Оглавление

[Введение 2](#_Toc73823053)

[1 Теоретическое описание протокола MTProto 3](#_Toc73823054)

[1.1 Краткий обзор компонентов 5](#_Toc73823055)

[1.2 Авторизация и криптография 7](#_Toc73823056)

[1.3 Транспорт 8](#_Toc73823057)

[2 Описание программной реализации MTProto 9](#_Toc73823058)

[2.1 Модуль server.cpp 9](#_Toc73823059)

[2.1.1 Функция **main** 9](#_Toc73823060)

[2.1.2 Функция **sendMsg** 10](#_Toc73823061)

[2.1.3 Прием пакета от клиента-отправителя 10](#_Toc73823062)

[2.2 Модуль client.cpp 11](#_Toc73823063)

[2.2.1 Функция **main** 11](#_Toc73823064)

[2.2.2 Функция **getMsg** 11](#_Toc73823065)

[Принимает зашифрованное сообщение от сервера 11](#_Toc73823066)

[2.3 Модуль keyExchange.cpp 12](#_Toc73823067)

[2.3.1 Сторона клиента 12](#_Toc73823068)

[2.3.2 Сторона сервера 13](#_Toc73823069)

[2.4 Модуль msg\_encr\_decr.cpp 13](#_Toc73823070)

[2.4.1 Функция **getEncryptedBlock** 13](#_Toc73823071)

[2.4.2 Формулы aes\_key, aes\_iv, msg\_key 13](#_Toc73823072)

[2.5 Модуль digits.cpp 14](#_Toc73823073)

[2.5.1 Функция **getDigit** 14](#_Toc73823074)

[Заключение 15](#_Toc73823075)

[Список используемых источников 16](#_Toc73823076)

[Приложение 18](#_Toc73823077)

# Введение

Одна из главных задач современного мира – обеспечение безопасности передачи информации. Существует множество криптографических протоколов, цель которых сделать недоступной для понимания злоумышленником информации. Компания Telegram Messenger выпустила на рынок свой продукт – мессенжер Telegram. Нам интересен этот мессенжер за счёт своего революционного протокола шифрования MTProto, который по сей день признан криптостойким и способным отражать все извесные на данный момент атаки на шифры. Мы заинтересовались этим протоколом и решили в рамках учебной практики исследовать протокол и написать его прототип на языке программирования С++.

# Теоретическое описание **протокола** MTProto

Протокол предназначен для доступа к серверному API с приложений, запущенных на мобильных устройствах. Подчеркнем, что интернет-браузер не считается таким приложением.

Протокол разбит на три почти независимых части:

* Высокоуровневая часть (язык запросов к API) — определяет, каким образом запросы к API и ответы на эти запросы преобразуются в двоичные *сообщения*.
* Криптографическая (авторизационная) прослойка — определяет, каким образом сообщения шифруются перед передачей через транспортный протокол.
* Транспортная часть — определяет, каким образом передаются сообщения между клиентом и сервером поверх какого-либо другого существующего сетевого протокола (например, http, https, tcp, udp).

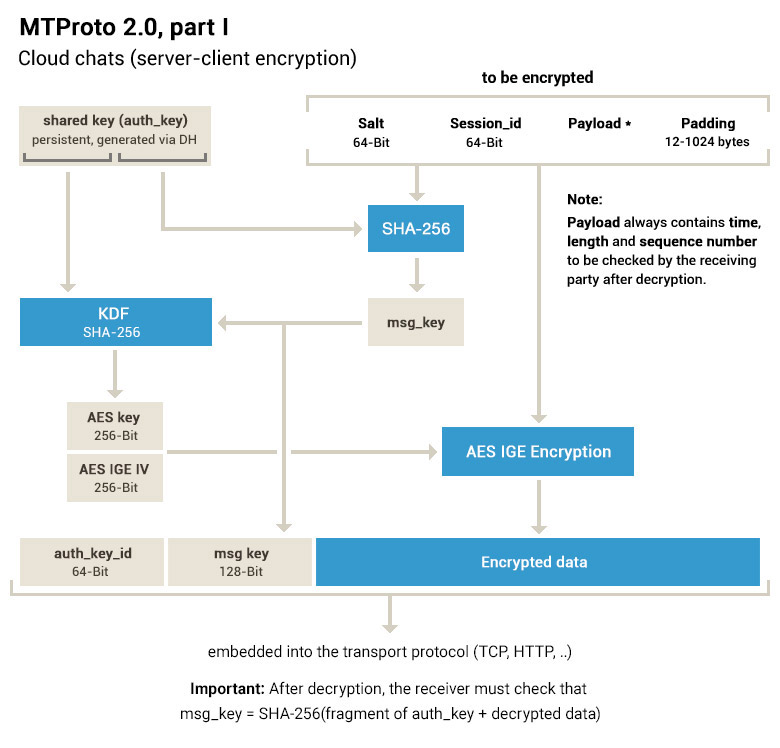


Рис. 1. Схема протокола MTProto для облачных чатов

***Примечание 1:*** Каждое текстовое сообщение, которое нужно зашифровать через MTProto, всегда содержит следующие данные, которые проверяются расшифровкой, чтобы сделать систему устойчивой против известных проблем с компонентами:

* server salt (соль сервера) (64-битная)
* session id (идентификатор сессии)
* message sequence number (порядковый номер сообщения)
* message length (длина сообщения)
* time (время)

## Краткий обзор компонентов

Высокоуровневая часть (язык RPC-запросов/API)

С точки зрения высокоуровневой части, клиент и сервер обмениваются сообщениями в рамках некоторой сессии. Сессия привязана к клиентскому устройству (вернее, приложению), но не к конкретному http/https/tcp-соединению. Кроме того, каждая сессия привязана к идентификатору пользовательского ключа, по которому фактически производится авторизация.

Может быть открыто несколько соединений к серверу; сообщения в ту или иную сторону могут идти по любому из них (ответ на запрос не обязан прийти по тому же соединению, по которому был отправлен сам запрос, хотя чаще всего это так; однако ни в коем случае сообщение не может быть возвращено в соединении, принадлежащем другой сессии). При использовании UDP-протокола может случиться, что ответ

на запрос приходит не с того IP, на который был оправлен запрос. Сообщения бывают нескольких типов:

* RPC-вызовы (от клиента к серверу)
* RPC-результаты (от сервера к клиенту)
* Подтверждение приема сообщени
* Запрос состояния сообщений
* Составное сообщение или контейнер

С точки зрения протоколов более низкого уровня, сообщение — это поток двоичных данных, выровненный по границе 4 или 16 байтов. Первые несколько полей сообщения фиксированы и используются системой криптографии/авторизации.

Каждое сообщение, отдельное или внутри контейнера, состоит из идентификатора сообщения (64 бита; см. ниже), порядкового номера сообщения в сессии (32 бита), длины (тела в байтах; 32 бита) и тела (любой размер, кратный 4 байтам). Кроме того, при отправке контейнера или одиночного сообщения, в егоначало дописывается внутренний заголовок (см. ниже), после чего все это шифруется, и в начало зашифрованного сообщения добавляется внешний заголовок (64-битный идентификатор ключа и 128-битный ключ сообщения).

Тело сообщения обычно состоит из 32-битного типа сообщения, за которым следуют параметры, зависящие от типа. В частности, каждой RPC-функции соответствует свой тип сообщения. Более подробно читайте в статье про [двоичную](https://tlgrm.ru/docs/mtproto/serialize) [сериализацию данных](https://tlgrm.ru/docs/mtproto/serialize) и [служебные сообщения](https://tlgrm.ru/docs/mtproto/service_messages).

.

Все числа записываются как little-endian. Однако очень большие числа (2048-битные), используемые в RSA и DH, записываются как big-endian, потому что так делает библиотека OpenSSL.

## Авторизация и криптография

Перед передачей сообщений (или составных сообщений) по сети посредством транспортного протокола они шифруются определенным образом; при этом перед сообщением приписывается внешний заголовок: 64-битный *идентификатор ключа* (однозначно определяющий *авторизационный ключ* для сервера,а также *пользователя*) и 128-битный *ключ* сообщения. Пользовательский ключ вместе с ключом сообщения определяют реальный 256-битный ключ, которым и зашифровано сообщение посредством шифра AES-256. Начало тела незашифрованного сообщения содержит некоторые данные (сессию, идентификатор сообщения, порядковый номер сообщения в сессии, серверную соль); *ключ сообщения* должен совпадать с младшими 128 битами SHA1 от тела сообщения (включая сессию, идентификатор сообщенияи т.п.). Составные сообщения шифруются как единое целое.

Первым делом клиентское приложение должно произвести создание авторизационного ключа, который обычно создается при первом запуске и практически никогда не изменяется.

Основной недостаток протокола — в том, что злоумышленник, пассивно перехватывающий сообщения, а затем каким-либо образом заполучивший авторизационный ключ (например, украв устройство) получит возможность расшифровать все перехваченные сообщения *post factum*. Вероятно, это не слишком серьезно (украв устройство, можно получить и всю закешированную на нем информацию, ничего не расшифровывая), однако для преодоления этих проблем можно сделать следующее:

* *Сессионные ключи*, генерируемые по протоколу Диффи-Хелмана, и используемые совместно с авторизационным ключом и ключом сообщения для выбора параметров AES. Для их создания клиент должен первым действием после создания новой сессии отправить серверу специальный RPC-запрос («сгенерировать сессионный ключ»), сервер ответит на него, после чего все последующие сообщения сессии шифруются с учетом и сессионного ключа.
* Защищать ключ, хранимый на клиентском устройством, (текстовым) паролем; этот пароль никогда не хранится в памяти и вводится пользователем при запуске приложения или чаще (в зависимости от настроек приложения).

Данные, хранимые (кэшируемые) на пользовательском устройстве, можно также защищать, шифруя с помощью авторизационного ключа, который, в свою очередь, надо защитить паролем. Тогда без ввода пароля невозможно будет получить доступ

## Транспорт

Позволяет доставлять уже зашифрованные контейнеры вместе с внешним заголовком (в дальнейшем — полезную нагрузку) от клиента к серверу и наоборот. Есть три типа транспорта:

* HTTP
* TCP
* UDP

# Описание программной реализации MTProto

Работа с потоками осуществляется при помощи класса std::thread

Работа с большими числами осуществляется при помощи библиотеки gmp

Также при помощи gmp реализован протокол Диффи-Хеллмана (модуль keyExchange.cpp) и функция getDigit (модуль digits.cpp).

Работа с шифрованием осуществляется при помощи библиотеки cryptopp

При помощи cryptopp реализованы такие алгоритмы, как:

* RAW-RSA (модуль rsa.cpp)
* AES256 (модуль aes.cpp)
* Base64Encoder (модуль aes.cpp)
* Base64Decoder (модуль aes.cpp)
* SHA256 (модуль sha256.cpp)

Кодировка base64 используется только в модуле aes

## Модуль server.cpp

### Функция **main**

Функция реализует работу с сокетами, инициализацию базы данных, создание нового потока.

Инициализация базы данных: создается база данных, содержащая таблицу:

**Таблица 1** - структура базы данных сервера.

|  |  |  |
| --- | --- | --- |
| SOCK\_ID | CHAR(64) | NOT NULL |
| SESSION\_ID | CHAR(64) | NOT NULL |
| AUTH\_KEY | CHAR(2048) | NOT NULL |

Где sock\_id - постоянный id клиента (дескриптор файла для сокета используется как id клиента на текущую сессию),

session\_id – id, необходимый для работы протокола mtproto,

auth\_key - уникальный ключ, использующийся для работы протокола mtproto,

### Функция **sendMsg**

Реализует протокол mtproto на стороне сервера. Принимает два параметра - массив sockets (содержащий sock\_id всех клиентов) и sock\_id отправителя. В рамках этой функции sockets выступает в качестве массива с sock\_id получателей.

Клиент и сервер обмениваются структурой package, в которой объявлены поля:

* sender\_session\_id - сессионный идентификатор клиента-отправителя, генерируется в ходе раунда обмена ключей (модуль keyExhange )
* recipient\_session\_id - сессионный идентификатор клиента-получателя (не используется)
* msg\_len - длина сообщения
* msg\_key - уникальный идентификатор сообщения
* encrypted\_data - зашифрованная информация AES256(salt+session\_id+msg+padding)

db\_decryption\_aes\_key – ключ для дешифрования данных из базы данных

db\_decryption\_aes\_iv - инициализационный вектор для дешифрования данных из базы данных

### Прием пакета от клиента-отправителя

Принимается информация от клиента-отправителя. Затем сервер, зная sock\_id клиента-отправителя, получает его auth\_key из базы данных.

Сервер расшифровывает auth\_key из базы данных при помощи db\_decryption\_aes\_key и db\_decryption\_aes\_iv

Сервер получает aes\_key и aes\_iv, используя auth\_key и msg\_key клиента- отправителя, расшифровывает encrypted\_data и получает decrypted\_data.

Далее необходимо зашифровать decrypted\_data для отправки клиентам- получателям

Сервер, зная sock\_id клиента-получателя, получает его auth\_key из базы данных.

Сервер расшифровывает auth\_key из базы данных при помощи db\_decryption\_aes\_key и db\_decryption\_aes\_iv

Сервер получает aes\_key и aes\_iv, используя auth\_key клиента- получателя и msg\_key клиента-отправителя, зашифровывает decrypted\_data и отправляет клиенту-получателю.

Повтор шагов 2.1-2.3 для всех активных sock\_id

## Модуль client.cpp

### Функция **main**

Реализует работу с сокетами.

RSA ключи клиента генерируются заного при запуске программы. В раунде обмена ключами публичный ключ клиента отправляется на сервер.

Инициализация базы данных -создается база данных, содержащая таблицу:

**Таблица 2** - Структура базы данных клиента.

|  |  |  |
| --- | --- | --- |
| SESSION\_ID | CHAR(64) | NOT NULL |
| AUTH\_KEY | CHAR(2048) | NOT NULL |

При подключении нового клиента создает для него два потока обработки сообщений - приём и отправка

Отправляет зашифрованное сообщение серверу Структура package идентична структуре package на сервере. Ход работы:

Получение auth\_key и session\_id из базы данных

Формирование блока to\_be\_encrypted (salt+msg+padding)

Формирование aes\_key и aes\_iv и шифрование блока to\_be\_encrypted AES256Encode(to\_be\_encrypted, aes\_key, aes\_iv)

Отправка на сервер

### Функция **getMsg**

### Принимает зашифрованное сообщение от сервера

Структура package идентична структуре package на сервере.

Ход работы:

1. Получение auth\_key и session\_id из базы данных
2. Формирование aes\_key и aes\_iv и дешифрование decrypted\_data
3. Вывод сообщения в консоль

## Модуль keyExchange.cpp

Генерация auth\_key посредством протокола Диффи-Хеллмана

Функция **getNewSession\_client** - Обмен ключами, сторона сервера

Функция **getNewSession\_server** - Обмен ключами, сторона клиента

Сервер обменивается с клиентом структурой package Структура package клиента аналогична структуре package сервера:

session\_id - сессионный идентификатор нового клиента

dh\_aes\_key – зашифрованный при помощи открытого ключа сервера/клиента AES ключ

dh\_aes\_iv - зашифрованный при помощи открытого ключа сервера/клиента AES IV.

p– 2048 битное число P, генерируется на сервере (см. digits.cpp)

g- 64 битное число g, генерируется на сервере (см. digits.cpp)

A – g^amod p для сервера или g^bmod p для клиента Сторона сервера:

Генерируется session\_id и присваивается новому клиенту. Этот параметр нужен для составления блока tobeencrypted в дальнейшем

Сервер принимает открытый RSA-ключ клиента

Сервер генерирует числа p, g, a(64 бит) и высчитывает число A = g^amod p

Сервер шифрует число A при помощи ключа dh\_aes\_key и dh\_aes\_iv, которые шифруются при помоощи открытого ключа клиента. Затем отправляет package клиенту

### Сторона клиента

Клиент генерирует 64 битное число b и принимает package от сервера. Затем расшифровывает число A при помощи dh\_aes\_key, dh\_aes\_iv, которые в свою очередь расшифровываются при помощи закрытого ключа клиента

Клиент высчитывает число B = g^bmod p и шифрует его при помощи ключа dh\_aes\_key и dh\_aes\_iv, затем отправляет package на сервер. AES параметры шифруются при помощи открытого ключа сервера.

При помощи А высчитывает auth\_key (A^bmod p).

### Сторона сервера

Сервер принимает package и расшифровывает dh\_aes\_key, dh\_aes\_iv при помощи закрытого ключа сервера.

При помощи dh\_aes\_key, dh\_aes\_iv расшифровывает число B и высчитывает auth\_key (B^amod p).

Шифрует auth\_key при помощи db\_aes\_key и db\_aes\_iv

Сохраняет в базу данных sock\_id, session\_id, auth\_key

## Модуль msg\_encr\_decr.cpp

### Функция **getEncryptedBlock**

Принимает два параметра: session\_id и msg

Возвращает строку to\_be\_encrypted длиной 1024 байта

to\_be\_encrypted = salt + session\_id + msg + padding, где padding нужен для дополнения до 1024 байт

Функция **get\_msg\_key**- возвращает msg\_key

Функция **get\_aes\_key** - возвращает aes\_key

Функция **get\_aes\_iv** - возвращает aes\_iv msg\_key,

### Формулы aes\_key, aes\_iv, msg\_key

aes\_key, aes\_iv высчитываются по специальным формулам:

msg\_key\_large = SHA256 (substr (auth\_key, 88+x, 32) + plaintext + random\_padding);

msg\_key = substr (msg\_key\_large, 8, 16);

sha256\_a = SHA256 (msg\_key + substr (auth\_key, x, 36));

sha256\_b = SHA256 (substr (auth\_key, 40+x, 36) + msg\_key);

aes\_key = substr (sha256\_a, 0, 8) + substr (sha256\_b, 8, 16) + substr (sha256\_a, 24, 8);

aes\_iv = substr (sha256\_b, 0, 8) + substr (sha256\_a, 8, 16) + substr (sha256\_b, 24, 8);

## Модуль digits.cpp

### Функция **getDigit**

Выработка чисел по заданным параметрам:

digit - сюда записывается результат

bits - количество битов

mode - если 1, то вырабатывать простое, если 0, то вырабатывать составное

base - если 10, вернуть число в DEC, если 16, вернуть число в H

# Заключение

На наш взгляд, протокол MTProto довольно революционен и перспективен. Его гибкость позволяет использовать его идеи и видоизменять протокол под свои нужды. Изучение и написание этого протокола дало нам бессменный опыт в разработке криптографических протоколов на С++, а так же в процессе исследования протокола мы не только познакомились с самим протоколом, но узнали для себя много новых понятий из области криптографии. Для запуска клиент - сервера необходимо сгенерировать ключи сервера командой *./server keygen* и публичный ключ скопировать в директорию с компилированной версией клиента.

# Список используемых источников

1. Официальная документация по протоколу MTProto 2.0 — Telegram Messenger Inc., 2021 — Электронный ресурс. Доступ по ссылке: <https://core.telegram.org/mtproto> (дата обращения: 15.05.2021)"
2. Библиотека криптографических функций Crypto++, свободное программное обеспечение, 2021 — Электронный ресурс. Доступ по ссылке: <https://www.cryptopp.com/wiki/Raw_RSA> (дата обращения: 15.05.2021)"
3. Библиотека криптографических функций Crypto++, свободное программное обеспечение, 2021 — Электронный ресурс. Доступ по ссылке: <https://cryptopp.com/wiki/Advanced_Encryption_Standard> (дата обращения: 15.05.2021)"
4. Библиотека криптографических функций Crypto++, свободное программное обеспечение, 2021 — Электронный ресурс. Доступ по ссылке: <https://www.cryptopp.com/wiki/Base64Encoder> (дата обращения: 15.05.2021)"
5. Библиотека криптографических функций Crypto++, свободное программное обеспечение, 2021 — Электронный ресурс. Доступ по ссылке: <https://www.cryptopp.com/wiki/Keys_and_Formats> (дата обращения: 15.05.2021)"
6. Библиотека криптографических функций Crypto++, свободное программное обеспечение, 2021 — Электронный ресурс. Доступ по ссылке: <https://www.cryptopp.com/wiki/Base64Decoder> (дата обращения: 15.05.2021)"
7. Библиотека криптографических функций Crypto++, свободное программное обеспечение, 2021 — Электронный ресурс. Доступ по ссылке: <https://www.cryptopp.com/wiki/SHA2> (дата обращения: 15.05.2021)"
8. Официальная документация по работе с библиотекой GMP, свободное программное обеспечение, 2021 — Электронный ресурс. Доступ по ссылке: <https://gmplib.org/> (дата обращения: 10.05.2021)"
9. Официальная документация по работе с библиотекой криптографических функций Crypto++, свободное программное обеспечение, 2021 — Электронный ресурс. Доступ по ссылке: <https://www.cryptopp.com/> (дата обращения: 10.05.2021)"

Приложение

Исходный код прототипа протокола MTProto

* <https://github.com/KomogorovKirill/MTProto.git>