ОГЛАВЛЕНИЕ

1. Постановка задачи

Целью данной работы является разработка клиент-серверного программного обеспечения, предназначенного для распределенной обработки данных в условиях ненадежного сетевого соединения. Приложение должно обеспечивать возможность эффективного распределения задач между сервером и несколькими клиентами, а также гарантировать надежную доставку данных и сбор результатов даже при наличии потерь пакетов в сети.

В соответствии с поставленной целью, к разрабатываемому приложению предъявляются следующие функциональные требования:

- Двухкомпонентная архитектура: Приложение должно состоять из двух частей: сервера и клиента, взаимодействующих по сети.
- Многопоточность: Обеспечение многопоточной обработки запросов как на стороне сервера, так и на стороне клиента.
- Распределение задач: Сервер должен иметь возможность рассылать задачи клиентам для обработки. В качестве примера задач могут выступать операции линейной алгебры над матрицами.
- Регистрация и управление клиентами: Клиенты должны иметь возможность регистрироваться на сервере, сообщать о своих вычислительных мощностях (количестве и типах обрабатываемых задач) и получать задачи для обработки.
- Обработка задач на клиенте: Клиент должен уметь принимать задачи от сервера, распределять их по доступным вычислительным ядрам (Execution Objects) и возвращать результаты обработки на сервер.
- Сбор и хранение результатов: Сервер должен обеспечивать сбор результатов обработки задач от всех клиентов и сохранение их в базу данных (CSV, JSON, YAML).
- Статистика и логирование: Приложение должно вести логирование выполненных задач, а также собирать статистику по работе сетевого уровня (потери пакетов, перепосылки, ошибки целостности).

Учитывая, что канал связи между сервером и клиентом не гарантирует надежной доставки пакетов, необходимо разработать двухуровневый стек протоколов поверх UDP:

Уровень L2 (Транспортный уровень):

- Обеспечивает надежную передачу пакетов данных фиксированного размера.
- Реализует механизмы контроля целостности, нумерации пакетов, подтверждения приема и повторной передачи потерянных пакетов.
- Производит фрагментацию и сборку данных, которыми оперирует уровень L3.

Уровень L3 (Логический уровень):

- Отвечает за реализацию прикладной логики взаимодействия между сервером и клиентом.
- Определяет следующие типы сообщений:
- Регистрация/дерегистрация клиента на сервере;
- Запросы на обработку задач и подтверждения обработки;
- Отправка результатов обработки;
- Сбор и передача статистики работы транспортного уровня (L2).

2. Описание протокола

2.1 Архитектура разработанного протокола

Протокол построен по принципу двухуровневой архитектуры, состоящей из канального и сессионного уровней. Такой подход позволяет эффективно организовать как управление соединениями, так и обмен данными внутри этих соединений.

2.1.1 Канальный уровень

На этом уровне решаются задачи установки, поддержания и разрыва соединения между клиентом и сервером. Здесь происходит инкапсуляция данных в UDP-пакеты, обеспечивается контроль целостности передаваемых данных.

2.1.2 Сессионный уровень

Внутри каждого установленного UDP-соединения протокол оперирует понятием "сессия". Сессия представляет собой логический канал обмена данными, идентифицируемый уникальным номером. Использование сессий позволяет организовать параллельную обработку запросов и ответов в рамках одного UDP-соединения, что значительно повышает эффективность использования сетевых ресурсов.

2.2 Реализация протокола и его особенности

При реализации протокола особое внимание уделялось вопросам его гибкости, масштабируемости и устойчивости к ошибкам, что было достигнуто за счет ряда архитектурных решений.

2.2.1 Поддержка параллельных передач

Ключевым преимуществом протокола является поддержка параллельных передач данных в рамках одного UDP-соединения (с использованием одного порта). Это достигается за счет использования сессий, каждая из которых функционирует независимо от других, обрабатывая свой собственный поток данных. Такой подход позволяет избежать блокировок и простоев, характерных для протоколов с последовательной обработкой запросов.

2.2.2 Устойчивость к ошибкам

Протокол передачи сообщений обладает встроенным механизмом повторной передачи пакетов, обеспечивающим надежную доставку данных даже в условиях ненадежной сети. В случае потери пакета отправитель инициирует его повторную отправку, что гарантирует получение данных получателем.

2.2.3 Гибкость и масштабируемость

Архитектура протокола предусматривает возможность легкого расширения функциональности путем добавления новых типов сессий. В текущей реализации уже доступны:

- **PingSession**: используется для проверки доступности узла и согласования версии протокола.
- **MessageSession**: предназначена для надежной передачи сообщений произвольной длины.

В дальнейшем, при необходимости, возможно добавление других типов сессий, например, StreamSession для организации потоковой передачи данных.

2.3 Тестирование с использованием UDP-proxy

Для всестороннего тестирования протокола был разработан UDP-proxy, эмулирующий различные сетевые условия, включая задержки и потери пакетов. Проведенные тесты подтвердили высокую устойчивость протокола к ошибкам и его способность надежно функционировать в неблагоприятных условиях.

2.4 Устройство работы приложения

Работа сетевого приложения, построенного на базе разработанного протокола, организована следующим образом:

Главный поток приложения:

- Принимает все входящие UDP-пакеты.
- Анализируя структуру пакета, определяет его тип – канальный или сессионный.
- Для канальных пакетов проверяется наличие установленного соединения. Если соединение отсутствует, пакет игнорируется.
- Сессионные пакеты перенаправляются в соответствующие объекты Connection, отвечающие за обработку данных в рамках конкретного UDP-соединения.

Объект Connection:

- Является центральным элементом архитектуры приложения на стороне клиента.
- Отвечает за обработку всех данных, полученных от определенного удаленного узла.
- Хранит очередь входящих пакетов и обрабатывает их последовательно в отдельном потоке.
- Для управления сессиями используется объект SessionManager.
- Автоматическое удаление неактивных сессий обеспечивается объектом SessionKiller.

2.4.1 Компонент SessionManager

Данный компонент играет роль прокси-сервера, перенаправляя поступающие сессионные пакеты в соответствующие им сессии. Перед передачей пакета SessionManager проверяет наличие активной сессии с указанным в пакете идентификатором. Если сессия не найдена, пакет отбрасывается.

2.4.2 Компонент SessionKiller

Объект SessionKiller отвечает за своевременное освобождение ресурсов, занимаемых неактивными сессиями. Он отслеживает время последней активности каждой сессии и автоматически удаляет ее по истечении заданного таймаута.

2.5 Примеры и схемы взаимодействий

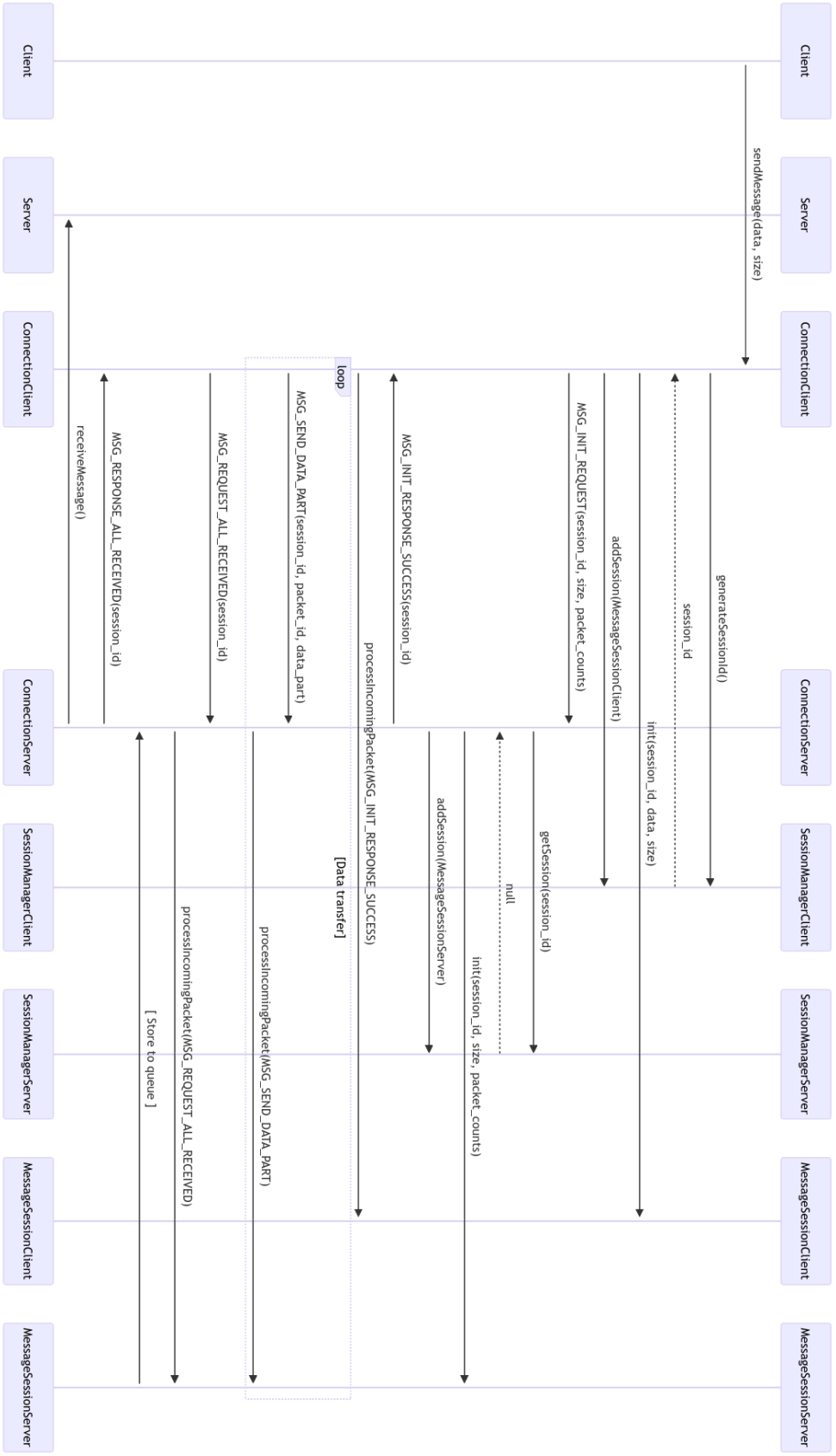

Рисунок 1. Взаимодействие программных компонентов при передаче сообщения

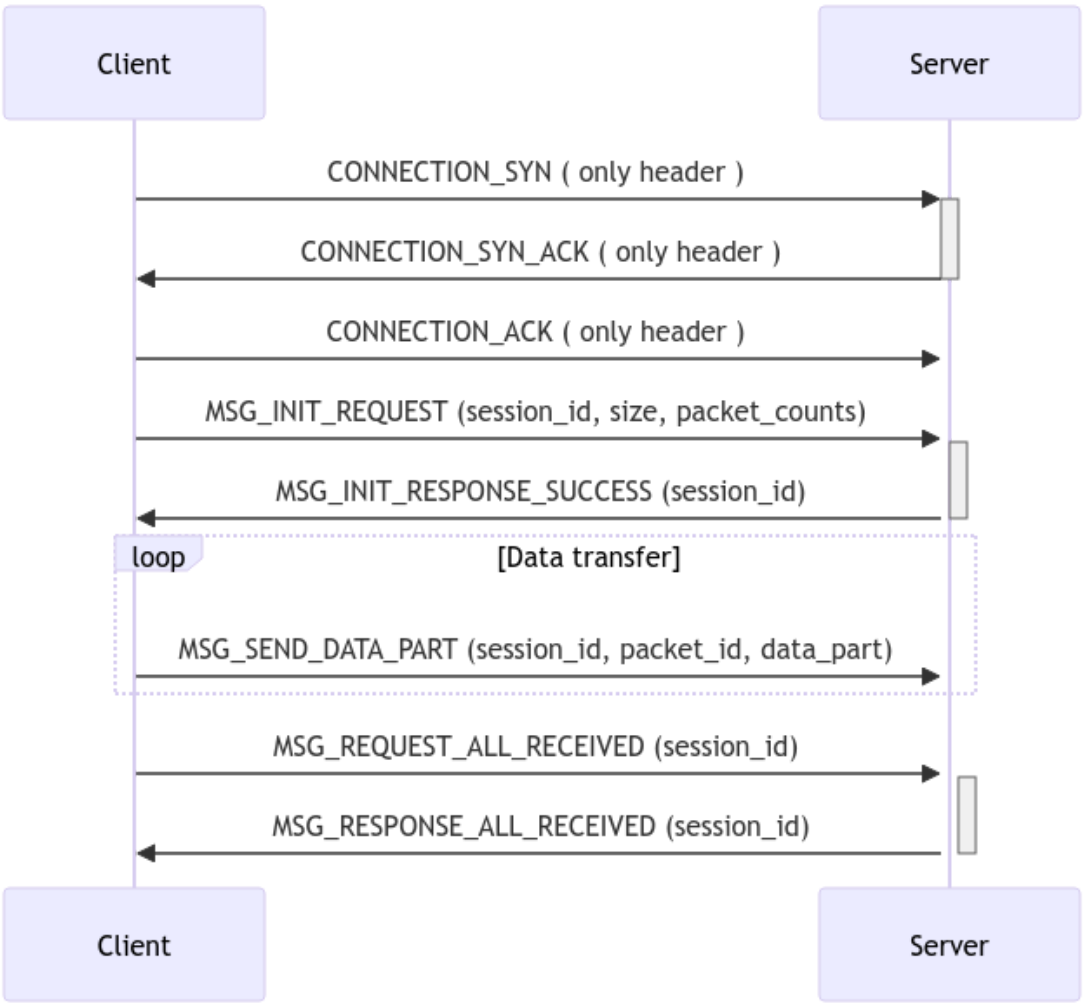

Рисунок 2. Схема передачи сообщения в нормальных условиях

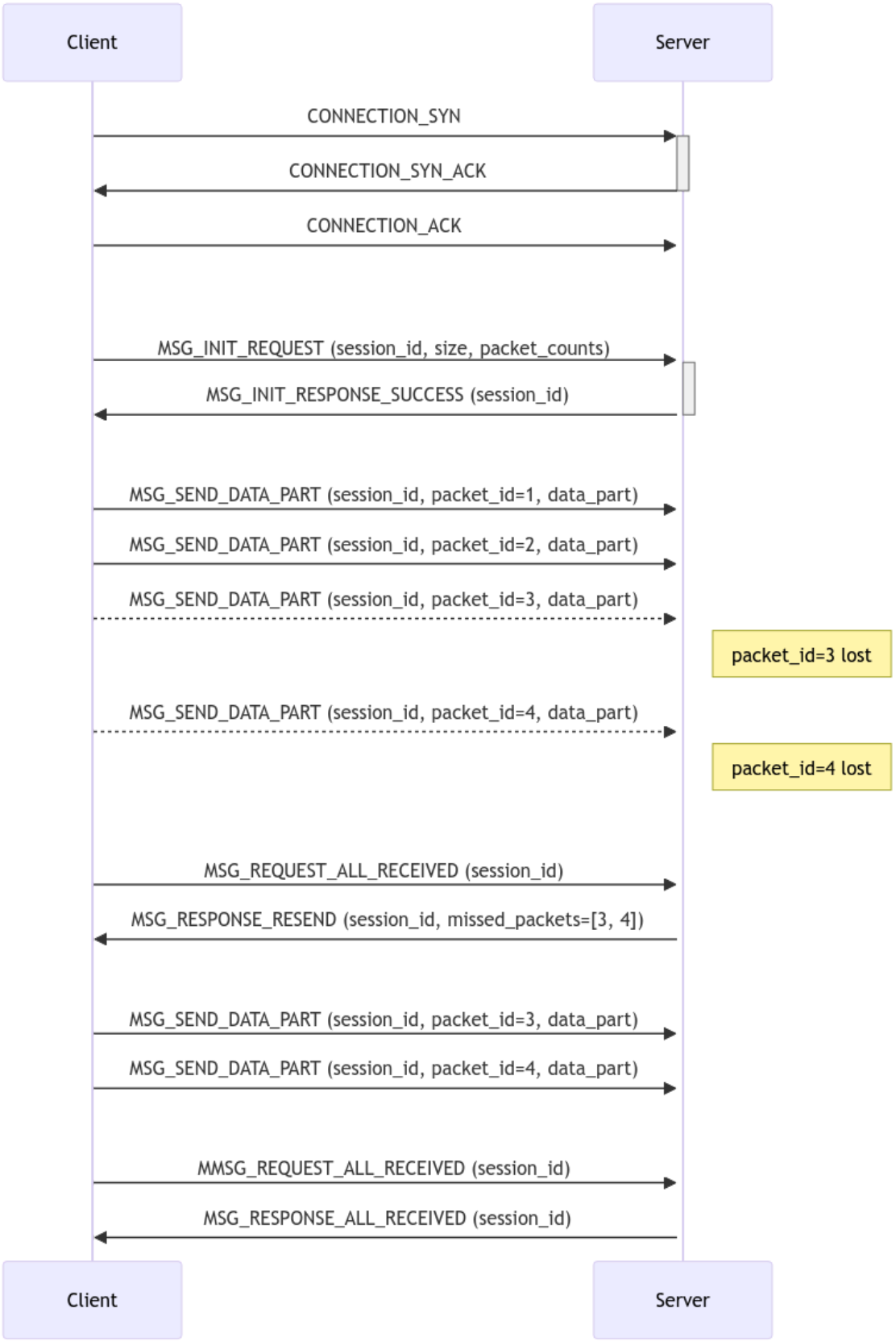

Рисунок 3. Схема передачи сообщения в канале с ошибками

2.6 Потенциал для улучшения протокола

Разработанный протокол обладает значительным потенциалом для дальнейшего развития и совершенствования. В частности, можно выделить следующие направления для будущих исследований:

- Реализация механизма постоянной пропинговки для автоматического разрыва неактивных соединений.
- Интеграция алгоритмов шифрования и аутентификации для повышения безопасности протокола.

3. Архитектура клиент-сервер

Приложение, созданное для демонстрации возможностей протокола, реализует систему распределенных вычислений, используя архитектуру клиент-сервер. Система состоит из трех ключевых компонентов:

Клиент: Отправляет задачи на сервер и получает результаты.

Сервер: Принимает задачи от клиентов, распределяет их между воркерами и возвращает результаты клиентам.

Воркер: Выполняет полученные от сервера задачи и отправляет результаты обратно.

3.1 Клиент

Клиентская часть системы отвечает за формирование задач, отправку их на сервер и получение результатов.

Запуск:

Клиент запускается из командной строки с помощью исполняемого файла RabbitClient и принимает следующие аргументы:

`-i, --id`: Идентификатор клиента (по умолчанию: "`1234`").

`-h, --host`: Адрес сервера (по умолчанию: "`localhost`").

`-p, --port`: Порт сервера (по умолчанию: `12345`).

Алгоритм работы:

1. Клиент подключается к серверу по UDP, используя протокол STIP, и регистрируется, отправляя свой идентификатор.
2. Пользователь выбирает тип задачи из предложенных вариантов.
3. Клиент запрашивает у пользователя необходимые входные данные для выбранной задачи.
4. Клиент формирует объект TaskRequest, содержащий идентификатор задачи, имя функции для выполнения, входные данные в формате JSON и количество ядер процессора, необходимых для выполнения задачи.
5. Клиент отправляет TaskRequest на сервер.

6. Клиент ожидает получения результата от сервера.
7. Получив результат, клиент выводит его в консоль.

3.2 Сервер

Сервер является центральным звеном системы, отвечающим за взаимодействие между клиентами и воркерами.

Запуск:

Сервер запускается из командной строки с помощью исполняемого файла RabbitServer и принимает следующий аргумент:

`-p, --port`: Порт, на котором сервер будет ожидать подключения (по умолчанию: `12345`).

Алгоритм работы:

1. Сервер запускает UDP-сокет на заданном порту и ожидает подключения клиентов и воркеров.
2. При подключении нового клиента сервер сохраняет его идентификатор и сетевое соединение.
3. При подключении нового воркера сервер сохраняет его идентификатор, количество ядер процессора и сетевое соединение.
4. При получении от клиента TaskRequest сервер проверяет наличие свободных воркеров с достаточным количеством ядер процессора.
5. Если свободный воркер найден, сервер отправляет ему TaskRequest и помечает воркер как занятой.
6. Если свободный воркер не найден, задача добавляется в очередь.
7. При получении от воркера результата TaskResult сервер помечает задачу как выполненную, обновляет состояние воркера, освобождая занятые ядра, и отправляет TaskResult клиенту, который отправил эту задачу.

8. Сервер постоянно проверяет очередь задач и отправляет задачи на выполнение свободным воркерам.

3.3 Воркер

Воркер представляет собой отдельный узел системы, предназначенный для выполнения полученных от сервера задач.

Запуск:

Воркер запускается из командной строки с помощью исполняемого файла RabbitWorker и принимает следующие аргументы:

`-i, --id`: Идентификатор воркера (по умолчанию: "`1234`").

`-h, --host`: Адрес сервера (по умолчанию: "`localhost`").

`-p, --port`: Порт сервера (по умолчанию: `12345`).

`-c, --cores`: Количество доступных ядер процессора (по умолчанию: `4`).

Алгоритм работы:

1. Воркер подключается к серверу по UDP, используя протокол STIP, и регистрируется, отправляя свой идентификатор и количество ядер процессора.
2. Воркер ожидает получения задач от сервера.
3. При получении TaskRequest воркер выбирает соответствующую функцию для выполнения на основе имени функции, переданного в TaskRequest.
4. Воркер выполняет функцию с заданными входными данными.
5. Результат выполнения функции упаковывается в объект TaskResult вместе с идентификатором задачи и отправляется обратно на сервер.

3.4 Схема взаимодействия

На рисунке ниже показана принципиальная схема взаимодействия основных компонентов приложения.

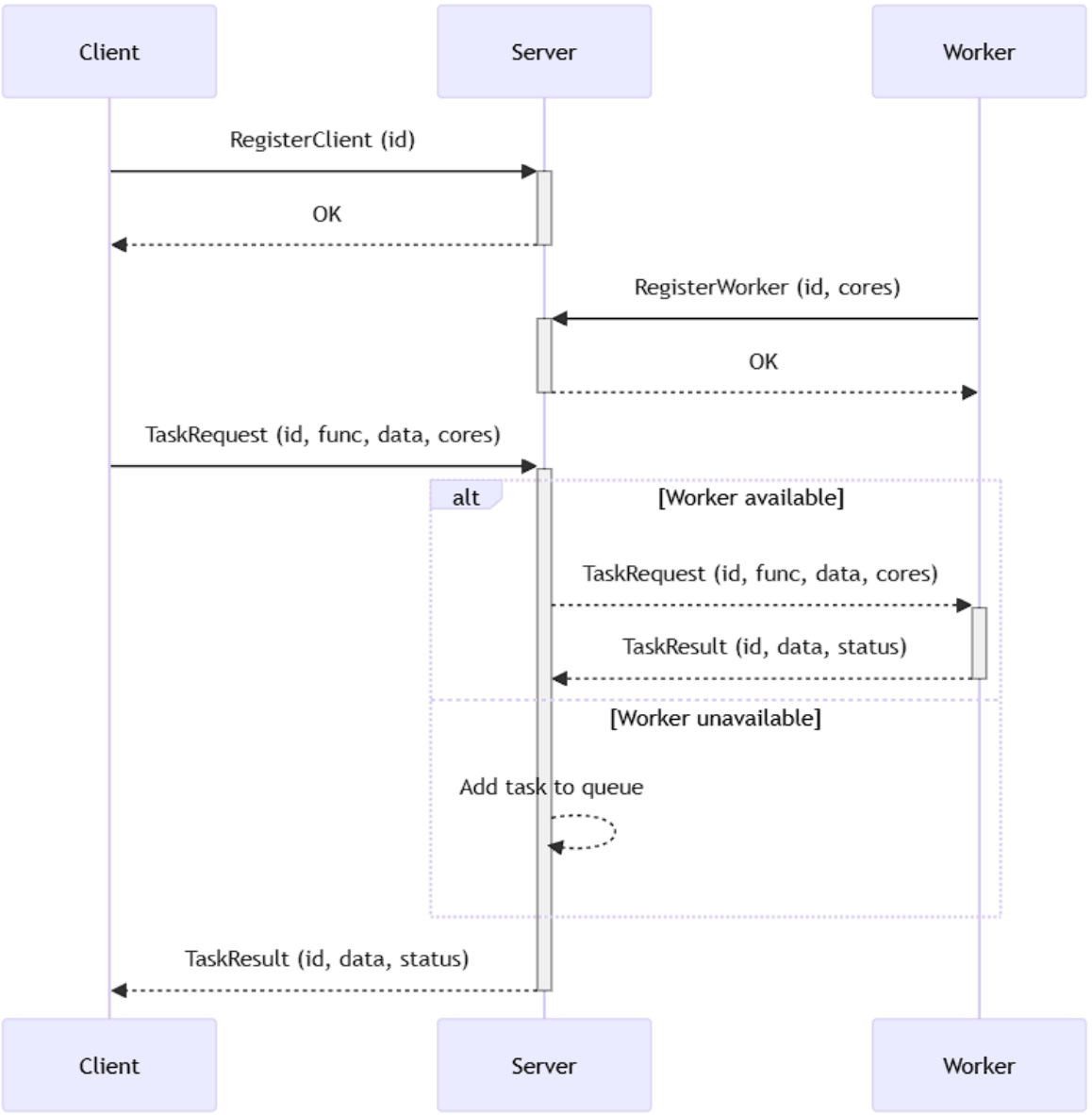

Рисунок 4. Схема взаимодействия основных компонентов приложения

4. Примеры работы программы

4.1 Работа с одним воркером в один поток

При отправке задачи сервер выбирает для работы самый малонагруженный воркер. У клиента есть возможность выбора количества потоков для выполнения его задачи.

Пример работы кода в одном потоке:

CLIENT:

```
Enter task number:
1 - simple math
2 - matrix determinant
3 - matrix multiplication
0 - exit console
> 3
Enter cores count:
> 1
Enter matrix size:
> 3

Task result:
[[5067, 4513, 2416],
[10451, 11867, 10993],
[5747, 8026, 5274]]
```

WORKER:

```
Worker registered with server
Received task request
Executing handler for func: matrixMultiplication, id: 41
RabbitWorker::matrixMultiplicationHandler - Handling matrix multiplication for request_id: 41
RabbitWorker::matrixMultiplicationHandler - Results sent for request_id: 41
RabbitWorker::matrixMultiplicationHandler - Execution time: 0.0045933s
```

SERVER:

```
Received TaskRequest from client: 2
Task added: 41 (Cores: 1)
Found worker: 1 (Cores: 4)
Update cores: Worker 1 used cores: 1
Task 41 sent to worker 1
Received TaskResult from worker: 1
Task marked as Ready: 41
Update cores: Worker 1 used cores: 0
Sent TaskResult to client: 2
Worker: 1, Cores: 4, Used cores: 0
Checking task queue for worker 1
Worker 1 has 4 free cores (cores: 4, used: 0)
```

4.2 Работа с одним воркером в несколько потоков

CLIENT:

```
Enter task number:
1 - simple math
2 - matrix determinant
3 - matrix multiplication
0 - exit console
> 2
Enter cores count:
> 1
Enter matrix size:
> 2000
Enter matrix count:
> 1

Enter task number:
1 - simple math
2 - matrix determinant
3 - matrix multiplication
0 - exit console
> 2
Enter cores count:
> 1
Enter matrix size:
> 2000
Enter matrix count:
> 1

Task result:
[963777302]

Task result:
[-782335630]
```

WORKER:

```
    Received task request
    RabbitWorker::determinantHandler - Handling determinant
for request_id: 1
    RabbitWorker::determinantHandler - Starting thread for
matrix 0
    RabbitWorker::determinant - Calculating determinant for
matrix of size 2000
    Received task request
    RabbitWorker::determinantHandler - Handling determinant
for request_id: 1
    RabbitWorker::determinantHandler - Starting thread for
matrix 0
    RabbitWorker::determinant - Calculating determinant for
matrix of size 2000
    RabbitWorker::determinant - Determinant result is
963777302
    RabbitWorker::determinantHandler - Results sent for
request_id: 41
    RabbitWorker::determinant - Determinant result is
-782335630
    RabbitWorker::determinantHandler - Results sent for
request_id: 18467
```

SERVER:

```
    Received TaskRequest from client: 2
    Task 41 added (Cores: 1)
    Found worker: 1 (Cores: 4)
    Update cores: Worker 1 used cores: 1
    Task 41 sent to worker 1
    Received TaskRequest from client: 2
    Task 18467 added (Cores: 1)
    Found worker: 1 (Cores: 4)
    Update cores: Worker 1 used cores: 2
    Task 18467 sent to worker 1
    Received TaskResult from worker: 1
    Task marked as Ready: 41
    Update cores: Worker 1 used cores: 1
```

```
Sent TaskResult to client: 2
Checking task queue for worker 1
Received TaskResult from worker: 1
Task marked as Ready: 18467
Update cores: Worker 1 used cores: 0
Sent TaskResult to client: 2
Checking task queue for worker 1
Worker 1 has 4 free cores (cores: 4, used: 0)
```

4.3 Работа с несколькими воркерами

При поступлении нескольких задач на сервер, он определяет самые малонагруженные воркеры и отправляет задачи им, в случае если задача имеет в аргументах количество потоков больше, чем есть у каких-либо воркеров на данный момент, но задача попадает в очередь до момента, пока воркеры не освободятся.

CLIENT:

```
Enter task number:
1 - simple math
2 - matrix determinant
3 - matrix multiplication
0 - exit console
> 2
Enter cores count:
> 2
Enter matrix size:
> 2000
Enter matrix count:
> 1

Enter task number:
1 - simple math
2 - matrix determinant
3 - matrix multiplication
0 - exit console
```

```
> 2
Enter cores count:
> 2
Enter matrix size:
> 2000
Enter matrix count:
> 1

Enter task number:
1 - simple math
2 - matrix determinant
3 - matrix multiplication
0 - exit console
> 2
Enter cores count:
> 2
Enter matrix size:
> 2000
Enter matrix count:
> 1

Enter task number:
1 - simple math
2 - matrix determinant
3 - matrix multiplication
0 - exit console
> 2
Enter cores count:
> 2
Enter matrix size:
> 2000
Enter matrix count:
> 1

Enter task number:
1 - simple math
2 - matrix determinant
3 - matrix multiplication
0 - exit console

Task result:
```

```
    [382831396]

    Task result:
    [1303567033]

    Task result:
    [-1903452625]

    Task result:
    [-2025022933]
```

WORKER 1:

```
    Polling started
    Received task request
    RabbitWorker::determinantHandler - Handling determinant
for request_id: 1
    RabbitWorker::determinantHandler - Starting thread for
matrix 0
    RabbitWorker::determinant - Calculating determinant for
matrix of size 2000
    Received task request
    RabbitWorker::determinantHandler - Handling determinant
for request_id: 1
    RabbitWorker::determinantHandler - Starting thread for
matrix 0
    RabbitWorker::determinant - Calculating determinant for
matrix of size 2000
    RabbitWorker::determinant - Determinant result is
382831396
    RabbitWorker::determinantHandler - Results sent for
request_id: 41
    RabbitWorker::determinant - Determinant result is
-1903452625
    RabbitWorker::determinantHandler - Results sent for
request_id: 6334
```

WORKER 2:

```
    Polling started
    Received task request
    RabbitWorker::determinantHandler - Handling determinant
for request_id: 3
    RabbitWorker::determinantHandler - Starting thread for
matrix 0
    RabbitWorker::determinant - Calculating determinant for
matrix of size 2000
    Received task request
    RabbitWorker::determinantHandler - Handling determinant
for request_id: 3
    RabbitWorker::determinantHandler - Starting thread for
matrix 0
    RabbitWorker::determinant - Calculating determinant for
matrix of size 2000
    RabbitWorker::determinant - Determinant result is
1303567033
    RabbitWorker::determinantHandler - Results sent for
request_id: 18467
    RabbitWorker::determinant - Determinant result is
-2025022933
    RabbitWorker::determinantHandler - Results sent for
request_id: 26500
```

SERVER:

```
    Received TaskRequest from client: 2
    Task 41 added (Cores: 2)
    Worker: 1, Cores: 4, Used cores: 0
    Worker: 3, Cores: 4, Used cores: 0
    Found worker: 1 (Cores: 4)
    Update cores: Worker 1 used cores: 2
    Task 41 sent to worker 1
    Received TaskRequest from client: 2
    Task 18467 added (Cores: 2)
    Worker: 1, Cores: 4, Used cores: 2
    Worker: 3, Cores: 4, Used cores: 0
    Found worker: 3 (Cores: 4)
    Update cores: Worker 3 used cores: 2
```

```
Task 18467 sent to worker 3
Received TaskRequest from client: 2
Task 6334 added (Cores: 2)
Worker: 1, Cores: 4, Used cores: 2
Worker: 3, Cores: 4, Used cores: 2
Found worker: 1 (Cores: 4)
Update cores: Worker before has used cores: 2
Update cores: Worker 1 used cores: 4
Task 6334 sent to worker 1
Received TaskRequest from client: 2
Task 26500 added (Cores: 2)
Worker: 1, Cores: 4, Used cores: 4
Worker: 3, Cores: 4, Used cores: 2
Found worker: 3 (Cores: 4)
Update cores: Worker before has used cores: 2
Update cores: Worker 3 used cores: 4
Task 26500 sent to worker 3
Received TaskResult from worker: 1
Task marked as Ready: 41
Update cores: Worker 1 used cores: 2
Sent TaskResult to client: 2
Checking task queue for worker 1
Received TaskResult from worker: 3
Task marked as Ready: 18467
Update cores: Worker 3 used cores: 2
Sent TaskResult to client: 2
Checking task queue for worker 3
Worker 3 has 4 free cores (cores: 4, used: 0)
Received TaskResult from worker: 1
Task marked as Ready: 6334
Update cores: Worker 1 used cores: 0
Sent TaskResult to client: 2
Checking task queue for worker 1
Received TaskResult from worker: 3
Task marked as Ready: 26500
Update cores: Worker 3 used cores: 0
Sent TaskResult to client: 2
Checking task queue for worker 3
Worker 3 has 4 free cores (cores: 4, used: 0)
```

ЗАКЛЮЧЕНИЕ

Разработанный в рамках данной курсовой работы сетевой протокол на базе UDP обладает рядом существенных преимуществ, которые делают его эффективным и перспективным инструментом для создания отказоустойчивых сетевых приложений. Поддержка параллельных передач, устойчивость к сетевым ошибкам, гибкость и масштабируемость – все это делает его пригодным для использования в широком спектре приложений, где требуется надежный и высокопроизводительный обмен данными по сети.

В рамках данной курсовой работы была разработана система распределенных вычислений, основанная на архитектуре клиент-сервер. Система позволяет распределять вычислительную нагрузку между несколькими узлами (воркерами), повышая эффективность обработки задач.

Благодаря модульной архитектуре и использованию JSON для обмена данными, система обладает гибкостью и расширяемостью. В дальнейшем возможно добавление новых типов задач, реализация различных алгоритмов распределения задач и балансировки нагрузки между воркерами.

В целом, разработанная система демонстрирует принципы работы распределенных систем и может служить основой для создания более сложных приложений, требующих распределенной обработки данных.

СПИСОК ЛИТЕРАТУРЫ

1. Документация к библиотеке Boost Asio C++ URL: https://www.boost.org/doc/libs/1_84_0/doc/html/boost_asio.html (дата обращения: 27.05.2024).
2. Кроссплатформенный многопоточный TCP/IP сервер на C++ // Habr URL: https://habr.com/ru/articles/503432 (дата обращения: 27.05.2024).
3. Протоколы семейства TCP/IP. Теория и практика URL: https://habr.com/ru/companies/ruvds/articles/759988/ (дата обращения: 27.05.2024).
4. C and C++ Package Manager Documentation // Conan 2 URL: https://docs.conan.io/2/ (дата обращения: 27.05.2024).