на запуск грейнов. Если вызов должен прийти на тот же самый сервер — среда выполнения это оптимизирует и вызов будет локальным.

Пользой виртуальных актеров в том, что это структура очень легко масштабируется в облаках: хранилища проверяют доступность друг друга и определяют, когда кто-то недоступен, перераспределяют зерна недоступного хранилища между собой. Кластер будет доступен и выполнять свои функции, пока есть хотя бы одно активное хранилище, и при подключении новых участников кластера — они получат свой диапазон грейнов и начнут активно обрабатывать запросы.

Также среда выполнения дает гарантии что при выполнении метода в зерне, никакой другой вызов на тоже самое зерно не придет, то есть среда выполнения гарантирует однопоточное выполнение кода, в рамках одного зерна.

Микросервисная архитектура — вариант сервис-ориентированной архитектуры программного обеспечения, ориентированный на взаимодействие небольших, насколько это возможно, слабо связанных и легко изменяемых модулей — микросервисов. Философия микросервисов гласит, что каждый сервис должен «делать что-то одно, и делать это хорошо» и взаимодействовать с другими сервисами простыми средствами [3]. Основные характеристики микросервисной архитектуры определяются принципами слабой связи между сервисами и высокой связанностью внутри сервиса. Приложение с микросервисной архитектурой отличается четкой структурой сервисов, которые представляют часть функционала приложения.

Философия микросервисной архитектуры ложится на модель структуры Orleans, где зерна выступают в роли сервисов, реализующих узкую часть бизнес логики приложения, которая изолирована в рамках зерна. Коммуникация между зернами-сервисам реализована на уровне среды выполнения Orleans, посредством интерфейсов реализованных зернами.

Однако Orleans не позволяет в полной степени реализовать все возможности микросервисной архитектуры. В первую очередь, разработку приложения не получится легко разделить между несколькими командами разработчиков, поскольку все сервисы описываются в одном проекте.Также одним из плюсов микросервисной архитектуры, который невозможно реализовать с Orleans, является использование различных языков программирования и технологий, подходящих под конкретные задачи. Еще к минусам реализации микросервисной архитектуры с помощью Orleans можно отнести отсутствие возможности развертывания и обновления только части сервисов, поскольку необходимо разворачивать хранилищи целиком со всеми зернами.

Однако микросервисная архитектура с использованием Orleans будет очень полезна для начального этапа разработки микросервисного приложения, поскольку сразу позволяет разбивать логику приложения между зернами, которые в дальнейшем могут стать настоящими сервисами. Также нет необходимости в дополнительных трудозатратах на масштабирование приложения и настройки коммуникации между сервисами, поскольку среда выполнения реализует все эти функции самостоятельно. Данные трудозатраты являются основными причинами отказа от микросервисной архитектуры на начальном этапе разработки, однако Orleans позволяет уменьшить издержки.

Список использованных источников:

1. Microsoft Orleans | Microsoft Orleans Documentation [Электронный ресурс]. – Электронные данные. – Режим доступа https://dotnet.github.io/orleans
2. Getting Started with Microsoft Orleans [Электронный ресурс]. – Электронные данные. – Режим доступа https://medium.com/@kritner/getting-started-with-microsoft-orleans-882cdac4307f
3. Wagner, .NET Microservices: Architecture for Containerized .NET Applications / Bill Wagner, Cesar de la Torre, Mike Rousos – Redmond, Washington, 2018.

СТРУКТУРЫ ДАННЫХ ДЛЯ РАБОТЫ СО СДЕЛКАМИ НА БИРЖЕ

Митьковец А.А.

Белорусский государственный университет информатики и радиоэлектроники
г. Минск, Республика Беларусь

Поттосин Ю.В. – к.ф.-м.н.,доцент

В статье обсуждается применение классических структур данных, позволяющих работать с потоком данных с биржи. Дается структура данных для ведения книги сделок на основе связного списка и хэш-таблицы. Обсуждается проведение сделок в контексте асинхронного программирования, даются рекомендации по использованию futures и promises.

Общение с биржей всегда строится по модели «клиент-сервер». Биржа представляет собой (готовый) сервер, который предоставляет API клиентам. В случае с конвенциальными биржами речь

может идти об интеграции с MetaTrader. Криптобиржи чаще предоставляют REST API для выполнения активных действий и WebSocket API для наблюдения за состоянием биржи. Для общности, будем считать, что абстрактная биржа предоставляет абстрактный синхронный и асинхронный канал для коммуникаций.

Для успешной торговли необходимо моделировать на клиенте состояние биржи – по большей части на основе данных из асинхронного канала. Для того, чтобы успешно оставлять сделки, на клиенте необходимо моделировать существующие списки сделок (order book) – в сторону продажи и покупки для каждого используемого инструмента. Такие списки формируются на основе исходной копии списка и дальнейших асинхронных частичных обновлений. Каждая сделка состоит из цены, объема и направления (покупка или продажа).

С теоретической стороны к идеальному списку предъявляются следующие требования:

1) чтение первых двух-трех вершин за O(1);

2) список всегда отсортирован по ценам;

3) при адресации по цене – вставка, чтение и удаление за O(1).

Простой структурой данных, которая похожа по требованиям, является order-statistic tree [1] – аугментированное красно-черное дерево, которое поддерживает выборку *i*-го крайнего значения за O(log n) в придачу к остальным обыкновенным операциям над деревом тоже за O(log n).

У списков сделок есть своя специфика – во-первых, на ликвидном рынке разница в цене между соседними сделками минимальна, во-вторых, допустимые ценовые уровни зачастую определяются биржей в виде дискретной сетки (с шагом, например, 10 рублей, $0,5).

При учете этой специфики, следующая структура данных позволяет добиться лучших показателей:

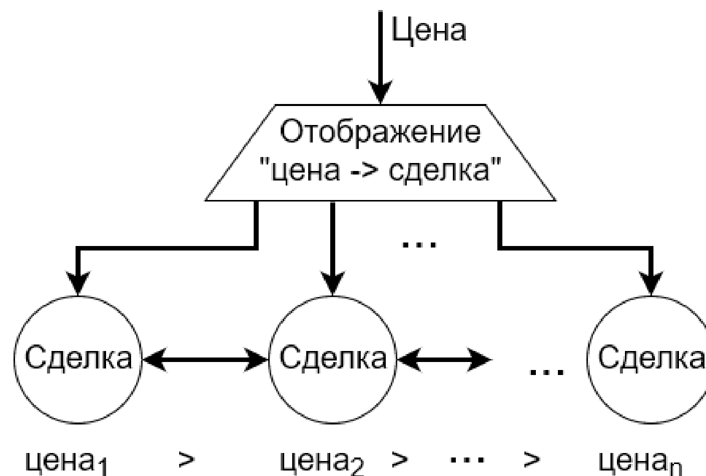

Рисунок 1 – Список сделок на основе двунаправленного списка и отображения

Сделки объединены в связный список, каждая сделка ссылается на две соседние по цене. Хранится только ссылка на первый элемент. Выборка первых N = const элементов происходит за O(1). Отображение из цен на сделки означает выборку по цене и удаление за O(1).

Вставка по цене с теоретической точки зрения занимает O(n) операций, где n – размер списка. На практике, для ликвидного рынка такие вставки, во-первых, редки, а во-вторых, ценовые дыры между сделками малы в силу ликвидности. Если цены на рынке дискретные, то перебор соседних цен через отображение приносит неплохие результаты.

Эмуляция состояния биржи на клиенте зависит от следующих главных асинхронных потоков данных:

1) обновления списка сделок;

2) обновления статуса собственных сделок (проведение торгов по ним, отмена, закрытие).

Поскольку обновления асихнронные, архитектура клиента должна это учитывать и предоставлять интерфейс, который бы скрывал хотя бы часть этой сложности. Например, типичная позиция на бирже включает в себя сделку на вход, экстренную сделку на выход (stop-loss) и take profit сделку на выход. Каждая сделка может быть в пяти состояниях: не выставлена, открыта, частично заполнена, выполнена, отменена. Для трех сделок это означает до 125 возможных состояний, причем почти все переходы случаются асинхронно. Вместе с этим необходимо получать обновления списка сделок.

Для контроля этой сложности хорошо зарекомендовал себя подход с созданием futures для ожидаемых переходов совместно с использованием их в языке, поддерживающем async/await конструкции.

Следующая схема помогает абстрагироваться от прямой работы с асинхронным каналом:

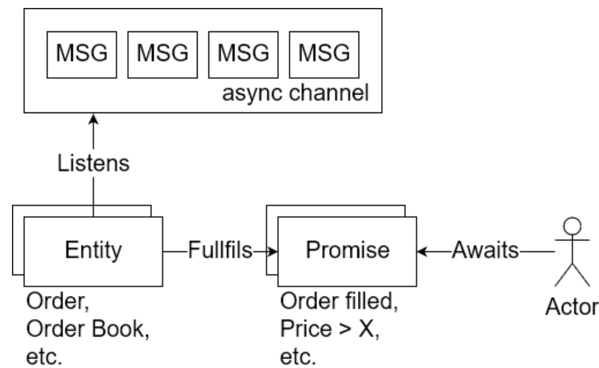

Рисунок 2 – сокрытие асинхронного канала при помощи promise/future без callbacks

Списку сделок достаточно иметь один вид следующие Promise: «цена покупки больше/меньше заданной». Не рекомендуется ждать строго указанной цены – в случае движения цены Promise сработает, однако он не сработает, если сделка окажется отмененной и этот ценовой уровень займет сделка в противоположную сторону.

Информация о собственных сделках также поступает из асинхронного источника, поэтому для них тоже нужны Promises:

1) Сделка отменена. Биржа может отменять сделки по своей инициативе (margin call, недостаточно средств и так далее).

2) Сделка частично выполнена. Может случиться при большом объеме, когда контрагентами по покупке/продаже выступают несколько других участников рынка.

3) Сделка полностью выполнена.

Имея следующие примитивы, можно реализовывать устойчивые и стабильные программы по работе с биржей. Программный код становится похож на набор последовательных инструкций – чего не происходит при прямом чтении из асинхронного канала или при использовании подхода с callbacks.

```
order_book = await OrderBook.create('USD/BYN')
price = await order_book.buy_price_lt(2.00)
buy_order = await Order.buy(price,  1000)
await buy_order.one_fill

take_profit = order_book.sell_price_gt(2.10)
stop_loss = order_book.sell_price_lt(1.90)
outcome = await first(take_profit, stop_loss)
cancel_buy = buy_order.cancel()

await (await Order.sell_all(outcome.value))
await cancel_buy
```

Рисунок 3 – псевдокод покупки и последующей продажи BYN

Список использованных источников:

1. Cormen, T. Introduction to Algorithms / T. Cormen, C. Leiserson, R. Rivest, C. Stein // MIT Ppess. – 1990. – P. 339–345.

УСТРАНЕНИЕ ДИСТОРСИИ НА ШИРОКОУГОЛЬНЫХ ИЗОБРАЖЕНИЯХ

Некревич С.А., Сапронова Ю.И.

Белорусский государственный университет информатики и радиоэлектроники
г. Минск, Республика Беларусь

Лукашевич М.М. – к.т.н., доцент

Показан метод нахождения суммарной дисторсии, основанный на измерениях искажений тестового изображения (шахматной доски или квадратной сетки). Для устранения дисторсии (получения изображения, близкого к идеальному) необходимо к координатам каждой точки реального изображения добавить величину полученной с помощью тестового изображения дисторсии. Рассмотрены проблемы, которые могут возникнуть в результате произведенной коррекции.