# Описание приложения

В директории с данной документацией вы можете найти:

* Видеопримеры работы пользователя с мессенджером
* Поведенческую диаграмму приложения (файл *model)*
* Диаграмму классов приложения (файл *uml*)

Ссылка на репозиторий в гитхабе: <https://github.com/smirnovlad/Messenger>

# Описание файлов

1. *Messenger.zip* содержит в себе реализацию серверной и клиентской частей. Сборка каждой из них производится через *conan* + *cmake*
2. *model*, *model.png* – поведенческая диаграмма приложения. С первым файлом можно работать на сайте [https://app.diagrams.net](https://app.diagrams.net/)
3. *uml* – диаграмма классов приложения (серверной и клиентской частей)
4. *example\_\** – видеопримеры использования мессенджера

# Сборка проекта

Для сборки проекта предлагается установить conan версии 1.59.0 и выполнить следующие команды из директории проекта (сервера и клиента по-отдельности):

1. *pip3 install conan==1.59.0*
2. *rm -rf build*
3. *mkdir build && cd build/*
4. *conan install .. --build=missing -c tools.system.package\_manager:mode=install -c tools.system.package\_manager:sudo=True*
5. *cmake ..*
6. *make*

Видео с примером: *build\_and\_run.webm*

Замечание: в видеозаписи файлы *conanfile.txt* и *CMakeLists.txt* могут находиться в неактуальном состоянии. Приложенный архив обновляется при каждом их изменении. Также актуальную версию можно найти на гите.

# Запуск приложения

Для запуска приложения необходимо выполнить следующие команды из директории проекта (сервера и клиента по-отдельности):

1. *cd build/bin/*
2. *export QT\_QPA\_PLATFORM=offscreen*
3. *./Client* в случае клиента и *./Server* в случае сервера

Видео с примером: *build\_and\_run.webm*

# Что реализовано

* Авторизация и регистрация пользователей
* Хранение истории сообщений
* Вход с нескольких устройств с синхронизацией сообщений
* Аутентификация пользователей на основе токенов
* Обработка запросов в случае падения сервера
* Синхронное редактирование сообщений

# Видеопримеры

* *build\_and\_run.webm:*
  + Пример сборки и запуска приложения
* *example\_1.webm:*
  + Пример синхронной переписки двух пользователей
  + Пример регистрации пользователя
* *example\_2.webm:* 
  + Пример синхронной работы в мессенджере двух пользователей, каждый из которых залогинен на двух клиентах
* *example\_3.webm:* 
  + Пример реконнекта к серверу после его отключения и последующего запуска
  + Пример инвалидации токена
  + Пример синхронной работы двух клиентов, залогиненных под одним пользователем
* *example\_4.webm:*
  + Пример синхронного редактирования сообщений
  + Пример переноса строки в сообщении

# Авторизация

* При запуске клиента осознанно требуется ввод логина/пароля для авторизации. Приложение можно обновить таким образом, что при запуске клиента будет происходить автоматическая авторизация с использованием сгенерированного токена, который хранится локально у пользователя. Так как предполагается работа нескольких пользователей с одного устройства, на данном устройстве могут храниться несколько токенов, поэтому и требуется ввод логина/пароля.
* При отправке со стороны клиента запросов серверу передаётся токен пользователя (хранящийся на локальной машине), что обеспечивает дополнительную безопасность приложения.
* Токен обновляется каждый календарный день. Также разница между датой генерации токена и текущим временем не должна превосходить 12 часов. В противном случае пользователь получает сообщение о невалидности токена, и для дальнейшей работы он обязан авторизоваться заново.

# База данных

* В качестве СУБД была выбрана *SQLite* в силу простоты её использования.
* Исходно база данных *MessengerDB* содержит в себе две пустые таблицы: таблицу пользователей мессенджера и таблицу соответствующих им токенов.
* В процессе создания(открытия) новых чатов в базу данных добавляются соответствующие таблицы по следующему принципу. При открытии пользователем *A* чата с пользователем *B* создаётся таблица (если ранее в базе данных не было ни таблицы *Chat\_A\_B*, ни таблицы *Chat\_B\_A*) *Chat\_B\_A*. Для хранения истории сообщений между пользователями A и B используется только одна таблица *Chat\_B\_A*
* Добавление групповых чатов представляется несложной задачей. Изменится структура базы данных

## Таблица Users

| *Название* | *Описание* | *Тип данных* | *Ограничение* |
| --- | --- | --- | --- |
| id | Идентификатор  пользователя | INTEGER | PRIMARY KEY |
| login | Логин пользователя | TEXT | UNIQUE |
| password | Пароль пользователя | TEXT | NOT NULL |

## Таблица Tokens

| *Название* | *Описание* | *Тип данных* | *Ограничение* |
| --- | --- | --- | --- |
| id | Идентификатор  токена | INTEGER | PRIMARY KEY |
| userId | Логин пользователя | INTEGER | UNIQUE |
| token | Токен | TEXT | NOT NULL |
| creationTimeStamp | Время последнего  обновления токена | TEXT | NOT NULL |

## Таблица Chat

| *Название* | *Описание* | *Тип данных* | *Ограничение* |
| --- | --- | --- | --- |
| id | Идентификатор  сообщения | INTEGER | PRIMARY KEY |
| fromID | Идентификатор  отправителя | INTEGER | NOT NULL |
| toID | Идентификатор  получателя | INTEGER | NOT NULL |
| message | Текст сообщения | TEXT | NOT NULL |
| timestamp | Время отправки  сообщения на сервере | TEXT | NOT NULL |

# Взаимодействие клиента и сервера

* Протокол взаимодействия представляет собой запросы и ответы.
* Для передачи данных используется TCP-соединение.
* Формат пакетов следующий:

При отправке запроса со стороны клиента:

* первые 4 байта пакета определяют тип запроса
* оставшиеся байты определяют параметры запроса.

При отправке ответа со стороны сервера:

* первые 4 байта пакета определяют тип запроса со стороны клиента, на который сервер отсылает соответствующий ответ
* следующие 4 байта определяют вердикт обработки запроса
* оставшиеся байты определяют результат обработки запроса

# Обработка запросов клиента и ответов сервера

Рассмотрим более подробно, что происходит, например, при успешной авторизации пользователя:

* После нажатия пользователем кнопки OK виджет авторизации обновляется на виджет списка пользователей
* После обновления виджета клиент отправляет серверу запрос получения списка пользователей
* Первые 4 байта запроса: “CTCS”
* Перед отправкой запроса серверу (запись запроса в сокет) проверяется наличие соединения между клиентом и сервером
  + В случае наличия соединения следующие байты запроса определяют токен пользователя (для его дальнейшей аутентификации на сервере)
  + В случае отсутствия соединения клиент получает сообщение об ошибке. Он может попробовать восстановить соединение, нажав на “Try to connect”, или же выйти из приложения, нажав на “Exit”
* После успешной отправки запроса сервер определяет по первым 4 байтам тип запроса “CTCS” и вызывает соответствующий обработчик с переданными аргументами. В данном случае аргумента два:
  + Первый – указатель на сокет клиента
  + Второй – переданный в запросе токен
* Обработчик сначала проверяет валидность токена:
  + Если токен не валиден, вердикт обработки запроса – “ITKN”
  + В противном случае, обработчик отправляет запрос базе данных на получение списка пользователей. Предполагается, что этот запрос обработается успешно, поэтому итоговый вердикт обработчика - “SCSS”
* База данных возвращает список пользователей в следующем формате:

id1 /s username1 /n id2 /s username2 /n …

* Обработчик вызывает функцию отправки ответа клиенту. Первым параметром функция принимает два аргумента:
  + указатель на сокет клиента,
  + строку, первые 4 байта которой – вердикт обработки запроса, следующие байты – список пользователей в описанном выше формате (который в случае невалидного токена является пустым)
* Функция отправки ответа добавляет к началу переданной строки 4 байта “CTCS” и записывает в переданный клиентский сокет полученную строку
* Получив ответ от сервера, клиент по первым 4 байтам определяет, на какой запрос был получен ответ. В данном случае клиент понимает, что сервер вернул ответ на запрос получения списка пользователей. Если вердикт обработки запроса “SCSS”, клиент парсит переданный сервером список пользователей
* Далее клиент вызывает функцию обработки графическим интерфейсом уже распарсенного списка пользователей, дополнительно передавая вердикт сервера:
  + Если вердикт сервера “ITKN”, выводится соответствующее сообщение об ошибке
  + В противном случае, вердикт сервера – “SCSS”. Графический интерфейс вызывает функцию инициализации списка пользователей

Отмечу, что парсинг ответов сервера происходит в клиенте и графический интерфейс ClientUI вместе с вердиктом получает уже обработанные данные.

# Детали реализации

* Приложение состоит из двух частей: клиентской и серверной.
* При запуске клиента с графическим интерфейсом отображается окно авторизации.
* После успешной авторизации пользователь получает список всех зарегистрированных пользователей мессенджера.
* В случае успешной обработки запроса на получение чата пользователь получает всю историю сообщений с выбранным собеседником.
* Пользователь может залогиниться в мессенджере с разных устройств. Сервер будет хранить соответствие номера этого пользователя с сокетами соответствующих соединений (реализовано с помощью multihash). Тогда при приёме сообщения какого-либо пользователя (также отправки сообщения самому себе) чат обновится на всех устройствах. Аналогичным образом обрабатывается случай, когда пользователь-отправитель залогинен с разных устройств, на которых открыт один чат. Видео с примером: *example\_2.webm*
* Под инвалидацией токена также понимается отсутствие на сервере информации о клиенте, который отправил запрос. Инвалидации такого типа происходят после перезапуска сервера
* После прерывания соединения с сервером и последующего соединения с ним пользователь получает сообщение об инвалидации токена
* Пусть пользователь залогинен с нескольких устройств, на каждом из которых открыт один чат. Тогда после падения сервера и его дальнейшего запуска при попытке установить с сервером соединение клиенту-отправителю будет предложено заново авторизоваться, при этом у остальных клиентов (под тем же логином, с тем же открытым чатом) инвалидируется токен, но изменений в самом мессенджере они не увидят
* Продолжим рассмотрение предыдущего сценария: несколько клиентов залогинены под одним пользователем, у каждого клиенты открыт один и тот же чат, сервер падает и заново запускается. Тогда после авторизации клиента-отправителя произойдёт следующее: при попытке отправить сообщение уже со второго клиента второй клиент получит сообщение о некорректном токене и ему будет предложено авторизоваться заново, в то время как клиент-отправитель (уже заново авторизовавшийся) сможет продолжить свою работу. Видео с примером: *example\_3.webm*
* При параллельной работе разных клиентов, залогиненных под одним пользователем, при отправке/приёме сообщений проверяется, находятся ли они в чате и, если находятся, то в каком именно. Видео с примером: *example\_2.webm*
* Для возможности работы на разных клиентах под одним пользователем нельзя обновлять (генерировать) токен при каждой авторизации, иначе бы токен уже залогиненных клиентов инвалидировался при каждой авторизации. Отмечу, что на одном компьютере клиенты разделяют общий файл с токеном, поэтому такой проблемы нет, но в общем случае она есть. Поэтому после успешной авторизации токен генерируется/обновляется, только если это первая авторизация пользователя или срок действия его токена истёк (токен инвалидировался)
* Для редактирования сообщения пользователь должен вызвать контекстное меню конкретного сообщения (rightclick) и выбрать “Edit”. Видео с примером: *example\_4.webm*
* На текущий момент сервер обрабатывает запросы и отвечает на них в одном потоке
* Независимо от нагрузки на сервер он обработает все отправленные запросы, поскольку после отправки клиентом запроса (записи в сокет), он попадает в очередь событий (event loop), для каждого из которых определён обработчик на стороне сервера

# Сценарии падения

* Если соединение между клиентом и сервером пропадёт в процессе обработки сервером запроса, клиент об ошибке не узнает и ответ на запрос не получит. Эту проблему можно решить при помощи таймаутов
* Для парсинга ответов со стороны сервера используются разделители /s и /n. Если, например, сообщение пользователя будет содержать в себе эти разделители, будет UB

# Тестирование решения

В процессе рассмотрения разных кейсов использования мессенджера я столкнулся с UB, которое достаточно долго не мог пофиксить. Иногда при попытке перейти в чат с выбранным пользователем я ловил сегфолт. В поиске ошибки мне помогли логирование и использование санитайзера (флаг компиляции -fsanitize=address). Путём комментирования некоторых инструкций я сделал вывод, что проблема заключалась в неправильной очистке основного (и единственного) окна приложения перед установкой виджета чата. Более детально, после клика на элемент списка в обработке соответствующего клика (сигнала) необходимо правильно удалять этот элемент. Обсуждение на эту тему: [Trouble diagnosing segfault in Qt application](https://stackoverflow.com/questions/11043994/trouble-diagnosing-segfault-in-qt-application)

Планирую добавить модуль для тестирования с использованием библиотеки googletest.

# Дальнейшие оптимизации

Оптимизировать работу приложения можно следующим образом:

* Обновить функции передачи относительно тяжеловесных данных типа истории сообщений пользователей: избегать их копирования при их передаче в качестве аргументов в функции
* В текущей реализации клиент хранит в себе логин (строку) пользователя, под которым он работает. Достаточно хранить номер этого пользователя в БД
* Для оптимизации работы БД можно использовать индексы. Такие данные, как, например, информация о зарегистрированных в мессенджере пользователях, редко изменяются или не изменяются вообще