Разработка модели и архитектуры для абстрактной системы с учётом потокобезопасности

Аннотация

Этот материал посвящён разработке архитектуры базы для многопоточных систем, сочетающей в себе потокобезопасность и отсутствие блокировок. В качестве примера готовой системы был выбран сервер для онлайн игры.

В статье рассмотрены основные существующие модели многопоточных прложений, способы синхронизации внутри них и представлены сравнительные таблицы. В качестве результата, разработано решение сервера онлайн игры на основе подготовленной системы.

Введение

При разработки многопоточных систем всегда встаёт проблема масштабируемости мощности выделяемой на бизнес логику при необходимости сохранения связности. В частности, в игровом проекте необходимо обеспечить единое игровое время для различных логических потоков обрабатывающих различные участки игровой карты для сохранения так называемого бесшовной карты. Так, как невозможно использовать один поток для обработки всех игроков на карте без деления этой карты на локации за адекватное время. Самым трудным моментом являются взаимодействия игроков, обрабатывающихся в разных логических потоках.

Актуальность

Распространение широкополосного интернета привело к увеличению числа онлайн игр привело к взрывному росту потенциальных игроков в онлайн игры и соответственно, числу онлайн-ореинтированных. Произошло несколько

бумов в разработке онлайн игр, имеющих больший срок жизни чем оффлайн побратимы: WEB, PC, консольный, а теперь и мобильный рынок наполнены огромным числом онлайн игр.

В соответствии с числом игр, растут и запросы потребителей на скорость и число одновременных игроков. Появилось большое число готовых серверных решений и даже услуги "сервер как услуга" где лишь необходимо было подготовить игровой графику и логическую начинку.

Тем не менее, так и не было создано оптимальное решение, которое могло бы удолетворить каждого разработчика и позволяло легко создать массовую многопользовательскую онлайн игру с достаточным уровнем уникальности.

Сравнение существующих архитектур с точки зрения многопоточного программирования

Монолитная архитектура

Тип архитектуры программная системы при котором она является единой и практически неделимой сущностью.

Сервис-ореинтированная архитектура

Модульный подход к разработке программного обеспечения, основанный на использовании распределённых, слабо связанных заменяемых компонентов, оснащённых стандартизированными интерфейсами для взаимодействия по стандартизированным протоколам.

Микросервисная архитектура

Архитектура в которой единое приложение представляется в виде набора небольших сервисов, каждый из которых работает в собственном процессе и коммуницирует с остальными используя легковесные механизмы. Данные аспекты архитектуры указаны в Evolve the Monolith to Microservices with Java and Node.

Критерии сравнения аналогов

Согласованность

Данный критерий оценивался по тому, насколько просто в архитектуре поддерживать согласованность кода, обрабатывать ошибки и т. д. Он важен в связи с необходимостью поддерживать и расширять проект в течении его жизни.

Доступность

Доступность подразумевает собой возможность функционирования системе при отказе части одной из её частей. Системы с низкой доступностью чаще всего бесмысленны так, как не могут взаимодействовать с пользователем и выполнять свою задачу.

Расширяемость

Расширяемость подразумевает собой возможность добавлять новые или изменять уже готовые функции в системе. Она очень похожа на согласованность, но это скорее архитектурный аспект.

Масштабируемость

Возможность размещения модулей системы на отдельных серверных узлах для увеличения производительности системы в целом. Одна из целей работы это создание системы, которая будет легко масштабируема.

Параллелизм

Параллелизм показывает наиболее привычный способ разнесения функционала по различным потокам. Стоит понимать, что данная характеристика не является абсолютным так, как ничто не мешает нам запустить все микросервисы в одном потоке. Что и является основным требованием системы.

Таблица сравнения по критериям

Критерий\Архитектура	Монолитная	Сервис-ореинтированная	Микросервисная
Согласованность	Высокая	Средняя	Низкая
Доступность	Низкая	Средняя	Высокая
Расширяемость	Низкая	Средняя	Высокая
Масштабируемость	Низкая	Высокая	Высокая
Параллелизм	Синхронизация средствами языка	Синхронизация средствами языка / Модуль-поток	Модуль-поток

Выводы по итогам сравнения

В соответствии с приведенными выше данными каждое из решений имеет своими плюсы и свои минусы. Например, монолитная архитектура имеет высокую согласованность, а значит, гораздо проще в реализации, управлении и развёртывании. А микросервисная архитектура, хотя и не может быть легко развёрнута, но зато позволяет обновлять приложение по частям и даже если один из них недоступен, это не приводит к сбою всего приложения. Сервис-ореинтированная предоставляет собой некий баланс между вышеперечисленными так, как её модули более функциональны чем у микросервисной, но всё ещё не так сильно связаны, как у монолитной. Это хорошо подчёркивается в Service Oriented Architecture with Java. Самые интересные характеристики это масштабируемость и параллелизм. Они показывают возможность разделения мощностей при сохраненнии логической

целостности системы. Хотя микросервисная архитектура и будет иметь преимущества за счёт большой гибкости, но намного труднее из-за низкой связности. Обратное верно и для монолитной архитектуры. Поэтому, была выбрана сервис-ореинтированная архитектура, как наиболее компромисный вариант.

Выбор метода решения

В результате обзора существующих архитектур с точки зрения многопоточного программирования были рассамотрены их характеристики из которых становилось ясна их низкая эффективность с точки зрения соотношения скорости разработки и производительности результата.

Поэтому, цель работы — разработка системы на основе архитектуры, которая сможет предоставить легко расширяемую и масштабируемую многопоточную среду при минимальных затратах времени разработчика.

При этом решение должно обладать следующими свойствами:

- Система должна быть доступной и продолжать функционировать даже при отказе одной из её частей.
- Должна сохранятся прозрачность и возможность отследить коммуникации между различными потоками.
- Необходим широкий набор базовых компонентов для построения универсального фреймворка без привязки к бизнес процессам.
- Поддержка unit тестов.
- Система должна иметь возможность добавления масштабируемости по разным машинам.
- Код должен быть согласован и в едином стиле.

Описание метода решения

В результате, было решены выбрать сервис-ореинтированную архитектуру и для синхронизации потоков взята за основу модель сообщений. Сервис ореинтированная архитектура представляет собой разумный баланс: её модули более функциональны чем у микросервисной, но всё ещё не так сильно связаны, как у монолитной.

Основная идея - взаимодействие потоков через сообщения. MessageSystem — объект для обмена данными. Address и Abonent. Аналогия с почтой. Message — иерархия наследования сообщений. AddressService. В идеале, абоненты могут находиться на разных физических машинах.

На иллюстрации вы можете увидеть принцип работы. Существуют два сервиса: Frontend и AccountService каждый из которых работает в своём потоке. Всё обращение между ними происходит через MessageSystem - при необходимости в неё передаются объекты с запросами данных или самими данными, которые добавляются в очередь сообщений каждого сервиса, а при возможности обрабатываются. Этот вариант работы называется асинхронным многопоточным.

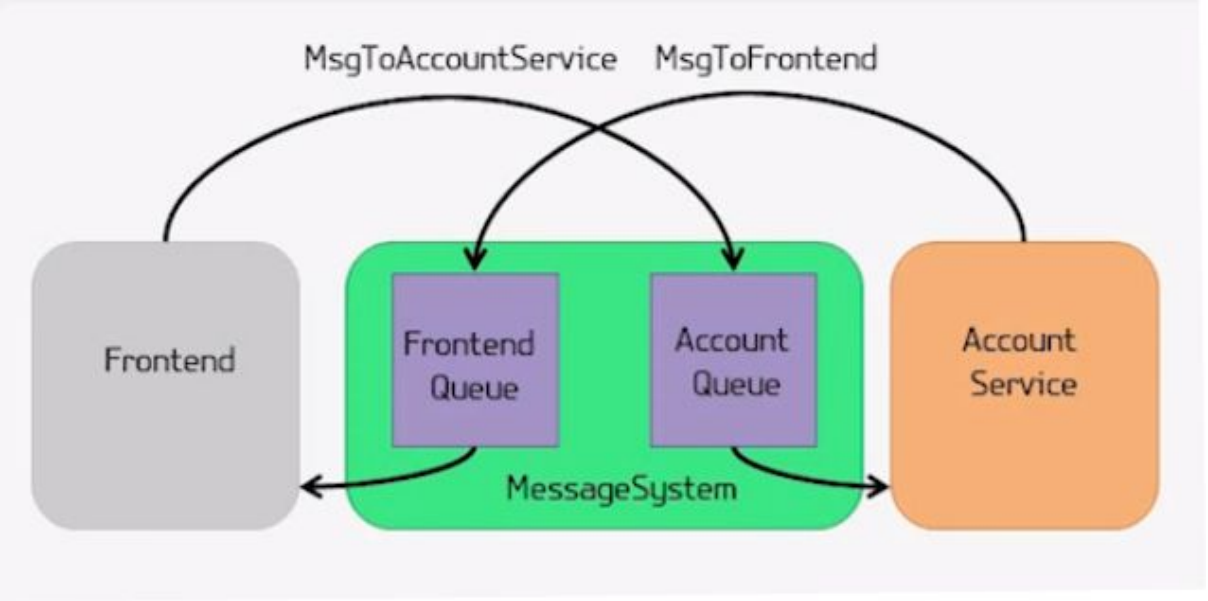

Рисунок 1 – Пример работы системы сообщений

Основной функционал системы

Разработанный код является основой для многопоточной системы, которая ореинтирована на добавление к себе различных модулей (сервисов) для решения задач.

В реализованном мною примере это система использоуется в качестве базы для серверной части многопоточной игры для чего были подготовлены сервисы: аккаунтов (работа с СУБД посредством Object-Relational Mapping), сетевого транспорта (передача информации игрокам), игровой логики, AI

(элементарые мобы).

Клиентская часть была выполнена на Unity3d с использованием C#; она и сетевая часть не являются предметом этой статьи.

Архитектура программной реализации

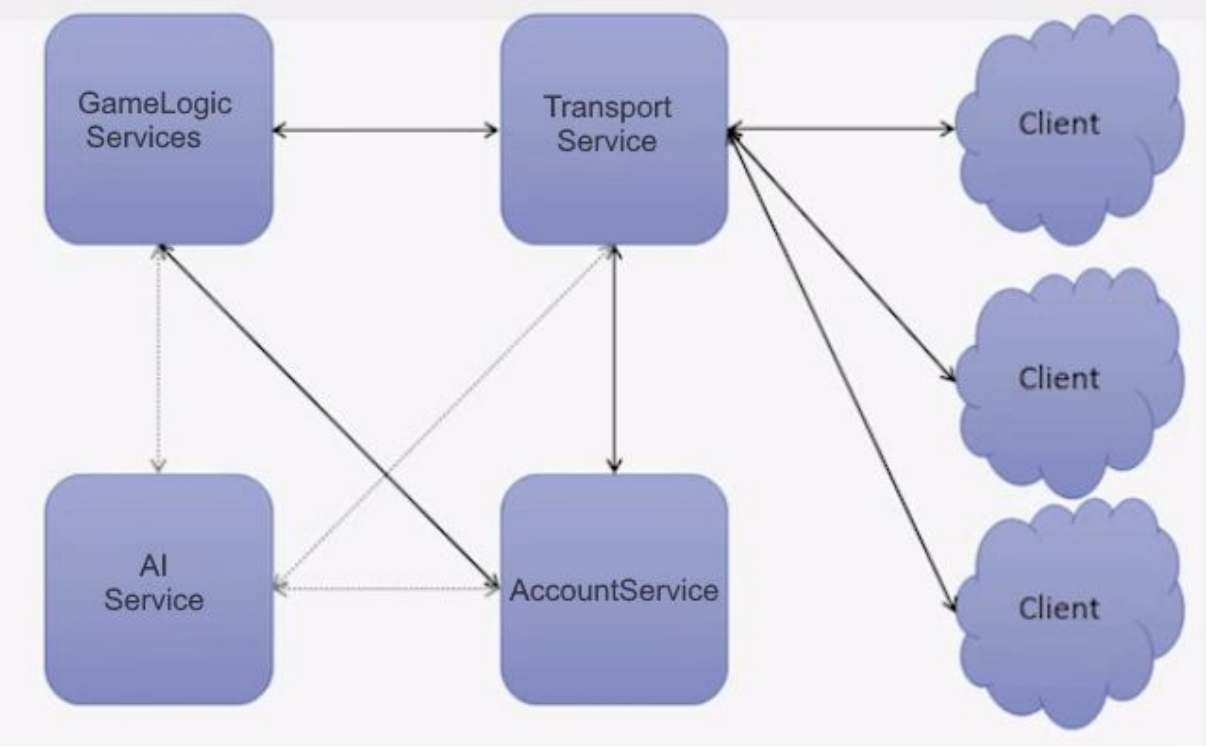

Рисунок 2 – Архитектура примера на основе системы

Архитектура представляет собой набор основных игровых сервисов, каждый из которых выполняется в отдельном потоке (в действующей системе инстансов одного сервиса может быть несколько для масштабирования). ####Transport Service Сервис, который отвечает за связь (отправку и получение данных) клиента (игрока) и сервера (игровой системы). ####GameLogic Service Сервис, в котором находится базовая игровая логика: физика, списки объектов, различные таймеры и триггеры событий. Именно в этот сервис отправляются сообщения о перемещении или атаках. ####Accounts Service Сервис, который записывает и запрашивает информацию в СУБД. Отвечает за загрузку и сохранение игрового прогресса пользователей. ####AI Service Сервис, который обращаясь к GameLogic Service опредеяет поведение игровых сущностей, не контролируемых игроками. Линии представляют собой основные каналы

отправки и получения сообщений.

Тестирование

В основном, базовая система покрыта лишь unit тестами направленными на проверку основных функций: отправка и получение сообщений.

Основной интерес представляют тесты уже построенного примера сервера игры.

Для примера выполнялось нагрузочное тестирование для определения оптимального числа объектов игровой логики (использовались эмуляторы активности нескольких сценариев игрока); проверка маршрутизации сообщений между инстансами игровой логики для потокобезопасных операций над игровыми объектами; поиск пути для ботов и стресс тесты на поведение системы при отключении части модулей.

Используемые технологи

Разработка велась в Intellij Idea с плагинами для тестирования и version control system (GIT). Код был написан на языке Java, используется СУБД MySQL.

Для тестирования используются библиотека JUnit. Она позволяют тестировать отдельные модули и методы системы, проверяя целостность работы, после обновлений.

В качестве io фреймворка был выбран Netty, как уже устоявшееся решения для сетевых приложений.

С целью упрощения развёртывания системы был подготовлен docker образ. Это упрощает и ускоряет перезапуск после критической ошибки, а также позволяет каждому иметь одинаковую среду для тестирования и разработки.

Выводы

Целью работы была — подготовка системы для разработки сервера онлайн игры, которая бы базировалась на многопоточной потокобезопасной архитектуре.

В процессе были рассмотрены различные архитектуры многопточных приложений и способы синхронизации. Разработана система классов для выбранной архитектуры и подготовлен на её основе пример сервера для многопоточной онлайн игры.

За недостатками решения можно принять то, что разработка велась лишь одним человеком без взгляда со стороны и всё же была заточена на конкретной пример. Разработка одним человеком может проявиться в ошибках и ясных лишь ему одному местах, а направленность под определённый пример отсутствием нужных уровней абстрации и некоторой гибкости.

Дальнейшими направлениями развития полученного решения является разработка полноценной онлайн игры псевдо реального времени.

Источники

1. paper_base/e-Reference.pdf
2. paper_base/akka-at-yandex.pdf
3. paper_base/microservices-for-java-developers.pdf
4. paper_base/Service Oriented Architecture with Java.pdf
5. paper_base/Evolve the Monolith to Microservices with Java and Node.pdf
6. https://habrahabr.ru/company/mailru/blog/259125/