In C++ move semantics, the object referenced by old variable is replaced by and „Empty“ object (an object that is safe to destruct, and its destruction will not invalidate copied object).

In rust, the invalidation of moved-from variables is enforced at compile time, and usage of invalidated variable will result in a compiler error. Rust provides only move semantics, with copy semantics emulated by the Clone trait.

Ownership and borrowing

Conceptually, the move semantic are used to express the concept of ownership. If a variable, contains an object, it „owns“ that object , and is responsible for its destruction. However, requiring programmer to transfer ownership of an object every time it is passed into a function would be extremely tedious on programmer side, and copying of an object would degrade performance.

Rust also provides a way to reference objects, without moving or copying them. By using `&` or `&mut` sigil, the programmer can create immutable and mutable reference respectively. The reason for 2 different reference types is ensuring memory safety.

We can create any number of immutable references to an object, but these references can't mutate referenced object, or we can create one mutable reference, and use this reference to mutate the object. Creation of multiple mutable references at the same time is not allowed, and will result in compiler error. Compiler uses the concept of a *lifetime* to ensure that created references do not overlap.

3.4.7 Concurrency primitives

The concept of ownership and borrowing is also used to ensure memory safety in multithreaded programs, and prevent data races

Data race occurs when 2 or more threads concurrently access same location of memory, one of these accesses is a write, and accesses are unsychronized. These types of bugs are extremely hard to discover, and have lead to death of several medical patients in one extreme case[**?**].

The ownership and borrowing system prevents these kinds of data races, but Rust also provides tools for ensuring other constraints in multithreaded programs. Primary building blocks are 2 marker traits. The Send and Sync traits.

The Send trait denotes that the implementor can be safely transferred to a different thread. This trait is automatically implemented by compiler, when appropriate. For example, objects that reference thread local storage do not implement Send.

The Sync trait is implemented for types that can be safely shared between threads(eg. immutable reference).

These 2 traits are then used by library abstractions like Mutex and Rwlock to ensure memory safety and data race free code. For example, the Mutex abstraction is used to protect an object with a mutex. The Mutex struct is generic, with one type parameter, that denotes contained value, which must implement Send trait. This ensures that the contained value can be safely shared between threads. The mutex itself implements both Send and Sync traits, meaning mutex can be safely shared between threads.

3.4.8 Asynchronous programming

One of the most recent additions the Rust language is the addition of primitives for lightweight asynchronous programming. The key component of this feature is the Future trait. This trait denotes that the implementing type represents an asynchronous computation that will result in a value or an error at a later time. The types of resulting item of error are represented as associated type on the Future trait.

```

pub trait Future {
    type Item;
    type Error;
    fn poll(&mut self) -> Poll<Self::Item>, Self::Error>;
}
```

Listing 3.3: Future trait

?? shows the definition of the Future trait. As you can see, in addition to two associated types, it also contains a method declaration for `poll` method. This method is called when a runtime executing the future is interested whether the future has completed. Implementations of this method should actively try to complete the work represented by the Future.

This design means that Rust futures are pull-based, as opposed to push-based implementations (eg. Javascripts promises). Our implementation heavily relies on this feature of Rust.

3.4.9 Build system and package manager

One area that low level languages are extremely outdated compared more high-level languages is modularity and code reuse. While these languages provide tools for creating modules, that can be combined to form a larger program, they lack tools for supporting this process. Because of this, the number of libraries, that a project uses is extremely low, and each project ends up reimplementing existing functionality. This is a large problem in C++, where most common libraries used are extremely large (QT, Boost), and domain specific libraries are virtually nonexistent.

Rust aims to solve this problem with **Cargo**. Cargo is primarily a build too and package manager, but it also provides testing and benchmarking support. Cargo operates on Crates. A crate is the smallest compilation unit. Each crate contains multiple source code files, and a manifest file which specifies metadata information about this crate, and lists dependencies.

Crates can be published and uploaded to **crates.io** repository, which is closely integrated with Cargo. Each crate can then also require dependencies from this repository.

This improvement encourages development and usage of small, domain specific libraries, which in turn allows for the standard library to be extremely small, on par with C++, without reducing productivity.

3.5 Actix

One of primary reasons for choosing rust was the choice of Actix[**?**] library. Actix is a library that provides abstractions for implementing applications with architecture based on actor model. Internally it is based upon the Futures feature describe earlier.

Core API provided by the library is a set of traits, that are used to for introducing semantics based on actor model to custom types. In addition to that, the library also provides several structs, that rely on these traits in order to provide communication between individual actors.

3.5.1 Actor in actix

Actors are types, that implement the Actor trait. The Actor trait has an associated type, that defines the context in which the actor will be run, and several methods for dealing with actor lifetime. The actor context defines how will the actor receive messages, manages actor mailbox, and several other supporting components, but its detailed description is out of the scope of this thesis.

The programmer will also have to implement Handler trait for every message that it wants to handle.

```
pub trait Handler<M> where Self: Actor, M: Message
{
    type Result: IntoResponse<Self, M>;

    /// Method is called for every message received by this Actor
    fn handle(&mut self, msg: M, ctx: &mut Self::Context) -> Self::Result;

}
```

Listing 3.4: Handler trait

The implemented `handle` method runs synchronously, but asynchronous computation can be started by returning a value that implements a Future trait, from the handle method, or by utilizing the `spawn` method on the ctx argument to spawn a new asynchronous task in same context.

After creating an actor, the actor must be started. This can be performed by several functions. the `Arbiter::start` function starts a new thread, runs an **Arbiter** actor inside it, and then starts our custom actor within this thread.

Or the `Actor::create` starts an actor within current thread.

```
impl Handler<IngestUpdate> for Ingest {
    type Result = Box<Future<Item=usize,Error=()>>;

    fn handle(&mut self, msg: IngestUpdate, ctx: &mut Context<Self>) {
        Box::new(self.db.send(db::SaveOhlc{
            id : msg.spec.pair_id().clone(),
            ohlc : msg.ohlc
        })
        .then(|v| if v.is_err() { panic!("DB Error")} else { return Ok(v.count) }))
    }
}
```

Listing 3.5: Asynchronous message handling example

?? shows implementation of the Handler trait with asynchronous message handling. The Ingest actor receives IngestUpdate message, and in response then sends SaveOhlc message to db actor. Then, using the `then` combinator, it verifies that the operation completed successfully, and returns number of written rows to original message sender.

The actual computation is not started in the handle method. Rather the computation is described by creation of a value that implements the Future trait in this method, and actual computation is started after returning the value from `handle` method, when the task created is spawned on event loop in which the actor is running.

3.5.2 Networked actors in actix

While actix provided extremely well designed base for implementing concurrent applications based on actor model, it has one glaring flaw. It does not provide tools for running actors on different computers, and thus can't really be used for distributed applications by itself.

To reify this issue, part of this thesis was the design and implementation of `actix-comm` library, which extends base actix library with primitives for communication between actors on different computers using ZeroMQ technology. This library is described in detail in **??**.

3.6 Cloud environment

While usage of low level language, with the support for Effective concurrent programming should provide a large performance advantage, this does not solve the scalability problem. One of the requirements was for the resulting system to be able to scale according to number of users, and resulting load on the system.

To solve the scalability problem, we have decided to utilize a dynamic computing environment. This approach is called Cloud Computing[?] This approach is characterized by shared pools of configurable system resources and services, that can be rapidly provisioned with low latency. Cloud computing relies on sharing of resources, and economies of scale to provide better cost to performance ratio than dedicated computing environments.

There are multiple providers of cloud computing environments, with different Service and deployment models. Most popular of these environments are Amazon web services, Microsoft Azure, Google Cloud Services, or Digital Ocean. For the purpose of this thesis, the Digital Ocean was chosen to be primary provider, but modifying created system for other environments should be relatively simple.

The basic component the necessary for function of cloud environment is virtualization. By virtualizing resources, the provider can ensure isolation of different customers, and fine-grained allocation of resources.

3.6.1 Virtual machine model

Conventional approach to virtualizing computing resources is the usage of a Virtual Machine(VM). However, the problem with virtual machines is that each virtual machine runs complete OS, that is running within the confines of another operating system. This redundancy creates unnecessary overhead. Another issue is the management of these VMs. Since each VM is running a complete OS, this OS must be periodically updated.

3.6.2 Container model

More fine-grained unit of isolation is a container. A container is a simple lightweight image, that contains only an application, and libraries needed by this application. It does not contain whole operating system. Conceptually, it provides isolation on a layer, that sits between a process and a virtual machine.

The issue with simple containers is that, the containers still need to be managed, and as application grows, this task becomes increasingly hard. To solve this problem, there must be another layer, on top of containers, that will manage them. This is called the orchestration layer.

3.6.3 Kubernetes

Kubernetes[?] is an open-source orchestration system. It's used for automating deployment, management and scaling of applications that run in containers. This system was initially released in 2014, based on the Borg[?] system, that was used for similar purposes internally in Google.

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 nodes, that can be dynamically added or removed.

Basic used primitives are:

- Namespace - Is a tool used to partition resources into disjoint sets.
- Pod - A pod is a basic scheduling unit, it contains one or more containers, has assigned unique IP address withing a cluster,and can define a storage volume, that it exposes to its containers.
- Service - Is a set of homogeneous pods, that work together. Its main goal is to expose information about running pods to internal DNS.
- Deployment - Serves as a watchdog that automatically ensures there are pods in a healthy state available to serve incoming requests
- Volume - Object representing a persistent storage.

3.7 Web applications

[[Low level details ?]] Another aspect of the designed system is user access to this system. Because the system will be distributed, and will run in cloud environment, there is only one usable approach to implementation of the user-facing side of the system. We will have to provide a web interface. There are several possible approaches for implementing such interfaces. The primary distinction between them is the location of the main application logic.

Server-side rendering & logic

We could implement the whole application as a set of static html pages. We would introduce dynamic behavior into the server side by rendering these pages using some kind of templating language. And some dynamic behavior into client side by embedding some javascript. while this approach could reduce development time, it would almost certainly yield bad user experience, and thus we chose not to go this way.

Client side rendering & Logic

State of the art approach for creating web applications is so called „single-page application. In this approach, the application consists of 2 parts, the *frontend* part runs in browser, and it connects to the *backed* that is running on the server. The frontend is implemented in javascript, and its primary functions include interpreting system data for user consumption, accepting user inputs and communicating with the *backend.* The *Backend* then runs on the server, connects to the database, and usually provides some kind of a API, most commonly in the form of REST[1] API.

[1]https://en.wikipedia.org/wiki/Representational_state_transfer

Chapter 4

Design

This chapter aims to outline a system, that satisfies the requirements listed in **??**, utilizing concepts and technologies outlined in **??**.

4.1 General system design

System is designed as a collection of loosely coupled components, running inside virtualized environment provided by kubernetes, which can be distributed across many computing nodes. Basic diagram of intended architecture can be found in **??**.

Each component will be implemented as a set of actors that run inside a single process and communicate with other components over ZeroMQ using both Request reply and Pub-Sub communication patterns.

The actual communication protocol will be implemented in `actix-comm` library that is described in **??**.

[[**Reword, mention React more**]] The user facing part of the system will be a web application, that will be also implemented using the Rust language and `actix-web` library for the back-end and `React` for the front end. This library builds upon the `actix` library mentioned earlier, and provides

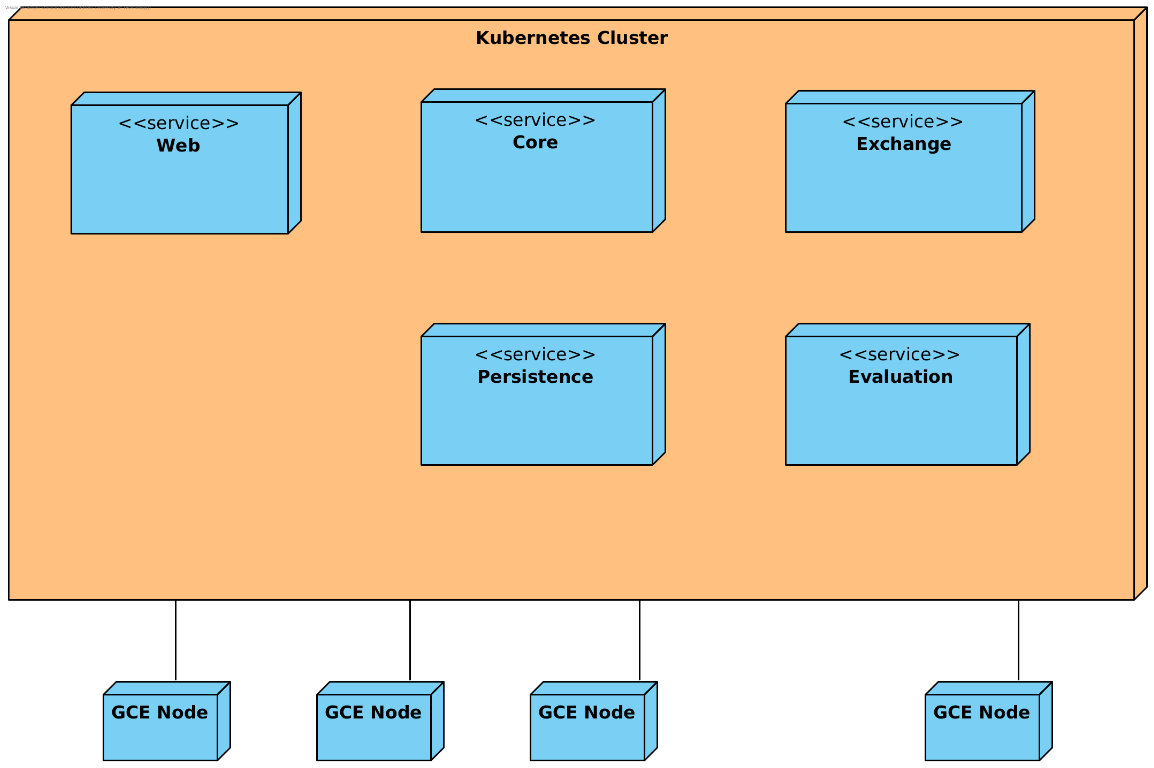

Figure 4.1: Basic architecture

4.2 Component design

By using Kubernetes we virtualized the computing environment. The environment as perceived by individual components will just be a set of connected Docker containers with access to DNS server, that contains information about other components. Kubernetes also provides virtualized network environment, making it appear as if all containers were on a same network.

This fact greatly simplifies architectural challenges. The only architectural challenge remaining, is the definition of how will the application components be connected and communicate. While the actual communication protocol is defined later in this chapter, here we are interested in more conceptual approach.

Application is divided into several components:

- Web component - Provides a web interface for users' inetraction with the system, and interacts with the database
- Core component - Accepts input from users, incoming data from exchanges, decides when to evaluate strategies, and when to create orders on exchanges
- Evaluation component - Evaluates strategies based on requests from Core service
- Persistence component - Stores tick data and user data upon provided by core service, and provides this data to all services if necessary

- Exchange components - Each of these components serves as an adapter, that connects to external exchange API, and maps its specific API onto internal communication channels.

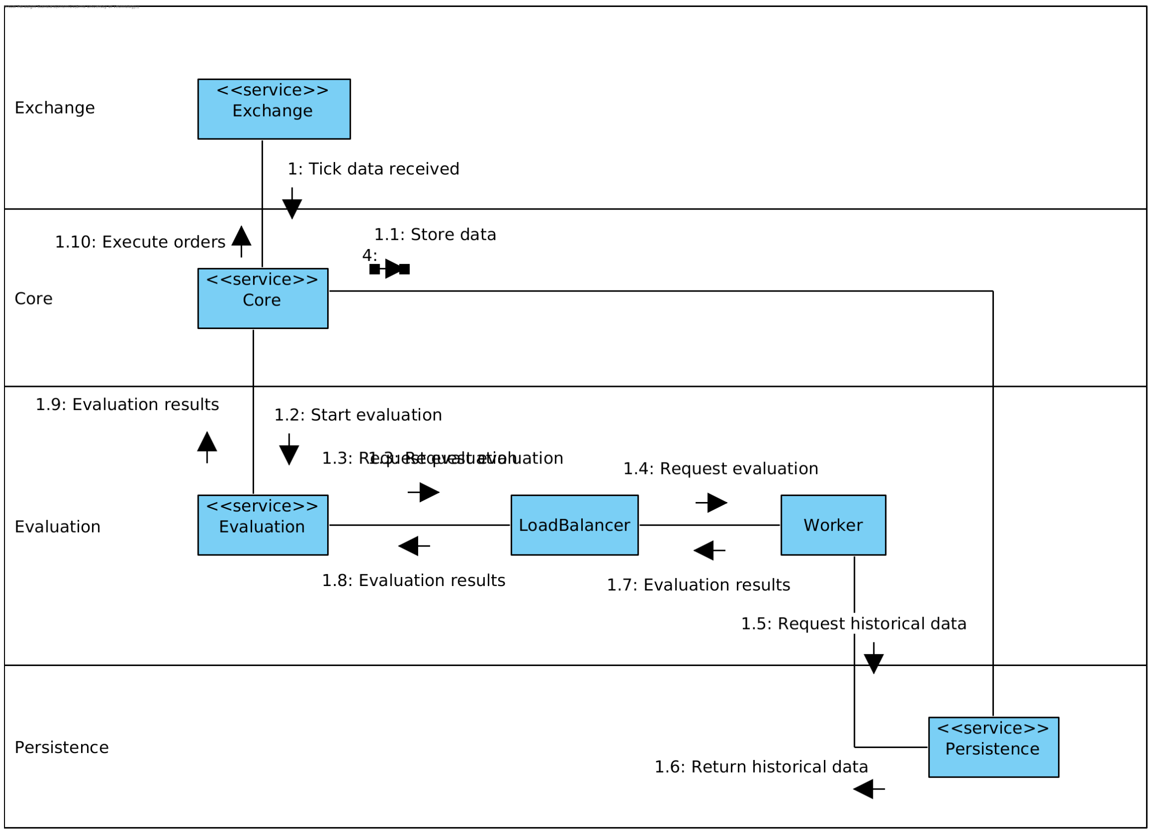

Figure 4.2: Service communication diagram

?? shows a communication diagram that describes communication between individual services, that will occur in response to receiving new financial data from an exchange. The system will store this information into persistent storage for later use, and if the information is up to date, it will initiate strategy evaluation for strategies applied to currency of incoming data.

4.2.1 Individual component architecture

Each of these services will be comprised of one or more kubernetes service-deployment pairs. The service part will ensure availability of information about individual pods on kubernetes' internal DNS server. The deployment part will ensure the availability of actual pods.

Each service will be comprised of one ore more pods that will be managed by deployment. Each pod will contain base communication actor from `actix-comm` library, and several other actors to support communication with other services. In addition to these supporting actors, it will also contain varying number of actors, that will collectively implement the desired functionality of a particular component.

4.3 Communication, actix-comm and actix-arch

As described earlier the Actix library does not provide tools for communication between actors on different machines. One of the goals of this thesis was to design & implement a library that would facilitate this functionality. The resulting library should be usable by other projects.

4.3.1 Underlying protocol

We chose to use ZeroMQ[**?**] as an underlying protocol instead of TCP/UDP because of several factors. The ZeroMQ can be used over many different transport types, including TCP, UDP, Unix pipes, PGM or shared memory. Another benefit is the added flexibility; While TCP requires establishing connection in a particular order (Bind then Connect), ZeroMQ does not have similar constraints.

Another possible approach would be usage of HTTP and/or Websockets. While these 2 communication protocols would probably satisfy the requirements, ZeroMQ was specifically designed for low-latency, low-overhead applications and seemed like a better fit into the global system architecture.

ZeroMQ

ZeroMQ is an asynchronous message-based communication library. It is aimed at low-latency distributed systems, and does not require a centralized message broker. It provides primitives for implementing different communication patterns. Our library will utilize 2 of these communication patterns.

The Router-Dealer socket types are used for asynchronous request-reply communication pattern, and our library uses them to implement **Request** and **Reply** actors respectively, which collectively implement described communication pattern. This type of communication is then used

The Pub-Sub socket types are used for publish-subscribe communication pattern. This communication pattern is implemented by **Publish** and **Subscribe** actors.

4.3.2 Library interface

Because the library is intended to be used exclusively with the Actix framework, the only provided interface will be in form of a several actors. These actors will respond to a set of messages that are also exported by the library. The implementation of additional functionality by creating a set of actors & messages is is very common within the Actix ecosystem. For example, the actix-web library, which is used to implement the web application, is also built with similar approach

The **Request** actor responds to **SendRequest** message, which denotes a request for sending another, wrapped request to remote machine. The **Request** anotates sent request with unique identifier, which is then inside the actor state along with an information used for notifying original **SendRequest** sender about the message response.

the **Reply** actor responds to **Register** message, which is used for registering another actor as a recipient of some message type. Then, whenever the **Register** actor receivies of said type, the message is forwarded to registered actor.

The **Publish Subscribe** actors operate similarly to **Request**, **Reply** actors respectively, but they do not send or receive message responses.

4.3.3 Communication protocol

Each of the defined communication actors contains one primary ZeroMQ socket, that is used for receiving and sending messages to other components. The actual communication protocols vary between individual actor types

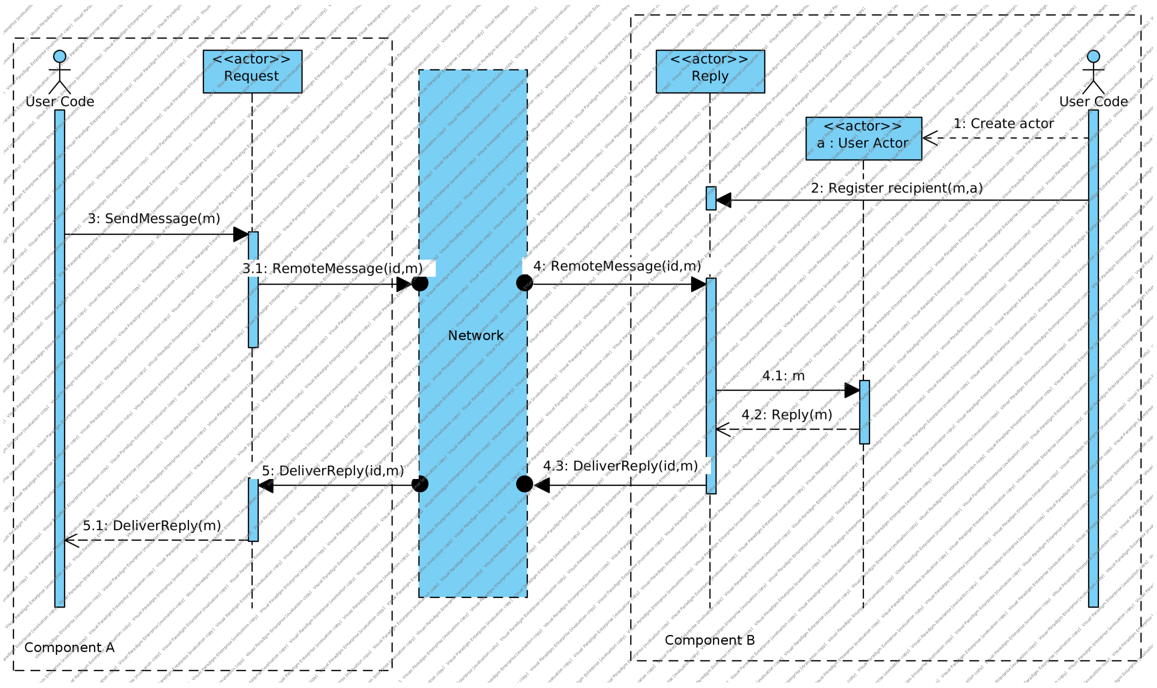

Figure 4.3: actix-comm communication

In **??** you can see simple sequence diagram containing basic request-reply communication pattern. The diagram contains 2 component. The component A contains a **Request** actor, that is used to send request to component B, which contains **Reply** actor and another, programmer defined, custom actor. The user code in component B first registers the user actor as a recipient of message type M. Then , the user code in component A sends a SendRequest to its actorRequest actor. It then stores some information about the message into its local state, and sends message data along with its type to ZeroMQ socket.

After the **Reply** actor in component B receives this message, it forwards the message to actor registered earlier, and upon a response from this actor, it sends response along with metadata received earlier back to ZeroMQ socket.

Then, upon receiving the response, the **Request** actor uses provided metadata to get apropriate notification channel from its local state, and uses it to notify originating code about the response.

4.3.4 Message format, actor state

[[Type erasure, message stored as a type name + json/Msgpack data]] [[If name not found, message not understood]] [[Fast, simple]]

Actor state

[[Identifier tokens, currently int]] [[Channels , rx is provided as asynchronous reply, tx stored internally]] [[Rx is also a future, a lot of futures used]]

4.3.5 Actix-arch

Is a library that was built on top of `actix-comm` simplify development of individual components. It contains implementations from common communication patterns, and components to support the development of communicating applications.

Service abstraction

Since most common communication pattern is Request-Response, this component was created to support this type of communication. It consists of 3 parts. The ServiceInfo trait, that is used for declaring crucial information about the service, like hostname, on which the service is available, and types of request and response.

```rust
pub trait ServiceInfo: 'static + Debug {
    type RequestType: Remotable + Debug;
    type ResponseType: Remotable + Debug;
    const ENDPOINT: &'static str;
}
```

Listing 4.1: ServiceInfo trait definition

Two other components are the ServiceHandler and ServiceConnection structs, which are generic over type paramenter S, that must implement ServiceInfo trait. The ServiceHandler struct has a register method, which can be used to register a handler of service messages. Both of these structs internally use ZeroMQ actors defined in `actix-comm` library, namely the **Request** and **Reply** actors.

Publish - subscribe abstraction

Is an abstraction for implementing Publish-Subscribe data flows. Is analogous to Service abstraction, utilizing 3 parts. The EndpointInfo trait defines the hostname, on which the binding endpoint can be found, its associated type FanType can be either FanOut or FanIn, and defines, which part of the communication channels binds to an address, and which connects to it. With FanIn, the subscriber binds to a port, and multiple publishers connect to it (used for receiving data from multiple exchange adapters), and with FanOut, the publisher binds to a socket, and subscribers connect to it.

```rust
pub trait EndpointInfo {
    type MsgType: RemoteMessage<Result=()> + Remotable;
    type FanType: FanType = FanOut;
    const ENDPOINT: &'static str;
}
```

Listing 4.2: EndpointInfo trait definition

The Publisher and Subscriber actors are generic over type parameter E, that must implement the EndpointInfo trait, and contain methods for publishing and subscribing to updates respectively.

Service load balancing

Other component necessary for our designed system was a way to perform load balancing for Service handlers. This will be mainly used in the strategy evaluation component, since this component will probably result in most of the computing load.

This component is implemented as 2 actors: the **LoadBalancer** and **WorkerProxy**, which both have one type parameter S that must implement ServiceInfo. Then, the LoadBalancer binds 2 ServiceHandlers to single port. The first one is used for receiving service requests from client, and the second one is used for receiving message for workers.

The workerProxy internally contains a ServiceConnection, that connects to **LoadBalancer**, and periodically subscribes for work. If **LoadBalancer** does not have any work available, it sends an empty response to **WorkerProxy**, or, if available, it sends a work unit to this worker. The **WorkerProxy** then sends received work to internal worker implementation, and after finishing, sends result to **LoadBalancer** as a separate request, that is also used for requesting more work units. [[**Diagram, cant explain this in words only**]] Therefore, the **LoadBalancer** therefore serves as a load balancing broker, which performs rendezvous between available workers and work units, communication using 2 service handlers.

4.4 Data storage

Another crucial aspect of designed system is the storage of both the financial data and general system data. These 2 types of data have different storage requirements.

4.4.1 Financial data

[[**Move this earlier ?**]] Financial data received from exchanges is in the form of several prices and volume that is aggregated over a time interval. These prices are:

- Open - Price, that was used in first transaction in this time interval
- High - Highest price that was used in transaction in this time interval
- Low - Lowest price that was used in transaction in this interval
- Close - Price, that was used in last transaction in this time interval

In adition the these price, each interval is associated with its starting timestamp, and cumulative volume of executed trades. We will refer to this type of data as OHLC data, based on the OHLC chart which is used for displaying these datasets ([[**Footnote https://www.investopedia.co**

There are several requirements put on storage solution that will be used for storing the OHLC data, mainly stemming from amount of this data, and operations that must be performed on it.

The chosen storage solution will have to support large ingest rates. For most basic implementation, supporting bitfinex exchange, we will have to store OHLC data for almost 100 assets, most of which are updated every 10 seconds. This would mean the insert rate of 10 rows per second.

Another aspect is periodic retrieval of latest data for evaluation of strategies. This retrieval is not a simple load from database, the stored data will have to be cleaned up, [[**Insert missing mpoints**]]

4.4.2 Application data

For storing application data, we have to store data that conforms to relational model. **[[ERD HERE]]** This ERD diagram shows basic data model of the system. This data model utilizes several foreign keys, and will require a relational database for storage.

4.4.3 Evaluated storage solutions

In order to store application data, we just have to use a relational database. These databases are most common types of databases used, and have extensive tooling built to support them. For the purposes of this application the **PostgreSQL** databases was chosen. This is an open source database which supports most of the modern SQL, and probably the only competitor to Oracle's offering in terms of functionality. One of the most exciting features of this database is support of extensions, that transform this simple relational database into general purpose storage solution. Some of the database extensions are

- Citus - Adds support for clustered deployments, in which several instances of PostgreSQL databases store data - improving redundancy and performance
- TimescaleDB - Improves upon built-in table partitioning, and adds custom storage engine optimized for time-series datasets.
- CStore - Adds columnar store, which can be used for storing data in ORC format.

Thanks to these features, the **PostgreSQL** was chosen as a database for storing application data.

However, the storage solution for storing OHLC data has different requirements. The chosen solution will have to support large insert rates mentioned earlier, fast retrieval of newly inserted subset of data, and preservation of these properties over time, when amount of stored data grows past the amount that could be stored in memory.

These requirements are common among the applications that deal with a steady stream of time-dependent data, also called time-series data.

There are several approaches to storing time-series data. We could store it as a normal postgresql table. This would ensure avaiability of analytical functions for working with data in the database engine, removing the need to implement it in application. But the drawback of this approach is scalability. While **PostgreSQL** is among the most peroformant database today, its performance quicky degrades with large tables**[[Cite timescaledb research on ingest rates]]**. Most common solution to this problem is table partitioning, that creates sub-tables for our time-series data table, and divides data into these subtables based on the temporal dimension. However, this requires additional tooling and management.

Another approach is to use a distributed database with multiple nodes. Then, the data would be partitioned across multiple nodes, with some degree of redundancy. Therefore, inserts and reads both can be easily parallelized, and we would gain redundancy, preserving data in case of node failure. Drawback of this approach is the overhead, that will use of multiple nodes incur. **ScyllaDB** was evaluated as an alternative, that utilizes the distriubuted approach.

But, after considering these multiple approaches, we have decided to use **PostgreSQL** with the **TimescaleDB** extension for storing OHLC data. The **TimescaleDB** extension targets time-series data storage, making it ideal for our needs. It creates a *hypertable* concept, that is an abstraction over standard table, perfroming automatic partitioning over temporal dimension, and optionally over several hash dimensions. It also provides to analytical functions for working with time series data (eg. `time_bucket` that allows working with data over different timescales). Thanks to the customized storage engine and automatic partitioning, this extension solves the main problem of reduced performance for large databases.

However, if this approach fails, there is experimental **ScyllaDB** storage layer for storing OHLC data, that seems to provide same performance as current **PostgreSQL** storage layer.

[[**SCYLLADB DOES NOT HAVE PROPER SQL -**]] [[**ERD**]] [[**Postgresql with timescaledb**]] [[**Hypertable - sharding**]] [[**Currently more htan enough, we'll see about future**]]

[[**Experimental Scylladb backend**]] [[**Not in production, postgresql more than enough**]]

4.5 Web

As mentioned earlier, we elected to implement the web application with the single-page approach. This requires dividing the application into 2 parts. The Backend part will run on the server, and provide information to the Frontend running on the clients web browser.

4.5.1 Backend

The backend part of the web application is implemented in `actix-web` library, which, like many other parts of this project, is built upon `actix` and its implementation of the actor model. This library provides several REST endpoints for working with system resources like strategies, trader accounts, or strategy assignments. Internally, these endpoints should perform validation of input data, and execute database queries, which modify the state of the system.

Other languages were not considered for this task, because utilizing them would mean that definitions of stored entities would have to be duplicated, and the cost of duplication would not be outweighed by the benefits provided, since the Backend part of the application is relatively small.

4.5.2 Frontend

For implementation of the Frontend part of the application, we evaluated several popular technologies.

Polymer

Is a library developed by Google. It utilizes custom webcomponents[[**Cite**]], Shadow DOM [[**Cite**]] and HTML Templating[[**Cite**]] technologies to achieve remarkable set of functionality with only a small extension build upon common web standards, that all major browsers implement. This technology is based on the `lit-html` library, which can be used for creating HTML templates directly from javascript. However, using polymer requires custom command line tool, in order to build applications made with this technology.

Angular

One of the most popular web frameworks, provides large amount of functionality. It does not use webcomponents or shadow dom, due to the fact that these advances came after it was created, since its predecessor was first of these kinds of libraries. It provides two-way databinding, utilizes MVC architecture and should be considered to be a framework instead of a library due to opinionated nature of this library. It uses Typescript instead of Javascript as main application implementation language.

React

Is is component based user interface library, with one of its targets being the web browser DOM. Other targets include Android , iOS, UWP , and are implemented in React-native branch of this library. React uses JSX indstead of plain javascript. This language is an extension to javascript, that allows programmer to write inline HTML inside normal javascript code, improving readability. React does not attempt to provide general application framework, its only targeted at building user interfaces.It is based on its virtual-DOM architecture. In this approach the entire application is rendered into memory representation of the resulting DOM, that is then compared to actual rendered DOM tree, and only changes are applied. This reduces number of updates that must be performed, increasing rendering performance at the cost of memory usage.

Flux & Redux

One of most important concepts used in React is the uni-directional data flow. Data flows from components to their children, and notifications from children to parents. To support this paradigm, we utilize the Flux architecture. In this architecture there are *actions*, that flow through the central *dispatcher* to a *store*, and changes in the store are propagated back to view. In react, this propagation is performed through component properties. This approach is similar to observer pattern[1] With this architecture, the properties passed to a components are immutable. Only way the application state can be affected is through sending *actions* to dispatcher.

Redux is the most well known implementation of this pattern. It features a single store, and dispatchers are called reducers. The reducers are pure functions, that respond to actions, and based on them modify this single source of truth.

4.5.3 Frontend application

After experimenting with each of these technologies, the **React** library with **Redux** was chosen ast the most suited technology for creating the *frontend* part of the web application. Another aspect of the *frontend* part of the application to consider is the user experience. The user must be able to:

- Create an account
- Login & Log out
- Create & edit strategies and exchange trading accounts
- Assign strategies and trading accounts to assets

[1]https://en.wikipedia.org/wiki/Observer_pattern

- Visualize the results of strategy evaluation and executed trades

4.6 Exchange adapters

While all cryptocurrency exchanges use similar technologies to implement their APIs, each of them is different. To reduce complexity of the system, these differences must be resolved at the edge of our system, and should not permeate into other components. To bridge the gap between external APIs, and internal communication, the system contains adapter component for each exchange.

This component connects to real-time webso cket API in order to receive notifications about market updates, then translates this data into OHLC format, and sends it to the core component. It also exposes a service endpoint for querying account state (wallet balances), and executing trades, to which the core's trader actor connects.

4.7 Strategies

In **??** we have described several implementations of automatic trading systems. Each one of them utilized some kind of programming language to define a trading strategy. In this regard, our system is very similar to others.

The system will have to support execution of user-written code. This fact poses a security concern. Because the user written code can perform arbitrary actions permitted by given programming language, we must carefully choose the programming language that will be used. Because implemented strategy will be operating with large amount of financial data, another concern is performance. And finally, since the intended users of this application are not programmers, the chosen language should be easy to use for beginners.

4.7.1 Language choice

These requirements severely limit possible choices. We can't accept user-compiled code, because of security concerns. Compiled languages like C/C++ are not acceptable because of large amount of infrastructure needed to support on-demand compilation of user written strategies.

Managed languages like Java or C# are a better choice, but they still require large runtimes with long start up times, therefore are not well suited for running short-lived scripts.

Scripting languages like Lua, Python or JavaScript seem like the best choice for this goal, with the drawback of reduced performance.

4.7.2 Lua

Ultimately, the Lua language was chosen as a primary language for implementing user defined strategies. There were several key properties, which caused this decision.

- Embeddability - Lua runtime is smaller than 256Kb, has virtually no start up time and can be embedded in application. as a simple library. In comparison, neither Python nor JavaScript runtimes can be embedded in the application, and both require large standard libraries

- Extendability - Basic lua standard library can be easily extended with code written in host language.
- Expressive power - While extremely simple, lua provides tools to model virtually any programming paradigm with ease
- Speed - While lua is a scripting language, that can't possibly compete with compiled language in this space, it is one of the fastest scripting languages available.

4.7.3 Safety

By using an intepreted language for implementation of user strategies, we have successfully eliminated a whole class of risks. By using a safe language, the probability of user code crashing the executing process is virtually none. However, there still are several safety issues, that have to be resolved, even with using an interpreted language.

Sandbox

As a basic strategy for ensuring the safety of strategy execution, we chose to utilize a sandboxing mechanism. This mechanism is supported by LUA very well. The sandbox mechanism consists of replacing the global environment table with a table, that contains only functions deemed safe when executing user code. This simple replacement restricts access to unsafe functions like `read`, removing the ability to of an malicious to to affect our system

Execution control

Another attack vector considered was the ability of an attacker to perform A Denial-of-Service(DOS) attack by submitting a strategy, that never terminates. Upon the start of evalauation, this strategy would lock currently evaluation actor, reducing the amount of available actors for strategy evaluation. After sufficient amount of attempts, this would leave no available actors, and then, the system would be unable to function.

4.7.4 Access to information

Method for strategy evaluation described so far should provide safe, and performant way for users to write custom code, that can be run inside our system. However, we must allow this code to access the financial data, and provide some kind of library for supporting the strategies.

We expose a vector of OHLC data items inside the `__ohlc` global variable. Each item in this vector has methods to access individual prices described in the **[[autoref OHLC definition]]**.

We also expose a set of functions implementing indicators for Technical analysis[2]. These are available inside the `ta` global table.

To ensure this attack is not possible, we limit number of LUA instructions that a particular strategy can execute. This is done by utilizing the `debug.sethook` function.

[2]https://en.wikipedia.org/wiki/Technical_analysis

4.8 Evaluation

Since users can crate multiple strategies, and then apply these trategies to multiple different assets on different exchanges, the amount of work associated with a single user can vary extremely. In addition to that, the number of users of our system can vary. This variability of computational load on the system was primary reason for designing the system as a distributed application.

The service that is most affected by this variability is the strategy evaluation service. This service needs to dynamically change the amount of used resources.

To implement this component, we will utilize the **LoadBalancer** actor defined earlier. The compoennt will consist of 2 sub-compontents. The load This will require specific architecture. The service will be divided into 2 parts. The control and the worker layers. The control layer will be a single Kubernetes pod, that will serve as an endpoint to rest of the system. It will receive strategy evaluation requests from Core service, and will pass them to individual workers int the worker layer. Each worker will be a single pod with multiple worker actors, each of which will register itself with control layer.

The control layer will also perform load balancing, ensuring that no single worker is over- or underutilized.

[[**Eval balancer implemented as a load balancing broker**]]

[[**TA library used to provide basic indicatiors (optimized) - Otherwise an iterator over raw values is available, user can manuall implement his logic**]]

Chapter 5

Implementation

In this chapter, we aim to present current state of the implementation, outline the problems faced when trying to satisfy the reuqirements outlined in previous chapters, what methodologies and approaches were taken to solve them, and also point out some interesting aspects of the implementation. Another goal of this chapter is to present the operations side of this project, meaning the processes used for building and deploying the system, since that was another important part of the implementation

5.1 Project structure

Like any other larger project, this project is also structured into several subdirectories. The main subdirectory is `code`, which contains the code for system compoennts,the `common` library and several other libraries in the `deps` directory. The `common` library groups dependencies that are shared by all components, and re-exports them for easier acess and centrailized version selection. The `deps` subdirectory contains for `actix-arch`, `actix-comm`, `db` libraries and other dependencies.

Then, the second core directory is the `ops` directory, that contains scripts and configuration files needed for deploying and managing the system.

The main directory contains `Cargo.toml` file, that is used by the **cargo** tool. This root file defines a workspace, and some configuration profiles for rust compilation. Each directory containing sub-project of a library or an executable also contains the `Cargo.toml` file.

The `target` directory contains build artifacts created byu cargo, and is also used to store intermediate files produced by our custom build scripts, that will be described in the following section.

5.2 Building and deploying

Thanks to the distributed architecture, the building and deployment processes also had to be modified with this choice in mind.

The rust language utilizes custom tool named **cargo** for building rust projects, and managing their dependencies. Having single tool manage both build process and dependency managment is extremely useful, but its not without drawbacks. **Cargo** cannot be used for building anything else than rust projects, and while it allows to customization of the build process by runing pre-build custom scripts, it does not support custimizing the build process with any kind of post-build steps.

Therefore, to solve these drawbacks, we have to wrap cargo in a meta-build system, that would manage building individual rust projects using **cargo**, and perform any additional steps that would be necessary.

5.2.1 Makefile meta-build managment

This meta-build system, does not have to do anything particularly complex, it only needs to track what projects were changed, and depending on that information re-build, and re-deploy them.

These requirements are perfectly satisfied by the ancient **make** unix tool. The project root contains the root Makefile, which references supporting makefiles stored in the `ops/make/` directory. It contains the `deploy` target, that builds all custom components, creates new docker images from them, re-evaluates the kubernetes configuration templates, and finally applies this configuration to currently active kubernetes cluster.

Building a component

One build target roughly corresponds to a **cargo** project, that produces an executable, and a docker container that contains this executable. Implementation of this build process is in `ops/make/App.mk` makefile. This makefile invokes cargo in an appropriate subdirectory, it then creates a docker container image according to `ops/docker/app.Dockerfile`, and emits container name and container tag into a file in `target/docker/` directory. This information is provided in a format that will later be loaded into an environment variable, and used for substitution when building kubernetes configuration templates.

5.2.2 Kuberentes configuration templating

This is very well known drawback of kubernetes. It does not support any kind of templating out of the box. There are existing solutions, that solve this issue, but many of them do so with opinionated approach, that is coupled with some kind of package management.

We elected to perform this templating manually, inside the makefiles implementing the build process. For this purpose, build of each component emits built docker image tag in the format of `{COMPONENT}_IMAGE={IMAGE TAG}`. This is then loaded into make environment using the `env` directive. Then, for each file in the `ops/k8s` directory the main makefile loads it, performs environment variable substitution using the `envsubst` tool, and saves modified file to `target/k8s` directory. There are dependencies between these steps, which are also listed in the makefile, and therefore it does not re-build components that were not modified.

Each image of a component is tagged with a special unique information identifying application version. Currently, this is the first 10 characters of the sha256 hash of the binary, but in the future, this should be changed to current git commit hash.

5.2.3 Build targets

Currently, the application is divided into 2 build targets. The `web` build target contains the implementation of the web application(both frontend and backend), and the `app` build target contains the implementation of every other component that in the system. This grouping is an artifact of how the system was developed, and in the future should be resolved by moving each system component into separate build target.

5.3 Component implementation

Most of the application is written in pure rust, with the exception of the frontend part of the web application, which is written in javascript. The project uses multiple advanced features of rust, that have not yet been stabilized, and therefore requires a nightly toolchain. Currently, the project uses nightly toolchain from 2019-04-07. Main reason for usin a nightly toolchain with a specific version is the fact, that the implementation utilizes async-await style programming, which has not yet been stabilized, and was recently changed. This features should be stabilized in the version 1.36, which will be released in several months.

[[Async await example]]

The usage of asynchronous code is prevalent throughout the codebase. Almost every actix actor is written in a way, that requires it to send and wait for message response when responding to a message itself. With synchronous code, this would mean, that the throughput of the system would be very low. With asynchronous code, the actor will perform some synchronous actions, will send messages other actors, and will then asynchronously wait for these messages. During this waiting, the actor can respond to other messages, greatly imrpoving throughput.

5.3.1 Core component

Core component is comprised of 3 main actors. As its name suggests, it implementes the core system logic, and other systems connect to it.

Ingest

The **Ingest** actor binds a **Subscriber** to a known port, and waits for publishers to connect to it. The publishers are created by exchange adapters, and the ingest actor uses this socket**[[or use actor ?]]** to receive messages that cointains newest OHLC data. The ingest actor then discards old data, and publishes new, valid updates to a proxy actor, that then sends it to **Rescaler** actor

Rescaler

This actor is responsible for creating aggregate data streams, that are not provided by the exchange adapter, but nonetheless are supported for trading by our system. In order to perform this task, the actor must have at least 12 hours of data available in memory. This data is loaded upon actor creation from the database, and the oldest data is periodically discarded during the normal operation of the actor.

The incoming data is then published along with the virtual aggregate data over proxy actor to the **Decision** actor.

Decision

This actor is responsible for deciding when a strategy should be evaluated. It periodically loads whole *assignments* table into memory. This table contains all assingments of strategies and trading accounts to individual accounts. It stores this information into B-Tree map with the asset as a key, and a vector of assignments as a value.

Then, upon receiving update from **Rescaler** actor, the **Decider** actor traverses its internal assignment map, and request strategy evaluation for each applicable strategy. This request is sent using the **ServiceConnection** to **eval** component. These requests are asynchronous, and therefore this actor can spawn hundreds of them without blocking the rest of the system.

Then, after receiving result from evaluation, the actor can send a request to **Trader** actor, but this is only possible, if the assignment used had a trading account associated with it.

Trader

This actor is responsible for executing trades on individal exchanges. It receives position change requests from the **Decision** actor. Then, it sends a request to get account balances to exchange adapter. After receiving a response, the **Trader** actor decides if the trading account has any available funds that could be used to strengthen the selected positon, and possibly executes a trade by sending a new request to the same exchange adapter.

Internally, the trader uses the `anymap` crate to store **ServiceConnections** to each exchange adapter, since each adapter has its separate **ServiceInfo**. The `anymap` crate allows this actor to store type-erased values, and then retrieve them on demand.

5.3.2 Eval component

This component is responsible for evaluation of user strategies. As mentioned earlier, this component consists of load balancing broker and a dynamic set of workers.

Load balancing broker

This broker is implemented by the **LoadBalancer** component described in previous chapter. In itself, it is not extremely interesting. It runs in separate kubernetes deployment, and is exposed by kubernetes service.

Workers

The evaluation workers are implemented by the **EvalWorker** actor. This actor uses a **WorkerProxy** actor to connect to the load balancer. The proxy actor handles communication with the balancer, and communicates with **EvalWorker**. The **EvalWorker** is written as a service handler, that accepts requests for strategy evalution. Each request contains the identifier of the strategy used, an asset identifier, and some additional information like the timestamp that denotes the point in the data stream, on which the strategy should be evaluated. The worker currently always reads the strategy, and the OHLC data from the persistent storage. Adding some kind of cache could be an easy optimization.

After receiving the strategy and OHLC data from the database, the worker creates a new LUA VM. It then initializes this VM by creating a technical analysis library and attaching OHLC Data.

It then executes the strategy in a sandbox, that removes any functions that could be used to access or modify system information from the environment. The execution is also timed, and if the script executes for too long, it is automatically killed.

After the strategy evaluation, the **EvalWorker** returns the result to the load balancer, and it in turn returns it to the original request source, which was the **Decision** actor in the core component.

The actual strategy evaluation is implemented in `strat-eval` library. This library could be extended to support other types of strategies.

5.3.3 Bitfinex adapter

This component consists of single actor. This actor currently performs several tasks at once, and should be divided into multiple actors in the future. During creation, it connects to Bitfinex websocket API, requests a list of available assets, and then subscribes for notifications for each asset.

After subscribing, the actor starts to receive notifications, that are then promptly translated into OHLC data and published to the **Ingest** actor in the core component.

Another task that is performed by this actor is the serving of wallet balance and trade requests from the trader actor. These request are translated into REST API calls, that are performed by the http client implemented in `actix-web` library.

The bitfinex adapter actor is also implemented in an asynchronous way, increasing throughput.

Future adapters will follow similar design to the bitfinex adapter, therefore extending the system to support additional exchange should be extremely easy.

The bitfinex adapter is also managed ky kubernetes deployment and exposed by a service.

API access implementation

Acces to exchange APIs is implemented in the `apis` library in the `code/deps` directory. This library implements primitives that could be shared between different exhange adapters, and contains a `bitfinex` submodule that contains definitions of types received from API calls, and implementations of these API calls as asynchronous rust functions.

This pattern of implementation of specific API exchange in a submodule will be continued when extending the system with other exchange adapters.

5.3.4 Persistence

This component is responsible for storing OHLC and user data. As mentioned earlier, we elected to use PostgreSQL with the TimescaleDB extension as the primary storage solution. This currently runs in a single large docker container, that is also managed by kubernetes deployment and exposed by a service. This could pose a problem, since usage of only one database instance provides a single point of failure. This issue could by fixed by sharding the database with simple streaming replication, but that approach would only ensure readability of data during issue with the masterserver, and would not allow writing of new data.

Probbly the only solution to this issue is the usage of distributed database, in the form of PostgreSQL with the citus extension, or using another database, like scyllaDb

Access

The access to the database, is implemented by the `db` library

[[Persistence - ERD]] [[Needs to be first do better understand other components]]

[[Implemented in the DB library]] [[Uses diesel for query generation]] [[Synchronous - uses R2d2 for connection pooling and SyncActors for concurrent access to DB]] [[Describe the Database wrapper struct]]

5.3.5 Web component

This system component implements the REST API neccesary to acces the system. This API is then in turn used by the frontend web application. In theory, this API could be published for consumption by other developers, that wish to extend our system.

The API is implemented using the `actix-web` library, that utilizes the actor model to its fullest, being one of the fastest HTTP library in the world**[[Cite techempower benchmark]]**. This library allows programmer to attach individual functions to HTTP URIs, and these funcitons are then invoked by a set of worker actors, whenever they receive a request matching said URI. The library also performas automatic parsing of path and query parameters and request bodies. Another core part of the library interface is the programmer defined state, that can be attached to a running web server. This state can be then retrieved inside every handler function.

Our implementation uses it to store handle to database access actors, which are then used for retrieving data. This is implemented using the **Database** wrapper struct, that was described earlier.

Most of the implemented endpoints act as a simple interface to the database, that performs some validation, but there are some custom logic**[[Describe assignments]]**

This part of the system heavily relies upon the async-await style of programming, that requires the nightly compiler. Every single handler method is implemented as an async function, that internally calls several other async functions.

```
async fn api_detail((req, id): (HttpRequest, Path<i32>))
      -> Result<impl Responder> {
    let db: Database = base.state.db.clone();
    let base = await_compat!(BaseReqInfo::from_request(&req))?;
    require_login!(base);

    let (strat, user) = await_compat!(db.strategy_data(id.into_inner()))?;
    let evals = await_compat!(db.get_evals(strat.id))?;
    require_cond!(strat.user_id == base.auth.uid, "Not authorized");

    Ok(Json(strat).respond_to(&req).unwrap())
}
```

Listing 5.1: Example web handler function

The **??** code sample shows an example of a handler method, that is used for retrieving a strategy from the database. First important aspect is the method argument, which is a tuple of the request and a path parameter, that is automatically extracted by the library.

The function first clones a handle to the database wrapper, and then retrieves basic info from the request. The **BaseReqInfo** struct primarily contains the authorization information. This retrieval must be asynchronous, since it might require a database access. This asynchronous operation is then wrapped in a `await_compat` macro, which internally wraps the `await` macro.

Then, the strategy is retrieved from database, and `require_cond` macro is used for ensuring that user can only access strategies that he owns.

Finally, the strategy is serialized wrapped in the `Json` struct, that performs serialization into Json in the `respond_to` method.

5.4 Web application frontend

As mentioned earlier, the frontend application is implemented as a react single page application. This approach was chosen after a long series of experiments with multiple approaches to web applications.

The first tried approach was completely server-side rendered site that utilizes static html forms for submitting data. While this approach did work, the problem was extremely poor UX, and very complex generated forms that utilized hidden fields in order to preserve information for form submission.

The completely server-side generated site was then progressively rewritten into combined application, that used forms for submitting data, but utilized custom web components based `LitElement` library to reduce complexity and provide some amount of dynamic behavior, improving user experience. However, there were problems with this approach too. The user experience still did not meet the standards expected by modern users, and the application logic was now in 2 languages distributed acros 2 codebases, greatly complicating futher development.

After both of these approaches failed, we elected to rewrite the web interface part of the system from scratch. This time, the backend was rewritten to the form of a REST API described earlier, and the frontend application was rewritten in React. We also decided to use Redux[[**Footnote**]] to manage applicaton state, and Material-UI library to provide set of base components, foregoing any kind of complex UI template.

5.4.1 React

As mentioned earlier, React is javascript library for writing user interfaces, that utilizes custom JSX syntax. The web applciation code resides in `code/web/app` and during compilation of the web component, the compiled web application is bundled inside the `web` binary using `includeddir` library.

This web applicaiton is then provided on the `/app` prefixed URI routes, while the `api` prefix contains all the REST endoints. The web server provides the same `index.html` page for every subroute under the `app` prefix, and this generated index file imports resources from the `/static` prefix, that refers to individual files of compiled react application.

5.4.2 Components & Routing

The application is divided into several root components, each root component roughly corresponding to single 'page' of the application. Because the server provides `index.html` for every URI under the `/app` prefix, the routing between indivudual components that reside under this prefix is performed in the browser using the `react-router` library. This library switches rendered component based on current path. Follwoing example shows the JSX for the root of the application.

```
<div className="App">
    <ConnectedRouter history={history}>
        <Switch>
            <Route exact path="/app/auth" component={Login}/>
            <Route path="/app" render={props => (
            <Dashboard>
                <Switch>
                    <Route exact path="/app/" component={Home}/>
                    <Route exact path="/app/strategies" component={StrategyList}/>
                    <Route exact path="/app/strategies/:id" component={StrategyDetail}/>
                    <Route exact path="/app/assignments" component={AssignmentList}/>
                    <Route exact path="/app/traders" component={TraderList}/>
                </Switch>
            </Dashboard>
        )}/>
        </Switch>
    </ConnectedRouter>
</div>
```

Listing 5.2: React application routing JSX

[[Remove subsubsection]]

Dashboard

This component contains the base layout of the web application, rendering the toolbar with title, navigation bar, and also rendering nested components. In our case, the nested component is the react-router **Switch**, that ensures only one route from a set of route elements is active at one time. This **Switch** component with the set of inner routes contains the individual pages.s

5.4.3 Redux

Another important aspect of the web applicaiton was the storage, and management of state. The earler attempts failed because the management of state was untenable with growing application complexity. the `Redux` library was developed precisely to solve this problem. As described earlier, the library is based on the **Flow** architecture pattern, that is based on 3 core concepts.

The redux store contains the application state, and is only mutable inside reducers. The redux reducers are used to modify state, based on the actions,that are published by components, and the componts then in turn react to changes of state.

[[Diagram here or in design chapter - Or move react to theory]]

Our usage of redux creates the state upon application creation, along with a single reducer which is responsible for storing data received from the REST API inside this state. The access to the REST API is implemented in `api/baseApi.js` file. This file contains a metadata object for each entity exposed by the API, and a `Api` class that contains methods for executing common methods.

However, this class is not used directly, its only used from the `actions/apiActions.js` file, that provides methods for contacting the API and then dispatching the results as redux actions. It utilizes the `redux-thunk` middleware.

These actions are then processed by the `dataReducer`, which is responsible for reflecting the changes inside the data store. First versions did so manually, but the growing complexity forced us to adopt the Redux-ORM library. This library provided simple ORM-like experience, and definitely made the development easier.

5.4.4 Material-UI

This library was chosen as a base library of components, upon which our custom components were built. It implements commonly needed components that conform to Google's Material design guidelines[[Cite]]. THese components are also written with first class support of mobile devices, opening up the pathway for making our web application into a Progressive Web Application[[Footnote PWA]].

Core component used in our custom components is the `Paper` component, which creates a surface with the appearance of an elevated piece of paper. This concept of an elevated surface is also at the core of Material design guidelines.

Each custom component is implemented as a list of these `Paper` surfaces, each one holding some kind of information. Most commonly, these surfaces contains tables that show existing entities of a given type, along with a button to add new entity.

EditDialog

The creation and editing of data entities is performed by editing a set of fields inside a modal dialog containing a form. During development, this pattern of a modal dialog with form organically appeared at multiple points, and therefore we decided to implement a single component that would replace individual forms.

This resulted in the implementation of **EditDialog** react component, which performs editing of a javascript object according to properties passed into it.

```
<EditDialog
    open={this.state.open}
    data={this.state.newTrader}
    title="New trader"
    text="Create a new trading account"
    onData={(d) => {
        this.setState({newTrader: d})
    }}
    attrs={[
        {name: "name", title: "Name", type: "text"},
        {name: "api_key", title: "Api key", type: "text"},
        {name: "api_secret", title: "Api secret", type: "text"},
        {name: "exchange", title: "Exchange", type: "select", values: values, text: (e) => e}
    ]}
    onDismiss={(save) => {
        this.setState({open: false});
        if (save) {
            dispatch(postOne(TYPE_TRADER, this.state.newTrader)).then(() => {
                this.setState({open: false});
            })
        }
    }}
/>
```

Listing 5.3: EditDialog for creation of Trader account

The **??** contains code for rendering an **EditDialog**, that performs creation of new trader account. It uses databinding to to pass properties, and callback functions into the dialog.

- open - Whether the dialog is shown or hidden
- data - Data object for editing
- title - Title of the dialog
- text - Information text of the dialog
- onData - Callback function that receives changed object
- attrs - A list of attributes, that should be mapped to editable fields
- onDismiss - Callback function invoked when dialog is dismissed, receives whether the dismissal was performed by the OK button or otherwise.

Chapter 6

Testing and evaluation

This chapter aims to describe the methodology that was used for evaluating the implemented system, and project results as a whole. Another goal is to outline authors experience with the used technolgies, and provide some guidelines for future projects of this type.

6.1 Testing

While 2 primary implementation languages of this project (Rust + React Javascript) have excellent support for unit testing, this approach for ensuring software correctness was used in minimal capacity. The most thoroughly tested part of this project were the `actix_comm` and a `actix_arch` library, that were implemented to serve as a layer between ZeroMQ, Actix and our system. **[[Code coverage table here]]**

To properly test the individual components and ensure their correct behavior, another approach was chosen. because of the fact, that the project uses kubernetes for managmennt, and deployment of the implemented system, allowed us to create separate testing environment. The system was then primarily deployed into this environment, and was observed. This approach allowed us to observe the system in virtually the same environment, in which it will be deployed. Several complex behaviors arisen from interaction of individual components were observed, and subsequently fixed. We wouldnt be able to observe these issues without all the components integrated together, so subjectivelly, we feel that the chosen approach was correct.

Example case - Silent disconnects

One of the observed complex interaction behaviors was the problem of long-lived connections in the environment, in which components are removed and added dynamically. The ZeroMQ library has a complex internal networking layer, that utilizes several sockets, and a separate thread for managing them.

Our `actix_net` uses ZeroMQ, and uses several different virtual socket types provided by it. These sockets are long-lived. They should stay open during the whol time the compoonent runs. However, in the kubernetes environment, components can be killed at virtually any time. This poses a problem. For ROuter and PUB socket types that were used in serviceHandler and publisher actors respectively, the ZeroMQ library does not detect disconnects, or detects them after some time.

This duration between disconnect and detection of it caused an issue with a large amount of missed messages, which destabilized the whole system. THe component that was affected the most, was the **Ingest** actor in the core component, which could not detect missing OHLC updates, since it was a binding SUB socket that created a SUB topology.

The approach, that we chose to fix this issue was to use features of the TCP protocol implementation know as Keepalive. This feature is so called, because it can be usd to keep a TCP conection alive and well, and determine when it was dropped from the other side.

The primary mechanism for performing this service is to periodically send empty TCP packets, that must be acknowledged from the other side. If a sequence of packets is not acknowledged in some pre-defined time, the connection is considered closd from the other side, and is dropped. Usage of this feature required the modification of `tokio-zmq` library, that provides asynchronous implemntation of ZeroMQ sockets. This change allowed setting custom options on the underlying sockets. **[[Something about parameters?]]**

6.1.1 Debugging

However, sometimes it was necessary to observe the internal state of a particular component while it is integrated in the system. Initially, this provided to b quite a difficult challenge, since kubernetes utilizes containers, and accessing the container internals is difficult.

But, after some research, we discovered the Telepresence[[**Footnote**]] command line tool. This tool is written in python, and it creates a vpn tunnel to a container that is running in the kubernetes cluster. Allowing local applications to run as if the were ran inside the actual kubernetes environment.

The fact that we could now run individual components locally, with a debugger greatly simplified the testing and bugfixing process.

6.1.2 Monitoring

Another important aspect was the monitoring of deployed system, with particular [[**Preklad dôraz**]] put upon the monitoring of system performance, and access to individual container logs.

At first, we only used the `kubectl` command line tool. But during during the development the system kept growing in size, and with it the number of components that needed to be monitored has grown too. We solved this issue by installing

a Kubernetes Dashboard[[**Footnote**]] into our cluster, and using this dashboard for monitoring, visualization and log access.

Currently, we use only the metrics provided by kubernetes dashboard out of the box, but in the future, we might export custom metrics specific to our application, that would allow us to better understand the internal system status.

6.2 Implementation evaluation

The need for low latency, scalability and predictable performance was one of the driving forces for multiple key decisions in this project. The implemtation language, communication technology, deployent strategy were all chosen in order to achieve these goals. This section aims to evaluate, whether we achieved these goals.

6.2.1 Measurement methodology

In order to precisely measure key aspects of the implementation, the system had to be modified. the primary change is addition of a `Measurement` component in the system, that receives data from other components and holds the “global„ clock. Because of the distributed nature, the true global clock does not exist, but using a component that will utilize its internal clock to annotate events in the system should provide enough precision for our purposes, since we are measuring on the scale of milliseconds.

Latency measurements

First step to allow for measuring latency is annotating a data update with unique identifier, which will be passed on throughout the system. The components will the use this identifier to uniquely identify all messages associated with a particular update. Then, throuchout normal system function, each component in the system will send messages denoting the state of a particular message along with its identifier to measuring component. This approach might add some latency, but considering that several of the processing steps take several milliseconds, it should be precise enough for our purposes.

Scalability measurements

The scalability measurement is closely associated with the latency measurement. We measured the scalability of the system by adding virtual strategies and corresponding assingments to the system, increasing the load put upon it. Then , we measured, how the system latency was affected. With the increase in computing requirements, the computing resources available to the kubernetes cluster were also increased.

6.3 Performance measurements

The performance measurements were taken in production environment

6.4 Results

[[Results - Performance figures]]

Chapter 7

Conclusion & Future work

The title of this thesis is a Design and development of distributed system that would perform a specific function. Most of the work performed so far was design of the system, and implementation of supporting technologies. The future development plan starts with finishing the `actix-comm` library. Then, designing and implementing a persistence service that would resolve bottlenecks found in current solution.

After issues with this part of the system are resolved, the next step will be implementation of the strategy evaluation service, and implementation of order execution functionality into exchange service. These 2 changes will make the system able to perform its primary function, that being automatic trading based on a strategy.

Next step will be the implementation of the web application that will make the system implemented in previous step accessible to users. This involves the implementation of user management, and inclusion of user data into the rest of the system.

After the web application implementation is complete, the performance of implemented system will be measured. The target of the measurements will be the latency between system receiving new financial data, and system being able to execute orders resulting from evaluation of strategies using this new data. The measurements will be performed with different amount of load on the system, but the system configuration will not be changed. The expected result of these measurements is the invariance of system latency based on system load.