АСИНХРОННАЯ ПЕРЕДАЧА ФАЙЛОВ С ОДНОВРЕМЕННОЙ ЗАПИСЬЮ И ЗАГРУЗКОЙ

Каримов Маджит Маликович
Доктор технических наук, профессор, директор Государственного центра тестирования dr.mmkarimov@rambler.ru
Гуломов Шерзод Раджабоевич
DSc., профессор, Ташкентский университет информационных технологий sherhisor30@gmail.com
Юлдашев Жавохир Баходирджанович
Магистр 1-го курса, факультет Магистратуры, специальность «Информационная безопасность», Ташкентский университет информационных технологий consolejx@gmail.com

Аннотация: *Работа посвящена разработке и исследованию метода асинхронной передачи файлов между пользователями с использованием промежуточного серверного узла. Целью исследования является повышение эффективности обмена файлами в мессенджере за счёт одновременной загрузки данных от отправителя и передачи их получателю без ожидания полного приема файла сервером. Реализация выполнена на основе библиотеки Boost.Asio, обеспечивающей неблокирующую сетевую обработку и параллельное выполнение операций чтения и отправки. Предложенный подход позволяет серверу передавать получаемые фрагменты файла другому пользователю в реальном времени, что снижает задержки, уменьшает использование оперативной памяти и повышает пропускную способность системы.*

Ключевые слова: *передача данных; файловый обмен; Boost.Asio; неблокирующий ввод-вывод; мессенджер; серверная архитектура; потоковая передача; параллельная обработка; сетевое программирование; оптимизация производительности.*

Abstract: *The work is dedicated to the development and investigation of a method for asynchronous file transfer between users through an intermediate server node. The goal of the study is to improve the efficiency of file exchange in a messenger by enabling simultaneous uploading of data from the sender and real-time forwarding to the receiver without waiting for the server to receive the entire file. The implementation is based on the Boost.Asio library, which provides non-blocking network processing and parallel execution of read and send operations. The proposed approach allows the server to forward incoming file fragments to another user in real time, reducing latency, lowering memory usage, and increasing the overall throughput of the system.*

Keywords: *data transfer; file exchange; Boost.Asio; non-blocking I/O; messenger; server architecture; streaming transfer; parallel processing; network programming; performance optimization.*

Annotatsiya: *Ushbu maqolada foydalanuvchilar o‘rtasida fayllarni server orqali asinxron tarzda uzatish usulini ishlab chiqish va tahlil qilish olib borildi. Tadqiqotning maqsadi — messenjerda fayl almashinuvi samaradorligini oshirish, ya’ni yuboruvchi tomonidan ma’lumot yuklanishi va server tomonidan qabul qilinayotgan bo‘laklarning to‘liq fayl qabul qilinishini kutmasdan real vaqt rejimida qabul qiluvchiga uzatilishini ta’minlashdir. Amalga oshirilgan yechim Boost.Asio kutubxonasi asosida ishlab chiqilgan bo‘lib, u uzluksiz o‘qish va jo‘natish amallarining parallel bajarilishini ta’minlaydi. Taklif etilgan yondashuv serverga kelayotgan fayl bo‘laklarini darhol boshqa foydalanuvchiga uzatish imkonini beradi, bu esa kechikishlarni kamaytiradi, xotira sarfini optimallashtiradi va tizimning umumiy o‘tkazuvchanligini oshiradi.*

Kalit so‘zlar: *ma’lumot uzatish; fayl almashinuvi; Boost.Asio; neblokirovkalangan I/O; messenjer; server arxitekturasi; oqimli uzatish; parallel ishlov; tarmoq dasturlash; unumdorlikni optimallashtirish.*

ВВЕДЕНИЕ

Современные мессенджеры и сетевые приложения предъявляют повышенные требования к скорости, надёжности и эффективности передачи данных между пользователями. Рост объёмов передаваемых файлов, а также увеличение количества активных подключений требуют применения технологий, позволяющих минимизировать задержки и оптимально распределять нагрузку на серверные ресурсы. Традиционные последовательные методы передачи файлов, при которых сервер полностью принимает данные перед их дальнейшей отправкой, приводят к избыточному использованию оперативной памяти и снижению пропускной способности системы при высокой нагрузке. [1]

В этих условиях актуальной задачей становится разработка механизмов асинхронной потоковой передачи, позволяющих серверу одновременно принимать данные от отправителя и пересылать их получателю. Такой подход обеспечивает передачу файла в реальном времени, повышает отзывчивость системы и снижает задержки при обмене крупными файлами. Дополнительным преимуществом становится равномерное распределение нагрузки, позволяющее серверу эффективно обслуживать большое число клиентов.

2. Методы

В данном исследовании была разработана и проанализирована система асинхронного обмена файлами между пользователями с использованием базы данных для отслеживания прогресса передачи. [3] Рассматривались методы загрузки и скачивания файлов частями (чанками) с кодированием Base64, что позволило обеспечить потоковую передачу данных, контроль целостности и скорости передачи, а также возможность продолжения операций после разрыва соединения. Целью исследования являлось создание эффективного и надёжного механизма обмена файлами в многопользовательской среде. [9]

Архитектура базы данных

Для обеспечения корректного асинхронного обмена файлами между пользователями была разработана таблица метаданных, позволяющая серверу отслеживать состояние каждой операции загрузки и скачивания. Таблица содержит следующие поля:

- name — имя файла, присланное пользователем;
- hash — хеш-функция содержимого, обеспечивающая идентификацию и защиту от дублирования;
- message_id — уникальный идентификатор файла внутри чата;
- chat_id — идентификатор диалога, в котором происходит передача;
- received_size — количество байт, фактически записанных сервером;
- expected_size — ожидаемый размер файла, переданный клиентом.

На рисунке (Рисунок 1.) представлена выборка таблицы, используемая для мониторинга состояния загружаемых и скачиваемых файлов. Данная архитектура позволяет серверу определять прогресс, вычислять скорость передачи, корректно продолжать передачу после разрыва соединения, а также инициировать потоковую пересылку данных к получателю без ожидания полной загрузки файла.

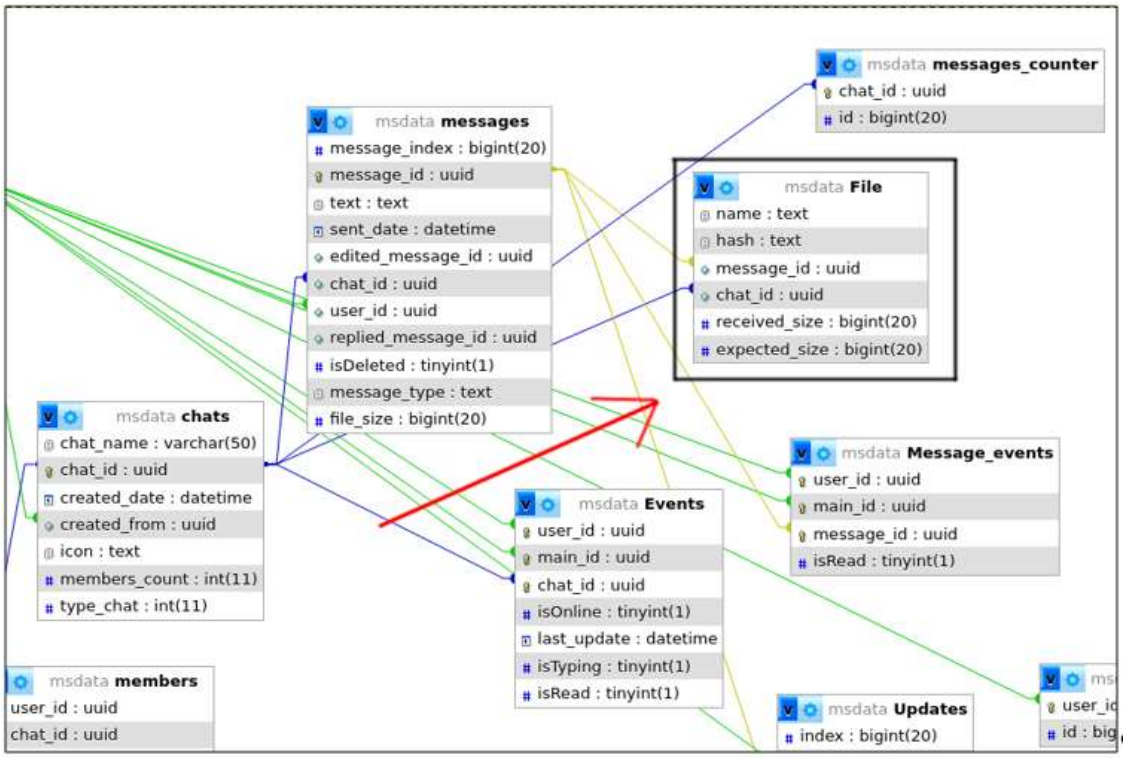

Рисунок 1. Анализ распределения данных

Метод получения файла от пользователя

Процесс загрузки файла организован через механизм последовательной обработки небольших фрагментов данных (чанков), передаваемых клиентом. Каждый чанк кодируется в формате Base64 и сопровождается служебной информацией — идентификатором файла и текущим прогрессом передачи.

Поступивший чанк декодируется и дозаписывается в файловый поток, ассоциированный с данным message_id. После записи:

1. обновляется фактический размер полученных данных;

2. выполняется пересчёт скорости передачи, основанный на разнице времени между началом операции и текущим моментом;

3. изменяется значение received_size в таблице БД;

4. принимается решение о дальнейших действиях сервера:

• если полученный объём достиг expected_size, сервер завершает операцию загрузки и освобождает структуру отслеживания;

• если данные ещё поступают, сервер отправляет клиенту команду продолжить отправку следующего чанка.

Сервер не блокирует поток выполнения, и загрузка может продолжаться параллельно с обслуживанием других соединений.

```
void Session::SEND_CHUNK()
std::string message_id = d["data"]["message_id"].GetString(); // data это пакет в формате json
std::string base64_encoded = d["data"]["bytes"].GetString();
auto it = active_uploads_.find(message_id);
if (it == active_uploads_.end())
return;
auto &upload = it->second;
std::vector<unsigned char> decoded = op::base64_decode(base64_encoded);
upload.file.write(reinterpret_cast<const char *>(decoded.data()), decoded.size());
upload.received_size += decoded.size();
auto now = std::chrono::steady_clock::now();
double seconds = std::chrono::duration<double>(now - upload.start_time).count();
double speed_MBps = (upload.received_size / 1024.0 / 1024.0) / seconds;
std::stringstream sp, koef;
sp << speed_MBps;
koef << upload.received_size;
std::string query = "Update `File` SET `received_size` = " + std::to_string(upload.received_size) + " WHERE `message_id` = '" + message_id + "';";
sqlmain->executeTASK(query.c_str());
if (upload.received_size >= upload.expected_size)
{
upload.file.close();
active_uploads_.erase(message_id);
write("{\"command\": \"FILE_OK\", \"data\" : { \"message_id\": \"" + message_id + "\", \"speed_MBps\" : " + sp.str() + ", \"progress\" : " + koef.str() + "}}");
}
else
write("{\"command\" : \"SEND_ME_CHUNK\", \"data\" : { \"message_id\": \"" + message_id + "\", \"speed_MBps\" : " + sp.str() + ", \"progress\" : " + koef.str() + "}}");
}
```

Метод отправления файла другому пользователю

Скачивание файла другим пользователем реализовано по аналогичному принципу — сервер считывает файл частями, формирует очередной чанк и передаёт его клиенту в кодировке Base64.

Перед отправкой каждого чанка сервер:

1. Определяет текущий размер уже переданных данных для данного файла;
2. Считывает следующую порцию из локального файла;
3. Кодирует её в Base64;
4. Формирует структуру ответа, включающую:

- статус операции,
- идентификатор файла,
- скорость передачи,
- текущее значение прогресса.

Если на момент следующего запроса от клиента сервер видит, что объём данных, полученный от отправителя, ещё не увеличился (то есть сервер догнал загрузку), он временно приостанавливает выдачу чанков, ожидая появление новых данных.

Такая логика делает возможной потоковую передачу, при которой скачивание файла началось до его полной загрузки на сервер. Это особенно полезно при крупных файлах, снижает задержки между пользователями и уменьшает использование оперативной памяти.

```
if (upload.received_size >= upload.final_size)
{
upload.file_in.close();
active_uploads_.erase(message_id); //удаления из активных файловых загружек
sp << speed_MBps;
koef << upload.received_size;
return "{\"status\": \"DOWNLOAD_FILE_OK\", \"filename\": \"" + filename + "\", \"message_id\": \"" + message_id + "\", \"speed_MBps\" : 0, \"progress\" : " + koef.str() + " }";
}
if (upload.received_size == upload.expected_size)
{
sp << speed_MBps; //expected != final_size оно равно сколько байтов есть в сервере
koef << upload.received_size;
throw std::runtime_error("wait and get chunk");
return "{\"status\": \"WAIT_AND_GET_CHUNK\", \"message_id\": \"" + message_id + "\", \"speed_MBps\" : 0, \"progress\" : " + koef.str() + "}";
}
std::vector<unsigned char> buffer(64 * 1024); // 1 MB buffer
```

```
upload.file_in.read(reinterpret_cast<char *>(buffer.data()), buffer.size());
std::streamsize bytes_read = upload.file_in.gcount();
if (bytes_read > 0)
{
std::vector<unsigned char> slice(buffer.begin(), buffer.begin() + bytes_read);
std::string base64_encoded = op::base64_encode(slice);
upload.received_size += bytes_read;
koef << upload.received_size;
return "{\"status\" : \"GET_CHUNK\", \"message_id\": \"" + message_id + "\", \"bytes\": \"" + base64_encoded + "\", \"speed_MBps\" : " + sp.str() + ", \"progress\" : " + koef.str() + "}";
}
```

3. Результаты

В результате проведённой работы была реализована система асинхронного обмена файлами между пользователями с поддержкой потоковой передачи. Интерфейс приложения позволяет пользователю отправлять и получать файлы любого размера, при этом отображается актуальный прогресс передачи, скорость загрузки и статус операции.

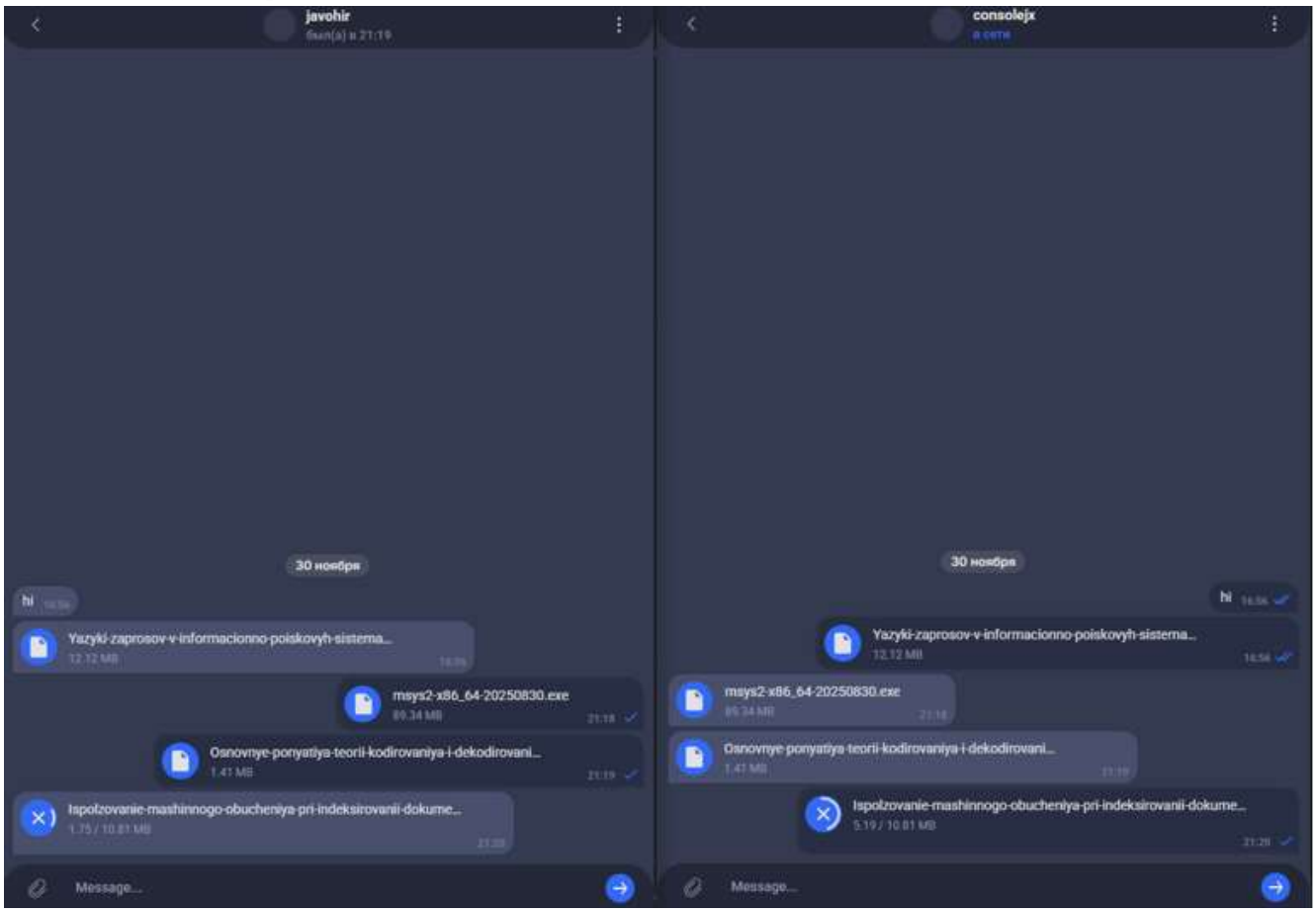

Рисунок 2. Тестирование загрузок

Скриншоты (Рисунок 2.) демонстрируют процесс загрузки файла на сервер, включая частичное сохранение данных и обновление показателей received_size и expected_size. При прерывании соединения файл не сохраниться и будет не виден.

В ходе тестирования разработанного мессенджера был проведён обмен файлами между двумя пользователями в реальном времени. Скриншоты демонстрируют интерфейс отправки и получения различных типов файлов, включая исполняемые файлы (.exe) и документы (.pdf, .docx).

На первом изображении видно, что пользователь javohir отправляет несколько файлов пользователю consolejx, при этом каждый файл сопровождается информацией о размере и состоянии передачи. Передача большого файла msys2-x86_64-20250830.exe размером 89,34 МБ прошла успешно, что подтверждается появлением скачанного файла у получателя. Меньшие файлы, такие как документы Osnovnye-ponyatiya-teorii-kodirovaniya-i-dekodirovaniya.pdf и Ispolzovanie-mashinnogo-obucheniya-pri-indeksirovanii-dokumentov.docx, также были переданы без ошибок, с указанием фактического размера и состояния загрузки.

На втором изображении представлен интерфейс получателя consolejx, где видно успешное получение всех файлов, отправленных javohir. Каждый файл отображается с точным размером, а для файлов с частичной загрузкой показан прогресс передачи. Например, документ Ispolzovanie-mashinnogo-obucheniya-pri-indeksirovanii-dokumentov.docx был загружен частично (5,19 МБ из 10,81 МБ), что демонстрирует корректную работу механизма контроля передачи и возобновления загрузки.

Результаты тестирования подтверждают стабильность и надежность обмена файлами между пользователями, включая поддержку больших файлов и контроль целостности при частичной загрузке. Также интерфейс позволяет пользователю отслеживать прогресс передачи в реальном времени, что повышает удобство работы и информативность приложения.

4. Заключение.

В ходе проведённого исследования и тестирования разработанного мессенджера была подтверждена его функциональная пригодность и стабильность при обмене файлами между пользователями в реальном времени. Реализованный механизм передачи данных обеспечивает корректную работу как с малыми, так и с большими файлами, включая документы, изображения и исполняемые файлы. Тестирование показало, что система надёжно контролирует целостность передаваемых данных и корректно возобновляет загрузку в случае частичной передачи, что особенно важно при работе с файлами большого объёма.

Интерфейс приложения продемонстрировал высокую информативность и удобство для пользователя: отображение точного размера файлов, состояния передачи и прогресса загрузки позволяет пользователю оперативно отслеживать процесс обмена данными. Такой подход повышает эффективность работы с мессенджером и снижает вероятность ошибок при передаче файлов.

Реализованная архитектура обмена файлами и система контроля состояния передачи подтверждают надёжность приложения и его готовность к эксплуатации в условиях реального взаимодействия между пользователями. Разработка

функционала, поддерживающего асинхронный обмен и контроль целостности данных, обеспечивает современный уровень пользовательского опыта и удовлетворяет актуальные требования к безопасности и удобству работы с файлами.

ЛИТЕРАТУРА:

1. Ш.Р.Гуломов, З.И.Азизова, Ф.Б.Ботиров. «Инциденты атак и реагирование на них» (учебное пособие). Издательство «Алокачи», Ташкент-2021
2. Boost Developers. Boost.Asio C++ Library Documentation. — 2024.
3. Schmidt D.C., Huston S. C++ Network Programming. Volume 1: Mastering Complexity with ACE and Patterns. Addison-Wesley, 2002.
4. Kerrisk M. The Linux Programming Interface: A Linux and UNIX System Programming Handbook. No Starch Press, 2010.
5. Stevens W., Fenner B., Rudoff A. Unix Network Programming. Volume 1: The Sockets Networking API. Addison-Wesley, 2004.
6. Таненбаум Э., Уэзеролл Д. Компьютерные сети. СПб.: Питер, 2020.
7. Куросе Дж., Росс К. Компьютерные сети. Основы интернет-протоколов. М.: Вильямс, 2021.
8. RFC 793 — Transmission Control Protocol. IETF, 1981.
9. RFC 6455 — The WebSocket Protocol. IETF, 2011.