КУРСОВАЯ ЛАЛАЛА

СОДЕРЖАНИЕ ЛАЛАЛА

1.Введение

В распределённых приложениях, таких как мессенджеры, критически важно выбрать оптимальную архитектуру для взаимодействия между клиентом и сервером. Разработчики либо используют транспортные протоколы TCP/UDP напрямую, создавая собственные протоколы обмена данными, либо применяют высокоуровневые фреймворки удалённого вызова процедур (RPC), такие как gRPC.

Выбор между низкоуровневым решением на основе TCP/UDP и RPC-фреймворком gRPC оказывает значительное влияние на сложность разработки, производительность приложения и последующее сопровождение.

В этом исследовании сравниваются две архитектуры сетевого взаимодействия: самописную на базе сокетов TCP/UDP и современную RPC-архитектуру на базе gRPC в контексте реализации клиент-серверного мессенджера.

**Актуальность темы:**

Данная тема актуальна, так как работа с сокетами осуществляет полный контроль над передачей данных, однако усложняет процесс разработки. С другой стороны, gRPC обещает упростить взаимодействие между процессами, хотя и требует дополнительных ресурсов.

**Цель исследования:**

Цель данного исследования — оценить эффективность и целесообразность использования двух архитектур для создания мессенджера: на основе сокетов TCP/UDP и с использованием RPC и фреймворка gRPC. Под эффективностью принимаются такие параметры, как производительность, использование ресурсов, а также качество архитектуры с точки зрения масштабируемости, безопасности и удобства сопровождения.

**Задачи исследования:**

Для достижения поставленной цели в работе будут решены следующие задачи:

1. Изучение и описание принципов работы архитектур, основанных на сокетах TCP/UDP и технологиях RPC, включая фреймворк gRPC. Выявление характеристик, преимуществ и недостатков.

В рамках нашего исследования планируется разработать два варианта мессенджера, которые будут иметь одинаковую функциональность. Один будет основан на классической клиент-серверной архитектуре с использованием TCP/UDP, а другой — на технологии RPC-вызовов с применением gRPC.

Чтобы обеспечить равные условия тестирования, будет создана воспроизводимая среда, которая будет включать скрипты сборки и контейнеризацию с использованием Docker. Это позволит запустить оба варианта мессенджера в одинаковых условиях, что позволит сравнить эффективность и особенности.

Чтобы провести объективное сравнение двух архитектурных решений, необходимо выполнить ряд экспериментов, которые будут включать в себя 3 параметра:

- задержка при передаче сообщений;

- пропускная способность;

- потребление ресурсов, таких как центральный процессор и оперативная память.

На основе полученных данных проводится сравнительный анализ двух архитектурных решений.

В заключение необходимо сформулировать рекомендации по использованию архитектуры, основанной на TCP/UDP или gRPC, в различных условиях и сценариях.

**2. Теоретическая часть**

**2.1 TCP/UDP протокол**

TCP (Transmission Control Protocol) и UDP (User Datagram Protocol) представляют собой фундаментальные протоколы транспортного уровня, используемые в разрабатывании клиент-серверных приложений. Архитектура, основанная на TCP/UDP, предполагает непосредственное взаимодействие приложения с сокетами для отправки и получения данных.Разработчики могут самостоятельно выбирать прикладной протокол обмена сообщениями, который будет использоваться поверх TCP/UDP, а также реализовывать логику обработки данных. Рассмотрим подробнее особенности такого подхода.

TCP (Transmission Control Protocol) — это протокол, обеспечивающий надёжное соединение, гарантируя доставку данных в исходном виде, а также контроль за перегрузкой. В архитектуре клиент-сервер, основанной на TCP, сервер прослушивает порт, ожидая входящих подключений клиентов.После установления соединения клиент и сервер обмениваются последовательностью байтов. В приложении-мессенджере, использующем протокол TCP, каждое текстовое сообщение или команда отправляются как пакет байтов через сокет. На принимающей стороне необходимо считывать поток байтов и выделять границы сообщений.Поскольку TCP это потокоориентированный протокол, то часто требуется реализация фрейминга (разделения) сообщений. Например, каждое сообщение предваряется заголовком с указанием длины или специальным разделителем, чтобы получатель собирал сообщение из потока.

UDP-сокеты представляют собой механизм передачи данных, функционирующий без установления соединения и оперирующий отдельными пакетами. Этот протокол не обеспечивает гарантированную доставку данных, что делает его проще и быстрее по сравнению с TCP. Однако надёжность и порядок доставки будут обеспечены на уровне приложения, если это необходимо.

В контексте мессенджера UDP может быть полезным в ситуациях, когда потеря пакетов некритична или когда требуется минимизировать задержку. Например, использоваться для передачи больших двоичных данных, таких как файлы, мультимедиа или потоковая информация, где потеря несущественна. Также UDP подходит, если разработчик готов самостоятельно реализовать простейший механизм подтверждения получения данных.

В рассматриваемом проекте исходный мессенджер использует UDP для ускорения передачи файлов, дополняя примитивным механизмом ACK (подтверждения получения) для каждого блока данных.GitHub.com предоставляет реализацию собственного протокола на основе UDP. Отправитель делит файл на блоки и передаёт в виде датаграмм, а получатель отправляет короткие подтверждения (ACK). В случае неполучения подтверждения блок может быть отправлен повторно, что обеспечивает надёжность, хотя и приводит к увеличению задержки и усложнению логики.

В некоторых проектах применяется комбинированное использование протоколов TCP и UDP. Это позволяет использовать преимущества каждого из них.В нашем случае, сообщения, требующие гарантированной доставки и правильного порядка, передаются по протоколу TCP. Передача файлов осуществляется через протокол UDP для повышения скорости.На сайте GitHub.com можно найти гибридные архитектурные решения, которые используются в различных областях. Например, в протоколах передачи голоса и видео (VoIP, стриминг) используется протокол UDP для передачи мультимедийных данных. Для служебных команд и текстовых данных применяется протокол TCP.Однако использование нескольких протоколов усложняет проект, так как требуется поддерживать две схемы связи и обеспечивать синхронизацию. Например, для протокола UDP необходимо знать адреса клиентов, которые передаются через TCP-соединение.

Программирование на сокетах TCP/UDP предоставляет широкие возможности, однако требует от разработчика тщательного подхода к реализации. Необходимо учитывать сетевые ошибки и тайм-ауты, обрабатывать разрывы соединений и корректно освобождать ресурсы.Для упрощения проектирования разработаны библиотеки, такие как Boost.Asio для C++, которые предоставляют асинхронные операции ввода-вывода и интегрируются с циклом событий. Однако даже при использовании таких библиотек разработчику необходимо вручную реализовывать обработчики событий, таких как подключение нового клиента, получение и отправка данных, а также следить за соблюдением протокола приложения.Например, исходный проект мессенджера использует асинхронную модель с использованием Boost.Asio, учитывая такие аспекты, как выравнивание STUN-пакетов в соответствии со стандартом RFC 5389, ручную сериализацию и десериализацию сообщений, обработку сетевых ошибок.Это указывает на то, что реализация проекта на низком уровне представляет собой сложную задачу. Для создания такого проекта необходимы познания в области сетевых протоколов и тщательно подходить к процессу кодирования. В противном случае могут возникнуть проблемы, связанные с совместимостью и надёжностью.

**Преимущества и недостатки подхода**

Архитектуры, основанные на использовании чистого TCP или UDP пользуются рядом преимуществ. Они предоставляют полный контроль над процессом передачи данных. Разработчик может оптимизировать собственный протокол в соответствии с конкретными потребностями, минимизируя накладные расходы в сообщениях и достигая минимальных задержек.

Кроме того, такие архитектуры не зависят от внешних компонентов, что упрощает процесс отладки сетевого трафика. Например, с помощью таких инструментов, как tcpdump или Wireshark, можно анализировать передаваемые пакеты.Также для реализации этих архитектур не требуется использование специализированного кода или фреймворков — достаточно стандартной библиотеки сокетов.

Однако у этого метода существует явный недостаток — сложность разработки и поддержки. Разработчику приходится самостоятельно решать задачи, которые уже решены в высокоуровневых фреймворках, такие как маршрутизация сообщений, форматирование и обработка исключений.

При непосредственной работе с сокетами код становится сложным и включать повторяющиеся фрагменты, необходимых для преобразования структур в байты и обратно, а также для обработки состояний соединения.

Также в обязанности разработчика входит обеспечение безопасности. Например, если необходимо использовать шифрование, то требуется вручную интегрировать TLS или другие методы.

Работа с необработанными TCP/UDP представляет собой более сложный и трудоёмкий подход. Однако он оправдан, если требуются специфические оптимизации или отсутствует готовое промежуточного программного обеспечения.

**2.2. RPC и фреймворк grpc.**

RPC (Remote Procedure Call) представляет собой подход к разработке распределённых систем, который разрешает программе обращаться к функции, выполняемой на другом компьютере (сервере), как к локальной.

В классическом варианте реализации RPC разработчик определяет интерфейс (набор процедур и методов) и использует специальный механизм, который автоматически передаёт параметры через сеть и получает результат.RPC охватывает технологии, которые разрешают программам обращаться к функциям и процедурам, расположенным в другом адресном пространстве, будь то удалённые узлы или даже другой процесс на том же компьютере.

Архитектура RPC традиционно состоит из двух компонентов:

1. Сетевой протокол, который обеспечивает клиент-серверный обмен сообщениями о вызовах и ответах.

2. Язык описания данных (средство сериализации) для представления параметров и возвращаемых значений в формате, который может быть перенесён между устройствами.

Принцип работы RPC (Remote Procedure Call) заключается в следующем: клиент вызывает удалённую функцию через специальный интерфейс, называемый «stub». Этот вызов перехватывается «stub» клиента, который формирует сетевой запрос, включая имя процедуры и параметры, и передаёт на сервер.На стороне сервера запрос перехватывается «stub» сервера, который декодирует, вызывает соответствующую локальную функцию и передаёт результат обратно клиенту. Для программиста этот процесс остаётся невидимым, так как работая с RPC как с локальной функцией, нет нужды вникать в детали передачи данных по сети.

Благодаря RPC становится проще организовывать распределённые вычисления и клиент-серверные архитектуры, так как взаимодействие между компонентами становится похожим на вызовы локальных функций. Разработано множество версий RPC, начиная с устаревших (Sun RPC, CORBA, DCOM) и заканчивая последними, такими как JSON-RPC, Apache Thrift и gRPC.

gRPC — это один из самых последних и популярных фреймворков RPC, разработанный компанией Google в 2015 году. Технология быстро получила популярность и стал использоваться такими компаниями, как Netflix, Docker, Spotify, Dropbox и другими, которые перевели архитектуру микросервисов на эту технологию.

По умолчанию для кодирования сообщений в gRPC применяется протокол Protocol Buffers (Protobuf), разработанный компанией Google. Это двоичный формат сериализации данных, который отличается компактностью и высокой скоростью кодирования и декодирования. В сочетании с протоколом HTTP/2 он демонстрирует впечатляющие результаты.

Protobuf генерирует небольшие по размеру сообщения и сериализует данные, что делает его частым выбором для сред с ограниченной пропускной способностью, таких как мобильные сети. Это делает взаимодействие через gRPC эффективным по сравнению с альтернативными вариантами, такими как REST/JSON API или самописные текстовые протоколы, особенно при большом количестве сообщений.gRPC использует универсальный язык описания интерфейсов — IDL (Interface Definition Language), который не зависит от конкретного языка программирования. Разработчики создают файлы .proto, описывающие сервисы (наборы RPC-методов) и типы сообщений (структуры данных для запросов и ответов).

На основе этих файлов специальные инструменты, такие как protoc с плагинами, генерируют код для выбранного языка программирования. В результате получаются классы сообщений, абстрактный класс сервиса (для сервера) и клиентский класс-стаб.Использование единого proto-файла гарантирует, что клиент и сервер «говорят на одном языке». Такой подход экономит время, особенно в архитектурах с большим количеством сервисов. Кроме того, он повышает читаемость кода, так как интерфейс легко просматривается в proto-файле.Это открывает новые возможности для реализации сценариев, требующих непрерывной передачи данных в режиме реального времени. Например, в чат-приложениях можно передавать поток сообщений, обновлять статус или передавать файл по частям без необходимости многократного вызова отдельных методов.Особенно это актуально для мессенджеров: gRPC позволяет организовать потоковую передачу сообщений сервера к клиенту по мере появления или встроенную потоковую передачу файла в RPC-вызов. Кроме того, gRPC предлагает встроенные средства управления вызовами, такие как поддержка тайм-аутов и отмену вызовов, что повышает стабильность распределённых проектов.

Преимуществом gRPC также является способность обеспечивать безопасность на транспортном уровне с помощью встроенного фреймворка, избегая необходимости вручную шифровать данные на уровне приложения. В отличие от этого, при использовании самописных решений на основе TCP/UDP добавление шифрования требует усилий. Например, необходимо использовать OpenSSL или аналогичный инструмент и интегрировать в схему обмена данными.

Подводя итоги, можно сделать вывод, что GRPC имеет высокую производительность благодаря использованию HTTP/2 и Protobuf, что обеспечивает удобство разработки за счёт генерации кода и абстракции вызовов и предлагает готовые и проверенные решения безопасности, в то время как самописные архитектуры могут быть уязвимыми и сложными для защиты. Это делает GRPC привлекательным выбором для современных распределённых архитектур.