МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное бюджетное образовательное учреждение высшего образования

«МОСКОВСКИЙ АВИАЦИОННЫЙ ИНСТИТУТ

(национальный исследовательский университет)» (МАИ)

Институт №8 «Информационные технологии и прикладная математика»

Кафедра 806 «Вычислительная математика и программирование»

КУРСОВАЯ РАБОТА

по дисциплине “Основы криптографии”

на тему: “Разработка клиент-серверного приложения передачи шифрованных данных. LOKI97. RC6”

Руководитель:

к.т.н., профессор кафедры 806

Романенков Александр Михайлович

Подпись _____

Выполнила:

студентка группы М8О-311Б-21

Соломатина Светлана Викторовна

Подпись _____

Оценка:

Дата:

Москва, 2024

Оглавление.

Введение.

Данная курсовая работа посвящена реализации системы защищенного обмена сообщениями между двумя клиентами. Целью работы является создание серверного приложения, взаимодействующего с клиентами через сетевой протокол, а также реализация клиентского приложения с графическим интерфейсом, обеспечивающего безопасный обмен сообщениями с использованием симметричного алгоритма шифрования и протокола обмена ключами Диффи-Хеллмана.

В курсовой работе будут реализованы симметричные алгоритмы шифрования LOKI97 и RC6, режимы шифрования ECB, CBC, PCBC, CFB, OFB, CTR, Random Delta, режимы набивки Zeros, ANSI X.923, PKCS7, ISO 10126. Также будут созданы интерфейсы для работы с симметричными алгоритмами шифрования, режимами шифрования и набивки, интерфейс для шифрования и дешифрования массива байт произвольной длины и файла. Кроме того, будет описан и реализован протокол Диффи-Хеллмана для безопасного распределения сеансового ключа.

Другими задачами, решаемыми в рамках этой работы, будут:

- разработка консольного серверного приложения, обслуживающего клиентов через сетевой протокол gRPC,
- разработка клиентского приложения с графическим интерфейсом, обеспечивающим удобное взаимодействие с сервером.

К подзадачам создания серверного приложения также относится реализация API сервера, обеспечивающего возможность создания и управления секретными чатами, распределения сеансового ключа с помощью протокола Диффи-Хеллмана и обработки сообщений между клиентами.

К подзадачам создания клиентского приложения относится внедрение брокера сообщений для обеспечения асинхронного взаимодействия между сервером и клиентами, реализация функциональности для создания секретных чатов, подключения и отключения от них, выбора алгоритма шифрования и режима шифрования, набивки, внедрение возможности шифрования и дешифрования данных, отправки и получения сообщений в секретных чатах.

Данная курсовая работа позволит получить практический опыт в разработке систем защищенного обмена сообщениями с использованием криптографических технологий и сетевых протоколов.

Теоретическая часть.

Описание реализованных алгоритмов шифрования.

LOKI97.

LOKI97 - симметричный блочный шифр, разработанный в 1997 году специально для конкурса Advanced Encryption Standard (AES). В основе шифра лежит сбалансированная сеть Фейстеля с использованием 16 циклов и функции Фейстеля, обеспечивающей смешивание входных битов.

Размер ключа: 128/192/256 бит.

Размер входного блока: 128 бит.

В алгоритме используются следующие операции:

$\oplus$: операция XOR - побитового исключающего “или”;

$+$: операция целочисленного сложения по модулю 2^{64};

$-$: операция целочисленного вычитания по модулю 2^{64}.

Спецификация алгоритма.

Шифрование алгоритмом LOKI97 выглядит следующим образом. Входной блок $[L|R]$ размера 128 бит делится на два 64-битных блока:

$L_0 = L$,

$R_0 = R$,

после чего эти два блока проходят 16 раундов (i = 1, 2, …, 16) сбалансированной сети Фейстеля по следующей схеме:

$R_i = L_{i-1} \oplus f(R_{i-1} + SK_{3i-2}, SK_{3i-1})$,

$L_i = R_{i-1} + SK_{3i-2} + SK_{3i}$,

где SK_i - i-й раундовый ключ, $f(A, B)$ - функция Фейстеля.

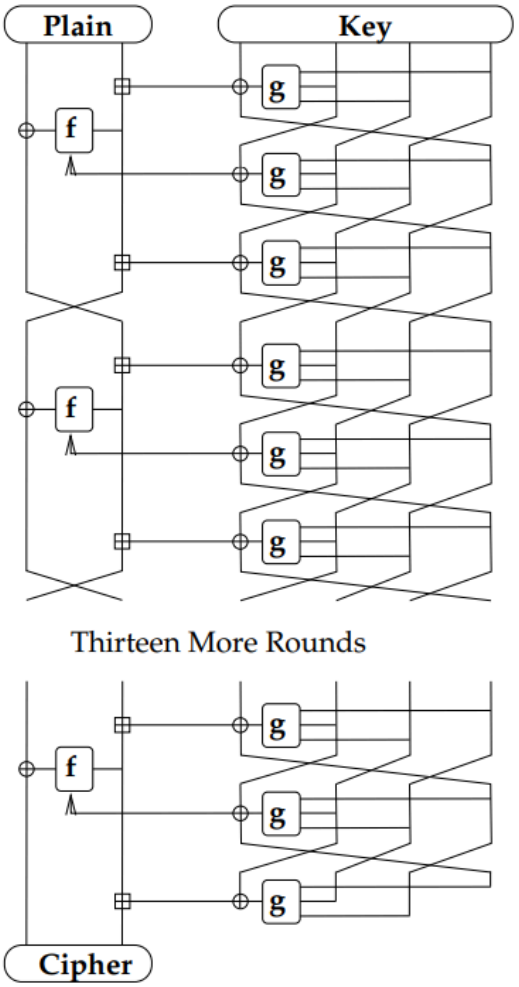

Рисунок 1 - Схема алгоритма LOKI97.

Получившийся шифротекст строится как $[R_{16}|L_{16}]$.

Дешифрование происходит аналогично шифрованию, но в обратном порядке: шифротекст $[R_{16}|L_{16}]$ проходит 16 раундов ($i = 16, 15, ..., 1$) сбалансированной сети Фейстеля по следующей схеме:

$R_{i-1} = L_i \oplus f(R_i - SK_{3i}, SK_{3i-1})$,

$L_{i-1} = R_{i-1} - SK_{3i} - SK_{3i-2}$.

Результат дешифрования - блок размером 128 бит $[L_0|R_0]$.

Инициализация ключей.

На вход алгоритма могут быть поданы ключи фиксированных размеров: 128, 192 или 256 бит.

Если на вход дан 256-битный ключ $[K_a|K_b|K_c|K_d]$, тогда

$[K4_0|K3_0|K2_0|K1_0] = [K_a|K_b|K_c|K_d]$.

Запись $[A|B] = [C|D]$ означает параллельное присвоение блоков, то есть A = C, B = D.

Если на вход дан 192-битный ключ $[K_a|K_b|K_c]$, тогда

$[K4_0|K3_0|K2_0|K1_0] = [K_a|K_b|K_c|f(K_a, K_b)]$.

Если на вход дан 128-битный ключ $[K_a|K_b]$, тогда

$[K4_0|K3_0|K2_0|K1_0] = [K_a|K_b|f(K_b, K_a)|f(K_a, K_b)]$.

Для каждого раунда нужно 3 раундовых ключа: SK_{3i-2}, SK_{3i-1} и SK_{3i}. Так как раундов 16, необходимо сгенерировать $3 * 16 = 48$ раундовых ключей. Для их генерации производятся 48 ($i = 1, 2, ..., 48$) следующих итераций:

$SK_i = K1_i = K4_{i-1} \oplus g_i(K1_{i-1}, K3_{i-1}, K2_{i-1})$,

$K4_i = K3_{i-1}$,

$K3_i = K2_{i-1}$,

$K2_i = K1_{i-1}$,

где

$g_i(K1, K3, K2) = f(K1 + K3 + (Delta * i), K2)$,

$Delta = (\sqrt{5} - 1) * 2^{63} = 9E3779B97F4A7C15_{16}$.

Листинг 1 - Функция $g(K1, K3, K2)$.

```
    private byte[] g(byte[] K1, byte[] K3, byte[] K2, int i) {
            long deltaMultiplyI = DELTA * i;
            return encryptionConversion.encrypt(
                    Operations.additionByteArraysLength8(K1,
                            Operations.additionByteArrayLength8AndLong(
                                    K3, deltaMultiplyI)), K2);
        }
```

Функция Фейстеля.

Функция Фейстеля в алгоритме LOKI97 описывается выражением:

$f(A, B) = S_b(P(S_a(E(KP(A, B)))), B)$,

где A, B - блоки размером 64 бит. Иллюстрация функции Фейстеля приведена на рисунке 2.

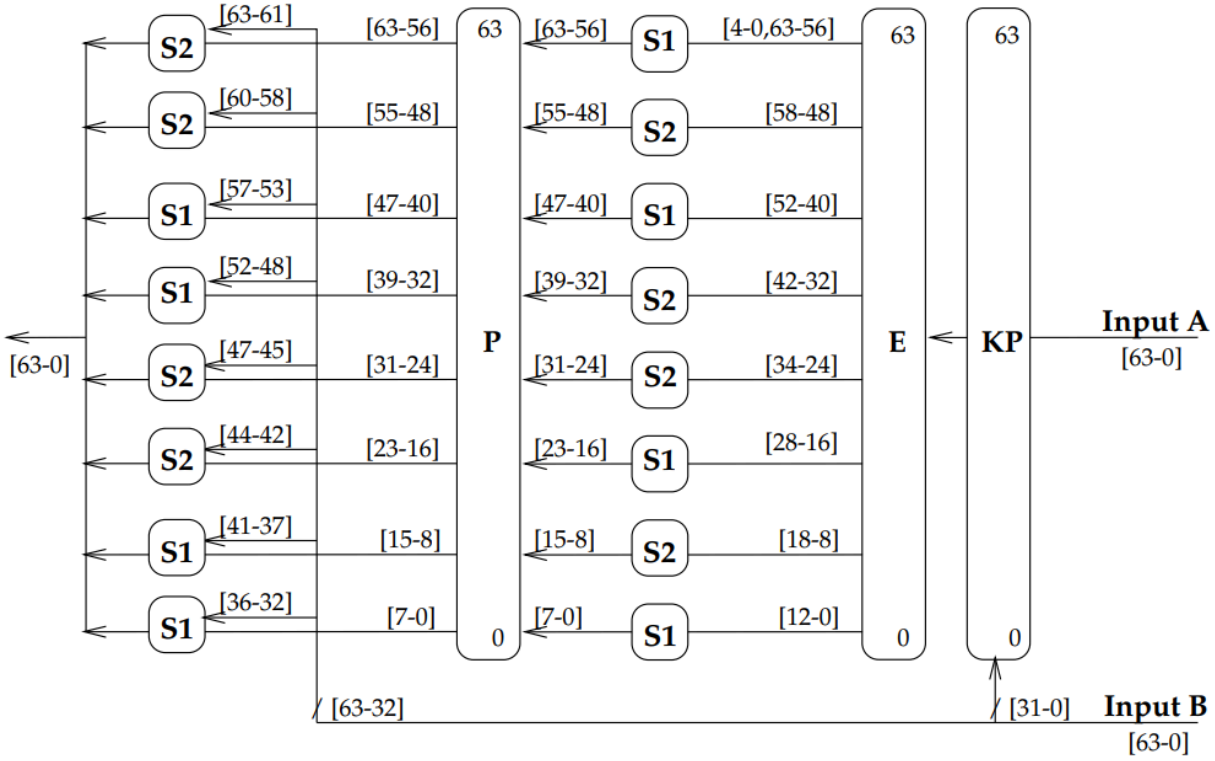

Рисунок 2 - Функция фейстеля LOKI97.

Дополнительная информация о каждой функции приведена ниже.

Используется следующее соглашение о нумерации битов: в каждом 64-битном слове бит 0 является самым правым и наименее значимым битом; бит с наибольшим номером (31, 63 или 127 в 4-, 8- или 16-байтовых словах соответственно) — это самый левый и самый значимый бит.

$KP(A,\ B)$

Функция перемешивания битов. Входные блоки A, B разбиваются следующим образом:

$A = [A_l, A_r]$,

$B = [B_l, B_r]$.

На выходе функция дает 64-битный результат согласно формуле

$KP([A_l|A_r],\ SK_r) = [((A_l\ \&\ \neg SK_r)\ ||\ (A_r\ \&\ SK_r))\ |\ ((A_r\ \&\ \neg SK_r)\ ||\ (A_l\ \&\ SK_r))]$,

где || - операция побитовой дизъюнкции;

& - операция побитовой дизъюнкции;

¬ - операция побитового отрицания.

В листинге 2 приведен код данной функции.

Листинг 2 - Функция KP(A, B).

```
private byte[] KP(byte[] A, byte[] roundKey) {
        int lenLeftRight = A.length / 2;
        byte[] leftA = new byte[lenLeftRight];
        byte[] rightA = new byte[lenLeftRight];
        byte[] rightRoundKey = new byte[lenLeftRight];
        System.arraycopy(A, 0, leftA, 0, lenLeftRight);
        System.arraycopy(A, lenLeftRight, rightA, 0, lenLeftRight);
        System.arraycopy(roundKey, lenLeftRight, rightRoundKey, 0,
lenLeftRight);
        return Operations.mergeByteArrays(
                Operations.or(
                        Operations.and(leftA,
Operations.negate(rightRoundKey)),
                        Operations.and(rightA, rightRoundKey)),
                Operations.or(
                        Operations.and(rightA,
Operations.negate(rightRoundKey)),
                        Operations.and(leftA, rightRoundKey)));
    }
```

$E(A)$

Функция расширения. Преобразует входное 64-битное слово в 96-битное по следующему закону:

$$[4-0|63-56|58-58|52-40|42-32|34-24|28-16|18-8|12-0].$$

Код функции можно увидеть в листинге 3.

Листинг 3 - Функция $E(A)$

```
private byte[] E(byte[] input) {
       return Permutation.permute(input, expansionTableEFromMostToLeast,
indexRule);
    }
```

$S_a(A),\ S_b(A)$

2 группы S-блоков. Строятся при помощи работы с элементами полей Галуа: элементы поля $GF(2^{13})$ для блока S_1 и $GF(2^{11})$ для блока S_2. Построение происходит следующим образом:

$$S1[x] = ((x\ xor\ 1FFF)^3\ mod\ 2911)\ \&\ FF,\ \ in\ \ GF(2^{13})$$
$$S2[x] = ((x\ xor\ 7FF)^3\ mod\ AA7)\ \&\ FF,\ \ in\ \ GF(2^{11})$$

Входные данные для $S_a(A)$ — 96-битное слово на выходе функции $E(C)$. Старшими битами слова для $S_b(A)$ являются старшие 32 бита слова B, использованного как второй аргумент функции $f(A, B)$, а младшими — результат действия функции $P(D)$.

Результатом применения функций являются группы блоков S1, размером 13 бит, и S2, размером 11 бит:

$S_a(A) = [S1, S2, S1, S2, S2, S1, S2, S1]$,

$S_b(A) = [S2, S2, S1, S1, S2, S2, S1, S1]$.

Реализация функции $S_b(A)$ приведена в листинге 4.

Листинг 4 - Функция $S_b(A)$.

```
private byte[] Sb(byte[] input, byte[] roundKey) {
      long inputLong = Operations.bytesArrToLong(input);
      long roundKeyLong = Operations.bytesArrToLong(roundKey);
      long f =
            (S2[(int) (((inputLong >>> 56) & 0xFF) | ((roundKeyLong >>> 53) &
0x700))] & 0xFFL) << 56 |
                  (S2[(int) (((inputLong >>> 48) & 0xFF) | ((roundKeyLong >>> 50) &
0x700))] & 0xFFL) << 48 |
                  (S1[(int) (((inputLong >>> 40) & 0xFF) | ((roundKeyLong >>> 45) &
0x1F00))] & 0xFFL) << 40 |
                  (S1[(int) (((inputLong >>> 32) & 0xFF) | ((roundKeyLong >>> 40) &
0x1F00))] & 0xFFL) << 32 |
                  (S2[(int) (((inputLong >>> 24) & 0xFF) | ((roundKeyLong >>> 37) &
0x700))] & 0xFFL) << 24 |
                  (S2[(int) (((inputLong >>> 16) & 0xFF) | ((roundKeyLong >>> 34) &
0x700))] & 0xFFL) << 16 |
                  (S1[(int) (((inputLong >>> 8) & 0xFF) | ((roundKeyLong >>> 29) &
0x1F00))] & 0xFFL) << 8 |
                  (S1[(int) ((inputLong & 0xFF) | ((roundKeyLong >>> 24) &
0x1F00))] & 0xFFL);
      return Operations.longToBytes(f);
   }
```

$P(A)$

Перестановка битов выходного блока функции S_a. 64 бита перемешиваются по следующему правилу (см. таблицу 1):

Таблица 1 - Таблица перестановок функции $P(A)$.

56	48	40	32	24	16	08	00	57	49	41	33	25	17	09	01
58	50	42	34	26	18	10	02	59	51	43	35	27	19	11	03
60	52	44	36	28	20	12	04	61	53	45	37	29	21	13	05
62	54	46	38	30	22	14	06	63	55	47	39	31	23	15	07

Функция P является основным путём перемешивания битов. При её построении преследовалась цель максимально уменьшить вероятность появления закономерностей в распределении входных и выходных битов. За основу таблицы перестановок был взят латинский квадрат.

RC6.

RC6 - симметричный блочный шифр, производный от алгоритма RC5. Был разработан в 1998 году специально для конкурса AES. В основе шифра лежит сбалансированная сеть Фейстеля. По сути RC6 - полностью параметризованная семья алгоритмов шифрования. Для спецификации алгоритма с конкретными параметрами принято обозначение RC6-w/r/b, где

w - длина машинного слова в битах,

r - количество раундов сети Фейстеля,

b - длина ключа в битах.

Размер входного блока: 128 бит.

В алгоритме используются следующие операции:

$\oplus$: операция XOR - побитового исключающего “или”;

$+$: операция целочисленного сложения по модулю 2^{32};

$-$: операция целочисленного вычитания по модулю 2^{32};

$\times$: операция целочисленного умножения по модулю 2^{32};

$a \ <<< \ b$: операция битового циклического сдвига влево на b бит;

$a \ >>> \ b$: операция битового циклического сдвига вправо на b бит.

Реализованный для данного Курсового проекта алгоритм имеет спецификацию RC6-32/20/128.

Спецификация алгоритма.

На вход алгоритму подается 4 w-битных слова A, B, C, D. Результат шифрования - объединённые те же 4 w-битных слова A, B, C, D. В основе алгоритма лежит стандартные r раундов сети Фейстеля.

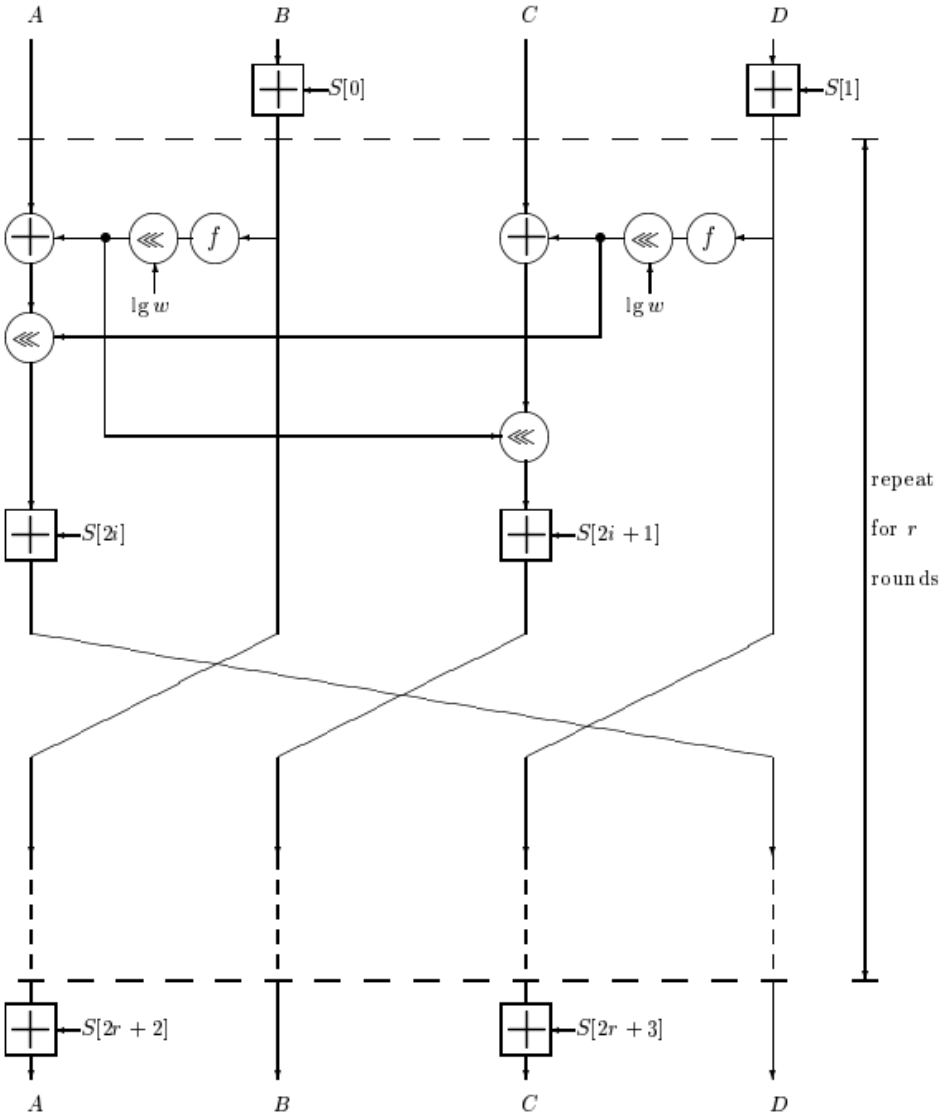

Рисунок 3 - Схема алгоритма RC6.

Реализация сети Фейстеля для шифрования алгоритмом RC6 представлена в листинге 5.

Листинг 5 - Метод для шифрования блока алгоритмом RC6.

```
public byte[] methodForConstructingBlockCiphersEncryption(byte[] input,
int roundCount) {
        if (roundKeys.length != 2 * (roundCount + 2)) {
            throw new
IllegalArgumentExceptionWithLog("methodForConstructingBlockCiphersEncryp
tion: " +
                    "Param roundCount must be 2 * (roundCount + 2) ==
roundKeys.length", log);
        }

        int lenInput = input.length;
        int wBytes = RC6.w / 8;

        if (lenInput != wBytes * 4) {
            throw new
IllegalArgumentExceptionWithLog("methodForConstructingBlockCiphersEncryp
tion:" + "Param input must be length 16 bytes", log);
        }
        int log2w = (int) (Math.log(wBytes * 8) / Math.log(2));

        // encryption RC6
        byte[] A = new byte[wBytes];
        byte[] B = new byte[wBytes];
        byte[] C = new byte[wBytes];
        byte[] D = new byte[wBytes];

        System.arraycopy(input, 0, A, 0, wBytes);
        System.arraycopy(input, wBytes, B, 0, wBytes);
        System.arraycopy(input, 2 * wBytes, C, 0, wBytes);
        System.arraycopy(input, 3 * wBytes, D, 0, wBytes);

        B = Operations.additionByteArraysLength4(B, roundKeys[0]);
        D = Operations.additionByteArraysLength4(D, roundKeys[1]);

        for (int i = 1; i <= roundCount; i++) {
            byte[] t = Operations.intToBytes(
                    Operations.cyclicShiftLeftInt(
                            Operations.bytesArrToInt(
Operations.multiplyingByteArrayLength4(B,
Operations.additionByteArrayLength4AndInt(
Operations.multiplyingByteArrayLength4AndInt(B, 2), 1))), log2w));

            byte[] u =
Operations.intToBytes(Operations.cyclicShiftLeftInt(
                    Operations.bytesArrToInt(
                            Operations.multiplyingByteArrayLength4(D,
Operations.additionByteArrayLength4AndInt(
Operations.multiplyingByteArrayLength4AndInt(D, 2), 1))), log2w));

            A = Operations.additionByteArraysLength4(
```

```
                    Operations.intToBytes(
                            Operations.cyclicShiftLeftInt(
                                    Operations.bytesArrToInt(
                                            Operations.xor(A, t)),
                                    Operations.bytesArrToInt(u))),
                    roundKeys[2 * i]);

            C = Operations.additionByteArraysLength4(
                    Operations.intToBytes(
                            Operations.cyclicShiftLeftInt(
                                    Operations.bytesArrToInt(
                                            Operations.xor(C, u)),
                                    Operations.bytesArrToInt(t))),
                    roundKeys[2 * i + 1]);

            byte[] tmp = A;
            A = B;
            B = C;
            C = D;
            D = tmp;
        }

        A = Operations.additionByteArraysLength4(A, roundKeys[2 *
roundCount + 2]);
        C = Operations.additionByteArraysLength4(C, roundKeys[2 *
roundCount + 3]);

        A = Operations.reverseByteArray(A);
        B = Operations.reverseByteArray(B);
        C = Operations.reverseByteArray(C);
        D = Operations.reverseByteArray(D);

        return Operations.mergeByteArrays(A,
                Operations.mergeByteArrays(B,
                        Operations.mergeByteArrays(C, D)));
    }
```

Дешифрование происходит в обратном порядке: операции сложения заменяются вычитанием, притом сам процесс дешифрования копирует действия шифрующего метода с конца.

Инициализация ключей.

Для r раундов сети Фейстеля необходимо $2r + 2$ раундовых ключей. Их генерация происходит при помощи поданного на вход b-битного ключа.

Для генерации понадобятся следующие константы:

$$Q_w = Odd((f - 1) \times 2^w,$$

$$P_w = Odd((e - 2) \times 2^w.$$

Для значения $w = 32$ в десятичном виде получаются следующие значения:

$$Q_w = 2654435769,$$

$$P_w = 3084996963.$$

Код алгоритма инициализации ключей продемонстрирован в листинге 6.

Листинг 6 - генерация ключей алгоритма RC6.

```
  public byte[][] generateRoundKeys(byte[] key) {
        int w = RC6.w;
        int wBytes = w / 8;
        int c = key.length / wBytes;
        byte[][] L = new byte[c][wBytes];
        for (int i = 0; i < c; i++) {
            System.arraycopy(key, i * wBytes, L[i], 0, wBytes);
        }
        int t = 2 * (ROUND_COUNT_RC6 + 2);
        byte[][] S = new byte[t][w / 8];

        S[0] = Operations.intToBytes(P32);
        for (int i = 1; i < t; i++) {
        S[i] = Operations.additionByteArrayLength4AndInt(S[i - 1], Q32);
        }
        int A = 0, B = 0, i = 0, j = 0;
        int log2w = (int) (Math.log(wBytes * 8) / Math.log(2));
        int v = 3 * max(c, t);
        for (int s = 1; s <= v; s++) {
            S[i] = Operations.intToBytes(
                    Operations.cyclicShiftLeftInt(
                            Operations.bytesArrToInt(S[i]) + A + B, 3));
            A = Operations.bytesArrToInt(S[i]);
            L[j] = Operations.intToBytes(
                    Operations.cyclicShiftLeftInt(
                            Operations.bytesArrToInt(L[j]) + A + B,
                            A + B));
            B = Operations.bytesArrToInt(L[j]);
            i = (i + 1) % t;
            j = (j + 1) % c;
        }
        return S;
    }
}
```

Описание протоколов взаимодействия.

Протокол Диффи-Хеллмана.

Протокол Диффи-Хеллмана – это криптографический протокол, позволяющий двум и более сторонам получить общий секретный ключ, используя незащищенный от прослушивания канал связи. Полученный ключ используется для шифрования дальнейшего обмена с помощью алгоритмов симметричного шифрования.

Описание алгоритма.

При работе алгоритма каждая сторона:

1. Знает числа g и p, не являющиеся секретными. g, p – простые числа, g – первообразный корень по модулю p.
2. Генерирует случайное натуральное число a – закрытый ключ.
3. Вычисляет открытый ключ A, используя преобразование над закрытым ключом$A = g^a mod\ p$.
4. Обменивается открытыми ключами с удалённой стороной.
5. Вычисляет общий секретный ключ K, используя открытый ключ удаленной стороны B и свой закрытый ключ a: $K = B^a mod\ p = g^{ab} mod\ p$, где b – закрытый ключ другой стороны.

Протокол клиент-серверного взаимодействия.

Клиент-серверное взаимодействие в курсовой работе построено с использованием сетевого протокола gRPC (система удаленного вызова процедур). В этом протоколе описание интерфейса взаимодействия производится с помощью Protocol Buffers – протокола сериализации структурированных данных, являющимся альтернативой текстовому формату xml.

В реализованном клиент-серверном взаимодействии считается, что если собеседник клиента отключился от сервера или удалил секретный чат с этим клиентом, то клиент должен удалить секретный чат с этим собеседником.

Рассмотрим реализованные интерфейсы удаленного вызова процедур.

Подключение клиента к серверу. Клиент посылает запрос, в который входит его логин. Сервер фиксирует, что клиент активен и возвращает ему число g, которое будет необходимо при применении протокола Диффи-Хеллмана.

Отключение клиента от сервера. Клиент посылает запрос, в который входит его логин. Сервер фиксирует, что клиент не активен и обновляет список

пользователей, которых нужно предупредить, что клиент отключился от сервера.

Проверка, отключился ли какой-либо собеседник от сервера. Клиент посылает запрос, в который входит его логин. Сервер отдает клиенту потоковый ответ, который содержит всех собеседников клиента, отключившихся от сервера или, если таких нет, включает только значение по умолчанию.

Запрос на создание комнаты. Клиент посылает запрос, в который входят: его логин, логин собеседника, а также выбранные клиентом характеристики чата: режим шифрования, режим набивки, алгоритм шифрования. Сервер, если собеседник активен и ранее не была создана комната между собеседником и клиентом или получен от собеседника запрос на создание комнаты с этим клиентом, кладет запрос клиента для ознакомления собеседника и отвечает клиенту сообщением, в котором содержатся: логин собеседника, с которым создается чат, режим шифрования, режим набивки, алгоритм шифрования, вектор инициализации и число p, необходимое для применения протокола Диффи-Хеллмана.

Проверка, есть ли какой-либо запрос на создание комнаты. Клиент посылает запрос, в котором содержится его логин. Сервер отдает потоковый ответ, в который входят все сообщения, несущие информацию о комнатах, инициализированной с этим клиентом, или, если таких нет, включает только значение по умолчанию. Каждое такое сообщение содержит логин собеседника, с которым создается чат, режим шифрования, режим набивки, алгоритм шифрования, вектор инициализации и число p, необходимое для применения протокола Диффи-Хеллмана.

Передача открытого ключа по протоколу Диффи-Хеллмана. Клиент посылает запрос, в котором содержится его логин, логин собеседника и открытый ключ. Сервер отвечает удачей или неудачей фиксации ключа для передачи указанному собеседнику.

Проверка, пришли ли какие-либо открытые ключи. Клиент посылает запрос, в котором содержится его логин. Сервер отдает потоковый ответ, в который входят все полученные для этого клиента открытые ключи с логином собеседника, передавшего ключ, или, если таких нет, включает только значение по умолчанию.

Удаление комнаты (закрытие чата). Клиент посылает на сервер запрос, в котором содержатся его логин и логин собеседника, с которым он хочет удалить комнату. Сервер отвечает удачей или неудачей попытки удаления комнаты с этим клиентом. Неудача может возникнуть, если такой комнаты не было либо собеседник уже отправил запрос на удаление этой комнаты. В случае удачи запрос клиента фиксируется для уведомления собеседника.

Проверка, удалил ли кто-либо комнату. Клиент посылает запрос, в котором содержится его логин. Сервер отдает потоковый ответ, в который входят все удаленные комнаты, или, если таких нет, включает только значение по умолчанию.

Запросы на проверку, отключился ли какой-либо собеседник от сервера, есть ли запрос на создание комнаты, пришли ли какие-либо открытые ключи, удалил ли какой-либо собеседник комнату выполняются в запланированном режиме, т.е. раз в некоторый промежуток времени.

Рассмотрим реализованные протоколы клиент-серверного взаимодействия.

Задача создания “комнаты” для организации защищенного обмена сообщениями между двумя клиентами (секретный чат) с возможностью выбора используемого алгоритма шифрования. Пусть клиент 1 хочет создать комнату с клиентом 2. Клиент 1 посылает на сервер запрос на создание комнаты. Если такой запрос удачный, то клиент 1 передает открытый ключ по протоколу Диффи-Хеллмана для клиента 2. Клиент 2 должен послать запрос на проверку, есть ли какой-либо запрос на создание комнаты. Он получает запрос от клиента 2 и тоже отсылает свой открытый ключ. После отправки открытого ключа оба клиента посылают запрос на проверку, пришел ли открытый ключ. В результате, у обоих клиентов есть информация о собеседнике, а также используемых в комнате режиме шифрования, режиме набивки, алгоритме, векторе инициализации и ключе. Схема создания “комнаты” изображена на рисунке 4.

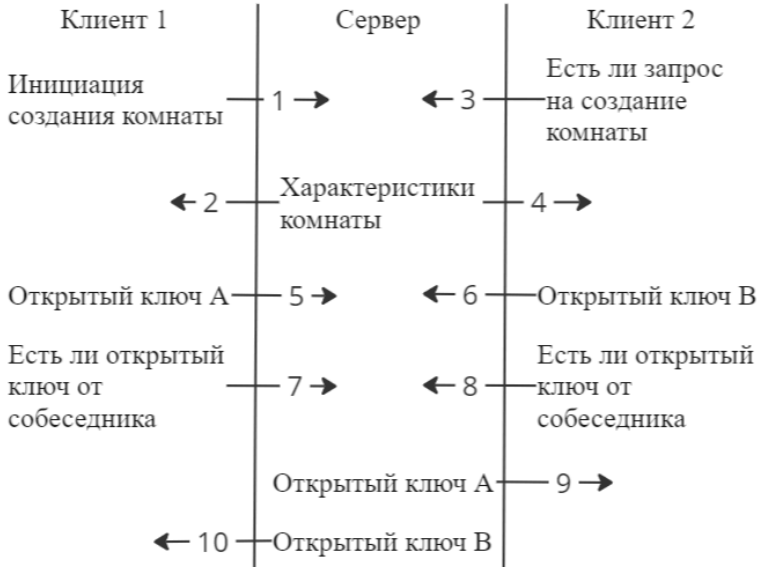

Рисунок 4. Клиент-серверное взаимодействие при создании секретного чата.

Задача закрытия ранее созданного секретного чата (рисунок 5). Пусть клиент 1 хочет закрыть чат с клиентом 2. Клиент 1 посылает на сервер запрос о закрытии чата с клиентом 2. Клиент 2 должен послать запрос на проверку,

удалил ли кто-либо комнату. В результате запроса он узнает, что клиент 1 удалил с ним комнату.

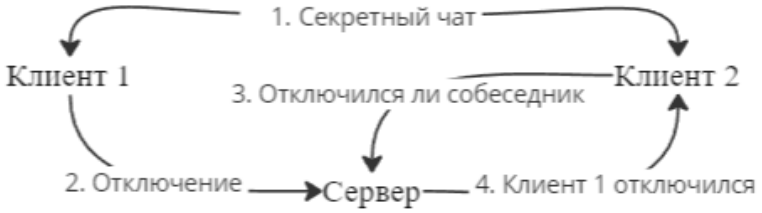

Рисунок 5. Клиент-серверное взаимодействие при закрытии ранее созданного чата.

Задача отключения клиента (рисунок 6). Пусть клиент хочет отключиться от сервера. Он посылает запрос на сервер о своем отключении. Все клиенты, которые являлись собеседниками отключившегося клиента должны послать запрос на проверку, отключился ли какой-либо собеседник от сервера. В результате они получат ответ, что их собеседник отключился.

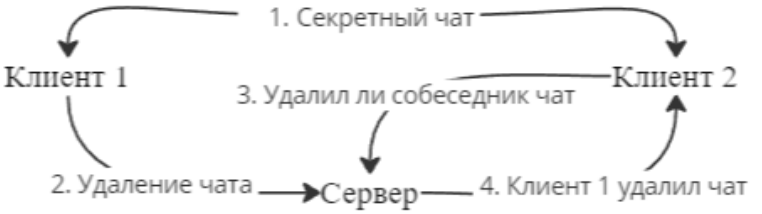

Рисунок 6. Клиент-серверное взаимодействие при отключении клиента от ранее созданного чата.

Задача приема и передачи данных о сообщениях реализована без участия сервера, напрямую между клиентами, в асинхронном стиле с использованием брокера сообщений Apache Kafka, предоставляющего очередь сообщений. Producer и consumer находятся на стороне клиента. Сообщениями, поступающими в брокер, являются фрагменты шифротекста. Объект передачи данных между клиентами имеет следующий формат сериализации: идентификатор сообщения (UUID), информация об отправителе, шифротекст, и, если передается файл – имя файла, количество разбиений, индекс этого разбиения.

Для процессов передачи и приема сообщений необходима созданная между клиентами комната. Это гарантирует наличие одинакового по характеристикам сервиса для шифрования и дешифрования, используемого обоими клиентами.

Общая схема приема и передачи сообщений такова, что сообщения передаются асинхронно, а получаются запланировано. Каждый клиент имеет свой producer и consumer. Последний подписан на специфический для этого клиента topic и раз в некоторый промежуток времени проверяет, не пришли ли в этот topic новые сообщения.

Таким образом, если клиент 1 хочет передать текстовое сообщение клиенту 2, он с помощью kafka producer кладет в специфический для клиента 2 topic объект передачи данных с идентификатором сообщения, своим логином, зашифрованным текстовым сообщением, а также пустым значением имени файла, количеством разбиений, равным одному, и нулевым индексом этого разбиения. Клиент 2 получает этот объект и дешифрует текстовое сообщение.

Если же клиент 1 хочет передать некоторый файл клиенту 2, то он предварительно зашифровывает файл и посылает уже зашифрованный файл по частям, указывая в объекте передачи данных название файла, количество разбиений и индекс конкретного разбиения. Клиент 2 должен убедиться, что он получил и записал все части зашифрованного файла. Тогда клиент 2 дешифрует полученный файл и получает исходный файл.

Практическая часть.

Описание архитектуры комплекса приложений.

Комплекс приложений написан на Java 17. Для реализации механизма внедрения зависимостей используется Spring Framework. Для реализации клиент-серверного взаимодействия применяется gRPC. Для логирования используется библиотека logback, для кодогенерации – библиотека Lombok. Обмен сообщений реализован при помощи платформы Apache Kafka, для чего применены технологии контейнеризации. Web-приложение написано с использованием фреймворка Vaadin.

Проект содержит три модуля: server, client и cryptography-starter. Рассмотрим каждый из них.

Серверная часть. Модуль Server.

Серверная часть состоит из точки входа в приложение, сервиса, обрабатывающего запросы клиентов, вспомогательных репозиториев данных и утилитных классов.

Этап развертывания серверного приложения реализован на основе абстракции инверсии управления (IoC), реализованной в виде механизма внедрения зависимостей (DI) с помощью Spring Framework.

Рассмотрим элементы, составляющие серверную часть.

Точка входа в приложение – класс Server. Он запускает gRPC сервер на прием запросов к сервису, управляющему чатами.

Класс ChatService – это gRPC сервис и Spring компонент. Он наследует генерируемый класс ChatServiceGrpc.ChatServiceImplBase и реализует его методы. Класс ChatService ответственен за обработку различных операций, относящихся к управлению чатами: подключение и отключение клиентов, инициализацию и удаление секретных чатов, обмен открытыми ключами по протоколу Диффи-Хеллмана. Он использует Spring репозитории ActiveUsersRepository, NotifyDisconnectedUsersRepository, DeleteRoomRequestsRepository, AddRoomRequestsRepository, DiffieHellmanNumbersRepository и ChatRoomRepository для управления секретными чатами, состоянием и соединением клиентов. Эти репозитории внедряются в конструктор класса, что позволяет объекту ChatService получать доступ к данным, необходимым для функциональности приложения, и манипулировать ими.

Класс ActiveUsersRepository ответственен за хранение активных клиентов. Для потокобезопасного доступа к данным об активных подключениях в классе

используется ConcurrentHashMap.KeySetView. Объект ActiveUsersRepository предоставляет методы для обновления и чтения коллекции активных пользователей.

Класс NotifyDisconnectedUsersRepository хранит информацию об отключившихся клиентах. В нем используется ConcurrentHashMap, где ключ – логин отключившегося пользователя, а значение – список пользователей, которых нужно уведомить об отключившемся пользователе (его собеседники).

Класс DeleteRoomRequestsRepository хранит информацию о клиентах, удаливших секретный чат. В нем используется ConcurrentHashMap, где ключ – логин собеседника, а значение – логин инициатора удаления комнаты.

Класс AddRoomRequestsRepository хранит информацию о клиентах, для которых было инициировано создание комнаты. В нем используется ConcurrentHashMap, где ключ – логин такого клиента, а значение – ассоциативный контейнер, в котором ключ – логин инициатора создания комнаты, значение – метаданные комнаты (режимы шифрования, набивки, алгоритм шифрования и прочее).

Класс DiffieHellmanNumbersRepository хранит информацию о клиентах, для которых был передан открытый ключ по протоколу Диффи-Хеллмана. В нем используется ConcurrentHashMap, где ключ – логин такого клиента, значение – ассоциативный контейнер, в котором ключ – логин отправителя, значение – открытый ключ, переданный этим отправителем.

Класс ChatRoomRepository хранит информацию об активных чатах клиентов. В нем используется коллекция ConcurrentHashMap.KeySetView, хранящая объекты класса CustomPair.

Класс CustomPair используется в репозитории ChatRoomRepository для репрезентации активных чатов. Это стандартная структура шаблонной пары, за исключением переопределенных методов Equals и HashCode таким образом, чтобы был неважен является ли ключом или значением логин конкретного собеседника.

Класс AlgorithmsConfigs – утилитный класс, инкапсулирующий константы применяемых симметричных алгоритмов, такие как длины ключа и блока. Используется при создании сервисом ChatService метаданных инициализированной комнаты.

Класс InitRoomResponseBuilder – утилитный класс, используемый для создания метаданных инициализированной комнаты, при ответе сервисом ChatService на соответствующий запрос.

В листинге 7 приведен proto-файл, применяемый для создания протокола взаимодействия сервера и клиентов.

Листинг 7. Proto-файл для определения контракта взаимодействия и структуры передаваемых между клиентом и сервером сообщений.

```
syntax = "proto3";
option java_multiple_files = true;
option java_package = "ru.mai";
package ru.mai;
service ChatService {
   rpc Connect (Login) returns (ConnectResponse);
   rpc Disconnect (Login) returns (Status);
   rpc CheckDisconnect (Login) returns (stream Login);
   rpc InitRoom (InitRoomRequest) returns (InitRoomResponse);
   rpc CheckForInitRoomRequest (Login) returns (stream
InitRoomResponse);
   rpc PassDiffieHellmanNumber (DiffieHellmanNumber) returns (Status);
   rpc AnyDiffieHellmanNumber (Login) returns (stream
DiffieHellmanNumber);
   rpc DeleteRoom (ChatRoomLogins) returns (CompanionAndStatus);
   rpc CheckForDeletedRoom (Login) returns (stream CompanionAndStatus);
}
message Status {
   EnumStatus enum_status = 1;
}
enum EnumStatus {
   ENUM_STATUS_OK = 0;
   ENUM_STATUS_ERROR = 1;
}
enum Algorithm {
   ALGORITHM_DES = 0;
   ALGORITHM_DEAL = 1;
   ALGORITHM_RIJNDAEL = 2;
   ALGORITHM_LOKI97 = 3;
   ALGORITHM_MARS = 4;
   ALGORITHM_RC6 = 5;
}
enum EncryptionMode {
   ENCRYPTION_MODE_ECB = 0;
   ENCRYPTION_MODE_CBC = 1;
   ENCRYPTION_MODE_PCBC = 2;
   ENCRYPTION_MODE_CFB = 3;
   ENCRYPTION_MODE_OFB = 4;
   ENCRYPTION_MODE_CTR = 5;
   ENCRYPTION_MODE_RANDOM_DELTA = 6;
}
enum PaddingMode {
   PADDING_MODE_ZEROES = 0;
   PADDING_MODE_ANSI_X_923 = 1;
   PADDING_MODE_PKCS7 = 2;
   PADDING_MODE_ISO10126 = 3;
}
message Login {
   string login = 1;
}
message ConnectResponse {
   string diffie_hellman_g = 1;
```

```
}
message CompanionAndStatus {
   string companion_login = 1;
   bool status = 2;
}
message InitRoomRequest {
   string own_login = 1;
   string companion_login = 2;
   Algorithm algorithm = 3;
   EncryptionMode encryption_mode = 4;
   PaddingMode padding_mode = 5;
}
message InitRoomResponse {
   string companion_login = 1;
   Algorithm algorithm = 2;
   EncryptionMode encryption_mode = 3;
   PaddingMode padding_mode = 4;
   string init_vector = 5;
   string diffie_hellman_p = 6;
}
message DiffieHellmanNumber {
   string own_login = 1;
   string companion_login = 2;
   string number = 3;
}
message ChatRoomLogins {
   string own_login = 1;
   string companion_login = 2;
}
message MessageToCompanion {
   string companion_login = 1;
   string uuid = 2;
   string sender = 3;
   string filename = 4;
   int32 partitions = 5;
   int32 curr_index = 6;
   bytes value = 7;
}
```

Модуль cryptography-starter.

Этот модуль предоставляет интерфейсы для шифрования и дешифрования массивов байт, потоков, файлов. cryptography-starter содержит контекст шифрования, реализации режимов шифрования, набивки алгоритмов шифрования, генераторы раундовых ключей, операции, используемые в алгоритмах шифрования.

Рассмотрим компоненты этого модуля.

Интерфейс EncryptionContext декларирует методы, необходимые для шифрования и дешифрования массива байт произвольной длины, входящего потока байт, файла.

Класс SymmetricEncryptionContextImpl реализует интерфейс EncryptionContext и репрезентирует контекст выполнения симметричного криптографического алгоритма, предоставляет объектный функционал по выполнению операций шифрования и дешифрования заданным ключом симметричного алгоритма с поддержкой одного из режимов шифрования: ECB, CBC, PCBC, CFB, OFB, CTR, Random Delta; а также с поддержкой одного из режимов набивки: Zeros, ANSI X.923, PKCS7, ISO 10126. Для режимов шифрования ECB, CTR, Random Delta реализована возможность шифрования в многопоточном режиме. Для режимов шифрования ECB, CTR, Random Delta, CBC, CFB реализовано распараллеливание вычислений при дешифровании. SymmetricEncryptionContextImpl отвечает за валидацию переданных параметров (режима шифрования, набивки, алгоритма, вектора инициализации), набивку входных данных для шифрования и удаление набивки при дешифровании, разбиение входного потока байт, передачу его в режим шифрования.

Интерфейс EncryptionMode декларирует методы для шифрования и дешифрования массива байт, длина которого кратна длине блока выбранного алгоритма.

Перечисление EncryptionModeEnum задает список идентификаторов режимов шифрования.

Абстрактный класс EncryptionModeAbstract наследует интерфейс EncryptionMode и содержит общие для алгоритмов параметры, конструкторы и методы.

Интерфейс EncryptionModeWithInitVector декларирует метод invokeAsNew, необходимый для повторного применения режимов шифрования, использующих вектор инициализации: CBC, PCBC, CFB, OFB, CTR, Random Delta.

Интерфейс EncryptionModeCounter расширяет интерфейс EncryptionModeWithInitVector и содержит методы, необходимые для реализации режимов CTR и Random Delta для шифрования и дешифрования в многопоточном режиме.

Интерфейс EncryptionModePreviousNeeded расширяет интерфейс EncryptionModeWithInitVector и содержит метод, необходимый для реализации режимов CBC и CFB для дешифрования в многопоточном режиме.

Класс ECB расширяет EncryptionModeAbstract и содержит реализацию режима электронной кодовой книги. Он полностью реализует методы шифрования и дешифрования из интерфейса EncryptionMode, принимающие

как параметр массив байт, по размеру кратный длине блока алгоритма шифрования. Схема шифрования режимом ECB приведена в рисунке 7.

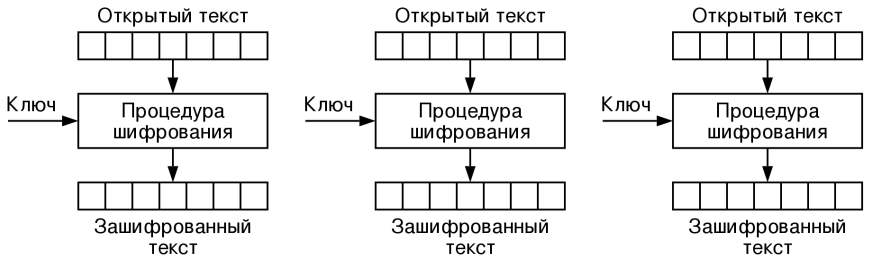

Рисунок 7. Шифрование в режиме ECB (режиме электронной кодовой книги).

Класс CBC расширяет EncryptionModeAbstract и наследует интерфейс EncryptionModePreviousNeeded. Он представляет собой реализацию режима сцепления блоков шифротекста. CBC реализует метод шифрования из интерфейса EncryptionMode и метод дешифрования из интерфейса EncryptionModePreviousNeeded. Последний метод может выполняться в многопоточном режиме. Схема шифрования режимом CBC приведена в рисунке 8.

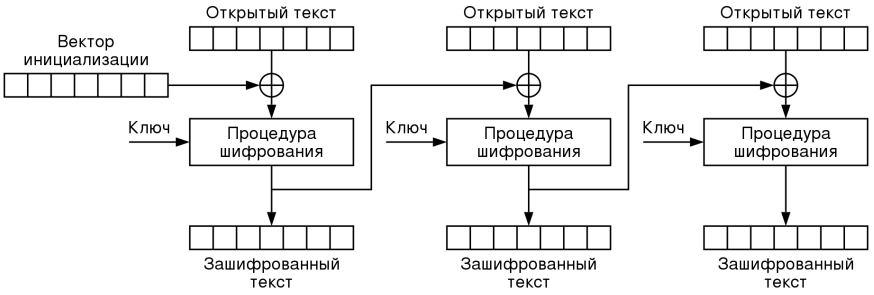

Рисунок 8. Шифрование в режиме CBC (режиме сцепления блоков шифротекста).

Класс PCBC расширяет EncryptionModeAbstract и наследует интерфейс EncryptionModeWithInitVector. Он представляет собой реализацию режима распространяющегося сцепления блоков шифротекста. PCBC полностью реализует методы шифрования и дешифрования из интерфейса EncryptionMode. Схема шифрования режимом PCBC приведена в рисунке 9.

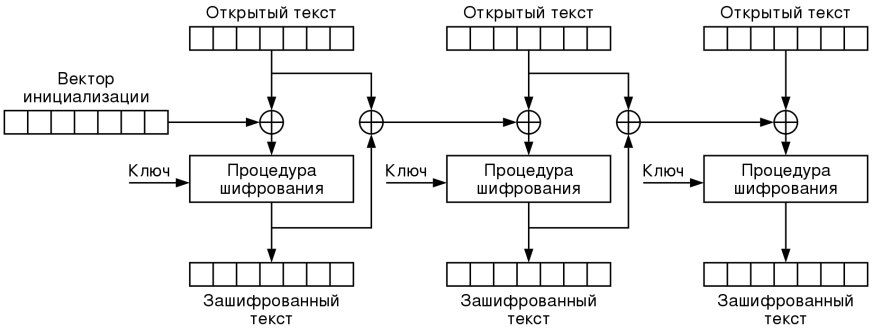

Рисунок 9. Шифрование в режиме PCBC (режиме распространяющегося сцепления блоков шифротекста).

Класс CFB расширяет EncryptionModeAbstract и наследует интерфейс EncryptionModePreviousNeeded. Он представляет собой реализацию режима обратной связи по шифротексту. CFB реализует метод шифрования из интерфейса EncryptionMode и метод дешифрования из интерфейса EncryptionModePreviousNeeded. Последний метод может выполняться в многопоточном режиме. Схема шифрования режимом CFB приведена в рисунке 10.

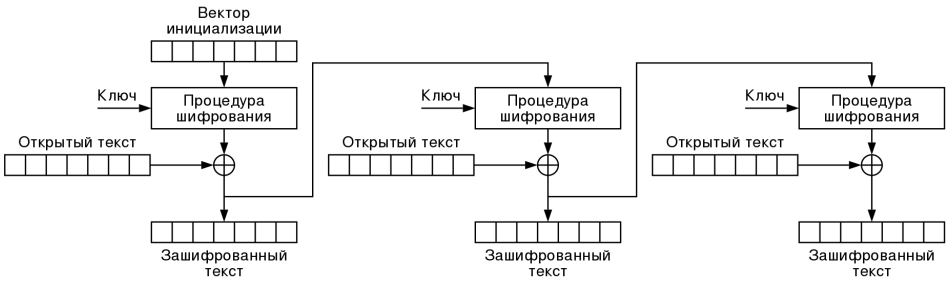

Рисунок 10. Шифрование в режиме CFB (режиме обратной связи по шифротексту)

Класс OFB расширяет EncryptionModeAbstract и наследует интерфейс EncryptionModeWithInitVector. Он представляет собой реализацию режима обратной связи по выходу. OFB полностью реализует методы шифрования и дешифрования из интерфейса EncryptionMode. Схема шифрования режимом OFB приведена в рисунке 11.

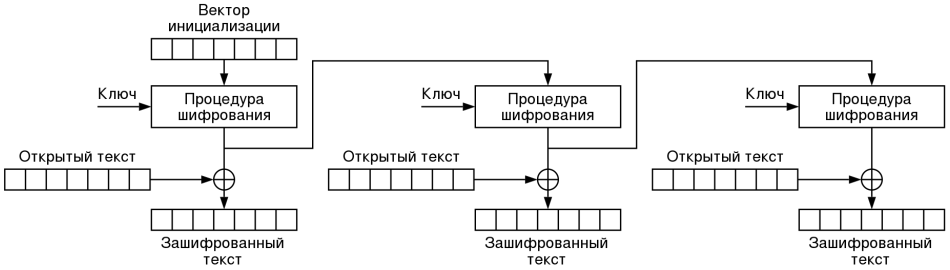

Рисунок 11. Шифрование в режиме OFB (режиме обратной связи по выходу)

Класс CTR расширяет EncryptionModeAbstract и наследует интерфейс EncryptionModeCounter. Он представляет собой реализацию режима счетчика. CTR полностью реализует методы шифрования и дешифрования из интерфейса EncryptionModeCounter для выполнения соответствующих операций в многопоточном режиме и не реализует методы интерфейса EncryptionMode. Схема шифрования режимом CTR приведена в рисунке 12.

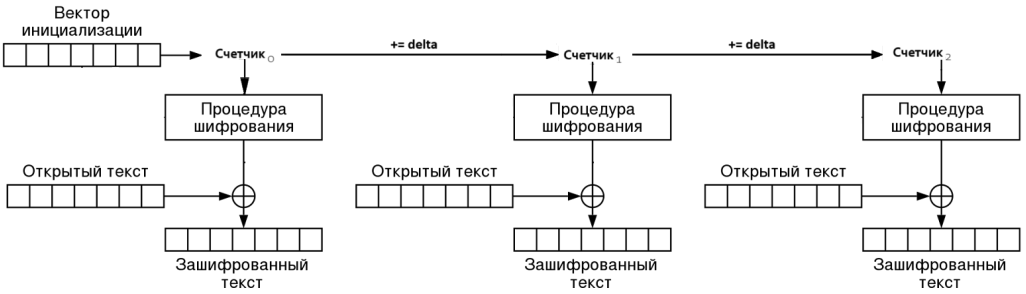

Рисунок 12. Шифрование в режиме CTR (режим счетчика)

Класс RandomDelta расширяет EncryptionModeAbstract и наследует интерфейс EncryptionModeCounter. Он представляет собой реализацию режима счетчика со случайной дельтой. RandomDelta полностью реализует методы шифрования и дешифрования из интерфейса EncryptionModeCounter для выполнения соответствующих операций в многопоточном режиме и не реализует методы интерфейса EncryptionMode. Схема шифрования режимом Random Delta аналогична схеме шифрования, приведенной в рисунке 12.

Интерфейс PaddingMode декларирует методы для набивки массива байт произвольной длины и удаления набивки.

Перечисление PaddingModeEnum задает список идентификаторов режимов набивки.

Абстрактный класс PaddingModeAbstract наследует интерфейс PaddingMode и содержит общие для режимов набивки поля и метод.

Класс Zeroes расширяет PaddingModeAbstract и представляет собой реализацию дополнения нулями – все байты, которые необходимо заполнить, заполняются нулями.

Класс PKCS7 расширяет PaddingModeAbstract и представляет собой реализацию дополнения в целых байтах. Значение каждого байта равно числу добавленных байтов, то есть добавляется N байт со значением N. Число добавленных байтов зависит от границы блока, до которого необходимо расширить сообщение.

Класс ISO 10126 расширяет PaddingModeAbstract и представляет собой реализацию дополнения случайными байтами, с тем, что последний байт должен содержать количество добавленных байтов.

Класс ANSI X.923 расширяет PaddingModeAbstract и представляет собой реализацию дополнения нулями с тем, что последний байт должен содержать количество добавленных байтов.

Интерфейс EncryptionAlgorithm декларирует методы для шифрования и дешифрования алгоритмом шифрования массива байт, кратного блоку шифрования, а также методы для получения размера блока и ключа алгоритма.

Класс LOKI97 выполняет методы интерфейса EncryptionAlgorithm и содержит реализацию одноименного алгоритма шифрования. Он использует

классы RoundKeyGenerationLOKI97 для генерации ключей, а FeistelCipherLOKI97, EncryptionConversionFeistelFunctionLOKI97 – для выполнения операций шифрования и дешифрования.

Класс RC6 выполняет методы интерфейса EncryptionAlgorithm и содержит реализацию одноименного алгоритма шифрования. Он использует классы FeistelCipherRC6 для генерации ключей, а FeistelCipherRC6 – для выполнения операций шифрования и дешифрования.

Класс MARS выполняет методы интерфейса EncryptionAlgorithm и содержит реализацию одноименного алгоритма шифрования. Он использует классы RoundKeyGenerationMARS для генерации ключей, а FeistelCipherMARS для выполнения операций шифрования и дешифрования.

Интерфейс RoundKeyGeneration декларирует метод для генерации раундовых ключей алгоритма шифрования.

Класс RoundKeyGenerationLOKI97 реализует интерфейс RoundKeyGeneration и представляет собой генератор раундовых ключей из заданного ключа для алгоритма LOKI97.

Класс RoundKeyGenerationRC6 реализует интерфейс RoundKeyGeneration и представляет собой генератор раундовых ключей из заданного ключа для алгоритма RC6.

Класс RoundKeyGenerationMARS реализует интерфейс RoundKeyGeneration и представляет собой генератор раундовых ключей из заданного ключа для алгоритма MARS.

Класс FeistelCipherLOKI97 реализует сбалансированную сеть Фейстеля, используемую в алгоритме шифрования LOKI97. FeistelCipherLOKI97 для шифрования и дешифрования применяет объект класса EncryptionConversionFeistelFunctionLOKI97, репрезентирующего функцию шифрования в сбалансированной сети Фейстеля для LOKI97.

Класс FeistelCipherRC6 реализует петлю сети Фейстеля, используемую в алгоритме шифрования RC6.

Класс FeistelCipherMARS реализует петлю сети Фейстеля, используемую в алгоритме шифрования MARS.

Клиентская часть. Модуль client.

Модуль client предоставляет серверную среду для веб-интерфейсов приложения, отвечает за обработку пользовательских взаимодействий с интерфейсами, посылку запросов к серверу и обработку ответов от сервера.

Для создания веб-интерфейсов в модуле применяется фреймворк Vaadin – он позволяет разрабатывать UI компоненты и управлять ими используя единственный язык программирования, Java, предлагает множество готовых компонентов и легкость их стилизации. Для написания запросов к серверу и обработки ответов используется gRPC. Механизм внедрения зависимостей в модуле реализован с помощью Spring Framework. Обмен зашифрованных сообщений осуществляется с применением брокера сообщений Apache Kafka.

Рассмотрим архитектуру модуля.

За создание начальной страницы отвечает класс LoginView, являющийся Vaadin Flow представлением. Он содержит методы для обработки входа пользователя в веб-приложение и перенаправления его на страницу чатов.

Класс ChatRoomsView – это Vaadin Flow представление, инициализирующее веб-интерфейс чата. Он отвечает за взаимодействие с пользователем, управление чатами, отправку и получение сообщений, а также отображение сообщений в удобной для пользователя форме, для чего использует объекты классов ChatClientService, ChatRepository, FilesToSendRepository и вложенные статические классы MessagesLayoutScrollerWrapper, ChatTab и ChatInfo. Кроме того, здесь создаются несколько исполняющих сервисов (объектов классов ExecutorService и ScheduledExecutorService) для обработки шифротекста, получения сообщений, передаваемых собеседниками, и уведомлений от gRPC сервера в фоновом режиме. Чат представлен отдельной вкладкой на панели вкладок с помощью классов ChatTab и ChatInfo, что позволяет легко переключаться между активными комнатами и отображать их для текущего пользователя. Класс MessagesLayoutScrollerWrapper ответственен за отображение переданных и полученных сообщений в надлежащем виде – текст как текст, картинки как картинки, остальные файлы – как компоненты UI, позволяющие сохранить файлы на локальное устройство при помощи стандартного диалога сохранения файла.

Класс ChatRepository – это репозиторий Spring, отвечающий за хранение информации о сообщениях, которые были отправлены и получены в чате с каждым конкретным собеседником.

Класс FilesToSendRepository – это репозиторий Spring, отвечающий за хранение данных для отправки файлов собеседнику.

Класс ChatClientService – это компонент Spring, ответственный за создание, отправку запросов к серверу и обработку ответов. Он является центральным звеном для взаимодействия с gRPC сервером в Vaadin-приложении. ChatClientService предоставляет методы для подключения и отключения клиента, управления чат-комнатами, отправки и получения сообщений, обработки сообщений, методы для проверки сервером запросов к текущему пользователю. Для этого применяются объекты классов ConnectionHandler, ChatRoomHandler, MessageHandler. В сервисах используется генерируемый класс ChatServiceGrpc.ChatServiceBlockingStub – блокирующий клиент, предназначенный для написания запросов и получения ответов от gRPC сервера.

Класс ClientConfig — это Spring-конфигурация, отвечающая за создание и настройку gRPC-соединения клиента с сервером чата. Класс предоставляет бин ChatServiceGrpc.ChatServiceBlockingStub и бин Channel для создания gRPC-канала для подключения блокирующего клиента к серверу. Другие Spring-компоненты, такие как ConnectionHandler, ChatRoomHandler, и MessageHandler, могут использовать ChatServiceGrpc.ChatServiceBlockingStub, созданный этим классом, для отправки запросов и получения ответов от сервера чата.

Класс ConnectionHandler – это Spring-компонент, сервис для работы с сетевым соединением. Он отвечает за установление и разрыв соединения с сервером чата и предоставляет методы для подключения, отключения, отслеживания отключений какого-либо собеседника.

Класс ChatRoomHandler – это Spring-компонент, сервис для создания, управления и удаления активных чат-комнат. Он предоставляет методы для: инициации создания, создания и удаления чатов, обмена ключами по протоколу Диффи-Хеллмана, отслеживания запросов на создание и удаление комнаты. ChatRoomHandler использует классы DiffieHellmanNumbersHandler для обработки ключей по протоколу Диффи-Хеллмана и генерации ключей для шифрования, InitMetadataRepository для хранения метаданных для создания чат-комнат, таких как алгоритмы шифрования, векторы инициализации и закрытые ключи, утилитный класс ContextBuilder для создания контекстов шифрования для конкретного собеседника с параметрами, полученными от сервера, и утилитный класс AlgorithmsConfigs, предоставляющий доступ к длинам блоков и ключей для каждого алгоритма шифрования, реализованного в курсовой работе.

Класс DiffieHellmanNumbersHandler – это утилитный класс для реализации протокола Диффи-Хеллмана: генерации закрытого ключа, получении открытого ключа и результирующего с использованием открытого ключа, полученного от собеседника.

Класс MessageHandler – это Spring-компонент, сервис для обработки сообщений. Он предоставляет методы для: отправки и получения сообщений и обработки полученных сообщений и файлов. Класс MessageHandler использует объект ContextsRepository для получения и хранения контекстов шифрования, необходимых для шифрования и дешифрования сообщений, FilesUnderDownloadRepository для отслеживания прогресса загрузки файлов, хранения информации о полученных частях файла и временных файлах для хранения частей файла, KafkaMessageHandler для отправки и получения сообщений через Apache Kafka.

Класс ContextsRepository — это Spring-репозиторий, реализующий хранилище для контекстов шифрования, используемых в чате.

Класс FileUnderDownloadRepository — это Spring-репозиторий, реализующий хранилище для отслеживания файлов, которые находятся в процессе загрузки. Он позволяет сохранять части загружаемых файлов во временные файлы, отслеживать количество полученных частей и удалять метаданные файла, когда загрузка завершена. Для репрезентации метаданных загружаемого файла используется класс FileUnderDownloadMetadata, в котором реализованы потокобезопасное получение имени временного файла и инкрементация счетчика полученных частей.

Класс KafkaMessageHandler — это Spring-компонент, который обеспечивает взаимодействие с Kafka для отправки и получения сообщений в чате. Он инициализирует KafkaProducer и KafkaConsumer с помощью KafkaManager, отвечает за отправку и получение сообщений, представляемых объектом класса MessageDto, через брокер сообщений.

Класс KafkaManager – это утилитарный класс, который обеспечивает управление Kafka-конфигурацией и создание Kafka-клиентов (производителя и потребителя) для обмена сообщениями в чате.

Класс MessageDto представляет объект передачи данных в Kafka и содержит идентификатор сообщения, логин отправителя, зашифрованный массив байт и, если передается часть файла, имя файла, количество разбиений и индекс передаваемой части.

Пример использования приложения.

Поскольку приложение развертывается локально на порту 8080, вход в него возможен по адресу “localhost:8080”. По этому адресу доступен вход в чат – страница предлагает ввести логин и перейти к основной странице приложения.

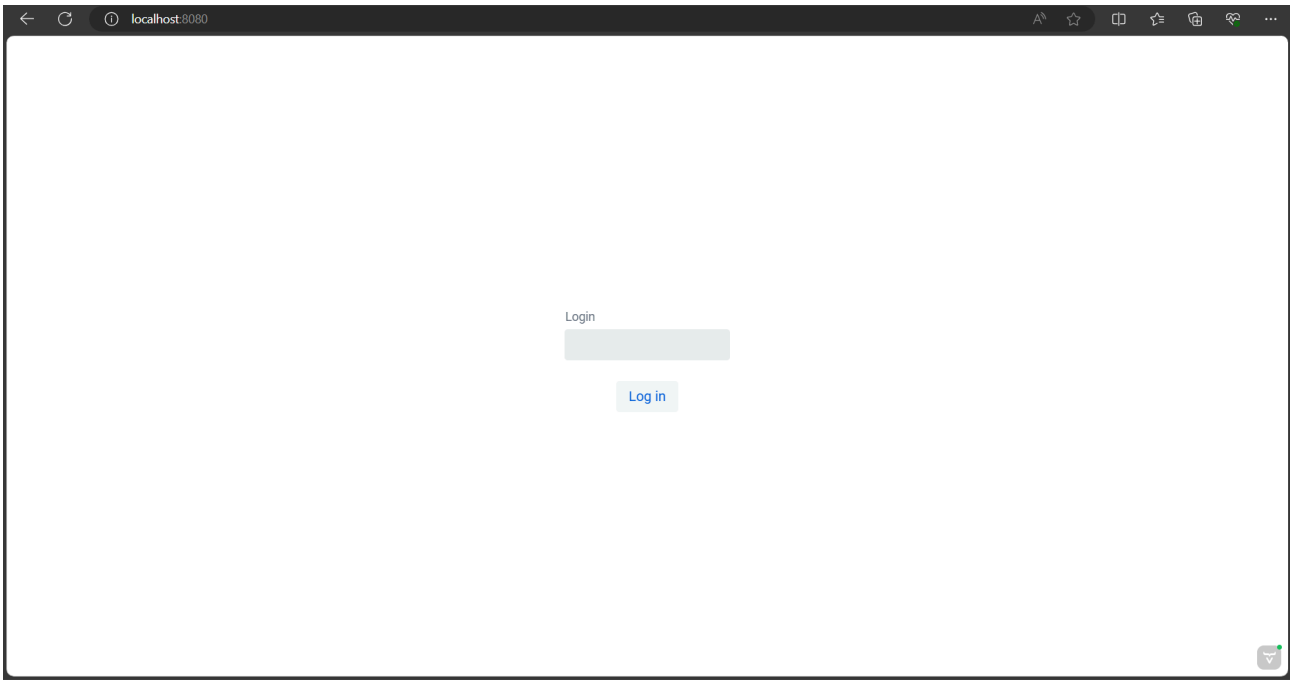

Рисунок 13. Вход в приложение.

Основная страница приложения содержит список открытых чатов, реализованный посредством Vaadin-компонента вкладок, и окна чата, содержащего компонент для отображения сообщений по выбранному собеседнику, кнопки для загрузки файлов, поле для ввода текстовых сообщений и кнопку для отправки файлов и текстовых сообщений.

Создание новой чат комнаты возможно по нажатию на кнопку “Добавить новую чат комнату”, находящуюся под списком активных чатов. Появится диалоговое окно, где пользователю нужно написать логин своего собеседника, выбрать один из режимов шифрования (ECB, CBC, PCBC, CFB, OFB, CTR, Random Delta) и набивки (Zeros, ANSI X.923, PKCS7, ISO 10126), алгоритм шифрования (LOKI97, RC6). По нажатии на кнопку “Подтвердить” происходит инициация выполнения протокола на создание секретного чата. Стороны обмениваются ключами по протоколу Диффи-Хеллмана, после чего инициализируется контекст шифрования этой чат комнаты и создается новая вкладка с логином собеседника. Переключение между чат-комнатами возможно по нажатию на другую вкладку.

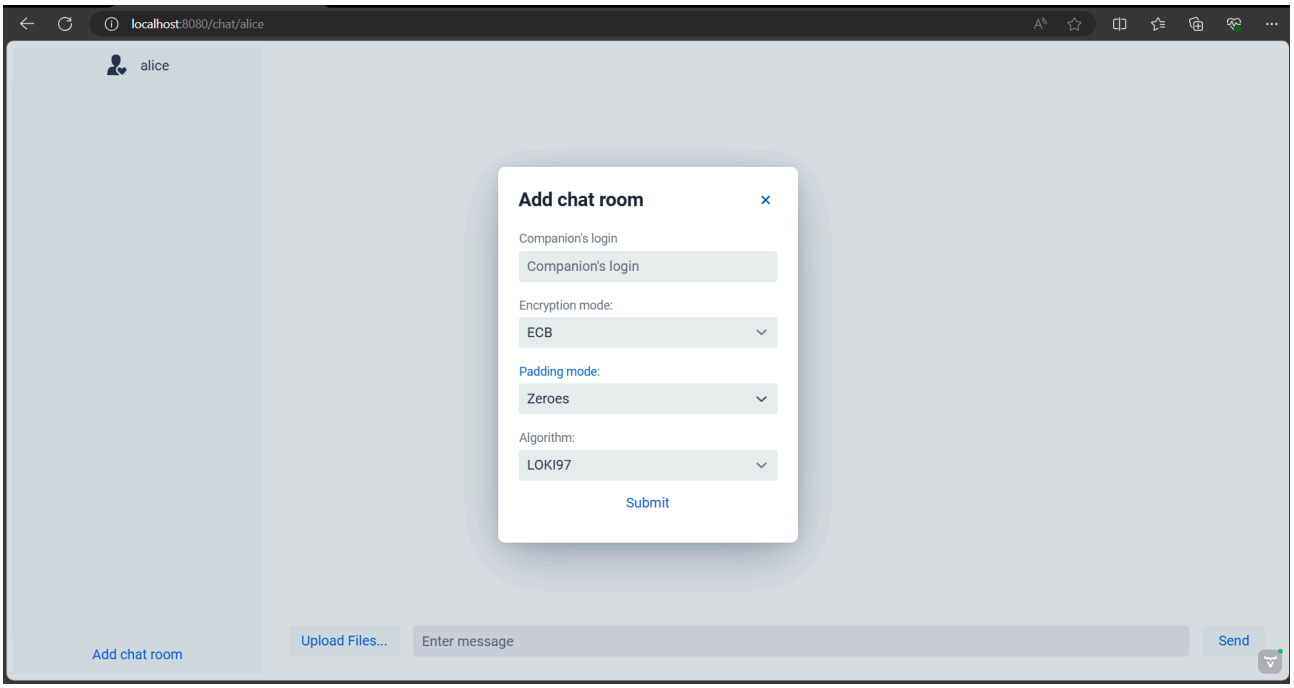

Рисунок 14. Инициализация выполнения протокола на создание секретного чата с указанием используемого алгоритма шифрования, режима шифрования и набивки.

Можно добавить нескольких собеседников с уникальным контекстом шифрования и историей сообщений. Чат-комнаты с новыми собеседниками будут отображаться слева в панели вкладок, рассортированные по времени инициации чат-комнаты. У каждой такой вкладки есть логин собеседника, чат-комнату с которым репрезентирует эта вкладка, и кнопка, по нажатию на которую инициализируется закрытие чата с этим собеседником.

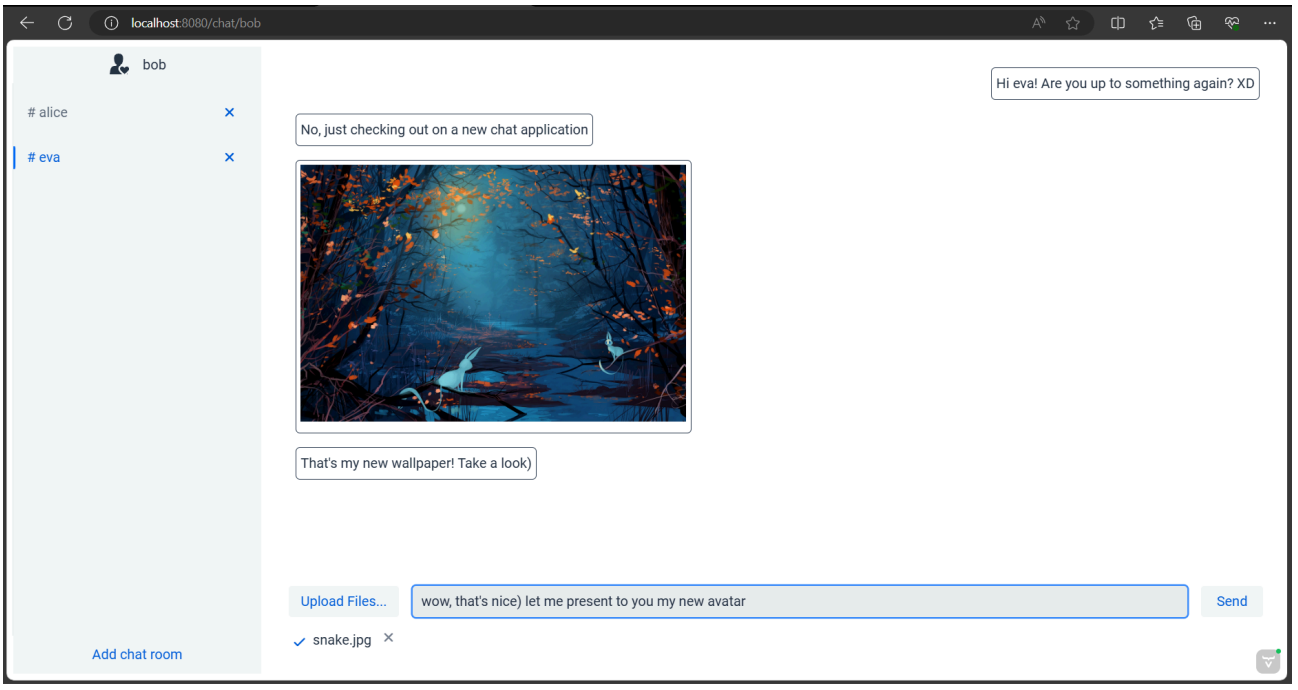

Рисунок 15. Отображение списка активных чатов и отправки сообщений.

Приложение позволяет отправлять данные для шифрования – текстовые сообщения, картинки, файлы. Можно инициировать шифрование и отправку сразу нескольких файлов.

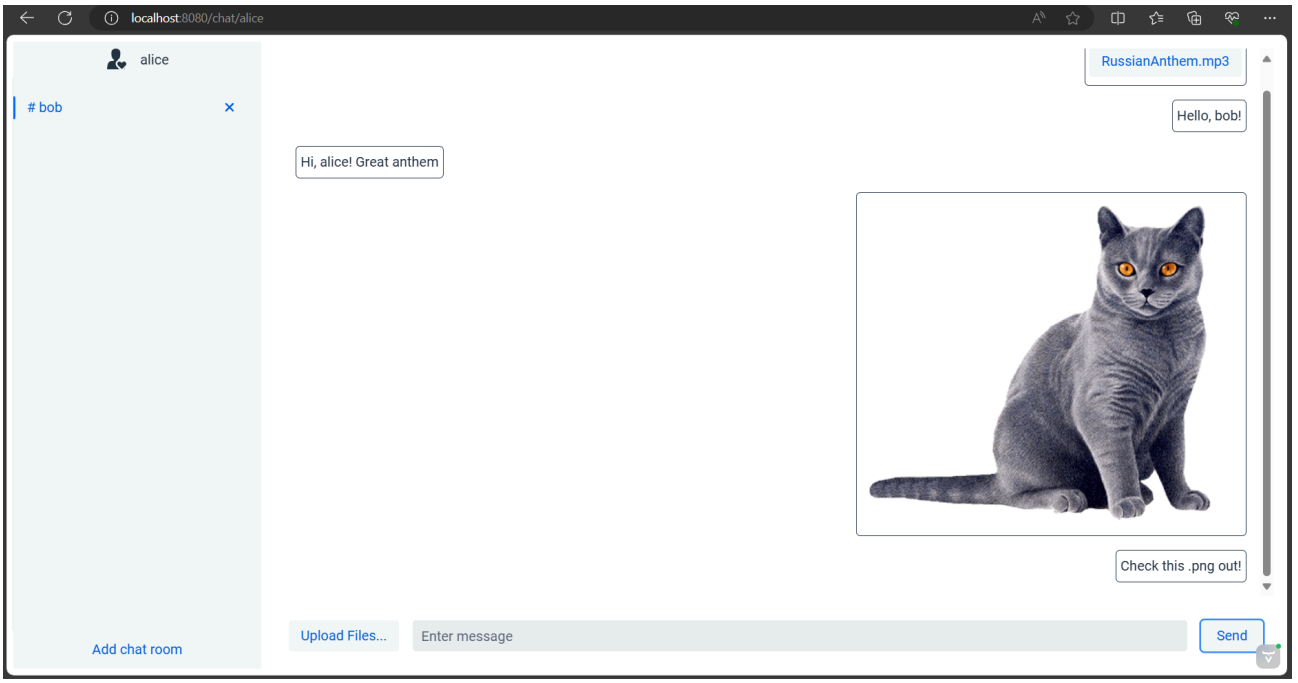

Рисунок 16. Отображение UI компонентов чата.

Закрытие страницы инициирует отключение клиента от сервера и закрытие всех активных вкладок.

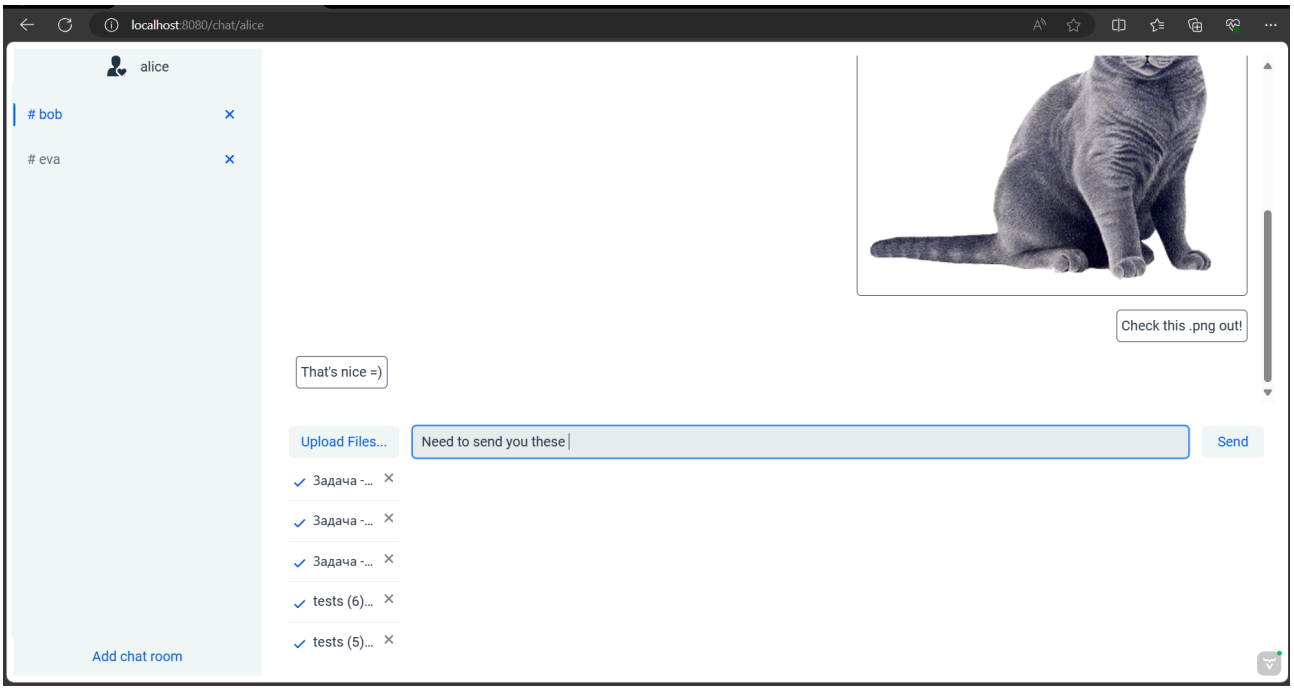

Рисунок 17. Отправка нескольких файлов.

Заключение.

В результате выполнения курсовой работы будет разработана система защищенного обмена сообщениями, передаваемых между двумя клиентами. Система была реализована с использованием симметричного шифрования, протокола Диффи-Хеллмана, а также брокера сообщений Apache Kafka для асинхронной передачи сообщений между клиентами.

Также было создано консольное серверное приложение, взаимодействующее с клиентами через gRPC и ответственное за обеспечение управления секретными чатами, распределение сеансового ключа, а также обработку и пересылку сообщений.

Разработанное клиентское web-приложение предоставляет удобный графический интерфейс для работы с системой защищенного обмена сообщениями. Пользователь может создавать секретные чаты, выбирать характеристики инициализированного чата, отправлять и получать сообщения.

В результате работы был получен практический опыт в разработке систем безопасного обмена сообщениями, применены алгоритмы шифрования LOKI97, RC6, фреймворки Vaadin, Spring Boot, протокол gRPC, использованы платформа Apache Kafka и технологии контейнеризации.

Список источников.

1. Лори Браун, Йозеф Пепшик Introducing the new LOKI97 Block Cipher. - 1998. - 22 с.
2. Рональд Райвест, М. Робшоу, Р. Сидни, The RC6 Block Cipher. - 1.1 изд. - 1998. - 21 с.
3. Филип М. Клуцник, Эрнест Эмблер, Джеймс Х. Берроуз DES MODES OF OPERATION. - 1981. - 17 с.
4. Моррис Дворкин Recommendation for Block Cipher Modes of Operation. - 2001. - 66 с.
5. Криптографический форум: Что такое режимы счетчика и PCBC? // iks-jena URL: http://altlasten.lutz.donnerhacke.de/mitarb/lutz/security/cryptfaq/q84.html (дата обращения: 05.06.2024).
6. ANSI X9.23 в режиме сцепления блоков шифротекста // ibm.com URL: https://www.ibm.com/docs/en/linux-on-systems?topic=processes-ansi-x923-cipher-block-chaining (дата обращения: 05.06.2024).
7. Дополнение (криптография) // wikipedia.org URL: https://en.wikipedia.org/wiki/Padding_(cryptography) (дата обращения: 05.06.2024).
8. Фреймворк Vaadin // vaadin.com URL: https://vaadin.com (дата обращения: 05.06.2024).
9. Фреймворк grpc // grpc.io URL: https://grpc.io (дата обращения: 05.06.2024).

ТУТ БУДЕТ СПИСОК ИСТОЧНИКОВ

^^

(github репу прикладывать?)