Санкт-Петербургский политехнический университет Петра Великого
Институт компьютерных наук и технологий
Кафедра компьютерных систем и программных технологий

Технологии компьютерных сетей

Отчет по лабораторной работе

Программирование серверного и клиентского приложения прикладного протокола

Работу выполнила:
Кузовкина Е.О.
Группа:
3530901/60202
Преподаватель:
Зозуля А.В.

Санкт-Петербург
2020

1. Задание

Разработать приложение–клиент и приложение–сервер банковского сервиса. Сервис позволяет пользователю открывать депозитные вклады, зарабатывать на процентах по вкладу, закрывать вклады.

2. Основные возможности

Серверное приложение должно реализовывать следующие функции:

1. Прослушивание определенного порта
2. Обработка запросов на подключение по этому порту от пользователей сервиса
3. Поддержка одновременной работы нескольких пользователей сервиса через механизм нитей
4. Прием запросов на:
    a. открытие вклада с фиксированной прибыльностью 10% для владельца и 5% для банка;
    b. пополнение вклада;
    c. начисление процентов по вкладу для владельца и банка (отдельной командой, которую можно вызывать неоднократно);
    d. состояние вкладов (для банка - всех);
    e. состояние баланса банка;
    f. закрытие вклада с выплатой процентов.
5. Обработка запроса на отключение клиента
6. Принудительное отключение клиента
7. Выдача списка подключенных клиентов в формате:
   № п/п IP-адрес порт

Клиентское приложение должно реализовывать следующие функции:

- Установление соединения с сервером
- Передача запросов серверу
- Получение ответов на запросы от сервера
- Разрыв соединения
- Обработка ситуации отключения клиента сервером

3. Настройки приложений

Разработанное клиентское приложение должно предоставлять пользователю настройку IP–адреса или доменного имени сервера и номера порта, используемого сервером.

4. Методика тестирования

Для тестирования приложений запускается сервер и несколько клиентов. В процессе тестирования проверяются основные возможности приложений по передаче и приему информации, параллельная обработка запросов.

5. Прикладной протокол

5.1. Реализация

Сервер – программа, написанная на языке C, поддерживаемая работу с множеством клиентов. В программе задействованы следующие виды потоков:

- Основной
  - создаёт сокет, принимающий новые соединения
  - создаёт поток, принимающий соединения
  - создаёт поток, слушающий пользовательский ввод
- Принимающий соединения
  - принимает соединения от новых клиентов
  - создаёт отдельный клиентские поток для каждого нового соединения
  - добавляет подключенного клиента в список подключённых клиентов
- Слушающий консоль
  - слушает пользовательские команды
  - выполняет действия в соответствии с введённой командой
- Клиентский
  - принимает пакеты от клиента
  - обрабатывает принятые пакеты в соответствии с их типом

В программе присутствуют следующие типы структур:

- связный список для хранения информации о клиентах
- связный список для хранения депозитов

Клиент представляет из себя программу на языке C.

Она предоставляет пользователю выбор из нескольких действий (открыть вклад, посмотреть открытые вклады, пополнить вклад, закрыть вклад, посмотреть счет банка, либо завершить работу программы).

В соответствии с выбором пользователя формируется и отправляется на сервер пакет определённого формата, затем происходит приём и обработка ответа сервера.

6. Описание архитектур

6.1. Структура TCP-клиента

Клиент протокола TCP создаёт экземпляр сокета, необходимый для взаимодействия с сервером, организует соединение, осуществляет обмен данными, в соответствии с протоколом прикладного уровня.

Типичная структура TCP-клиента представлена на рис.6.1.

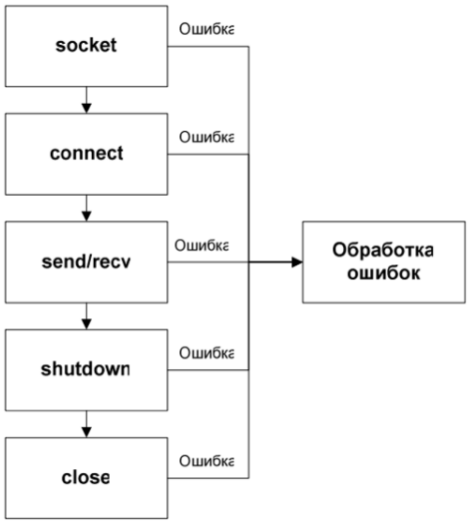

Рисунок 6.1. Типичная структура TCP-клиента

Если инициатором разрыва соединения является клиентское приложение, то далее следует вызвать функцию shutdown и после этого закрыть сокет.

Каждый вызов функций библиотеки сокетов должен сопровождаться проверкой на наличие ошибочной ситуации и обработкой этой ситуации.

6.2. Структура TCP-сервера

Организация TCP-сервера отличается от TCP-клиента в первую очередь созданием слушающего сокета (см. рис. 6.2 а). Такой сокет находится в состоянии listen и предназначен только для приёма входящих соединений. В случае прихода запроса на соединение создаётся дополнительный сокет, который и занимается обменом данными с клиентом.

Типичная структура TCP-сервера и взаимосвязь сокетов изображена на рис. 6.2 а) и б).

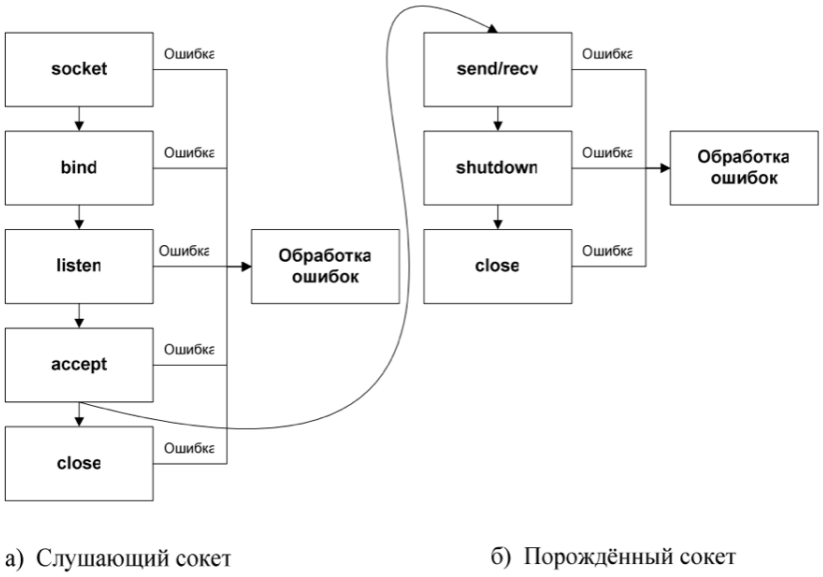

Рисунок 6.2. Типичная структура TCP-сервера

6.3. Структура UDP-клиента

Структура UDP-клиента ещё более простая, чем у TCP-клиента, так как нет необходимости создавать и разрывать соединение. Варианты организации UDP-клиента изображены на рис. 6.3.

Наличие двух вариантов организации связано с возможностью в UDPприложениях использовать вызов connect, устанавливающий значения по умолчанию для IP-адреса и порта сервера.

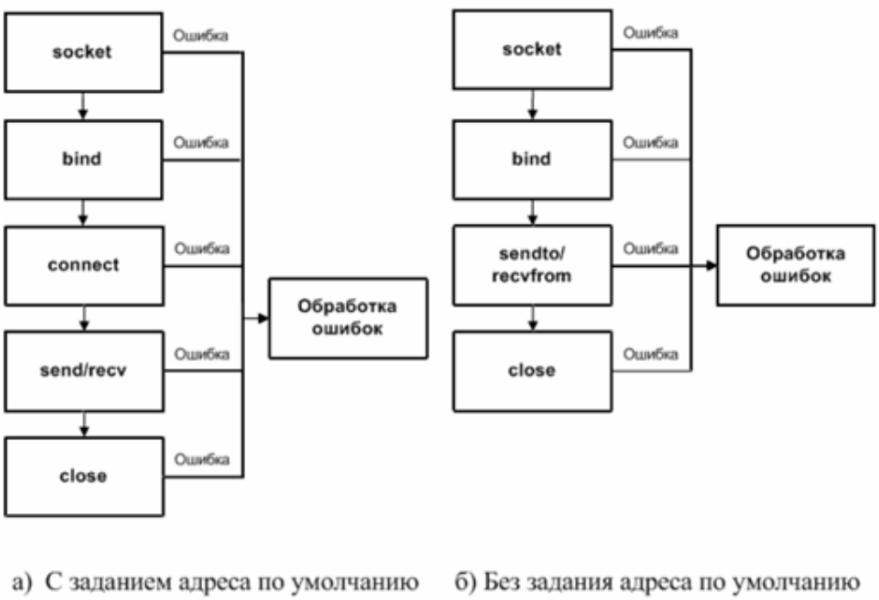

Рисунок 6.3. Типичная структура UDP-клиента

6.4. Структура UDP-сервера

Ввиду того, что в протоколе UDP не устанавливается логический канал связи между клиентом и сервером, то для обмена данными между несколькими клиентами и сервером нет необходимости использовать со стороны сервера несколько сокетов. Для определения источника полученной дейтаграммы серверный сокет может использовать поля структуры from вызова recvfrom. Типичный способ организации UDP-сервера приведён на рис. 6.4.

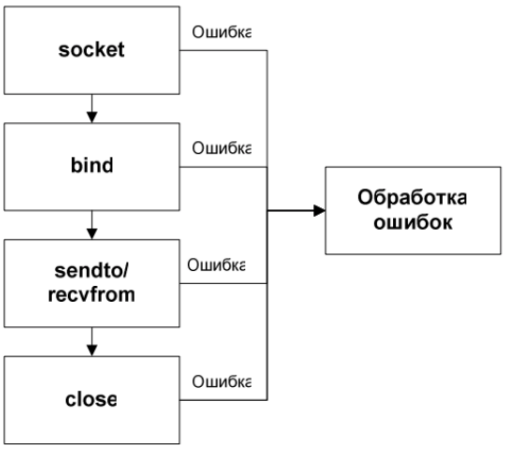

Рисунок 6.4. Типичная структура UDP-сервера

7. Особенности реализации сетевых и многопоточных приложений

Для чтения из сокета заданного количества байт использовался метод readn():

```
1  int readn(int sockfd, void *dst, size_t len){
2      int total_number_read = 0;
3      int local_number_read;
4
5      while (len > 0) {
6          local_number_read = read(sockfd, (char*) dst + total_number_read, len);
7
8          if (local_number_read == 0) {
9              return total_number_read;
10         }
11
12         if (local_number_read < 0) {
13             return local_number_read;
14         }
15
16         total_number_read += local_number_read;
17         len -= local_number_read;
18     }
19     return total_number_read;
20 }
```

Для контроля и завершения всех потоков использовался join. Ожидание завершения всех потоков:

```
1  pthread_join(accepting_thread, NULL);
2  pthread_join(console_listening_thread, NULL);
```

Для работы с разделяемыми ресурсами использовались мьютексы. Инициализация мьютексов:

```
1 init_list_of_clients_mutex();
2 init_list_of_deposits_mutex();
```

Код требующий использования разделяемых ресурсов:

```
1 pthread_mutex_lock(&list_of_clients_mutex);
2 pthread_mutex_unlock(&list_of_clients_mutex);
3 pthread_mutex_lock(&list_of_deposits_mutex);
4 pthread_mutex_unlock(&list_of_deposits_mutex);
```

В TCP сервере использовалась структура для создания нового клиента со следующими полями:

```
1  User_info* make_new_client(int sockfd, int port, char* address, pthread_t
       ↪ client_thread) {
2      User_info* new_client = (User_info*) malloc(sizeof(User_info));
3
4      new_client -> port = port;
5      new_client -> address = address;
6      new_client -> sockfd = sockfd;
7      new_client -> next = NULL;
8      new_client -> client_thread = client_thread;
9
10     return new_client;
11 }
```

В UDP сервере использовалась похожая структура для создания нового клиента, но с дополнительными полями:

```
1  new_client -> required_index = 1;
2  new_client -> last_answer = NULL;
3  new_client -> packet_size = 0;
```

При использовании протокола UDP была необходимость в контроле: порядка посылок, дублирования посылок и потери.

Для решения этих проблем в каждый пакет вкладывался номер этого пакета. Для определения, куда отправлять ответ (так как нет выделенного сокета) требовалось хранить историю по каждому соединению. Каждый клиент заносился в структуру данных.

8. Сервис депозитов

8.1. Возможности сервера

- Вывод списка клиентов
- Отключение клиента

- Начисление процентов
- Вывод счета банка
- Вывод всех вкладов
- Завершение работы

8.2. Типы пакетов

```
1 #define ERROR_PACKET 1
2 #define ACKNOWLEDGMENT_PACKET 2
3 #define LIST_OF_DEPOSITS_PACKET 3
4 #define OPEN_DEPOSIT_PACKET 4
5 #define REFILL_DEPOSIT_PACKET 5
6 #define CLOSE_DEPOSIT_PACKET 6
7 #define GET_LIST_OF_DEPOSITS_PACKET 7
8 #define GET_BANK_AMOUNT 8
9 #define SHOW_BANK_AMOUNT 9
10 #define PLEASE_ADD_PERCENTS 10
```

8.2.1. ERROR PACKET

Посылается сервером, когда не удалось открыть вклад/пополнить вклад/закрыть вклад.

Таблица 8.1

Формат пакета

length	type	text
4	2	4

8.2.2. ACKNOWLEDGMENT PACKET

Посылается сервером в ответ на запрос открытия вклада/пополнения вклада/начиления процентов по вкладам/закрытия вклада.

Таблица 8.2

Формат пакета

length	type	ack_type	deposit_id	initial_amount
4	2	2	4	4

8.2.3. LIST OF DEPOSITS PACKET

Посылается сервером в ответ на запрос списка депозитов.

Таблица 8.3

Формат пакета

length	type	init_amount	curr_amount	...	dep_id	init_amount	curr_amount	...
4	2	4	4	...	4	4	4	...

8.2.4. OPEN DEPOSIT PACKET

Посылается клиентом для открытия депозита.

Таблица 8.4

Формат пакета

length	type	initial_amount
4	2	4

8.2.5. REFILL DEPOSIT PACKET

Посылается клиентом для пополнения депозита.

Таблица 8.5

Формат пакета

length	type	deposit_id	amount
4	2	4	4

8.2.6. CLOSE DEPOSIT PACKET

Посылается клиентом для закрытия депозита.

Таблица 8.6

Формат пакета

length	type	deposit_id
4	2	4

8.2.7. GET LIST OF DEPOSITS PACKET

Посылается клиентом для получения списка своих депозитов.

Таблица 8.7

Формат пакета

length	type
4	2

8.2.8. GET BANK AMOUNT

Посылается клиентом для запроса счета банка.

Таблица 8.8

Формат пакета

length	type
4	2

8.2.9. SHOW BANK AMOUNT

Посылается сервером в ответ на запрос счета банка.

Таблица 8.9

Формат пакета

length	type	bank_amount
4	2	4

8.2.10. PLEASE ADD PERCENTS

Посылается клиентом для начисления процентов по вкладам.

Таблица 8.10

Формат пакета

length	type
4	2

8.3. Изменения пакетов в UDP

При использовании протокола UDP произошли небольшие изменения в формате пакетов:

8.3.1. ERROR PACKET

Посылается сервером, когда не удалось открыть вклад/пополнить вклад/закрыть вклад.

Таблица 8.11

Формат пакета

type	index	text
2	4	string

8.3.2. ACKNOWLEDGMENT PACKET

Посылается сервером в ответ на запрос открытия вклада/пополнения вклада/начиления процентов по вкладам/закрытия вклада.

Таблица 8.12

Формат пакета

type	index	ack_type	deposit_id (block_number)
2	4	2	4

8.3.3. LIST OF DEPOSITS PACKET

Посылается сервером в ответ на запрос списка депозитов.

Таблица 8.13

Формат пакета

type	index	block_number	deposit_id	initial_amount	current_amount
2	4	4	4	4	4

8.3.4. OPEN DEPOSIT PACKET

Посылается клиентом для открытия депозита.

Таблица 8.14

Формат пакета

type	index	initial_amount
4	2	4

8.3.5. REFILL DEPOSIT PACKET

Посылается клиентом для пополнения депозита.

Таблица 8.15

Формат пакета

type	index	deposit_id	amount
2	4	4	4

8.3.6. CLOSE DEPOSIT PACKET

Посылается клиентом для закрытия депозита.

Таблица 8.16

Формат пакета

type	index	deposit_id
2	4	4

8.3.7. GET LIST OF DEPOSITS PACKET

Посылается клиентом для получения списка своих депозитов.

Таблица 8.17

Формат пакета

type	index
2	4

8.3.8. GET BANK AMOUNT

Посылается клиентом для запроса счета банка.

Таблица 8.18

Формат пакета

type	index
2	4

8.3.9. SHOW BANK AMOUNT

Посылается сервером в ответ на запрос счета банка.

Таблица 8.19

Формат пакета

type	index	bank_amount
2	4	4

8.3.10. PLEASE ADD PERCENTS

Посылается клиентом для начисления процентов по вкладам.

Таблица 8.20

Формат пакета

type	index
2	4

9. Результаты тестирования приложения

9.1. TCP

Было проведено полноценное тестирование и отладка приложения на нескольких клиентах. Пример плана тестирования всего функционала приведен ниже.

После каждой операции на клиенте и сервере появляется меню действий, из которого клиент или сервер выбирает нужный пункт (рис. 9.1. и 9.2.).

```
1) Вывести список клиентов
2) Отключить клиента
3) Начислить проценты
4) Показать счёт банка
5) Показать все вклады
6) Завершить работу
```

Рисунок 9.1. Меню сервера

```
1) Открыть вклад
2) Посмотреть вклады
3) Пополнить вклад
4) Закрыть вклад
5) Завершить работу
6) Показать счет банка
7) Начислить проценты по вкладам
1
Enter amount of deposit: 70
You opened deposit with id 1 and amount 70.
```

Рисунок 9.2. Меню клиента и клиент 1 открыл первый вклад размером 70

```
1
Enter amount of deposit: 60
You opened deposit with id 2 and amount 60.
```

Рисунок 9.3. Клиент 2 открыл первый вклад размером 60

```
1
Enter amount of deposit: 50
You opened deposit with id 3 and amount 50.
```

Рисунок 9.4. Клиент 1 открыл второй вклад размером 50

```
2
-----------------
id of deposit: 1
initial amount of deposit: 70
current amount of deposit: 70.000000
-----------------
id of deposit: 3
initial amount of deposit: 50
current amount of deposit: 50.000000
-----------------
```

Рисунок 9.5. Просмотр вкладов клиентом 1

```
7
Percents to all your deposits were added
```

Рисунок 9.6. Клиент 1 попросил начислить ему по всем вкладам проценты

```
5
client deposit_id initial_amount current_amount
     4          1             70      77.000000
     5          2             60      60.000000
     4          3             50      55.000000
```

Рисунок 9.7. Сервер смотрит текущее состояние вкладов после начисления процентов одному клиенту

```
6
Bank amount: 6.000000
```

Рисунок 9.8. Просмотр клиентом счета банка

```
3
Enter id of deposit: 3
Enter refilling amount: 100
You refilled deposit with id 3 and amount 100.
```

Рисунок 9.9. Клиент 1 пополнил второй вклад на 100

```
2
------------------
id of deposit: 1
initial amount of deposit: 70
current amount of deposit: 77.000000
------------------
id of deposit: 3
initial amount of deposit: 50
current amount of deposit: 155.000000
------------------
```

Рисунок 9.10. Клиент 1 посмотрел состояние своих вкладов

```
4
Enter id of deposit: 1
You deleted deposit with id 1 and amount 77.
```

Рисунок 9.11. Клиент 1 закрыл свой первый вклад

```
5
client deposit_id initial_amount current_amount
     5          2             60      60.000000
     4          3             50     155.000000
```

Рисунок 9.12. Сервер посмотрел состояние вкладов

```
5

Process finished with exit code 0
```

Рисунок 9.13. Клиент 1 отключился

```
1
index   socket     port          address
    1        5     5009        127.0.0.1
    3        4     5009        127.0.0.1
```

Рисунок 9.14. Сервер смотрит подключения

```
2
Enter client socket number: 5
CLIENT 5 client disconnected
```

Рисунок 9.15. Отключение 2 клиента

```
1
index   socket     port          address
    1        4     5009        127.0.0.1
```

Рисунок 9.16. Сервер просматривает вновь список клиентов

```
6
CLIENT 4 client disconnected

Process finished with exit code 0
```

Рисунок 9.17. Сервер отключается и в процессе отключения отключает подключенных клиентов

```
1
Enter amount of deposit: 30

Process finished with exit code 13
```

Рисунок 9.18. Клиент пытается взаимодействовать с сервером после его отключения

9.2. UDP

Функционал приложения с использованием UDP не поменялся, поэтому предыдущий план тестирования на TCP и UDP совпал. Для UDP приложения было проведено дополнительное тестирование в виду особенностей протокола.

Сервер и клиент при передаче показывают индекс пакетов для удобства отладки приложения. На рисунке видно, что пакеты отправляются по очереди, ошибки при передаче отсутствуют.

```
1) Начислить проценты
2) Показать счёт банка
3) Показать все вклады
4) Завершить работу
5) Посмотреть список клиентов
6) Удалить вклады клиента
Отправлен пакет с индексом: 1
Отправлен пакет с индексом: 2
Отправлен пакет с индексом: 3
Отправлен пакет с индексом: 3
Отправлен пакет с индексом: 4
Отправлен пакет с индексом: 5
Отправлен пакет с индексом: 6
Отправлен пакет с индексом: 7
Отправлен пакет с индексом: 8
Отправлен пакет с индексом: 9
```

Рисунок 9.19. Нормальная работа сервера

```
3
   address   port deposit_id initial_amount current_amount
 127.0.0.1  52705          1             50      55.000000
 127.0.0.1  52705          2             30      33.000000
```

Рисунок 9.20. Вывод списка вкладов на сервере

```
6
Введите порт клиента:
52705
Введите адрес клиента:
127.0.0.1
Вклады клиента удалены
```

Рисунок 9.21. Отключение клиента на сервере

```
3
No deposits
```

Рисунок 9.22. Проверка состояния вкладов на клиенте после его удаления с сервера

Протестируем приложение при перемешивании пакетов - 1, 2, 4, 3 (индексы пакетов).

```
1
Enter amount of deposit: 50
index: 1
Вклад открыт, id: 1
```

Рисунок 9.23. Пакет с индексом 1

```
2
index: 2
-----------------
id of deposit: 1
accrued amount: 50
amount: 50.000000
```

Рисунок 9.24. Пакет с индексом 2

```
3
Enter id of deposit: 1
Enter refilling amount: 60
index: 4
-----------------
id of deposit: 1
accrued amount: 50
amount: 50.000000
```

Рисунок 9.25. Пакет с индексом 4, поэтому сервер отправляет предыдущий ack

```
1
Enter amount of deposit: 40
index: 3
Вклад открыт, id: 2
```

Рисунок 9.26. Пакет с индексом 3, как и ожидалось, поэтому уже новый ack

```
Отправлен пакет с индексом: 1
Отправлен пакет с индексом: 2
Отправлен пакет с индексом: 2
Отправлен пакет с индексом: 3
```

Рисунок 9.27. Отправка ack'ов сервером

Протестируем приложение при потере пакетов - 1, 2, 4 (индексы пакетов).

```
1
Enter amount of deposit: 80
index: 1
Вклад открыт, id: 1
```

Рисунок 9.28. Пакет с индексом 1

```
1
Enter amount of deposit: 20
index: 2
Вклад открыт, id: 2
```

Рисунок 9.29. Пакет с индексом 2

```
1
Enter amount of deposit: 40
index: 4
Вклад открыт, id: 2
```

Рисунок 9.30. Пакет с индексом 4, ожидался пакет с индексом 3

```
Отправлен пакет с индексом: 1
Отправлен пакет с индексом: 2
Отправлен пакет с индексом: 2
3
          address   port deposit_id initial_amount current_amount
        127.0.0.1  38861          1             80      80.000000
        127.0.0.1  38861          2             20      20.000000
```

Рисунок 9.31. Третий вклад не был создан, так как пакет имел индекс 4

Протестируем приложение при дублировании пакетов - 1, 2, 2 (индексы пакетов).

```
1
Enter amount of deposit: 30
index: 1
Вклад открыт, id: 1
```

Рисунок 9.32. Пакет с индексом 1

```
1
Enter amount of deposit: 40
index: 2
Вклад открыт, id: 2
```

Рисунок 9.33. Пакет с индексом 2

```
1
Enter amount of deposit: 50
index: 2
Вклад открыт, id: 2
```

Рисунок 9.34. Снова пакет с индексом 2, ожидался пакет с индексом 3

```
Отправлен пакет с индексом: 1
Отправлен пакет с индексом: 2
Отправлен пакет с индексом: 2
3
         address   port deposit_id initial_amount current_amount
       127.0.0.1  65235          1             30      30.000000
       127.0.0.1  65235          2             40      40.000000
```

Рисунок 9.35. Третий вклад не был создан, так как пакет имел индекс 2

10. Выводы

Был получен опыт разработки клиент-серверного приложения, написания прикладного протокола, работа с транспортными протоколами TCP/UDP. Разработан сервер-банк, в котором был реализован весь функционал согласно исходному заданию.

Для перехода с TCP на UDP необходимо было бы внести следующие изменения в прикладной протокол:

- Контроль очерёдности, потери и дублирования данных
- Необходимость добавления номера пакета в сам пакет

При реализации указанных изменений скорость передачи данных сильно возрастает, что является основным достоинством UDP.

Программирование TCP проще, однако скорость работы с использование UDP выше, так как в UDP отсутствует постоянное соединение между клиентом и сервером.

UDP предоставляет сервис ненадежных дейтаграмм, в отличие от TCP, где присутствует модель рукопожатий для обеспечения надежности, упорядочивания и целостности данных.

В виду отсутствия постоянного соединения реализация сервиса депозитов на UDP была чуть сложнее.

Проблемы потери, дублирования и перемешивания решала система контроля индексов всех посылаемых пакетов, а также система таймаутов и повторной посылки данных.

11. Приложения

11.1. TCP

11.1.1. Клиент

Листинг 1: main.h

```
1 #ifndef DEPOSIT_SERVICE_CLIENT_MAIN_H
```

```
2 #define DEPOSIT_SERVICE_CLIENT_MAIN_H
3
4 #include <netinet/in.h>
5 #include <stdlib.h>
6 #include <stdint.h>
7 #include <string.h>
8 #include <stdio.h>
9 #include <netdb.h>
10 #include <netinet/in.h>
11 #include <pthread.h>
12 #include <stdlib.h>
13 #include <string.h>
14 #include <stdio.h>
15 #include <sys/socket.h>
16 #include <arpa/inet.h>
17 #include <errno.h>
18 #include <fcntl.h>
19 #include <unistd.h>
20 #include <stdint.h>
21
22 #define ERROR_PACKET 1
23 #define ACKNOWLEDGMENT_PACKET 2
24 #define LIST_OF_DEPOSITS 3
25 #define OPEN_DEPOSIT_PACKET 4
26 #define REFILL_DEPOSIT 5
27 #define CLOSE_DEPOSIT 6
28 #define GET_LIST_OF_DEPOSITS_PACKET 7
29 #define GET_BANK_AMOUNT 8
30 #define SHOW_BANK_AMOUNT 9
31 #define PLEASE_ADD_PERCENTS 10
32
33 #define DEPOSIT_WAS_OPENED 1
34 #define DEPOSIT_WAS_REFILLED 2
35 #define DEPOSIT_WAS_DELETED 3
36 #define PERCENTS_ADDED 4
37
38 #define SIZE_OF_PACKET_ACK_TYPE 2
39 #define SIZE_OF_PACKET_LENGTH 4
40 #define SIZE_OF_PACKET_TYPE 2
41 #define SIZE_OF_PACKET_AMOUNT 4
42 #define SIZE_OF_ID_DEPOSIT 4
43 #define SIZE_OF_REFILL_AMOUNT 4
44 #define SIZE_OF_CURRENT_AMOUNT sizeof(double)
45 #define SIZE_OF_INITIAL_AMOUNT 4
46
47 #define SHIFT 3
48
49 #endif
```

Листинг 2: main.c

```
1 #include "main.h"
2 int check_number_of_args(int argc, char *argv[]) {
3     if (argc < 3) {
4         fprintf(stderr, "usage␣%s␣hostname␣port\n", argv[0]);
5         return -1;
6     } else {
7         return 1;
8     }
9 }
10
```

```
11 int read_amount() {
12     char buffer[sizeof(uint32_t) + 1];
13     fprintf(stdout, "Enter amount of deposit: ");
14     fflush(stdout);
15     fgets(buffer, sizeof(uint32_t), stdin);
16     return atoi(buffer);
17 }
18
19 int readn(int sockfd, void *dst, size_t len){
20     int total_number_read = 0;
21     int local_number_read;
22
23     while (len > 0) {
24         local_number_read = read(sockfd, (char*) dst + total_number_read, len);
25
26         if (local_number_read == 0) {
27             return total_number_read;
28         }
29
30         if (local_number_read < 0) {
31             return local_number_read;
32         }
33
34         total_number_read += local_number_read;
35         len -= local_number_read;
36     }
37     return total_number_read;
38 }
39
40 void* receive_packet(int sockfd, uint32_t * packet_length, int* number_received)
    ↪  {
41     void* packet;
42
43     *number_received += readn(sockfd, packet_length, SIZE_OF_PACKET_LENGTH);
44     if (number_received <= 0) {
45         return NULL;
46     }
47     packet = malloc(*packet_length);
48     memcpy(packet, packet_length, SIZE_OF_PACKET_LENGTH);
49
50     *number_received += readn(sockfd, (char*) packet + SIZE_OF_PACKET_LENGTH, *
    ↪ packet_length - SIZE_OF_PACKET_LENGTH);
51     return packet;
52 }
53
54 void handle_acknowledgment_packet(void* packet) {
55     uint16_t* ack_type = (uint16_t *) ((char *) packet + SIZE_OF_PACKET_LENGTH +
    ↪  SIZE_OF_PACKET_TYPE);
56     uint16_t* deposit_id = (uint16_t *) ((char *) ack_type +
    ↪ SIZE_OF_PACKET_ACK_TYPE);
57     uint32_t* amount = (uint32_t *) ((char *) deposit_id + SIZE_OF_ID_DEPOSIT);
58
59     switch (*ack_type) {
60         case DEPOSIT_WAS_OPENED:
61             printf("You opened deposit with id %d and amount %d.\n", *deposit_id
    ↪ , *amount);
62             break;
63         case DEPOSIT_WAS_REFILLED:
64             printf("You refilled deposit with id %d and amount %d.\n", *
    ↪ deposit_id, *amount);
```

```
65              break;
66          case DEPOSIT_WAS_DELETED:
67              printf("You deleted deposit with id %d and amount %d.\n", *
    ↪ deposit_id, *amount);
68              break;
69          case PERCENTS_ADDED:
70              printf("Percents to all your deposits were added\n");
71              break;
72          default:
73              break;
74      }
75  }
76
77  void handle_list_of_deposits(void* packet, uint32_t packet_length) {
78      char* cur_packet_pointer = (char*) packet + SIZE_OF_PACKET_LENGTH +
    ↪ SIZE_OF_PACKET_TYPE;
79      uint64_t list_of_deposits_size = packet_length - SIZE_OF_PACKET_LENGTH -
    ↪ SIZE_OF_PACKET_TYPE;
80      uint32_t * deposit_id;
81      uint32_t * amount;
82      uint32_t * initial_amount;
83
84      if (list_of_deposits_size <= 0) {
85          printf("--------------------\n");
86          printf("List of deposits is empty.\n");
87      }
88
89      while (list_of_deposits_size > 0) {
90          deposit_id = (uint32_t *) cur_packet_pointer;
91          amount = (uint32_t *) ( (char*) deposit_id + SIZE_OF_ID_DEPOSIT);
92          initial_amount = (uint32_t *) ((char*) amount + SIZE_OF_PACKET_AMOUNT);
93
94          printf("--------------------\n");
95          printf("id of deposit: %d\n", *deposit_id);
96          printf("accrued amount: %d\n", *initial_amount);
97          printf("amount: %d\n", *amount);
98
99          cur_packet_pointer += (SIZE_OF_ID_DEPOSIT + SIZE_OF_CURRENT_AMOUNT +
    ↪ SIZE_OF_INITIAL_AMOUNT); //next
    item
100         list_of_deposits_size -= (SIZE_OF_ID_DEPOSIT + SIZE_OF_CURRENT_AMOUNT +
    ↪ SIZE_OF_INITIAL_AMOUNT);
101     }
102     printf("--------------------\n");
103 }
104
105 void handle_bank_amount(void* packet, uint32_t packet_length){
106     double bank_amount = *(double *)  ((char*)packet+ SIZE_OF_PACKET_LENGTH +
    ↪ SIZE_OF_PACKET_TYPE);
107     printf("Bank amount: %f\n", bank_amount);
108 }
109
110 void handle_received_packet(void* packet, int packet_length) {
111     uint16_t packet_type = *(uint16_t*) ((char*) packet + SIZE_OF_PACKET_LENGTH)
    ↪ ;
112
113     if (packet_type == ERROR_PACKET) {
114         char* error_msg = (char*) packet + SIZE_OF_PACKET_LENGTH +
    ↪ SIZE_OF_PACKET_TYPE;
115         printf("ERROR: %s\n", error_msg);
116         return;
```

```
117     }
118
119     if (packet_type == ACKNOWLEDGMENT_PACKET) {
120         handle_acknowledgment_packet(packet);
121         return;
122     }
123
124     if (packet_type == LIST_OF_DEPOSITS) {
125         handle_list_of_deposits(packet, packet_length);
126         return;
127     }
128
129     if (packet_type == SHOW_BANK_AMOUNT){
130         handle_bank_amount(packet, packet_length);
131         return;
132     }
133 }
134
135 int get_user_choice() {
136     printf("1) Открыть вклад\n");
137     printf("2) Посмотреть вклады\n");
138     printf("3) Пополнить вклад\n");
139     printf("4) Закрыть вклад\n");
140     printf("5) Завершить работу\n");
141     printf("6) Показать счет банка\n");
142     printf("7) Начислить проценты по вкладам\n");
143
144     char buffer[sizeof(int) + 1];
145     fgets(buffer, sizeof(int), stdin);
146     return atoi(buffer);
147 }
148
149 void* create_open_deposit_packet(uint32_t* packet_size, uint32_t amount) {
150     *packet_size = SIZE_OF_PACKET_LENGTH + SIZE_OF_PACKET_TYPE +
    ↪ SIZE_OF_PACKET_AMOUNT;
151     uint16_t packet_type = OPEN_DEPOSIT_PACKET;
152     char* packet = (char*) malloc(*packet_size);
153
154     memcpy(packet, packet_size, SIZE_OF_PACKET_LENGTH);
155     memcpy(packet + SIZE_OF_PACKET_LENGTH, &packet_type, SIZE_OF_PACKET_TYPE);
156     memcpy(packet + SIZE_OF_PACKET_LENGTH + SIZE_OF_PACKET_TYPE, &amount,
    ↪ SIZE_OF_PACKET_AMOUNT);
157     return packet;
158 }
159
160 int send_open_deposit_packet(int sockfd, uint32_t amount) {
161     uint32_t packet_size;
162     void* packet =  create_open_deposit_packet(&packet_size, amount);
163     int number_written = write(sockfd, packet, packet_size);
164     free(packet);
165     return number_written;
166 }
167
168 void close_client(int sockfd){
169     shutdown(sockfd, SHUT_RDWR);
170     close(sockfd);
171 }
172
173 void* create_get_list_packet(uint32_t* packet_size){
174     *packet_size = SIZE_OF_PACKET_LENGTH + SIZE_OF_PACKET_TYPE;
```

```
175     uint16_t packet_type = GET_LIST_OF_DEPOSITS_PACKET;
176     char* packet = (char*) malloc(*packet_size);
177 
178     memcpy(packet, packet_size, SIZE_OF_PACKET_LENGTH);
179     memcpy(packet + SIZE_OF_PACKET_LENGTH, &packet_type, SIZE_OF_PACKET_TYPE);
180     return packet;
181 }
182 
183 int send_open_list_packet(int sockfd){
184     uint32_t packet_size;
185     void* packet =  create_get_list_packet(&packet_size);
186     int number_written = write(sockfd, packet, packet_size);
187     free(packet);
188     return number_written;
189 }
190 
191 int read_id() {
192     char buffer[sizeof(uint32_t) + 2];
193     fprintf(stdout, "Enter id of deposit: ");
194     fflush(stdout);
195     fgets(buffer, sizeof(uint32_t), stdin);
196     return atoi(buffer);
197 }
198 
199 int read_refill_amount() {
200     char buffer[sizeof(uint32_t) + 2];
201     fprintf(stdout, "Enter refilling amount: ");
202     fflush(stdout);
203     fgets(buffer, sizeof(uint32_t)+2, stdin);
204     return atoi(buffer);
205 }
206 
207 void* create_refill_deposit_packet(uint32_t* packet_size, uint32_t deposit_id,
    ↪ uint32_t refill_amount){
208     *packet_size = SIZE_OF_PACKET_LENGTH + SIZE_OF_PACKET_TYPE +
    ↪ SIZE_OF_ID_DEPOSIT + SIZE_OF_REFILL_AMOUNT;
209     uint16_t packet_type = REFILL_DEPOSIT;
210     char* packet = (char*) malloc(*packet_size);
211 
212     memcpy(packet, packet_size, SIZE_OF_PACKET_LENGTH);
213     memcpy(packet + SIZE_OF_PACKET_LENGTH, &packet_type, SIZE_OF_PACKET_TYPE);
214     memcpy(packet + SIZE_OF_PACKET_LENGTH + SIZE_OF_PACKET_TYPE, &deposit_id,
    ↪ SIZE_OF_ID_DEPOSIT);
215     memcpy(packet + SIZE_OF_PACKET_LENGTH + SIZE_OF_PACKET_TYPE +
    ↪ SIZE_OF_ID_DEPOSIT, &refill_amount, SIZE_OF_REFILL_AMOUNT);
216     return packet;
217 }
218 
219 int send_refill_deposit_packet(int sockfd, uint32_t deposit_id, uint32_t
    ↪ refill_amount) {
220     uint32_t packet_size;
221     void* packet =  create_refill_deposit_packet(&packet_size, deposit_id,
    ↪ refill_amount);
222     int number_written = write(sockfd, packet, packet_size);
223     free(packet);
224     return number_written;
225 }
226 
227 void* create_close_deposit_packet(uint32_t* packet_size, uint32_t deposit_id){
228     *packet_size = SIZE_OF_PACKET_LENGTH + SIZE_OF_PACKET_TYPE +
```

```
    ↪ SIZE_OF_ID_DEPOSIT;
229     uint16_t packet_type = CLOSE_DEPOSIT;
230     char* packet = (char*) malloc(*packet_size);
231 
232     memcpy(packet, packet_size, SIZE_OF_PACKET_LENGTH);
233     memcpy(packet + SIZE_OF_PACKET_LENGTH, &packet_type, SIZE_OF_PACKET_TYPE);
234     memcpy(packet + SIZE_OF_PACKET_LENGTH + SIZE_OF_PACKET_TYPE, &deposit_id,
    ↪ SIZE_OF_ID_DEPOSIT);
235     return packet;
236 }
237 void* create_show_bank_amount_packet(uint32_t* packet_size){
238     *packet_size = SIZE_OF_PACKET_LENGTH + SIZE_OF_PACKET_TYPE;
239     uint16_t packet_type = GET_BANK_AMOUNT;
240     char* packet = (char*) malloc(*packet_size);
241 
242     memcpy(packet, packet_size, SIZE_OF_PACKET_LENGTH);
243     memcpy(packet + SIZE_OF_PACKET_LENGTH, &packet_type, SIZE_OF_PACKET_TYPE);
244     return packet;
245 }
246 
247 int send_close_deposit_packet(int sockfd, uint32_t deposit_id){
248     uint32_t packet_size;
249     void* packet = create_close_deposit_packet(&packet_size, deposit_id);
250     int number_written = write(sockfd, packet, packet_size);
251     free(packet);
252     return number_written;
253 }
254 
255 void* create_add_percent_packet(uint32_t* packet_size){
256     *packet_size = SIZE_OF_PACKET_LENGTH + SIZE_OF_PACKET_TYPE;
257     uint16_t packet_type = PLEASE_ADD_PERCENTS;
258     char* packet = (char*) malloc(*packet_size);
259     memcpy(packet, packet_size, SIZE_OF_PACKET_LENGTH);
260     memcpy(packet + SIZE_OF_PACKET_LENGTH, &packet_type, SIZE_OF_PACKET_TYPE);
261     return packet;
262 }
263 
264 int send_add_percent_packet(int sockfd){
265     uint32_t packet_size;
266     void* packet = create_add_percent_packet(&packet_size);
267     int number_written = write(sockfd, packet, packet_size);
268     free(packet);
269     return number_written;
270 }
271 
272 int send_show_bank_amount(int sockfd){
273     uint32_t packet_size;
274     void* packet = create_show_bank_amount_packet(&packet_size);
275     int number_written = write(sockfd, packet, packet_size);
276     free(packet);
277     return (number_written == (int) packet_size) ? number_written : -1;
278 }
279 
280 void show_list_of_deposits(void* packet, int packet_length){
281     char* packet_pointer = (char*) packet + SIZE_OF_PACKET_LENGTH +
    ↪ SIZE_OF_PACKET_TYPE;
282     uint64_t size = packet_length - SIZE_OF_PACKET_LENGTH - SIZE_OF_PACKET_TYPE;
283     uint32_t deposit_id;
284     uint32_t initial_amount;
285     double current_amount;
```

```
286
287     if (size <= 0 ){
288         printf("--------------------\n");
289         printf("List is empty.\n");
290     }
291
292     while (size > 0){
293         deposit_id = *(uint32_t *) packet_pointer;
294         initial_amount = *(uint32_t *) ( (char*) packet_pointer +
    ↪ SIZE_OF_ID_DEPOSIT);
295         current_amount = *(double *) ( (char*) packet_pointer +
    ↪ SIZE_OF_ID_DEPOSIT + SIZE_OF_INITIAL_AMOUNT);
296
297         printf("--------------------\n");
298         printf("id of deposit: %d\n", deposit_id);
299         printf("initial amount of deposit: %d\n", initial_amount);
300         printf("current amount of deposit: %f\n", current_amount);
301
302         packet_pointer += (SIZE_OF_ID_DEPOSIT + SIZE_OF_CURRENT_AMOUNT +
    ↪ SIZE_OF_INITIAL_AMOUNT);
303         size -= (SIZE_OF_ID_DEPOSIT + SIZE_OF_CURRENT_AMOUNT +
    ↪ SIZE_OF_INITIAL_AMOUNT);
304     }
305     printf("--------------------\n");
306 }
307
308 int main(int argc, char *argv[]) {
309     int sockfd;
310     uint16_t portno;
311     struct sockaddr_in serv_addr;
312     struct hostent *server;
313
314     if (check_number_of_args(argc, argv) < 0){
315         return 0;
316     }
317
318     portno = (uint16_t) atoi(argv[2]);
319     sockfd = socket(AF_INET, SOCK_STREAM, 0);
320
321     if (sockfd < 0) {
322         perror("ERROR opening socket");
323         return 0;
324     }
325
326     server = gethostbyname(argv[1]);
327     if (server == NULL) {
328         fprintf(stderr, "ERROR, no such host\n");
329         return 0;
330     }
331
332     bzero((char *) &serv_addr, sizeof(serv_addr));
333     serv_addr.sin_family = AF_INET;
334     bcopy(server->h_addr, (char *) &serv_addr.sin_addr.s_addr, (size_t) server->
    ↪ h_length);
335     serv_addr.sin_port = htons(portno);
336
337     if (connect(sockfd,(struct sockaddr *) &serv_addr, sizeof(serv_addr)) < 0) {
338         perror("ERROR connecting");
339         return 0;
340     }
```

```
341
342     void * received_packet;
343     uint32_t packet_length;
344     int number_received;
345     int amount;
346     int deposit_id;
347     int refill_amount;
348
349     while (1) {
350         switch (get_user_choice()) {
351             case 1:
352                 if ((amount = read_amount()) < 0){
353                     printf("ERROR: invalid value\n");
354                     continue;
355                 }
356
357                 if (send_open_deposit_packet(sockfd, amount) < 0){
358                     printf("ERROR: server disconnected.\n");
359                     close_client(sockfd);
360                     return 0;
361                 }
362
363                 received_packet = receive_packet(sockfd, &packet_length, &
    ↪ number_received);
364
365                 if (received_packet == NULL || number_received < packet_length)
    ↪ {
366                     printf("ERROR: server disconnected.\n");
367                     close_client(sockfd);
368                     return 0;
369                 }
370
371                 handle_received_packet(received_packet, packet_length);
372                 free(received_packet);
373                 break;
374
375             case 2:
376                 if (send_open_list_packet(sockfd) < 0){
377                     printf("ERROR: server disconnected.\n");
378                     close_client(sockfd);
379                     return 0;
380                 }
381
382                 received_packet = receive_packet(sockfd, &packet_length, &
    ↪ number_received);
383
384                 if (received_packet == NULL || number_received < packet_length)
    ↪ {
385                     printf("ERROR: server disconnected.\n");
386                     free(received_packet);
387                     close_client(sockfd);
388                     return 0;
389                 }
390
391                 show_list_of_deposits(received_packet, packet_length);
392                 break;
393
394             case 3:
395                 if ((deposit_id = read_id()) < 0){
396                     printf("ERROR: invalid value\n");
```

```
397                     continue;
398                 }
399
400                 if ((refill_amount = read_refill_amount()) < 0){
401                     printf("ERROR: invalid value\n");
402                     continue;
403                 }
404
405                 if (send_refill_deposit_packet(sockfd, deposit_id, refill_amount
    ↪ ) < 0){
406                     printf("ERROR: server disconnected.\n");
407                     close_client(sockfd);
408                     return 0;
409                 }
410
411                 received_packet = receive_packet(sockfd, &packet_length, &
    ↪ number_received);
412
413                 if (received_packet == NULL || number_received < packet_length)
    ↪ {
414                     printf("ERROR: server disconnected.\n");
415                     close_client(sockfd);
416                     return 0;
417                 }
418
419                 handle_received_packet(received_packet, packet_length);
420                 free(received_packet);
421                 break;
422
423             case 4:
424                 if ((deposit_id = read_id()) < 0){
425                     printf("ERROR: invalid value\n");
426                     continue;
427                 }
428
429                 if (send_close_deposit_packet(sockfd, deposit_id) < 0){
430                     printf("ERROR: server disconnected.\n");
431                     close_client(sockfd);
432                     return 0;
433                 }
434
435                 received_packet = receive_packet(sockfd, &packet_length, &
    ↪ number_received);
436
437                 if (received_packet == NULL || number_received < packet_length)
    ↪ {
438                     printf("ERROR: server disconnected.\n");
439                     close_client(sockfd);
440                     return 0;
441                 }
442
443                 handle_received_packet(received_packet, packet_length);
444                 free(received_packet);
445                 break;
446
447             case 5:
448                 close_client(sockfd);
449                 return 0;
450
451             case 6:
```

```
452                 if (send_show_bank_amount(sockfd) < 0){
453                     printf("ERROR: server disconnected.\n");
454                     close_client(sockfd);
455                     return 0;
456                 }
457 
458                 received_packet = receive_packet(sockfd, &packet_length, &
    ↪ number_received);
459 
460                 if (received_packet == NULL || number_received < packet_length)
    ↪ {
461                     printf("ERROR: server disconnected.\n");
462                     close_client(sockfd);
463                     return 0;
464                 }
465 
466                 handle_received_packet(received_packet, packet_length);
467                 free(received_packet);
468                 break;
469 
470             case 7:
471                 if (send_add_percent_packet(sockfd) < 0){
472                     printf("ERROR: server disconnected.\n");
473                     close_client(sockfd);
474                     return 0;
475                 }
476 
477                 received_packet = receive_packet(sockfd, &packet_length, &
    ↪ number_received);
478 
479                 if (received_packet == NULL || number_received < packet_length)
    ↪ {
480                     printf("ERROR: server disconnected.\n");
481                     close_client(sockfd);
482                     return 0;
483                 }
484 
485                 handle_received_packet(received_packet, packet_length);
486                 free(received_packet);
487                 break;
488 
489             default:
490                 printf("ERROR: wrong choice\n");
491             break;
492         }
493     }
494 }
```

11.1.2. Сервер

Листинг 3: main.h

```
1 #ifndef SERVER_MAIN_H
2 #define SERVER_MAIN_H
3 
4 #include <netinet/in.h>
5 #include <pthread.h>
6 #include <stdlib.h>
7 #include <string.h>
```

```c
#include <stdio.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <errno.h>
#include <fcntl.h>
#include <unistd.h>
#include <stdint.h>
#include <math.h>

typedef struct accepting_thread_input {
    int port;
    int* initial_sockfd;
} Accepting_thread_input;

pthread_t create_accepting_thread(int port_number, int* initial_sockfd);

typedef struct {
    int sockfd;
    int port;
    struct sockaddr client_addr;
    uint64_t client_addr_len;
} Listening_thread_input;

typedef struct User_info {
    char* address;
    int port;
    int sockfd;
    pthread_t client_thread;
    struct User_info* next;
} User_info;

typedef struct deposit_info {
    uint32_t deposit_id;
    double current_amount;
    int client_sockfd;
    uint32_t initial_amount;
    struct deposit_info* next;
} Deposit_info;

#define SIZE_OF_PACKET_ACK_TYPE 2
#define SIZE_OF_PACKET_LENGTH 4
#define SIZE_OF_PACKET_TYPE 2
#define SIZE_OF_PACKET_AMOUNT 4
#define SIZE_OF_ID_DEPOSIT 4
#define SIZE_OF_REFILL_AMOUNT 4
#define SIZE_OF_CURRENT_AMOUNT sizeof(double)
#define SIZE_OF_INITIAL_AMOUNT 4
#define SIZE_OF_PACKET_AMOUNT 4

#define SIZE_OF_ZERO_CHAR 1

#define ERROR_PACKET 1
#define ACKNOWLEDGMENT_PACKET 2
#define LIST_OF_DEPOSITS_PACKET 3
#define OPEN_DEPOSIT_PACKET 4
#define REFILL_DEPOSIT_PACKET 5
#define CLOSE_DEPOSIT_PACKET 6
#define GET_LIST_OF_DEPOSITS_PACKET 7
#define GET_BANK_AMOUNT 8
#define SHOW_BANK_AMOUNT 9
```

```
68 #define PLEASE_ADD_PERCENTS 10
69
70 #define DEPOSIT_WAS_ADDED 1
71 #define DEPOSIT_WAS_REFILLED 2
72 #define DEPOSIT_WAS_REMOVED 3
73 #define PERCENTS_ADDED 4
74
75 void list_of_deposits_remove_all_client_deposits(int client_socket);
76 void list_of_clients_add(User_info* new_client);
77 User_info* make_new_client(int sockfd, int port, char* address, pthread_t
       ↪ client_thread);
78 int list_of_clients_remove(int sockfd);
79 void list_of_deposits_send(int client_sockfd);
80 void handle_add_deposit_packet(int client_sockfd, void* packet);
81 void handle_remove_deposit_packet(int client_sockfd, void* packet);
82 pthread_t create_listening_thread(int sockfd);
83 void list_of_clients_remove_all();
84 void init_list_of_clients_mutex();
85 void init_list_of_deposits_mutex();
86 pthread_t create_user_listening_thread(int* initial_socket);
87 void list_of_deposits_remove_all();
88 void handle_refill_deposit_packet(int client_sockfd, void* packet);
89 int refill_deposit(int client_sockfd, uint32_t deposit_id, uint32_t added_amount
       ↪ );
90 double list_of_deposits_remove(int client_sockfd, uint32_t deposit_id);
91 void list_of_deposits_add(Deposit_info* new_deposit);
92 void list_of_deposits_add_percents(void);
93 uint32_t generate_deposit_id(void);
94 void list_of_clients_export(FILE* dst_fd);
95 void list_of_deposits_export_bank_amount();
96 Deposit_info* make_new_deposit(int deposit_id, int initial_amount,int
       ↪ client_sockfd);
97 void list_of_deposits_all_deposits(FILE* output_file);
98 void send_bank_amount(int client_sockfd);
99 void handle_added_percents_packet(int client_sockfd, void* packet);
100 int add_percents_to_client(int client_socket);
101 pthread_t list_of_clients_get_client_thread(int sockfd);
102
103 #endif
```

Листинг 4: main.c

```
1  #include "main.h"
2  int checkArguments(int argc, char* argv[]) {
3      if (argc != 2) {
4          fprintf(stderr, "usage␣%s␣port␣\n", argv[0]);
5          return -1;
6      }
7      return 0;
8  }
9
10 int main(int argc, char* argv[]) {
11     int initial_socket;
12
13     if (checkArguments(argc, argv) < 0){
14         return 0;
15     }
16
17     const uint16_t port_number = (uint16_t)  atoi(argv[1]);
18     initial_socket = socket(AF_INET, SOCK_STREAM, 0);
19
```

```
20      if (initial_socket < 0) {
21          fprintf(stderr, "ERROR opening init socket\n");
22          exit(1);
23      }
24
25      init_list_of_clients_mutex();
26      init_list_of_deposits_mutex();
27
28      pthread_t accepting_thread = create_accepting_thread(port_number, &
        ↪ initial_socket);
29      pthread_t console_listening_thread = create_user_listening_thread(&
        ↪ initial_socket);
30
31      pthread_join(accepting_thread, NULL);
32      pthread_join(console_listening_thread, NULL);
33
34      list_of_clients_remove_all();
35      list_of_deposits_remove_all();
36      return 0;
37  }
```

Листинг 5: accepting_thread.c

```
1  #include "main.h"
2  static void make_socket_reusable(int sockfd) {
3      if (setsockopt(sockfd, SOL_SOCKET, SO_REUSEADDR, &(int){ 1 }, sizeof(int)) <
       ↪  0) {
4          fprintf(stderr, "ERROR: setsockopt(SO_REUSEADDR) failed\n");
5      }
6  }
7
8  void fill_server_info(struct sockaddr_in* server_addr, int port) {
9      bzero(server_addr, sizeof(*server_addr));
10     server_addr -> sin_port = htons(port);
11     server_addr -> sin_addr.s_addr = INADDR_ANY;
12     server_addr -> sin_family = AF_INET;
13 }
14
15 static int bind_socket(const int* sockfd, struct sockaddr_in* server_addr) {
16     if (bind(*sockfd, (struct sockaddr *) server_addr, sizeof(*server_addr)) <
       ↪ 0) {
17         perror("ERROR on binding\n");
18         return -1;
19     }
20     return 0;
21 }
22
23 void* accepting_thread(void* arg) {
24     struct sockaddr_in server_addr, client_addr;
25     int* initial_socket;
26     int newsockfd;
27     unsigned int client_len;
28     int port;
29
30     initial_socket = ((Accepting_thread_input*)arg)->initial_sockfd;
31     port = ((Accepting_thread_input*) arg) -> port;
32
33     if (initial_socket < 0) {
34         fprintf(stderr, "ERROR opening init socket\n");
35         exit(1);
36     }
```

```
37
38      make_socket_reusable(*initial_socket);
39      fill_server_info(&server_addr, port);
40
41      if (bind_socket(initial_socket, &server_addr) < 0) {
42          shutdown(*initial_socket, SHUT_RDWR);
43          close(*initial_socket);
44          return NULL;
45      }
46
47      listen(*initial_socket, 5);
48      client_len = sizeof(client_addr);
49
50      while(1) {
51          newsockfd = accept(*initial_socket, (struct sockaddr *) &client_addr, &
    ↪ client_len);
52
53          if (newsockfd < 0) {
54              return NULL;
55          }
56
57          pthread_t client_thread = create_listening_thread(newsockfd);
58          list_of_clients_add(make_new_client(newsockfd, port, inet_ntoa(
    ↪ client_addr.sin_addr), client_thread));
59      }
60  }
61
62  Accepting_thread_input* init_accepting_thread_input_structure(int port, int*
    ↪ initial_sockfd) {
63      Accepting_thread_input* new_input_structure = (Accepting_thread_input*)
    ↪ malloc(sizeof(Accepting_thread_input));
64
65      new_input_structure -> port = port;
66      new_input_structure -> initial_sockfd = initial_sockfd;
67
68      return new_input_structure;
69  }
70
71  pthread_t create_accepting_thread(int port, int* initial_sockfd) {
72      pthread_t new_thread;
73      Accepting_thread_input* accepting_thread_input =
    ↪ init_accepting_thread_input_structure(port, initial_sockfd);
74
75      if (pthread_create(&new_thread, NULL, accepting_thread, (void*)
    ↪ accepting_thread_input) != 0) {
76          return -1;
77      }
78
79      return new_thread;
80  }
```

Листинг 6: client_thread.c

```
1 #include "main.h"
2 static int readn(int sockfd, void *dst, size_t len) {
3     int total_number_read = 0;
4     int local_number_read;
5
6     while (len > 0) {
7         local_number_read = recv(sockfd, dst + total_number_read, len,
    ↪ MSG_WAITALL);
```

```
8
9           if (local_number_read == 0) {
10              return total_number_read;
11          }
12
13          if (local_number_read < 0) {
14              return -1;
15          }
16
17          total_number_read += local_number_read;
18          len -= local_number_read;
19      }
20      return total_number_read;
21  }
22
23  void *socket_listening_thread(void *arg) {
24      int client_sockfd = ((Listening_thread_input *) arg)->sockfd;
25      uint32_t packet_length;
26      uint16_t packet_type;
27      int number_read;
28      void *packet;
29
30      while (1) {
31          number_read = readn(client_sockfd, &packet_length, SIZE_OF_PACKET_LENGTH
    ↪ );
32
33          if (number_read < SIZE_OF_PACKET_LENGTH) {
34              list_of_clients_remove(client_sockfd);
35              list_of_deposits_remove_all_client_deposits(client_sockfd);
36              free(arg);
37              return NULL;
38          }
39
40          packet = malloc(packet_length);
41          number_read = readn(client_sockfd, packet, packet_length -
    ↪ SIZE_OF_PACKET_LENGTH);
42
43          if (number_read < packet_length - SIZE_OF_PACKET_LENGTH) {
44              list_of_clients_remove(client_sockfd);
45              list_of_deposits_remove_all_client_deposits(client_sockfd);
46              free(packet);
47              free(arg);
48              return NULL;
49          }
50
51          if ((packet_type = *(uint16_t *) packet) == OPEN_DEPOSIT_PACKET) {
52              handle_add_deposit_packet(client_sockfd, packet);
53
54          } else if (packet_type == CLOSE_DEPOSIT_PACKET) {
55              handle_remove_deposit_packet(client_sockfd, packet);
56
57          } else if (packet_type == GET_LIST_OF_DEPOSITS_PACKET) {
58              list_of_deposits_send(client_sockfd);
59
60          } else if (packet_type == REFILL_DEPOSIT_PACKET) {
61              handle_refill_deposit_packet(client_sockfd, packet);
62
63          } else if (packet_type == GET_BANK_AMOUNT) {
64              send_bank_amount(client_sockfd);
65
```

```
66          } else if (packet_type == PLEASE_ADD_PERCENTS) {
67              handle_added_percents_packet(client_sockfd, packet);
68          }
69
70          free(packet);
71      }
72  }
73
74  Listening_thread_input *init_listening_thread_input_structure(int sockfd) {
75      Listening_thread_input *new_input_structure = (Listening_thread_input *)
    ↪ malloc(sizeof(Listening_thread_input));
76
77      new_input_structure->sockfd = sockfd;
78
79      return new_input_structure;
80  }
81
82  pthread_t create_listening_thread(int sockfd) {
83      pthread_t listening_thread;
84
85      Listening_thread_input *listening_thread_input =
    ↪ init_listening_thread_input_structure(sockfd);
86
87      if (pthread_create(&listening_thread, NULL, socket_listening_thread,
    ↪ listening_thread_input)) {
88          return -1;
89      }
90
91      return listening_thread;
92  }
```

Листинг 7: console_thread.c

```
1  #include "main.h"
2
3  int get_user_choice() {
4      printf("1) Вывести список клиентов\n");
5      printf("2) Отключить клиента\n");
6      printf("3) Начислить проценты\n");
7      printf("4) Показать счёт банка\n");
8      printf("5) Показать все вклады\n");
9      printf("6) Завершить работу\n");
10
11     char buffer[sizeof(int) + 2];
12     fgets(buffer, sizeof(int)+2, stdin);
13     return atoi(buffer);
14 }
15
16 void* console_listening_thread(void* arg) {
17     int sockfd;
18
19     while(1) {
20         switch (get_user_choice()){
21             case 1:
22                 list_of_clients_export(stdout);
23                 break;
24
25             case 2:
26                 printf("Enter client socket number: ");
27                 fflush(stdout);
28                 scanf("%d", &sockfd);
```

```
29                  shutdown(sockfd, SHUT_RDWR);
30                  close(sockfd);
31
32                  pthread_t client_thread;
33                  if ((client_thread = list_of_clients_get_client_thread(sockfd))
    ↪ == -1) {
34                      printf("ERROR: no such user.\n");
35                  } else {
36                      pthread_join(client_thread, NULL);
37                  }
38                  break;
39
40              case 3:
41                  list_of_deposits_add_percents();
42                  break;
43
44              case 4:
45                  list_of_deposits_export_bank_amount();
46                  break;
47
48              case 5:
49                  list_of_deposits_all_deposits(stdout);
50                  break;
51
52              case 6:
53                  shutdown(*(int*) arg, SHUT_RDWR);
54                  close(*(int*) arg);
55                  return NULL;
56
57              default:
58                  printf("ERROR: wrong choice\n");
59                  break;
60          }
61      }
62  }
63
64  pthread_t create_user_listening_thread(int* initial_socket) {
65      pthread_t user_listening_thread;
66
67      if (pthread_create(&user_listening_thread, NULL, console_listening_thread, (
    ↪ void*) initial_socket)) {
68          return -1;
69      }
70
71      return user_listening_thread;
72  }
```

Листинг 8: list_of_clients.c

```
1  #include "main.h"
2  static pthread_mutex_t list_of_clients_mutex;
3  static User_info* root = NULL;
4
5  void init_list_of_clients_mutex() {
6      pthread_mutex_init(&list_of_clients_mutex, NULL);
7  }
8
9  User_info* make_new_client(int sockfd, int port, char* address, pthread_t
    ↪ client_thread) {
10     User_info* new_client = (User_info*) malloc(sizeof(User_info));
11
```

```
12      new_client -> port = port;
13      new_client -> address = address;
14      new_client -> sockfd = sockfd;
15      new_client -> next = NULL;
16      new_client -> client_thread = client_thread;
17
18      return new_client;
19  }
20
21  pthread_t list_of_clients_get_client_thread(int sockfd) {
22      User_info* iterator = root;
23
24      while (iterator != NULL && iterator->sockfd != sockfd) {
25          iterator = iterator->next;
26      }
27
28      return (iterator == NULL ) ? -1 : iterator->client_thread;
29  }
30
31  void list_of_clients_add(User_info* new_client) {
32      pthread_mutex_lock(&list_of_clients_mutex);
33
34      if (root == NULL) {
35          root = new_client;
36      } else {
37          User_info* iterator;
38          for (iterator = root; iterator -> next != NULL; iterator = iterator ->
        ↪ next) {
39          }
40          iterator -> next = new_client;
41      }
42
43      pthread_mutex_unlock(&list_of_clients_mutex);
44      printf("CLIENT␣␣%d␣client␣connected\n", new_client->sockfd);
45  }
46
47  int list_of_clients_remove(int sockfd) {
48      pthread_mutex_lock(&list_of_clients_mutex);
49
50      if (root != NULL) {
51          User_info* iterator = root;
52          User_info* prev = NULL;
53
54          while (iterator != NULL && iterator->sockfd != sockfd) {
55              prev = iterator;
56              iterator = iterator->next;
57          }
58
59          if (iterator == NULL) {
60              pthread_mutex_unlock(&list_of_clients_mutex);
61              return -1;
62          }
63
64          if (prev == NULL) {
65              shutdown(root -> sockfd, SHUT_RDWR);
66              close(root -> sockfd);
67              pthread_join( root->client_thread, NULL);
68
69              User_info* old_root = root;
70              root = root -> next;
```

```
71              free(old_root);
72
73              pthread_mutex_unlock(&list_of_clients_mutex);
74              printf("CLIENT %d client disconnected\n", sockfd);
75              return 1;
76          }
77
78          prev -> next = iterator -> next;
79          shutdown(iterator -> sockfd, SHUT_RDWR);
80          close(iterator -> sockfd);
81          pthread_join( iterator->client_thread, NULL);
82          free(iterator);
83
84          pthread_mutex_unlock(&list_of_clients_mutex);
85          printf("CLIENT %d client disconnected\n", sockfd);
86          return 1;
87      }
88
89      pthread_mutex_unlock(&list_of_clients_mutex);
90      return -1;
91  }
92
93  void list_of_clients_export(FILE* dst_fd) {
94
95      pthread_mutex_lock(&list_of_clients_mutex);
96
97      if (root == NULL) {
98          fprintf(dst_fd, "No clients\n");
99          pthread_mutex_unlock(&list_of_clients_mutex);
100         return;
101     }
102
103     User_info* iterator = root;
104     fprintf(dst_fd,"%5s %8s %8s %16s\n", "index", "socket", "port", "address");
105
106     for (int index = 1; iterator != NULL; index++, iterator = iterator->next) {
107         fprintf(dst_fd, "%5d %8d %8d %16s\n", index++, iterator->sockfd,
    ↪ iterator->port, iterator->address);
108     }
109
110     pthread_mutex_unlock(&list_of_clients_mutex);
111 }
112
113 void list_of_clients_remove_all() {
114     pthread_t thread;
115
116     while (root != NULL) {
117         pthread_mutex_lock(&list_of_clients_mutex);
118         thread = root->client_thread;
119         shutdown(root->sockfd, SHUT_RDWR);
120         close(root->sockfd);
121         pthread_mutex_unlock(&list_of_clients_mutex);
122         pthread_join(thread, NULL);
123     }
124
125 }
```

Листинг 9: list_of_deposits.c

```
1 #include "main.h"
2 static double bank_amount = 0.0;
```

```
3  static pthread_mutex_t list_of_deposits_mutex;
4  static Deposit_info* root = NULL;
5
6  void init_list_of_deposits_mutex() {
7      pthread_mutex_init(&list_of_deposits_mutex, NULL);
8  }
9
10 Deposit_info* make_new_deposit(int deposit_id, int initial_amount,int
       ↪ client_sockfd) {
11     Deposit_info* new_deposit = (Deposit_info*) malloc(sizeof(Deposit_info));
12
13     new_deposit->client_sockfd = client_sockfd;
14     new_deposit->current_amount = initial_amount;
15     new_deposit->initial_amount = initial_amount;
16     new_deposit->deposit_id = deposit_id;
17     new_deposit->next = NULL;
18
19     return new_deposit;
20 }
21
22 uint32_t generate_deposit_id() {
23     uint32_t deposit_id = 1;
24
25     for (Deposit_info* iterator = root; iterator != NULL; iterator = iterator->
       ↪ next) {
26         deposit_id = iterator->deposit_id + 1;
27     }
28
29     return deposit_id;
30 }
31
32 void list_of_deposits_add(Deposit_info* new_deposit) {
33     pthread_mutex_lock(&list_of_deposits_mutex);
34     Deposit_info* iterator = root;
35
36     while (iterator != NULL && iterator->next != NULL) {
37         iterator = iterator->next;
38     }
39
40     if (iterator == NULL) {
41         root = new_deposit;
42         pthread_mutex_unlock(&list_of_deposits_mutex);
43         return;
44     }
45
46     iterator->next = new_deposit;
47     pthread_mutex_unlock(&list_of_deposits_mutex);
48 }
49
50 void list_of_deposits_remove_all_client_deposits(int client_socket) {
51     pthread_mutex_lock(&list_of_deposits_mutex);
52     Deposit_info* iterator = root;
53     Deposit_info* iterator_prev = NULL;
54
55     while (iterator != NULL) {
56         if (iterator->client_sockfd == client_socket) {
57             if(iterator_prev == NULL) {
58                 Deposit_info* old_root = root;
59                 root = root->next;
60                 free(old_root);
```

```
61                  iterator = root;
62              } else {
63                  iterator_prev->next = iterator->next;
64                  free(iterator);
65                  iterator = iterator_prev->next;
66              }
67          } else {
68              iterator_prev = iterator;
69              iterator = iterator->next;
70          }
71      }
72
73      pthread_mutex_unlock(&list_of_deposits_mutex);
74  }
75
76  double list_of_deposits_remove(int client_sockfd, uint32_t deposit_id) {
77      pthread_mutex_lock(&list_of_deposits_mutex);
78      Deposit_info* iterator = root;
79      Deposit_info* iterator_prev = NULL;
80
81      while (iterator != NULL && iterator->deposit_id != deposit_id) {
82          iterator_prev = iterator;
83          iterator = iterator->next;
84      }
85
86      if (iterator == NULL || iterator->client_sockfd != client_sockfd) {
87          pthread_mutex_unlock(&list_of_deposits_mutex);
88          return -1;
89      }
90
91      if (iterator_prev == NULL) {
92          double final_amount = root->current_amount;
93
94          Deposit_info* old_root = root;
95          root = root->next;
96          free(old_root);
97
98          pthread_mutex_unlock(&list_of_deposits_mutex);
99          return final_amount;
100     }
101
102     double final_amount = iterator->current_amount;
103     iterator_prev->next = iterator->next;
104     free(iterator);
105
106     pthread_mutex_unlock(&list_of_deposits_mutex);
107     return final_amount;
108 }
109
110 void list_of_deposits_add_percents(){
111     pthread_mutex_lock(&list_of_deposits_mutex);
112     Deposit_info* iterator = root;
113
114     while (iterator != NULL) {
115         bank_amount +=  iterator->current_amount * 0.05;
116         iterator->current_amount +=  iterator->current_amount * 0.1;
117         iterator = iterator -> next;
118     }
119
120     pthread_mutex_unlock(&list_of_deposits_mutex);
```

```
121 }
122
123 static uint64_t count_list_length(int client_sockfd) {
124     uint64_t result;
125     Deposit_info* iterator;
126
127     for (iterator = root,  result = 0; iterator != NULL ; iterator = iterator->
    ↪ next) {
128         if(client_sockfd == iterator->client_sockfd){
129             result += (SIZE_OF_ID_DEPOSIT + SIZE_OF_INITIAL_AMOUNT +
    ↪ SIZE_OF_CURRENT_AMOUNT);
130         }
131     }
132     return result;
133 }
134
135 void send_bank_amount(int client_sockfd){
136     uint32_t packet_length = SIZE_OF_PACKET_LENGTH + SIZE_OF_PACKET_TYPE +
    ↪ sizeof(double);
137     uint16_t packet_type = SHOW_BANK_AMOUNT;
138     void* packet = malloc(packet_length);
139
140     memcpy(packet, &packet_length, SIZE_OF_PACKET_LENGTH);
141     memcpy(packet + SIZE_OF_PACKET_LENGTH, &packet_type, SIZE_OF_PACKET_TYPE);
142     memcpy(packet + SIZE_OF_PACKET_LENGTH + SIZE_OF_PACKET_TYPE, &bank_amount,
    ↪ sizeof(double));
143
144     if (write(client_sockfd, packet, packet_length) < packet_length ) {
145         printf("ERROR␣sending␣bank_amount␣packet␣to␣client.␣Client␣socket␣number
    ↪ :␣%d.\n", client_sockfd);
146     }
147
148     free(packet);
149 }
150
151 void list_of_deposits_send(int client_sockfd) {
152     uint32_t packet_length = SIZE_OF_PACKET_LENGTH + SIZE_OF_PACKET_TYPE +
    ↪ count_list_length(client_sockfd);
153     uint16_t packet_type = LIST_OF_DEPOSITS_PACKET;
154
155     write(client_sockfd, &packet_length, SIZE_OF_PACKET_LENGTH);
156     write(client_sockfd, &packet_type, SIZE_OF_PACKET_TYPE);
157
158     for (Deposit_info* iterator = root; iterator != NULL; iterator = iterator->
    ↪ next) {
159         if(client_sockfd == iterator->client_sockfd){
160             write(client_sockfd, &(iterator->deposit_id), SIZE_OF_ID_DEPOSIT);
161             write(client_sockfd, &(iterator->initial_amount),
    ↪ SIZE_OF_INITIAL_AMOUNT);
162             write(client_sockfd, &(iterator->current_amount),
    ↪ SIZE_OF_CURRENT_AMOUNT);
163         }
164     }
165 }
166
167 void list_of_deposits_remove_all() {
168     Deposit_info* iterator;
169     Deposit_info* iterator_next;
170     pthread_mutex_lock(&list_of_deposits_mutex);
171
```

```
172     for (iterator = root; iterator != NULL ; iterator = iterator_next) {
173         iterator_next = iterator->next;
174         free(iterator);
175     }
176 
177     pthread_mutex_unlock(&list_of_deposits_mutex);
178 }
179 
180 void list_of_deposits_all_deposits(FILE* output_file){
181     Deposit_info* iterator;
182     pthread_mutex_lock(&list_of_deposits_mutex);
183 
184     if (root == NULL) {
185         fprintf(output_file, "No deposits\n");
186         pthread_mutex_unlock(&list_of_deposits_mutex);
187         return;
188     }
189 
190     fprintf(output_file,"%6s %10s %14s %14s\n", "client", "deposit_id", "
    ↪ initial_amount", "current_amount");
191 
192     for (iterator = root; iterator != NULL ; iterator = iterator->next) {
193         fprintf(output_file, "%6d %10d %14d %14f\n", iterator->client_sockfd,
    ↪ iterator->deposit_id,
194                 iterator->initial_amount, iterator->current_amount);
195     }
196 
197     pthread_mutex_unlock(&list_of_deposits_mutex);
198 }
199 
200 void list_of_deposits_export_bank_amount() {
201     printf("Bank amount: %f\n", bank_amount);
202 }
203 
204 int refill_deposit(int client_sockfd, uint32_t deposit_id, uint32_t added_amount
    ↪ ){
205     pthread_mutex_lock(&list_of_deposits_mutex);
206     Deposit_info* iterator = root;
207 
208     while (iterator != NULL && iterator->deposit_id != deposit_id ) {
209         iterator = iterator->next;
210     }
211 
212     if (iterator == NULL || iterator->client_sockfd != client_sockfd) {
213         pthread_mutex_unlock(&list_of_deposits_mutex);
214         return -1;
215     }
216 
217     iterator->current_amount += added_amount;
218 
219     pthread_mutex_unlock(&list_of_deposits_mutex);
220     return iterator->current_amount;
221 }
222 
223 int add_percents_to_client(int client_socket){
224     pthread_mutex_lock(&list_of_deposits_mutex);
225 
226     int client_has_deposits = -1;
227     Deposit_info* iterator = root;
228 
```

```
229     while (iterator != NULL) {
230         if (iterator->client_sockfd == client_socket){
231             bank_amount +=  iterator->current_amount * 0.05;
232             iterator->current_amount +=  iterator->current_amount * 0.1;
233             client_has_deposits = 1;
234         }
235             iterator = iterator -> next;
236     }
237     pthread_mutex_unlock(&list_of_deposits_mutex);
238     return client_has_deposits;
239 }
```

Листинг 10: packet_handler.c

```
1 #include "main.h"
2 void send_acknowledgment_packet(int client_sockfd, uint16_t ack_type, uint32_t
    ↪ deposit_id, uint32_t amount) {
3     uint32_t packet_length = SIZE_OF_PACKET_LENGTH + SIZE_OF_PACKET_TYPE +
    ↪ SIZE_OF_PACKET_ACK_TYPE + SIZE_OF_ID_DEPOSIT + SIZE_OF_PACKET_AMOUNT;
4     uint16_t packet_type = ACKNOWLEDGMENT_PACKET;
5     void* packet = malloc(packet_length);
6
7     memcpy(packet, &packet_length, SIZE_OF_PACKET_LENGTH);
8     memcpy(packet + SIZE_OF_PACKET_LENGTH, &packet_type, SIZE_OF_PACKET_TYPE);
9     memcpy(packet + SIZE_OF_PACKET_LENGTH + SIZE_OF_PACKET_TYPE, &ack_type,
    ↪ SIZE_OF_PACKET_ACK_TYPE);
10    memcpy(packet + SIZE_OF_PACKET_LENGTH + SIZE_OF_PACKET_TYPE +
    ↪ SIZE_OF_PACKET_ACK_TYPE, &deposit_id, SIZE_OF_ID_DEPOSIT);
11    memcpy(packet + SIZE_OF_PACKET_LENGTH + SIZE_OF_PACKET_TYPE +
    ↪ SIZE_OF_PACKET_ACK_TYPE + SIZE_OF_ID_DEPOSIT,
12            &amount, SIZE_OF_INITIAL_AMOUNT);
13
14    if (write(client_sockfd, packet, packet_length) < packet_length) {
15        printf("ERROR sending acknowledgment packet to client. Client socket
    ↪ number: %d.\n", client_sockfd);
16    }
17
18    free(packet);
19 }
20
21 void send_error_packet(int client_sockfd, char* msg, uint64_t msg_size) {
22    uint32_t packet_length = SIZE_OF_PACKET_LENGTH + SIZE_OF_PACKET_TYPE +
    ↪ msg_size + SIZE_OF_ZERO_CHAR;
23    uint16_t packet_type = ERROR_PACKET;
24    void* packet = malloc(packet_length);
25    char zero_char = '\0';
26
27    memcpy(packet, &packet_length, SIZE_OF_PACKET_LENGTH);
28    memcpy(packet + SIZE_OF_PACKET_LENGTH, &packet_type, SIZE_OF_PACKET_TYPE);
29    memcpy(packet+SIZE_OF_PACKET_LENGTH+SIZE_OF_PACKET_TYPE, msg, msg_size);
30    memcpy(packet+SIZE_OF_PACKET_LENGTH+SIZE_OF_PACKET_TYPE + msg_size, &
    ↪ zero_char, SIZE_OF_ZERO_CHAR);
31
32    if (write(client_sockfd, packet, packet_length) < packet_length ) {
33        printf("ERROR sending error packet to client. Client socket number: %d.\
    ↪ n", client_sockfd);
34    }
35
36    free(packet);
37 }
38
```

```
39 void handle_add_deposit_packet(int client_sockfd, void* packet) {
40     uint32_t initial_amount = *(uint32_t*) (packet + SIZE_OF_PACKET_TYPE);
41     uint32_t deposit_id = generate_deposit_id();
42 
43     list_of_deposits_add(
44             make_new_deposit(deposit_id, initial_amount, client_sockfd)
45             );
46     send_acknowledgment_packet(client_sockfd, DEPOSIT_WAS_ADDED, deposit_id,
    ↪ initial_amount);
47 }
48 
49 void handle_refill_deposit_packet(int client_sockfd, void* packet) {
50     uint32_t deposit_id = *(uint32_t*) (packet + SIZE_OF_PACKET_TYPE);
51     uint32_t amount = *(uint32_t*) (packet + SIZE_OF_PACKET_TYPE +
    ↪ SIZE_OF_ID_DEPOSIT);
52 
53     if (refill_deposit(client_sockfd, deposit_id, amount) == -1 ) {
54         char* error_msg = "ERROR: couldn't refill deposit";
55         send_error_packet(client_sockfd, error_msg, strlen(error_msg));
56     } else {
57         send_acknowledgment_packet(client_sockfd, DEPOSIT_WAS_REFILLED,
    ↪ deposit_id, amount);
58     }
59 }
60 
61 void handle_remove_deposit_packet(int client_sockfd, void* packet) {
62     uint32_t deposit_id = *(uint32_t*) (packet + SIZE_OF_PACKET_TYPE);
63     double final_amount = list_of_deposits_remove(client_sockfd, deposit_id);
64 
65     if (final_amount == -1 ) {
66         char* error_msg = "ERROR: couldn't remove deposit";
67         send_error_packet(client_sockfd, error_msg, strlen(error_msg));
68     } else {
69         send_acknowledgment_packet(client_sockfd, DEPOSIT_WAS_REMOVED,
    ↪ deposit_id, final_amount);
70     }
71 }
72 
73 void send_acknowledgment_about_percents_packet(int sockfd, uint16_t ack_type){
74     uint32_t packet_length = SIZE_OF_PACKET_LENGTH + SIZE_OF_PACKET_TYPE +
    ↪ SIZE_OF_PACKET_ACK_TYPE;
75     uint16_t packet_type = ACKNOWLEDGMENT_PACKET;
76     void* packet = malloc(packet_length);
77     memcpy(packet, &packet_length, SIZE_OF_PACKET_LENGTH);
78     memcpy(packet + SIZE_OF_PACKET_LENGTH, &packet_type, SIZE_OF_PACKET_TYPE);
79     memcpy(packet + SIZE_OF_PACKET_LENGTH + SIZE_OF_PACKET_TYPE, &ack_type,
    ↪ SIZE_OF_PACKET_ACK_TYPE);
80 
81     if (write(sockfd, packet, packet_length) < packet_length){
82         printf("ERROR sending acknowledgment packet about added percents to
    ↪ client. Client socket number: %d\n", sockfd);
83     }
84     free(packet);
85 }
86 
87 void handle_added_percents_packet(int client_sockfd, void* packet){
88     if (add_percents_to_client(client_sockfd) == -1){
89         char* error_msg = "ERROR couldn't add percents to deposits";
90         send_error_packet(client_sockfd, error_msg, strlen(error_msg));
91     } else {
```

```
92          send_acknowledgment_about_percents_packet(client_sockfd, PERCENTS_ADDED)
        ↪ ;
93      }
94  }
```

11.2. UDP

11.2.1. Клиент

Листинг 11: main.h

```
1  #ifndef DEPOSIT_SERVICE_CLIENT_UDP_MAIN_H
2  #define DEPOSIT_SERVICE_CLIENT_UDP_MAIN_H
3
4  #include <stdio.h>
5  #include <stdlib.h>
6  #include <sys/socket.h>
7  #include <strings.h>
8  #include <stdint.h>
9  #include <netdb.h>
10 #include <unistd.h>
11 #include <sys/select.h>
12 #include <string.h>
13
14 #define MAX_PACKET_SIZE 519
15
16 #define SIZE_OF_PACKET_ACK_TYPE 2
17 #define SIZE_OF_PACKET_TYPE 2
18 #define SIZE_OF_PACKET_INDEX 4
19 #define SIZE_OF_PACKET_AMOUNT 4
20 #define SIZE_OF_ID_DEPOSIT 4
21 #define SIZE_OF_ACK_NUMBER 4
22 #define SIZE_OF_REFILL_AMOUNT 4
23 #define SIZE_OF_CURRENT_AMOUNT 4
24 #define SIZE_OF_INITIAL_AMOUNT 4
25 #define SIZE_OF_PACKET_BLOCK_NUMBER 4
26
27 #define ERROR_PACKET 1
28 #define ACKNOWLEDGMENT_PACKET 2
29 #define LIST_OF_DEPOSITS_PACKET 3
30 #define OPEN_DEPOSIT_PACKET 4
31 #define REFILL_DEPOSIT_PACKET 5
32 #define CLOSE_DEPOSIT_PACKET 6
33 #define GET_LIST_OF_DEPOSITS_PACKET 7
34 #define GET_BANK_AMOUNT_PACKET 8
35 #define SHOW_BANK_AMOUNT_PACKET 9
36 #define PLEASE_ADD_PERCENTS_PACKET 10
37
38 #define DEPOSIT_WAS_OPENED 1
39 #define DEPOSIT_WAS_REFILLED 2
40 #define DEPOSIT_WAS_DELETED 3
41 #define PACKET_WAS_RECEIVED 4
42 #define PERCENTS_ADDED 5
43
44 void* create_add_deposit_packet(uint32_t* packet_length, uint32_t index,
        ↪ uint32_t initial_amount);
45 void* create_remove_deposit_packet(uint32_t* packet_size, uint32_t index,
        ↪ uint32_t deposit_id);
```

```
46 void* create_get_list_of_deposits_packet(uint32_t *packet_length, uint32_t index
       ↪ );
47 void* create_acknowledgment_packet(uint32_t* packet_length, uint16_t ack_type,
       ↪ uint32_t index, uint32_t number);
48 void* create_refill_deposit_packet(uint32_t* packet_length, uint32_t index,
       ↪ uint32_t deposit_id, uint32_t amount);
49 void* create_show_bank_amount_packet(uint32_t* packet_size, uint32_t index);
50 void* create_please_add_percents_packet(uint32_t* packet_length, uint32_t index)
       ↪ ;
51
52 #endif
```

Листинг 12: main.c

```
1  #include "main.h"
2  int read_id() {
3      char buffer[sizeof(uint32_t) + 2];
4      fprintf(stdout, "Enter id of deposit: ");
5      fflush(stdout);
6      fgets(buffer, sizeof(uint32_t), stdin);
7      return atoi(buffer);
8  }
9
10 int read_refill_amount() {
11     char buffer[sizeof(uint32_t) + 2];
12     fprintf(stdout, "Enter refilling amount: ");
13     fflush(stdout);
14     fgets(buffer, sizeof(uint32_t) + 2, stdin);
15     return atoi(buffer);
16 }
17
18 int read_amount() {
19     char buffer[sizeof(uint32_t) + 1];
20     fprintf(stdout, "Enter amount of deposit: ");
21     fflush(stdout);
22     fgets(buffer, sizeof(uint32_t), stdin);
23     return atoi(buffer);
24 }
25
26 int get_user_choice() {
27     printf("1) Открыть вклад\n");
28     printf("2) Посмотреть вклады\n");
29     printf("3) Пополнить вклад\n");
30     printf("4) Закрыть вклад\n");
31     printf("5) Завершить работу\n");
32     printf("6) Показать счет банка\n");
33     printf("7) Наличислить проценты по вкладам\n");
34     char buffer[sizeof(int) + 1];
35     fgets(buffer, sizeof(int), stdin);
36     return atoi(buffer);
37 }
38
39 int check_number_of_args(int argc, char *argv[]) {
40     if (argc < 3) {
41         fprintf(stderr, "usage %s hostname port\n", argv[0]);
42         return -1;
43     } else {
44         return 1;
45     }
46 }
47
```

```
48 uint32_t generate_new_index(uint32_t old_index) {
49     // (1 2 4 3)
50     if (old_index == 2) {
51         return 4;
52     } else if (old_index == 4) {
53         return 3;
54     } else {
55         return old_index + 1 > 65535 ? 1 : old_index + 1;
56     }
57 
58     // (1 2 4)
59     return old_index > 1 ? old_index + 2 : old_index + 1;
60 
61     return old_index + 1 > 65535 ? 1 : old_index + 1;
62 }
63 
64 int handle_received_packet(int number_received, const char received_packet[
    ↪ MAX_PACKET_SIZE], uint32_t cur_index,
65                            int deposit_id, uint32_t *expected_block_number) {
66     if ((number_received > 0) {
67         if (*(uint16_t *) received_packet == ACKNOWLEDGMENT_PACKET &&
68             *(uint16_t *) (received_packet + SIZE_OF_PACKET_TYPE +
    ↪ SIZE_OF_PACKET_INDEX) == DEPOSIT_WAS_OPENED) {
69             printf("Вклад открыт, id: %d\n",
70                    *(uint32_t *) (received_packet + SIZE_OF_PACKET_TYPE +
    ↪ SIZE_OF_PACKET_INDEX +
71                                   SIZE_OF_PACKET_ACK_TYPE));
72             return 1;
73         }
74 
75         if (*(uint16_t *) received_packet == LIST_OF_DEPOSITS_PACKET &&
76             (*(uint16_t *) (received_packet + SIZE_OF_PACKET_TYPE +
    ↪ SIZE_OF_PACKET_INDEX) == *expected_block_number ||
77              *(uint16_t *) (received_packet + SIZE_OF_PACKET_TYPE +
    ↪ SIZE_OF_PACKET_INDEX) == 0)) {
78             printf("-------------------\n");
79             printf("id of deposit: %d\n",
80                    *(uint32_t *) (received_packet + SIZE_OF_PACKET_TYPE +
    ↪ SIZE_OF_PACKET_INDEX +
81                                   SIZE_OF_PACKET_BLOCK_NUMBER));
82             printf("accrued amount: %d\n",
83                    *(uint32_t *) (received_packet + SIZE_OF_PACKET_TYPE +
    ↪ SIZE_OF_PACKET_INDEX +
84                                   SIZE_OF_PACKET_BLOCK_NUMBER +
    ↪ SIZE_OF_ID_DEPOSIT));
85             printf("amount: %f\n",
86                    *(double *) (received_packet + SIZE_OF_PACKET_TYPE +
    ↪ SIZE_OF_PACKET_INDEX +
87                                 SIZE_OF_PACKET_BLOCK_NUMBER + SIZE_OF_ID_DEPOSIT
    ↪  +
88                                 SIZE_OF_INITIAL_AMOUNT));
89             *expected_block_number =
90                     *(uint32_t *) (received_packet + SIZE_OF_PACKET_TYPE +
91                                    SIZE_OF_PACKET_INDEX) ==
92                     0 ? 0 : *expected_block_number + 1;
93             return 1;
94         }
95 
96         if (*(uint16_t *) received_packet == ACKNOWLEDGMENT_PACKET &&
97                 *(uint16_t *) (received_packet + SIZE_OF_PACKET_TYPE +
```

```
    ↪ SIZE_OF_PACKET_INDEX) == DEPOSIT_WAS_REFILLED) {
98              printf("You refilled deposit with id %d and amount is %f.\n",
    ↪ deposit_id,
99                     (double) (*(uint32_t *) (received_packet +
    ↪ SIZE_OF_PACKET_TYPE + SIZE_OF_PACKET_INDEX +
100                                             SIZE_OF_PACKET_ACK_TYPE)));
101             return 1;
102         }
103
104         if (*(uint16_t *) received_packet == ACKNOWLEDGMENT_PACKET &&
105             *(uint16_t *) (received_packet + SIZE_OF_PACKET_TYPE +
    ↪ SIZE_OF_PACKET_INDEX) == DEPOSIT_WAS_DELETED) {
106             printf("Вклад удалён, id: %d\n", *(uint32_t *) (received_packet +
    ↪ SIZE_OF_PACKET_TYPE +
107                                                             SIZE_OF_PACKET_INDEX
    ↪  + SIZE_OF_PACKET_ACK_TYPE));
108             return 1;
109
110         }
111
112         if (*(uint16_t *) received_packet == SHOW_BANK_AMOUNT_PACKET) {
113             printf("Счет банка: %f\n", *(double *) (received_packet +
    ↪ SIZE_OF_PACKET_TYPE + SIZE_OF_PACKET_INDEX));
114             return 1;
115         }
116
117         if (*(uint16_t *) received_packet == ACKNOWLEDGMENT_PACKET &&
118                 *(uint16_t *) (received_packet + SIZE_OF_PACKET_TYPE +
    ↪ SIZE_OF_PACKET_INDEX) == PERCENTS_ADDED) {
119             printf("Проценты по вкладам начислены.\n");
120             return 1;
121         }
122
123         if (*(uint16_t *) received_packet == ERROR_PACKET) {
124             printf("ERROR. error message: %s.\n",
125                    received_packet + SIZE_OF_PACKET_TYPE + SIZE_OF_PACKET_INDEX)
    ↪ ;
126             *expected_block_number = 0;
127             return 1;
128         }
129     }
130     return -1;
131 }
132
133 int main(int argc, char *argv[]) {
134     int sockfd;
135     uint16_t portno;
136     struct sockaddr_in serv_addr;
137     struct hostent *server;
138
139     if (check_number_of_args(argc, argv) < 0) {
140         return 0;
141     }
142
143     portno = (uint16_t) atoi(argv[2]);
144     sockfd = socket(AF_INET, SOCK_DGRAM, 0);
145     if (sockfd < 0) {
146         perror("ERROR opening socket");
147         return 0;
148     }
```

```
149
150     server = gethostbyname(argv[1]);
151     if (server == NULL) {
152         fprintf(stderr, "ERROR, no such host\n");
153         return 0;
154     }
155
156     bzero((char *) &serv_addr, sizeof(serv_addr));
157     serv_addr.sin_family = AF_INET;
158     bcopy(server->h_addr, (char *) &serv_addr.sin_addr.s_addr, (size_t) server->
    ↪ h_length);
159     serv_addr.sin_port = htons(portno);
160
161     void *packet;
162     char received_packet[MAX_PACKET_SIZE];
163     uint32_t amount;
164     uint32_t packet_size;
165     struct sockaddr_in received_addr;
166     int received_addr_len;
167     int deposit_id = 0;
168     int refill_amount;
169     uint32_t cur_index = 0;
170     uint32_t expected_block_number;
171     int number_received;
172     int result;
173     fd_set inputs;
174     struct timeval timeout;
175     int i = 0;
176
177     while (1) {
178         switch (get_user_choice()) {
179             case 1:
180                 if ((amount = read_amount()) < 0) {
181                     printf("ERROR: invalid value\n");
182                     continue;
183                 }
184
185                 cur_index = generate_new_index(cur_index);
186                 printf("index: %d\n", cur_index);
187                 packet = create_add_deposit_packet(&packet_size, cur_index,
    ↪ amount);
188                 FD_ZERO(&inputs);
189                 FD_SET(sockfd, &inputs);
190
191                 for (i = 0; i < 3; ++i) {
192                     sendto(sockfd, packet, packet_size, MSG_WAITALL, (const
    ↪ struct sockaddr *) &serv_addr,
193                             sizeof(serv_addr));
194                     timeout.tv_sec = 5;
195                     timeout.tv_usec = 0;
196                     result = select(FD_SETSIZE, &inputs, NULL, NULL, &timeout);
197
198                     if (result == 0) {
199                         continue;
200                     }
201
202                     bzero(&received_addr, sizeof(received_addr));
203                     number_received = recvfrom(sockfd, received_packet,
    ↪ MAX_PACKET_SIZE, MSG_WAITALL,
204                                                 (struct sockaddr *) &
```

```
    ↪ received_addr,
205                                                 (socklen_t * ) &
    ↪ received_addr_len);
206                     expected_block_number = 0;
207                     if (handle_received_packet(number_received, received_packet,
    ↪  cur_index, deposit_id,
208                                                 &expected_block_number) == 1) {
209                         break;
210                     }
211                 }
212
213                 if (i == 3) {
214                     fprintf(stdout, "Server is not responding\n");
215                 }
216                 free(packet);
217                 break;
218
219             case 2:
220                 cur_index = generate_new_index(cur_index);
221                 printf("index: %d\n", cur_index);
222                 expected_block_number = 1;
223
224                 while (expected_block_number != 0) {
225                     if (expected_block_number == 1) {
226                         packet = create_get_list_of_deposits_packet(&packet_size
    ↪ , cur_index);
227                     } else {
228                         packet = create_acknowledgment_packet(&packet_size,
    ↪ PACKET_WAS_RECEIVED, cur_index,
229
    ↪ expected_block_number - 1);
230                     }
231
232                     FD_ZERO(&inputs);
233                     FD_SET(sockfd, &inputs);
234
235                     for (i = 0; i < 3; ++i) {
236                         sendto(sockfd, packet, packet_size, MSG_WAITALL,
237                                (const struct sockaddr *) &serv_addr, sizeof(
    ↪ serv_addr));
238                         timeout.tv_sec = 5;
239                         timeout.tv_usec = 0;
240                         result = select(FD_SETSIZE, &inputs, NULL, NULL, &
    ↪ timeout);
241
242                         if (result == 0) {
243                             continue;
244                         }
245
246                         bzero(&received_addr, sizeof(received_addr));
247                         number_received = recvfrom(sockfd, received_packet,
    ↪ MAX_PACKET_SIZE, MSG_WAITALL,
248                                                    (struct sockaddr *) &
    ↪ received_addr,
249                                                    (socklen_t *) &
    ↪ received_addr_len);
250
251                         if (handle_received_packet(number_received,
    ↪ received_packet, cur_index, deposit_id,
252                                                    &expected_block_number) == 1) {
```

```
253                         break;
254                     }
255                 }
256
257                 if (i == 3) {
258                     fprintf(stdout, "Server is not responding\n");
259                     free(packet);
260                     break;
261                 }
262             }
263             break;
264
265         case 3:
266             if ((deposit_id = read_id()) < 0) {
267                 printf("ERROR: invalid value\n");
268                 continue;
269             }
270
271             if ((refill_amount = read_refill_amount()) < 0) {
272                 printf("ERROR: invalid value\n");
273                 continue;
274             }
275
276             cur_index = generate_new_index(cur_index);
277             printf("index: %d\n", cur_index);
278             packet = create_refill_deposit_packet(&packet_size, cur_index,
    ↪ deposit_id, refill_amount);
279             FD_ZERO(&inputs);
280             FD_SET(sockfd, &inputs);
281
282             for (i = 0; i < 3; ++i) {
283                 sendto(sockfd, packet, packet_size, MSG_WAITALL, (const
    ↪ struct sockaddr *) &serv_addr,
284                        sizeof(serv_addr));
285                 timeout.tv_sec = 5;
286                 timeout.tv_usec = 0;
287                 result = select(FD_SETSIZE, &inputs, NULL, NULL, &timeout);
288
289                 if (result == 0) {
290                     continue;
291                 }
292
293                 bzero(&received_addr, sizeof(received_addr));
294                 number_received = recvfrom(sockfd, received_packet,
    ↪ MAX_PACKET_SIZE, MSG_WAITALL,
295                                            (struct sockaddr *) &
    ↪ received_addr,
296                                            (socklen_t *) &received_addr_len)
    ↪ ;
297                 expected_block_number = 0;
298                 if (handle_received_packet(number_received, received_packet,
    ↪  cur_index, deposit_id,
299                                            &expected_block_number) == 1) {
300                     break;
301                 }
302             }
303
304             if (i == 3) {
305                 fprintf(stdout, "Server is not responding\n");
306             }
```

```
307                 free(packet);
308                 break;
309
310             case 4:
311                 if ((deposit_id = read_id()) < 0) {
312                     printf("ERROR: invalid value\n");
313                     continue;
314                 }
315
316                 cur_index = generate_new_index(cur_index);
317                 printf("index: %d\n", cur_index);
318                 packet = create_remove_deposit_packet(&packet_size, cur_index,
    ↪ deposit_id);
319                 FD_ZERO(&inputs);
320                 FD_SET(sockfd, &inputs);
321
322                 for (i = 0; i < 3; ++i) {
323                     sendto(sockfd, packet, packet_size, MSG_WAITALL, (const
    ↪ struct sockaddr *) &serv_addr,
324                            sizeof(serv_addr));
325                     timeout.tv_sec = 5;
326                     timeout.tv_usec = 0;
327                     result = select(FD_SETSIZE, &inputs, NULL, NULL, &timeout);
328
329                     if (result == 0) {
330                         continue;
331                     }
332
333                     bzero(&received_addr, sizeof(received_addr));
334                     number_received = recvfrom(sockfd, received_packet,
    ↪ MAX_PACKET_SIZE, MSG_WAITALL,
335                                                (struct sockaddr *) &
    ↪ received_addr,
336                                                (socklen_t *) &received_addr_len)
    ↪ ;
337                     expected_block_number = 0;
338                     if (handle_received_packet(number_received, received_packet,
    ↪  cur_index, deposit_id,
339                                                &expected_block_number) == 1) {
340                         break;
341                     }
342                 }
343
344                 if (i == 3) {
345                     fprintf(stdout, "Server is not responding\n");
346                 }
347                 free(packet);
348                 break;
349
350             case 5:
351                 shutdown(sockfd, SHUT_RDWR);
352                 close(sockfd);
353                 return 0;
354
355             case 6:
356                 cur_index = generate_new_index(cur_index);
357                 printf("index: %d\n", cur_index);
358                 packet = create_show_bank_amount_packet(&packet_size, cur_index)
    ↪ ;
359                 FD_ZERO(&inputs);
```

```
360                 FD_SET(sockfd, &inputs);
361
362                 for (i = 0; i < 3; ++i) {
363                     sendto(sockfd, packet, packet_size, MSG_WAITALL, (const
    ↪ struct sockaddr *) &serv_addr,
364                            sizeof(serv_addr));
365                     timeout.tv_sec = 5;
366                     timeout.tv_usec = 0;
367                     result = select(FD_SETSIZE, &inputs, NULL, NULL, &timeout);
368
369                     if (result == 0) {
370                         continue;
371                     }
372
373                     bzero(&received_addr, sizeof(received_addr));
374                     number_received = recvfrom(sockfd, received_packet,
    ↪ MAX_PACKET_SIZE, MSG_WAITALL,
375                                                (struct sockaddr *) &
    ↪ received_addr,
376                                                (socklen_t *) &received_addr_len)
    ↪ ;
377                     expected_block_number = 0;
378                     if (handle_received_packet(number_received, received_packet,
    ↪  cur_index, deposit_id,
379                                                &expected_block_number) == 1) {
380                         break;
381                     }
382                 }
383
384                 if (i == 3) {
385                     fprintf(stdout, "Server is not responding\n");
386                 }
387                 free(packet);
388                 break;
389
390             case 7:
391                 cur_index = generate_new_index(cur_index);
392                 printf("index: %d\n", cur_index);
393                 packet = create_please_add_percents_packet(&packet_size,
    ↪ cur_index);
394                 FD_ZERO(&inputs);
395                 FD_SET(sockfd, &inputs);
396
397                 for (i = 0; i < 3; ++i) {
398                     sendto(sockfd, packet, packet_size, MSG_WAITALL, (const
    ↪ struct sockaddr *) &serv_addr,
399                            sizeof(serv_addr));
400                     timeout.tv_sec = 5;
401                     timeout.tv_usec = 0;
402                     result = select(FD_SETSIZE, &inputs, NULL, NULL, &timeout);
403
404                     if (result == 0) {
405                         continue;
406                     }
407
408                     bzero(&received_addr, sizeof(received_addr));
409                     number_received = recvfrom(sockfd, received_packet,
    ↪ MAX_PACKET_SIZE, MSG_WAITALL,
410                                                (struct sockaddr *) &
    ↪ received_addr,
```

```
411                                                 (socklen_t *) &received_addr_len)
    ↪ ;
412                         expected_block_number = 0;
413                         if (handle_received_packet(number_received, received_packet,
    ↪  cur_index, deposit_id,
414                                                    &expected_block_number) == 1) {
415                             break;
416                         }
417                     }
418
419                     if (i == 3) {
420                         fprintf(stdout, "Server is not responding\n");
421                     }
422                     free(packet);
423                     break;
424
425                 default:
426                     printf("ERROR: wrong choice\n");
427                     break;
428             }
429         }
430 }
```

Листинг 13: packet_creator.c

```
1 #include "main.h"
2 void* create_add_deposit_packet(uint32_t* packet_length, uint32_t index,
    ↪ uint32_t initial_amount) {
3     *packet_length = SIZE_OF_PACKET_TYPE + SIZE_OF_PACKET_INDEX +
    ↪ SIZE_OF_PACKET_AMOUNT;
4     void* packet = malloc(*packet_length);
5     uint16_t packet_type = OPEN_DEPOSIT_PACKET;
6     memcpy(packet, &packet_type, SIZE_OF_PACKET_TYPE);
7     memcpy(packet + SIZE_OF_PACKET_TYPE, &index, SIZE_OF_PACKET_INDEX);
8     memcpy(packet + SIZE_OF_PACKET_TYPE + SIZE_OF_PACKET_INDEX, &initial_amount,
    ↪  SIZE_OF_INITIAL_AMOUNT);
9     return packet;
10 }
11
12 void* create_remove_deposit_packet(uint32_t* packet_size, uint32_t index,
    ↪ uint32_t deposit_id) {
13     *packet_size = SIZE_OF_PACKET_TYPE + SIZE_OF_PACKET_INDEX +
    ↪ SIZE_OF_ID_DEPOSIT;
14     void* packet = malloc(*packet_size);
15     uint16_t packet_type = CLOSE_DEPOSIT_PACKET;
16     memcpy(packet, &packet_type, SIZE_OF_PACKET_TYPE);
17     memcpy(packet + SIZE_OF_PACKET_TYPE, &index, SIZE_OF_PACKET_INDEX);
18     memcpy(packet + SIZE_OF_PACKET_TYPE + SIZE_OF_PACKET_INDEX, &deposit_id,
    ↪ SIZE_OF_ID_DEPOSIT);
19     return packet;
20 }
21
22 void* create_show_bank_amount_packet(uint32_t* packet_size, uint32_t index){
23     *packet_size = SIZE_OF_PACKET_TYPE + SIZE_OF_PACKET_INDEX;
24     void* packet = malloc(*packet_size);
25     uint16_t packet_type = GET_BANK_AMOUNT_PACKET;
26     memcpy(packet, &packet_type, SIZE_OF_PACKET_TYPE);
27     memcpy(packet + SIZE_OF_PACKET_TYPE, &index, SIZE_OF_PACKET_INDEX);
28     return packet;
29 }
30
```

```
31 void* create_get_list_of_deposits_packet(uint32_t * packet_length, uint32_t
       ↪ index) {
32     *packet_length = SIZE_OF_PACKET_TYPE + SIZE_OF_PACKET_INDEX;
33     void* packet = malloc(*packet_length);
34     uint16_t packet_type = GET_LIST_OF_DEPOSITS_PACKET;
35     memcpy(packet, &packet_type, SIZE_OF_PACKET_TYPE);
36     memcpy(packet + SIZE_OF_PACKET_TYPE, &index, SIZE_OF_PACKET_INDEX);
37     return packet;
38 }
39
40 void* create_acknowledgment_packet(uint32_t* packet_length, uint16_t ack_type,
       ↪ uint32_t index, uint32_t number) {
41     *packet_length = SIZE_OF_PACKET_TYPE + SIZE_OF_PACKET_INDEX +
       ↪ SIZE_OF_PACKET_ACK_TYPE + SIZE_OF_ACK_NUMBER;
42     void* packet = malloc(*packet_length);
43     uint16_t packet_type = ACKNOWLEDGMENT_PACKET;
44     memcpy(packet, &packet_type, SIZE_OF_PACKET_TYPE);
45     memcpy(packet + SIZE_OF_PACKET_TYPE, &index, SIZE_OF_PACKET_INDEX);
46     memcpy(packet + SIZE_OF_PACKET_TYPE + SIZE_OF_PACKET_INDEX, &ack_type,
       ↪ SIZE_OF_PACKET_ACK_TYPE);
47     memcpy(packet + SIZE_OF_PACKET_TYPE + SIZE_OF_PACKET_INDEX +
       ↪ SIZE_OF_PACKET_ACK_TYPE, &number, SIZE_OF_ACK_NUMBER);
48     return packet;
49 }
50
51 void* create_refill_deposit_packet(uint32_t* packet_length, uint32_t index,
       ↪ uint32_t deposit_id, uint32_t amount) {
52     *packet_length = SIZE_OF_PACKET_TYPE + SIZE_OF_PACKET_INDEX +
       ↪ SIZE_OF_ID_DEPOSIT + SIZE_OF_PACKET_AMOUNT;
53     void* packet = malloc(*packet_length);
54     uint16_t packet_type = REFILL_DEPOSIT_PACKET;
55     memcpy(packet, &packet_type, SIZE_OF_PACKET_TYPE);
56     memcpy(packet + SIZE_OF_PACKET_TYPE, &index, SIZE_OF_PACKET_INDEX);
57     memcpy(packet + SIZE_OF_PACKET_TYPE + SIZE_OF_PACKET_INDEX, &deposit_id,
       ↪ SIZE_OF_ID_DEPOSIT);
58     memcpy(packet + SIZE_OF_PACKET_TYPE + SIZE_OF_PACKET_INDEX +
       ↪ SIZE_OF_ID_DEPOSIT, &amount, SIZE_OF_PACKET_AMOUNT);
59     return packet;
60 }
61
62 void* create_please_add_percents_packet(uint32_t* packet_length, uint32_t index)
       ↪ {
63     *packet_length = SIZE_OF_PACKET_TYPE + SIZE_OF_PACKET_INDEX;
64     void* packet = malloc(*packet_length);
65     uint16_t packet_type = PLEASE_ADD_PERCENTS_PACKET;
66     memcpy(packet, &packet_type, SIZE_OF_PACKET_TYPE);
67     memcpy(packet + SIZE_OF_PACKET_TYPE, &index, SIZE_OF_PACKET_INDEX);
68     return packet;
69 }
```

11.2.2. Сервер

Листинг 14: main.h

```
1 #ifndef SERVER_MAIN_H
2 #define SERVER_MAIN_H
3
4 #define MAX_PACKET_SIZE 516
5
```

```c
#include <arpa/inet.h>
#include <netinet/in.h>
#include <pthread.h>
#include <stddef.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/socket.h>
#include <sys/types.h>
#include <unistd.h>

typedef struct {
  int port;
  int *initial_sockfd;
} Receiving_thread_input;

typedef struct {
  struct sockaddr_in cliaddr;
  int sockfd;
  int cliaddr_len;
  void *packet;
} Listening_thread_input;

typedef struct deposit_info {
  uint32_t deposit_id;
  double current_amount;
  int client_sockfd;
  uint32_t initial_amount;
  struct sockaddr_in cliaddr;
  struct deposit_info *next;
} Deposit_info;

typedef struct User_info {
  char *address;
  int port;
  int sockfd;
  pthread_t client_thread;
  struct sockaddr_in cliaddr;
  uint32_t required_index;
  void* last_answer;
  uint32_t packet_size;
  struct User_info *next;
} User_info;

#define DEPOSIT_WAS_OPENED 1
#define DEPOSIT_WAS_REFILLED 2
#define DEPOSIT_WAS_DELETED 3
#define PACKET_WAS_RECEIVED 4
#define PERCENTS_ADDED 5

#define ERROR_PACKET 1
#define ACKNOWLEDGMENT_PACKET 2
#define LIST_OF_DEPOSITS_PACKET 3
#define OPEN_DEPOSIT_PACKET 4
#define REFILL_DEPOSIT_PACKET 5
#define CLOSE_DEPOSIT_PACKET 6
#define GET_LIST_OF_DEPOSITS_PACKET 7
#define GET_BANK_AMOUNT_PACKET 8
#define SHOW_BANK_AMOUNT_PACKET 9
#define PLEASE_ADD_PERCENTS_PACKET 10
```

```
66
67 #define SIZE_OF_PACKET_ACK_TYPE 2
68 #define SIZE_OF_PACKET_TYPE 2
69 #define SIZE_OF_PACKET_AMOUNT 4
70 #define SIZE_OF_PACKET_INDEX 4
71 #define SIZE_OF_ID_DEPOSIT 4
72 #define SIZE_OF_ACK_NUMBER 4
73 #define SIZE_OF_ERR_TYPE 2
74 #define SIZE_OF_REFILL_AMOUNT 4
75 #define SIZE_OF_CURRENT_AMOUNT 4
76 #define SIZE_OF_INITIAL_AMOUNT 4
77 #define SIZE_OF_PACKET_BLOCK_NUMBER 4
78
79 void list_of_clients_add(User_info *new_client);
80 User_info *make_new_client(int sockfd, int port, char *address, pthread_t
    ↪ client_thread);
81 int list_of_clients_remove(int sockfd);
82 void list_of_clients_remove_all();
83 void init_list_of_clients_mutex();
84 pthread_t create_user_listening_thread(int *initial_socket);
85 void list_of_clients_export(FILE *dst_fd);
86 uint32_t list_of_clients_get_client_index(int port, char *address);
87 void list_of_clients_set_client_index(int port, char *address, uint32_t index);
88 void list_of_clients_set_required_index(int port, char* address);
89 void list_of_clients_set_last_answer(int port, char* address, void *answer,
    ↪ uint32_t packet_size);
90 void* list_of_clients_get_last_answer(int port, char* address, uint32_t *
    ↪ packet_size);
91 uint32_t list_of_clients_get_required_index(int port, char* address);
92 void list_of_deposits_remove_all();
93 void init_list_of_deposits_mutex();
94 double list_of_deposits_refill_deposit(int port, const char *address, uint32_t
    ↪ deposit_id, uint32_t added_amount);
95 int32_t list_of_deposits_remove(int port, const char *address, uint32_t
    ↪ deposit_id);
96 void list_of_deposits_send(int client_sockfd);
97 void list_of_deposits_add(Deposit_info *new_deposit);
98 void list_of_deposits_add_percents(void);
99 uint32_t generate_deposit_id(void);
100 void list_of_deposits_export_bank_amount();
101 void list_of_deposits_all_deposits(FILE *output_file);
102 Deposit_info *make_new_deposit(int deposit_id, int initial_amount, int
    ↪ client_sockfd, struct sockaddr_in cliaddr);
103 Deposit_info* list_of_deposits_get_deposit(int port, const char *address, int
    ↪ number, int *is_deposit_last);
104 void *create_add_deposit_packet(uint32_t *packet_length, uint32_t index,
    ↪ uint32_t initial_amount);
105 void *create_remove_deposit_packet(uint32_t *packet_size, uint32_t index,
    ↪ uint32_t deposit_id);
106 void *create_get_list_of_deposits_packet(uint32_t index);
107 void *create_acknowledgment_packet(uint32_t *packet_size, uint32_t index,
    ↪ uint16_t ack_type, uint32_t number);
108 void *create_error_packet(uint32_t *packet_length, uint32_t index, char *
    ↪ err_text, uint64_t msg_size);
109 void *create_refill_deposit_packet(uint32_t *packet_length, uint32_t index,
    ↪ uint32_t deposit_id, uint32_t amount);
110 void *create_list_of_deposit_packet(uint32_t *packet_length, uint32_t index,
    ↪ uint32_t deposit_id, uint32_t block_number,
111                                     uint32_t initial_amount, double
    ↪ current_amount);
```

```
112 double list_of_deposits_get_bank_amount(void);
113 pthread_t create_receiving_thread(int port, int *initial_sockfd);
114 pthread_t create_listening_thread(int sockfd, void *packet, struct sockaddr_in
    ↪ cliaddr, int cliaddr_len);
115 void* create_show_bank_amount_packet(uint32_t* packet_length, uint32_t index,
    ↪ double bank_amount);
116 int add_percents_to_client(int port, const char *address);
117 int remove_client_deposits(int port, const char *address);
118
119 #endif
```

Листинг 15: main.c

```
 1 #include "main.h"
 2 int checkArguments(int argc, char* argv[]) {
 3     if (argc != 2) {
 4         fprintf(stderr, "usage %s port \n", argv[0]);
 5         return -1;
 6     }
 7     return 1;
 8 }
 9
10 int main(int argc, char* argv[]) {
11     struct sockaddr_in cliaddr;
12     int sockfd;
13     int cliaddr_len;
14     struct sockaddr_in servaddr;
15
16     if (checkArguments(argc, argv) < 0) {
17         return 0;
18     }
19     const uint16_t port = (uint16_t) atoi(argv[1]);
20
21     if ((sockfd = socket(AF_INET, SOCK_DGRAM, 0)) < 0) {
22         printf("ERROR: socket creation failed.\n");
23         return 0;
24     }
25
26     memset(&servaddr, 0, sizeof(servaddr));
27     servaddr.sin_family = AF_INET;
28     servaddr.sin_port = htons(port);
29     servaddr.sin_addr.s_addr = INADDR_ANY;
30
31     if (setsockopt(sockfd, SOL_SOCKET, SO_REUSEADDR, &(int){ 1 }, sizeof(int)) <
    ↪ 0) { //
32         fprintf(stderr, "ERROR: setsockopt(SO_REUSEADDR) failed");
33     }
34
35     if (bind(sockfd, (const struct sockaddr *)&servaddr, sizeof(servaddr)) < 0)
    ↪ {
36         printf("ERROR: bind failed");
37         return 0;
38     }
39
40     init_list_of_clients_mutex();
41     init_list_of_deposits_mutex();
42
43     pthread_t receiving_thread = create_receiving_thread(port, &sockfd);
44     pthread_t user_listening_thread = create_user_listening_thread(&sockfd);
45     pthread_join(receiving_thread, NULL);
46     pthread_join(user_listening_thread, NULL);
```

```
47
48      list_of_clients_remove_all();
49      list_of_deposits_remove_all();
50  }
```

Листинг 16: client_thread.c

```
1  #include "main.h"
2  void *socket_listening_thread(void *arg) {
3
4      void *packet = ((Listening_thread_input *) arg)->packet;
5      int sockfd = ((Listening_thread_input *) arg)->sockfd;
6      struct sockaddr_in cliaddr = ((Listening_thread_input *) arg)->cliaddr;
7      int cliaddr_len = ((Listening_thread_input *) arg)->cliaddr_len;
8      uint32_t deposit_id;
9      uint32_t packet_size;
10     void *answer_packet;
11     uint32_t amount;
12     Deposit_info *deposit;
13     int is_deposit_last;
14
15     switch (*(uint16_t *) packet) {
16         case OPEN_DEPOSIT_PACKET:
17             if (list_of_clients_get_required_index(cliaddr.sin_port, inet_ntoa(
       ↪ cliaddr.sin_addr)) != 0 &&
18                 ((*(uint32_t *) ((char *) packet + SIZE_OF_PACKET_TYPE) ==
19                   list_of_clients_get_required_index(cliaddr.sin_port, inet_ntoa
       ↪ (cliaddr.sin_addr)) - 1))) {
20
21                 answer_packet = list_of_clients_get_last_answer(cliaddr.sin_port
       ↪ , inet_ntoa(cliaddr.sin_addr),
22                                                                 &packet_size);
23                 if (answer_packet != NULL) {
24                     sendto(sockfd, answer_packet, packet_size, MSG_WAITALL, (
       ↪ const struct sockaddr *) &cliaddr,
25                            cliaddr_len);
26                     printf("Отправлен пакет с индексом: %d\n",
27                            *(uint32_t *) ((char *) answer_packet +
       ↪ SIZE_OF_PACKET_TYPE));
28                 }
29             }
30
31             if (*(uint32_t *) ((char *) packet + SIZE_OF_PACKET_TYPE) ==
32                 list_of_clients_get_required_index(cliaddr.sin_port, inet_ntoa(
       ↪ cliaddr.sin_addr))) {
33
34                 if ((*(uint32_t *) ((char *) packet + SIZE_OF_PACKET_TYPE)) > 2)
       ↪  {
35                     answer_packet = list_of_clients_get_last_answer(cliaddr.
       ↪ sin_port, inet_ntoa(cliaddr.sin_addr),
36                                                                     &packet_size
       ↪ );
37                     if (answer_packet != NULL) {
38                         sendto(sockfd, answer_packet, packet_size, MSG_WAITALL,
       ↪ (const struct sockaddr *) &cliaddr,
39                                cliaddr_len);
40                         printf("Отправлен пакет с индексом: %d\n",
41                                *(uint32_t *) ((char *) answer_packet +
       ↪ SIZE_OF_PACKET_TYPE));
42                     }
43                 } else {
```

```
44                      deposit_id = generate_deposit_id();
45                      list_of_deposits_add(make_new_deposit(deposit_id,
46                                                            *(uint32_t *) ((char
    ↪ *) packet + SIZE_OF_PACKET_TYPE +
47
    ↪ SIZE_OF_PACKET_INDEX), sockfd,
48                                                            cliaddr));
49                      answer_packet = create_acknowledgment_packet(&packet_size,
50                                                                   *(uint32_t *)
    ↪ ((char *) packet + SIZE_OF_PACKET_TYPE),
51
    ↪ DEPOSIT_WAS_OPENED, deposit_id);
52
53                      list_of_clients_set_last_answer(cliaddr.sin_port, inet_ntoa(
    ↪ cliaddr.sin_addr), answer_packet,
54                                                      packet_size);
55                      list_of_clients_set_required_index(cliaddr.sin_port,
    ↪ inet_ntoa(cliaddr.sin_addr));
56                      if (answer_packet != NULL) {
57                          sendto(sockfd, answer_packet, packet_size, MSG_WAITALL,
    ↪ (const struct sockaddr *) &cliaddr,
58                                 cliaddr_len);
59                          printf("Отправлен пакет с индексом: %d\n",
60                                 *(uint32_t *) ((char *) answer_packet +
    ↪ SIZE_OF_PACKET_TYPE));
61                      }
62                  }
63              }
64              break;
65
66          case CLOSE_DEPOSIT_PACKET:
67              if (list_of_clients_get_required_index(cliaddr.sin_port, inet_ntoa(
    ↪ cliaddr.sin_addr)) != 0 &&
68                  *(uint32_t *) ((char *) packet + SIZE_OF_PACKET_TYPE) ==
69                  list_of_clients_get_required_index(cliaddr.sin_port, inet_ntoa(
    ↪ cliaddr.sin_addr)) - 1) {
70
71                  answer_packet = list_of_clients_get_last_answer(cliaddr.sin_port
    ↪ , inet_ntoa(cliaddr.sin_addr),
72                                                                  &packet_size);
73                  if (answer_packet != NULL) {
74                      sendto(sockfd, answer_packet, packet_size, MSG_WAITALL, (
    ↪ const struct sockaddr *) &cliaddr,
75                             cliaddr_len);
76                      printf("Отправлен пакет с индексом: %d\n",
77                             *(uint32_t *) ((char *) answer_packet +
    ↪ SIZE_OF_PACKET_TYPE));
78                  }
79              }
80
81              if (*(uint32_t *) ((char *) packet + SIZE_OF_PACKET_TYPE) ==
82                  list_of_clients_get_required_index(cliaddr.sin_port, inet_ntoa(
    ↪ cliaddr.sin_addr))) {
83
84                  if ((*(uint32_t *) ((char *) packet + SIZE_OF_PACKET_TYPE)) > 2)
    ↪  {
85                      answer_packet = list_of_clients_get_last_answer(cliaddr.
    ↪ sin_port, inet_ntoa(cliaddr.sin_addr),
86                                                                      &packet_size
    ↪ );
```

```
87                      if (answer_packet != NULL) {
88                          sendto(sockfd, answer_packet, packet_size, MSG_WAITALL,
    ↪ (const struct sockaddr *) &cliaddr,
89                                 cliaddr_len);
90                          printf("Отправлен пакет с индексом: %d\n",
91                                 *(uint32_t *) ((char *) answer_packet +
    ↪ SIZE_OF_PACKET_TYPE));
92                      }
93                  } else {
94                      deposit_id = *(uint32_t *) ((char *) packet +
    ↪ SIZE_OF_PACKET_TYPE + SIZE_OF_PACKET_INDEX);
95                      double id = list_of_deposits_remove(cliaddr.sin_port,
    ↪ inet_ntoa(cliaddr.sin_addr), deposit_id);
96
97                      if (id < 0) {
98                          char *error_msg = "ERROR: couldn't close deposit";
99                          answer_packet = create_error_packet(&packet_size,
100                                                              *(uint32_t *) ((char
    ↪  *) packet + SIZE_OF_PACKET_TYPE),
101                                                             error_msg, strlen(
    ↪ error_msg));
102                     } else {
103                         answer_packet = create_acknowledgment_packet(&
    ↪ packet_size,
104                                                                      *(uint32_t
    ↪ *) ((char *) packet +
105
    ↪     SIZE_OF_PACKET_TYPE),
106
    ↪ DEPOSIT_WAS_DELETED, id);
107                     }
108
109                     list_of_clients_set_last_answer(cliaddr.sin_port, inet_ntoa(
    ↪ cliaddr.sin_addr), answer_packet,
110                                                     packet_size);
111                     list_of_clients_set_required_index(cliaddr.sin_port,
    ↪ inet_ntoa(cliaddr.sin_addr));
112                     if (answer_packet != NULL) {
113                         sendto(sockfd, answer_packet, packet_size, MSG_WAITALL,
    ↪ (const struct sockaddr *) &cliaddr,
114                                cliaddr_len);
115                         printf("Отправлен пакет с индексом: %d\n",
116                                *(uint32_t *) ((char *) answer_packet +
    ↪ SIZE_OF_PACKET_TYPE));
117                     }
118                 }
119             }
120             break;
121
122         case GET_LIST_OF_DEPOSITS_PACKET:
123             if (list_of_clients_get_required_index(cliaddr.sin_port, inet_ntoa(
    ↪ cliaddr.sin_addr)) != 0 &&
124                 *(uint32_t *) ((char *) packet + SIZE_OF_PACKET_TYPE) ==
125                 list_of_clients_get_required_index(cliaddr.sin_port, inet_ntoa(
    ↪ cliaddr.sin_addr)) - 1) {
126
127                 answer_packet = list_of_clients_get_last_answer(cliaddr.sin_port
    ↪ , inet_ntoa(cliaddr.sin_addr),
128                                                                 &packet_size);
129                 if (answer_packet != NULL) {
```

```
130                     sendto(sockfd, answer_packet, packet_size, MSG_WAITALL, (
    ↪ const struct sockaddr *) &cliaddr,
131                             cliaddr_len);
132                     printf("Отправлен пакет с индексом: %d\n",
133                             *(uint32_t *) ((char *) answer_packet +
    ↪ SIZE_OF_PACKET_TYPE));
134                 }
135             }
136 
137             if (*(uint32_t *) ((char *) packet + SIZE_OF_PACKET_TYPE) ==
138                 list_of_clients_get_required_index(cliaddr.sin_port, inet_ntoa(
    ↪ cliaddr.sin_addr))) {
139 
140                 if ((*(uint32_t *) ((char *) packet + SIZE_OF_PACKET_TYPE)) > 2)
    ↪  {
141                     answer_packet = list_of_clients_get_last_answer(cliaddr.
    ↪ sin_port, inet_ntoa(cliaddr.sin_addr),
142                                                                     &packet_size
    ↪ );
143                     if (answer_packet != NULL) {
144                         sendto(sockfd, answer_packet, packet_size, MSG_WAITALL,
    ↪ (const struct sockaddr *) &cliaddr,
145                                cliaddr_len);
146                         printf("Отправлен пакет с индексом: %d\n",
147                                *(uint32_t *) ((char *) answer_packet +
    ↪ SIZE_OF_PACKET_TYPE));
148                     }
149                 } else {
150                     if ((deposit = list_of_deposits_get_deposit(cliaddr.sin_port
    ↪ , inet_ntoa(cliaddr.sin_addr), 1,
151                                                                 &is_deposit_last
    ↪ )) == NULL) {
152 
153                         char *error_msg = "ERROR: you have no deposit opened";
154                         answer_packet = create_error_packet(&packet_size,
155                                                             *(uint32_t *) ((char
    ↪  *) packet + SIZE_OF_PACKET_TYPE),
156                                                             error_msg, strlen(
    ↪ error_msg));
157 
158                     } else {
159                         answer_packet = create_list_of_deposit_packet(
160                                 &packet_size,
161                                 *(uint32_t *) (packet + SIZE_OF_PACKET_TYPE),
162                                 deposit->deposit_id,
163                                 is_deposit_last == 1 ? 0 : 1,
164                                 deposit->initial_amount,
165                                 deposit->current_amount);
166                     }
167 
168                     list_of_clients_set_last_answer(cliaddr.sin_port, inet_ntoa(
    ↪ cliaddr.sin_addr), answer_packet,
169                                                     packet_size);
170                     if (is_deposit_last == 1) {
171                         list_of_clients_set_required_index(cliaddr.sin_port,
    ↪ inet_ntoa(cliaddr.sin_addr));
172                     }
173                     if (answer_packet != NULL) {
174                         sendto(sockfd, answer_packet, packet_size, MSG_WAITALL,
    ↪ (const struct sockaddr *) &cliaddr,
```

```
175                                 cliaddr_len);
176                         printf("Отправлен пакет с индексом: %d\n",
177                                 *(uint32_t *) ((char *) answer_packet +
    ↪ SIZE_OF_PACKET_TYPE));
178                     }
179                 }
180             }
181             break;
182
183         case ACKNOWLEDGMENT_PACKET:
184             if (list_of_clients_get_required_index(cliaddr.sin_port, inet_ntoa(
    ↪ cliaddr.sin_addr)) != 0 &&
185                 *(uint32_t *) ((char *) packet + SIZE_OF_PACKET_TYPE) ==
186                 list_of_clients_get_required_index(cliaddr.sin_port, inet_ntoa(
    ↪ cliaddr.sin_addr)) - 1) {
187                 answer_packet = list_of_clients_get_last_answer(cliaddr.sin_port
    ↪ , inet_ntoa(cliaddr.sin_addr),
188                                                                 &packet_size);
189                 if (answer_packet != NULL) {
190                     sendto(sockfd, answer_packet, packet_size, MSG_WAITALL, (
    ↪ const struct sockaddr *) &cliaddr,
191                            cliaddr_len);
192                     printf("Отправлен пакет с индексом: %d\n",
193                             *(uint32_t *) ((char *) answer_packet +
    ↪ SIZE_OF_PACKET_TYPE));
194                 }
195             }
196
197             if (*(uint32_t *) ((char *) packet + SIZE_OF_PACKET_TYPE) ==
198                 list_of_clients_get_required_index(cliaddr.sin_port, inet_ntoa(
    ↪ cliaddr.sin_addr))) {
199
200                 if ((*(uint32_t *) ((char *) packet + SIZE_OF_PACKET_TYPE)) > 2)
    ↪  {
201                     answer_packet = list_of_clients_get_last_answer(cliaddr.
    ↪ sin_port, inet_ntoa(cliaddr.sin_addr),
202                                                                     &packet_size
    ↪ );
203                     if (answer_packet != NULL) {
204                         sendto(sockfd, answer_packet, packet_size, MSG_WAITALL,
    ↪ (const struct sockaddr *) &cliaddr,
205                                cliaddr_len);
206                         printf("Отправлен пакет с индексом: %d\n",
207                                 *(uint32_t *) ((char *) answer_packet +
    ↪ SIZE_OF_PACKET_TYPE));
208                     }
209                 } else {
210                     if ((deposit = list_of_deposits_get_deposit(cliaddr.sin_port
    ↪ , inet_ntoa(cliaddr.sin_addr),
211                                                                 (int) (*(
    ↪ uint32_t *) (packet + SIZE_OF_PACKET_TYPE +
212
    ↪         SIZE_OF_PACKET_INDEX +
213
    ↪         SIZE_OF_PACKET_ACK_TYPE) + 1),
214                                                                 &is_deposit_last
    ↪ )) == NULL) {
215
216                         char *error_msg = "ERROR: you have no deposit opened";
217                         answer_packet = create_error_packet(&packet_size,
```

```
218                                                         *(uint32_t *) ((char
    ↪  *) packet + SIZE_OF_PACKET_TYPE),
219                                                         error_msg, strlen(
    ↪ error_msg));
220
221                     } else {
222                         answer_packet = create_list_of_deposit_packet(
223                                 &packet_size,
224                                 *(uint32_t *) (packet + SIZE_OF_PACKET_TYPE),
225                                 deposit->deposit_id,
226                                 is_deposit_last == 1 ? 0 :
227                                 *(uint32_t *) (packet + SIZE_OF_PACKET_TYPE +
    ↪ SIZE_OF_PACKET_INDEX +
228                                                     SIZE_OF_PACKET_ACK_TYPE) + 1,
229                                 deposit->initial_amount,
230                                 deposit->current_amount);
231                     }
232
233                     list_of_clients_set_last_answer(cliaddr.sin_port, inet_ntoa(
    ↪ cliaddr.sin_addr), answer_packet,
234                                                     packet_size);
235                     if (is_deposit_last == 1) {
236                         list_of_clients_set_required_index(cliaddr.sin_port,
    ↪ inet_ntoa(cliaddr.sin_addr));
237                     }
238                     if (answer_packet != NULL) {
239                         sendto(sockfd, answer_packet, packet_size, MSG_WAITALL,
    ↪ (const struct sockaddr *) &cliaddr,
240                                cliaddr_len);
241                         printf("Отправлен пакет с индексом: %d\n",
242                                 *(uint32_t *) ((char *) answer_packet +
    ↪ SIZE_OF_PACKET_TYPE));
243                     }
244                 }
245             }
246             break;
247
248         case REFILL_DEPOSIT_PACKET:
249             if (list_of_clients_get_required_index(cliaddr.sin_port, inet_ntoa(
    ↪ cliaddr.sin_addr)) != 0 &&
250                 *(uint32_t *) ((char *) packet + SIZE_OF_PACKET_TYPE) ==
251                 list_of_clients_get_required_index(cliaddr.sin_port, inet_ntoa(
    ↪ cliaddr.sin_addr)) - 1) {
252
253                 answer_packet = list_of_clients_get_last_answer(cliaddr.sin_port
    ↪ , inet_ntoa(cliaddr.sin_addr),
254                                                                 &packet_size);
255                 if (answer_packet != NULL) {
256                     sendto(sockfd, answer_packet, packet_size, MSG_WAITALL, (
    ↪ const struct sockaddr *) &cliaddr,
257                            cliaddr_len);
258                     printf("Отправлен пакет с индексом: %d\n",
259                             *(uint32_t *) ((char *) answer_packet +
    ↪ SIZE_OF_PACKET_TYPE));
260                 }
261             }
262
263             if (*(uint32_t *) ((char *) packet + SIZE_OF_PACKET_TYPE) ==
264                 list_of_clients_get_required_index(cliaddr.sin_port, inet_ntoa(
    ↪ cliaddr.sin_addr))) {
```

```
265
266                 if ((*(uint32_t *) ((char *) packet + SIZE_OF_PACKET_TYPE)) > 2)
    ↪  {
267                     answer_packet = list_of_clients_get_last_answer(cliaddr.
    ↪ sin_port, inet_ntoa(cliaddr.sin_addr),
268                                                                     &packet_size
    ↪ );
269                     if (answer_packet != NULL) {
270                         sendto(sockfd, answer_packet, packet_size, MSG_WAITALL,
    ↪ (const struct sockaddr *) &cliaddr,
271                                cliaddr_len);
272                         printf("Отправлен␣пакет␣с␣индексом:␣%d\n",
273                                *(uint32_t *) ((char *) answer_packet +
    ↪ SIZE_OF_PACKET_TYPE));
274                     }
275                 } else {
276                     deposit_id = *(uint32_t *) ((char *) packet +
    ↪ SIZE_OF_PACKET_TYPE + SIZE_OF_PACKET_INDEX);
277                     amount = *(uint32_t *) ((char *) packet +
    ↪ SIZE_OF_PACKET_TYPE + SIZE_OF_PACKET_INDEX +
278                                             SIZE_OF_ID_DEPOSIT);
279                     double deposit_amount;
280                     if ((deposit_amount = list_of_deposits_refill_deposit(
    ↪ cliaddr.sin_port, inet_ntoa(cliaddr.sin_addr),
281
    ↪ deposit_id, amount)) == -1) {
282                         char *error_msg = "ERROR:␣couldn't␣refill␣deposit";
283                         answer_packet = create_error_packet(&packet_size,
284                                                             *(uint32_t *) ((char
    ↪  *) packet + SIZE_OF_PACKET_TYPE),
285                                                             error_msg, strlen(
    ↪ error_msg));
286                     } else {
287                         answer_packet = create_acknowledgment_packet(&
    ↪ packet_size,
288                                                                      *(uint32_t
    ↪ *) ((char *) packet +
289
    ↪     SIZE_OF_PACKET_TYPE),
290
    ↪ DEPOSIT_WAS_REFILLED,
291                                                                      (uint32_t)
    ↪ deposit_amount);
292                     }
293
294                     list_of_clients_set_last_answer(cliaddr.sin_port, inet_ntoa(
    ↪ cliaddr.sin_addr), answer_packet,
295                                                     packet_size);
296                     list_of_clients_set_required_index(cliaddr.sin_port,
    ↪ inet_ntoa(cliaddr.sin_addr));
297                     if (answer_packet != NULL) {
298                         sendto(sockfd, answer_packet, packet_size, MSG_WAITALL,
    ↪ (const struct sockaddr *) &cliaddr,
299                                cliaddr_len);
300                         printf("Отправлен␣пакет␣с␣индексом:␣%d\n",
301                                *(uint32_t *) ((char *) answer_packet +
    ↪ SIZE_OF_PACKET_TYPE));
302                     }
303                 }
304             }
```

```
305             break;
306
307         case GET_BANK_AMOUNT_PACKET:
308             if (list_of_clients_get_required_index(cliaddr.sin_port, inet_ntoa(
    ↪ cliaddr.sin_addr)) != 0 &&
309                 *(uint32_t *) ((char *) packet + SIZE_OF_PACKET_TYPE) ==
310                 list_of_clients_get_required_index(cliaddr.sin_port, inet_ntoa(
    ↪ cliaddr.sin_addr)) - 1) {
311                 answer_packet = list_of_clients_get_last_answer(cliaddr.sin_port
    ↪ , inet_ntoa(cliaddr.sin_addr),
312                                                                 &packet_size);
313                 if (answer_packet != NULL) {
314                     sendto(sockfd, answer_packet, packet_size, MSG_WAITALL, (
    ↪ const struct sockaddr *) &cliaddr,
315                            cliaddr_len);
316                     printf("Отправлен пакет с индексом: %d\n",
317                            *(uint32_t *) ((char *) answer_packet +
    ↪ SIZE_OF_PACKET_TYPE));
318                 }
319             }
320
321             if (*(uint32_t *) ((char *) packet + SIZE_OF_PACKET_TYPE) ==
322                 list_of_clients_get_required_index(cliaddr.sin_port, inet_ntoa(
    ↪ cliaddr.sin_addr))) {
323
324                 if ((*(uint32_t *) ((char *) packet + SIZE_OF_PACKET_TYPE)) > 2)
    ↪  {
325                     answer_packet = list_of_clients_get_last_answer(cliaddr.
    ↪ sin_port, inet_ntoa(cliaddr.sin_addr),
326                                                                     &packet_size
    ↪ );
327                     if (answer_packet != NULL) {
328                         sendto(sockfd, answer_packet, packet_size, MSG_WAITALL,
    ↪ (const struct sockaddr *) &cliaddr,
329                                cliaddr_len);
330                         printf("Отправлен пакет с индексом: %d\n",
331                                *(uint32_t *) ((char *) answer_packet +
    ↪ SIZE_OF_PACKET_TYPE));
332                     }
333                 } else {
334                     answer_packet = create_show_bank_amount_packet(&packet_size,
335                                                                    *(uint32_t *)
    ↪  ((char *) packet +
336
    ↪   SIZE_OF_PACKET_TYPE),
337
    ↪ list_of_deposits_get_bank_amount());
338
339                     list_of_clients_set_last_answer(cliaddr.sin_port, inet_ntoa(
    ↪ cliaddr.sin_addr), answer_packet,
340                                                     packet_size);
341                     list_of_clients_set_required_index(cliaddr.sin_port,
    ↪ inet_ntoa(cliaddr.sin_addr));
342                     if (answer_packet != NULL) {
343                         sendto(sockfd, answer_packet, packet_size, MSG_WAITALL,
    ↪ (const struct sockaddr *) &cliaddr,
344                                cliaddr_len);
345                         printf("Отправлен пакет с индексом: %d\n",
346                                *(uint32_t *) ((char *) answer_packet +
    ↪ SIZE_OF_PACKET_TYPE));
```

```
347                     }
348                 }
349             }
350             break;
351 
352         case PLEASE_ADD_PERCENTS_PACKET:
353             if (list_of_clients_get_required_index(cliaddr.sin_port, inet_ntoa(
    ↪ cliaddr.sin_addr)) != 0 &&
354                 *(uint32_t *) ((char *) packet + SIZE_OF_PACKET_TYPE) ==
355                 list_of_clients_get_required_index(cliaddr.sin_port, inet_ntoa(
    ↪ cliaddr.sin_addr)) - 1) {
356                 answer_packet = list_of_clients_get_last_answer(cliaddr.sin_port
    ↪ , inet_ntoa(cliaddr.sin_addr),
357                                                                 &packet_size);
358                 if (answer_packet != NULL) {
359                     sendto(sockfd, answer_packet, packet_size, MSG_WAITALL, (
    ↪ const struct sockaddr *) &cliaddr,
360                            cliaddr_len);
361                     printf("Отправлен пакет с индексом: %d\n",
362                            *(uint32_t *) ((char *) answer_packet +
    ↪ SIZE_OF_PACKET_TYPE));
363                 }
364             }
365 
366             if (*(uint32_t *) ((char *) packet + SIZE_OF_PACKET_TYPE) ==
367                 list_of_clients_get_required_index(cliaddr.sin_port, inet_ntoa(
    ↪ cliaddr.sin_addr))) {
368 
369                 if ((*(uint32_t *) ((char *) packet + SIZE_OF_PACKET_TYPE)) > 2)
    ↪  {
370                     answer_packet = list_of_clients_get_last_answer(cliaddr.
    ↪ sin_port, inet_ntoa(cliaddr.sin_addr),
371                                                                     &packet_size
    ↪ );
372                     if (answer_packet != NULL) {
373                         sendto(sockfd, answer_packet, packet_size, MSG_WAITALL,
    ↪ (const struct sockaddr *) &cliaddr,
374                                cliaddr_len);
375                         printf("Отправлен пакет с индексом: %d\n",
376                                *(uint32_t *) ((char *) answer_packet +
    ↪ SIZE_OF_PACKET_TYPE));
377                     }
378                 } else {
379                     if (add_percents_to_client(cliaddr.sin_port, inet_ntoa(
    ↪ cliaddr.sin_addr)) > 0) {
380                         answer_packet = create_acknowledgment_packet(&
    ↪ packet_size,
381                                                                      *(uint32_t
    ↪ *) ((char *) packet +
382 
    ↪     SIZE_OF_PACKET_TYPE),
383 
    ↪ PERCENTS_ADDED, 0);
384                     } else {
385                         char *error_msg = "ERROR: couldn't add percents";
386                         answer_packet = create_error_packet(&packet_size,
387                                 *(uint32_t *) ((char *) packet +
    ↪ SIZE_OF_PACKET_TYPE), error_msg, strlen(error_msg));
388                     }
389                     list_of_clients_set_last_answer(cliaddr.sin_port, inet_ntoa(
```

```
    ↪ cliaddr.sin_addr), answer_packet,
390                                                     packet_size);
391                     list_of_clients_set_required_index(cliaddr.sin_port,
    ↪ inet_ntoa(cliaddr.sin_addr));
392                     if (answer_packet != NULL) {
393                         sendto(sockfd, answer_packet, packet_size, MSG_WAITALL,
    ↪ (const struct sockaddr *) &cliaddr,
394                                cliaddr_len);
395                         printf("Отправлен пакет с индексом: %d\n",
396                                *(uint32_t *) ((char *) answer_packet +
    ↪ SIZE_OF_PACKET_TYPE));
397                     }
398                 }
399             }
400             break;
401
402         default:
403             break;
404     }
405     free(packet);
406     return NULL;
407 }
408
409 Listening_thread_input *
410 init_listening_thread_input_structure(int sockfd, void *packet, struct
    ↪ sockaddr_in *cliaddr,
411                                       const int *cliaddr_len) {
412     Listening_thread_input *new_input_structure = (Listening_thread_input *)
    ↪ malloc(sizeof(Listening_thread_input));
413
414     new_input_structure->sockfd = sockfd;
415     memcpy(&new_input_structure->cliaddr, cliaddr, sizeof(struct sockaddr_in));
416     new_input_structure->cliaddr_len = *cliaddr_len;
417     new_input_structure->packet = packet;
418     return new_input_structure;
419 }
420
421 pthread_t create_listening_thread(int sockfd, void *packet, struct sockaddr_in
    ↪ cliaddr, int cliaddr_len) {
422     pthread_t listening_thread;
423
424     Listening_thread_input *listening_thread_input =
    ↪ init_listening_thread_input_structure(sockfd, packet, &cliaddr,
425
    ↪             &cliaddr_len);
426     if (pthread_create(&listening_thread, NULL, socket_listening_thread,
    ↪ listening_thread_input)) {
427         return -1;
428     }
429     return listening_thread;
430 }
```

Листинг 17: console_thread.c

```
1 #include "main.h"
2 int get_user_choice() {
3     printf("1) Начислить проценты\n");
4     printf("2) Показать счёт банка\n");
5     printf("3) Показать все вклады\n");
6     printf("4) Завершить работу\n");
7     printf("5) Посмотреть список клиентов\n");
```

```
8       printf("6) Удалить вклады клиента\n");
9       char buffer[sizeof(int) + 2];
10      fgets(buffer, sizeof(int) + 2, stdin);
11      return atoi(buffer);
12  }
13
14  void* console_listening_thread(void* arg) {
15      int port;
16      char address[17];
17
18      while(1) {
19          switch (get_user_choice()){
20              case 1:
21                  list_of_deposits_add_percents();
22                  break;
23
24              case 2:
25                  list_of_deposits_export_bank_amount();
26                  break;
27
28              case 3:
29                  list_of_deposits_all_deposits(stdout);
30                  break;
31
32              case 4:
33                  shutdown(*(int*) arg, SHUT_RDWR);
34                  close(*(int*) arg);
35                  return NULL;
36
37              case 5:
38                  list_of_clients_export(stdout);
39                  break;
40
41              case 6:
42                  printf("Введите порт клиента:\n");
43                  scanf("%d", &port);
44                  printf("Введите адрес клиента:\n");
45                  scanf("%s", address);
46                  if (remove_client_deposits(port, address) < 0){
47                      printf("ERROR: can't remove client deposits.");
48                  } else {
49                      printf("Вклады клиента удалены\n");
50                  }
51                  break;
52
53              default:
54                  printf("ERROR: wrong choice\n");
55                  break;
56          }
57      }
58  }
59
60  pthread_t create_user_listening_thread(int* initial_socket) {
61      pthread_t user_listening_thread;
62
63      if (pthread_create(&user_listening_thread, NULL, console_listening_thread, (
        ↪ void*) initial_socket)) {
64          return -1;
65      }
66      return user_listening_thread;
```

```
67 }
```

Листинг 18: list_of_clients.c

```
1 #include "main.h"
2 static pthread_mutex_t list_of_clients_mutex;
3 static User_info* root = NULL;
4
5 void init_list_of_clients_mutex() {
6     pthread_mutex_init(&list_of_clients_mutex, NULL);
7 }
8
9 User_info* make_new_client(int sockfd, int port, char* address, pthread_t
    ↪ client_thread) {
10     User_info* new_client = (User_info*) malloc(sizeof(User_info));
11
12     new_client -> port = port;
13     new_client -> address = strdup(address);
14     new_client -> sockfd = sockfd;
15     new_client -> client_thread = client_thread;
16     new_client -> next = NULL;
17     new_client -> required_index = 1;
18     new_client -> last_answer = NULL;
19     new_client -> packet_size = 0;
20     return new_client;
21 }
22
23 uint32_t list_of_clients_get_required_index(int port, char* address) {
24     pthread_mutex_lock(&list_of_clients_mutex);
25     User_info *iterator = root;
26
27     while (iterator != NULL && (iterator->port != port || strcmp(iterator->
    ↪ address, address) != 0) ) {
28         iterator = iterator->next;
29     }
30
31     if (iterator != NULL) {
32         pthread_mutex_unlock(&list_of_clients_mutex);
33         return iterator->required_index;
34     }
35     pthread_mutex_unlock(&list_of_clients_mutex);
36     return 0;
37 }
38
39 void list_of_clients_set_required_index(int port, char* address) {
40     pthread_mutex_lock(&list_of_clients_mutex);
41     User_info *iterator = root;
42
43     while (iterator != NULL && (iterator->port != port || strcmp(iterator->
    ↪ address, address) != 0) ) {
44         iterator = iterator->next;
45     }
46
47     if (iterator != NULL) {
48         iterator->required_index = (iterator->required_index + 1 == 65535) ? 1 :
    ↪  iterator->required_index + 1;
49     }
50     pthread_mutex_unlock(&list_of_clients_mutex);
51 }
52
53 void list_of_clients_set_last_answer(int port, char *address, void *answer,
```

```
    ↪ uint32_t packet_size) {
54      pthread_mutex_lock(&list_of_clients_mutex);
55      User_info *iterator = root;
56
57      while (iterator != NULL && (iterator->port != port || strcmp(iterator->
    ↪ address, address) != 0)) {
58          iterator = iterator->next;
59      }
60
61      if (iterator != NULL) {
62          free(iterator->last_answer);
63          iterator->packet_size = packet_size;
64          iterator->last_answer = answer;
65      }
66      pthread_mutex_unlock(&list_of_clients_mutex);
67  }
68
69  void* list_of_clients_get_last_answer(int port, char* address, uint32_t *
    ↪ packet_size) {
70      pthread_mutex_lock(&list_of_clients_mutex);
71      User_info *iterator = root;
72
73      while (iterator != NULL && (iterator->port != port || strcmp(iterator->
    ↪ address, address) != 0) ) {
74          iterator = iterator->next;
75      }
76
77      if (iterator != NULL) {
78          *packet_size = iterator->packet_size;
79          pthread_mutex_unlock(&list_of_clients_mutex);
80          return iterator->last_answer;
81      }
82      pthread_mutex_unlock(&list_of_clients_mutex);
83      return NULL;
84  }
85
86  void list_of_clients_add(User_info* new_client) {
87      pthread_mutex_lock(&list_of_clients_mutex);
88
89      if (root == NULL) {
90          root = new_client;
91          pthread_mutex_unlock(&list_of_clients_mutex);
92          return;
93      }
94      User_info *iterator;
95      User_info *iterator_prev = root;
96
97      for (iterator = root; iterator->next != NULL && (new_client->port !=
    ↪ iterator->port ||
98                              strcmp(new_client->address, iterator->address) != 0);
99                              iterator_prev = iterator, iterator = iterator->next) {
100     }
101
102     if (iterator != NULL) {
103         pthread_mutex_unlock(&list_of_clients_mutex);
104         return;
105     }
106     iterator_prev->next = new_client;
107     pthread_mutex_unlock(&list_of_clients_mutex);
108 }
```

```
109
110 void list_of_clients_export(FILE* dst_fd) {
111     pthread_mutex_lock(&list_of_clients_mutex);
112
113     if (root == NULL) {
114         fprintf(dst_fd, "No clients\n");
115         pthread_mutex_unlock(&list_of_clients_mutex);
116         return;
117     }
118     User_info* iterator = root;
119     fprintf(dst_fd,"%5s %8s %16s\n", "index", "port", "address");
120
121     for (int index = 1; iterator != NULL; index++, iterator = iterator->next) {
122         fprintf(dst_fd, "%5d %8d %16s\n", index++, iterator->port, iterator->
    ↪ address);
123     }
124     pthread_mutex_unlock(&list_of_clients_mutex);
125 }
126
127 void list_of_clients_remove_all() {
128     pthread_mutex_lock(&list_of_clients_mutex);
129     User_info* iterator = root;
130     User_info* iterator_next;
131
132     while(iterator != NULL) {
133         iterator_next = iterator->next;
134         shutdown(iterator -> sockfd, SHUT_RDWR);
135         close(iterator -> sockfd);
136         pthread_join(iterator->client_thread, NULL);
137         free(iterator->last_answer);
138         free(iterator);
139         iterator = iterator_next;
140     }
141     pthread_mutex_unlock(&list_of_clients_mutex);
142 }
```

Листинг 19: list_of_deposits.c

```
1 #include "main.h"
2 static pthread_mutex_t list_of_deposits_mutex;
3 static Deposit_info* root = NULL;
4 static double bank_amount = 0;
5
6 void init_list_of_deposits_mutex() {
7     pthread_mutex_init(&list_of_deposits_mutex, NULL);
8 }
9
10 double list_of_deposits_get_bank_amount() {
11     return bank_amount;
12 }
13
14 Deposit_info* make_new_deposit(int deposit_id, int initial_amount,int
    ↪ client_sockfd, struct sockaddr_in cliaddr) {
15     Deposit_info* new_deposit = (Deposit_info*) malloc(sizeof(Deposit_info));
16
17     new_deposit->client_sockfd = client_sockfd;
18     new_deposit->current_amount = (double) initial_amount;
19     new_deposit->initial_amount = initial_amount;
20     new_deposit->deposit_id = deposit_id;
21     new_deposit->cliaddr = cliaddr;
22     new_deposit->next = NULL;
```

```
23      return new_deposit;
24  }
25
26  uint32_t generate_deposit_id() {
27      pthread_mutex_lock(&list_of_deposits_mutex);
28      uint32_t deposit_id = 1;
29
30      for (Deposit_info* iterator = root; iterator != NULL; iterator = iterator->
        ↪ next) {
31          deposit_id = iterator->deposit_id + 1;
32      }
33      pthread_mutex_unlock(&list_of_deposits_mutex);
34      return deposit_id;
35  }
36
37  void list_of_deposits_add(Deposit_info* new_deposit) {
38      pthread_mutex_lock(&list_of_deposits_mutex);
39      Deposit_info* iterator = root;
40
41      while (iterator != NULL && iterator->next != NULL) {
42          iterator = iterator->next;
43      }
44
45      if (iterator == NULL) {
46          root = new_deposit;
47          pthread_mutex_unlock(&list_of_deposits_mutex);
48          return;
49      }
50      iterator->next = new_deposit;
51      pthread_mutex_unlock(&list_of_deposits_mutex);
52  }
53
54  int32_t list_of_deposits_remove(int port, const char *address, uint32_t
        ↪ deposit_id) {
55      pthread_mutex_lock(&list_of_deposits_mutex);
56      Deposit_info* iterator = root;
57      Deposit_info* previous =  NULL;
58
59      while (iterator != NULL && iterator->deposit_id != deposit_id) {
60          previous = iterator;
61          iterator = iterator->next;
62      }
63
64      if (iterator == NULL || iterator->cliaddr.sin_port != port || inet_ntoa(
        ↪ iterator->cliaddr.sin_addr) != address) {
65          pthread_mutex_unlock(&list_of_deposits_mutex);
66          return -1;
67      }
68
69      if (iterator == root){
70          Deposit_info* old_root = root;
71          root = root->next;
72          free(old_root);
73          pthread_mutex_unlock(&list_of_deposits_mutex);
74          return deposit_id;
75      }
76      previous->next = iterator->next;
77      free(iterator);
78      pthread_mutex_unlock(&list_of_deposits_mutex);
79      return deposit_id;
```

```
80 }
81
82 void list_of_deposits_add_percents(){
83     pthread_mutex_lock(&list_of_deposits_mutex);
84     Deposit_info* iterator = root;
85
86     while (iterator != NULL) {
87         bank_amount +=  iterator->current_amount * 0.05;
88         iterator->current_amount +=  iterator->current_amount * 0.1;
89         iterator = iterator -> next;
90     }
91     pthread_mutex_unlock(&list_of_deposits_mutex);
92 }
93
94 void list_of_deposits_remove_all() {
95     Deposit_info* iterator;
96     Deposit_info* iterator_next;
97     pthread_mutex_lock(&list_of_deposits_mutex);
98
99     for (iterator = root; iterator != NULL ; iterator = iterator_next) {
100         iterator_next = iterator->next;
101         free(iterator);
102     }
103     pthread_mutex_unlock(&list_of_deposits_mutex);
104 }
105
106 void list_of_deposits_all_deposits(FILE* output_file){
107     Deposit_info* iterator;
108     pthread_mutex_lock(&list_of_deposits_mutex);
109
110     if (root == NULL) {
111         fprintf(output_file, "No deposits\n");
112         pthread_mutex_unlock(&list_of_deposits_mutex);
113         return;
114     }
115     fprintf(output_file, "%17s %6s %10s %14s %14s\n", "address", "port", "
    ↪ deposit_id", "initial_amount",
116             "current_amount");
117
118     for (iterator = root; iterator != NULL; iterator = iterator->next) {
119         fprintf(output_file, "%17s %6d %10d %14d %14f\n", inet_ntoa(iterator->
    ↪ cliaddr.sin_addr),
120                 iterator->cliaddr.sin_port, iterator->deposit_id,
121                 iterator->initial_amount, iterator->current_amount);
122     }
123     pthread_mutex_unlock(&list_of_deposits_mutex);
124 }
125
126 void list_of_deposits_export_bank_amount() {
127     printf("Bank amount: %f\n", bank_amount);
128 }
129
130 double list_of_deposits_refill_deposit(int port, const char *address, uint32_t
    ↪ deposit_id, uint32_t added_amount) {
131     pthread_mutex_lock(&list_of_deposits_mutex);
132     Deposit_info *iterator = root;
133
134     while (iterator != NULL && iterator->deposit_id != deposit_id) {
135         iterator = iterator->next;
136     }
```

```
137
138     if (iterator == NULL || iterator->cliaddr.sin_port != port || inet_ntoa(
    ↪ iterator->cliaddr.sin_addr) != address) {
139         pthread_mutex_unlock(&list_of_deposits_mutex);
140         return -1;
141     }
142     iterator->current_amount += added_amount;
143     pthread_mutex_unlock(&list_of_deposits_mutex);
144     return iterator->current_amount;
145 }
146
147 int add_percents_to_client(int port, const char *address){
148     pthread_mutex_lock(&list_of_deposits_mutex);
149     int client_has_deposits = -1;
150     Deposit_info* iterator = root;
151
152     while (iterator != NULL) {
153         if (iterator->cliaddr.sin_port == port && strcmp(inet_ntoa(iterator->
    ↪ cliaddr.sin_addr), address) == 0 ){
154             bank_amount +=  iterator->current_amount * 0.05;
155             iterator->current_amount +=  iterator->current_amount * 0.1;
156             client_has_deposits = 1;
157         }
158         iterator = iterator->next;
159     }
160     pthread_mutex_unlock(&list_of_deposits_mutex);
161     return client_has_deposits;
162 }
163
164 int remove_client_deposits(int port, const char *address){
165     pthread_mutex_lock(&list_of_deposits_mutex);
166     Deposit_info* iterator = root;
167     Deposit_info* previous =  NULL;
168
169     while (iterator != NULL) {
170         if (iterator->cliaddr.sin_port == port && strcmp(inet_ntoa(iterator->
    ↪ cliaddr.sin_addr), address) == 0){
171             if (iterator == root){
172                 Deposit_info* old_root = root;
173                 root = root->next;
174                 free(old_root);
175             } else {
176                 previous->next = iterator->next;
177                 free(iterator);
178             }
179         }
180         previous = iterator;
181         iterator = iterator->next;
182     }
183     pthread_mutex_unlock(&list_of_deposits_mutex);
184     return 1;
185 }
186
187 Deposit_info *list_of_deposits_get_deposit(int port, const char *address, int
    ↪ number, int *is_deposit_last) {
188     pthread_mutex_lock(&list_of_deposits_mutex);
189     Deposit_info *iterator = root;
190     Deposit_info *result;
191
192     while (iterator != NULL && number != 0) {
```

```
193             if (iterator->cliaddr.sin_port == port && inet_ntoa(iterator->cliaddr.
    ↪ sin_addr) == address) {
194                 number--;
195             }
196             if (number != 0) {
197                 iterator = iterator->next;
198             }
199         }
200         result = iterator;
201         *is_deposit_last = 1;
202         while (iterator != NULL) {
203             iterator = iterator->next;
204             if (iterator != NULL && iterator->cliaddr.sin_port == port && inet_ntoa(
    ↪ iterator->cliaddr.sin_addr) == address) {
205                 *is_deposit_last = -1;
206             }
207         }
208         pthread_mutex_unlock(&list_of_deposits_mutex);
209         return result;
210 }
```

Листинг 20: packet_handler.c

```
1  #include "main.h"
2  void* create_add_deposit_packet(uint32_t* packet_length, uint32_t index,
       ↪ uint32_t initial_amount) {
3      *packet_length = SIZE_OF_PACKET_TYPE + SIZE_OF_PACKET_INDEX +
       ↪ SIZE_OF_PACKET_AMOUNT;
4      void* packet = malloc(*packet_length);
5      uint16_t packet_type = OPEN_DEPOSIT_PACKET;
6      memcpy(packet, &packet_type, SIZE_OF_PACKET_TYPE);
7      memcpy(packet + SIZE_OF_PACKET_TYPE, &index, SIZE_OF_PACKET_INDEX);
8      memcpy(packet + SIZE_OF_PACKET_TYPE + SIZE_OF_PACKET_INDEX, &initial_amount,
       ↪  SIZE_OF_INITIAL_AMOUNT);
9      return packet;
10 }
11
12 void* create_remove_deposit_packet(uint32_t* packet_size, uint32_t index,
       ↪ uint32_t deposit_id) {
13     *packet_size = SIZE_OF_PACKET_TYPE + SIZE_OF_PACKET_INDEX +
       ↪ SIZE_OF_ID_DEPOSIT;
14     void* packet = malloc(*packet_size);
15     uint16_t packet_type = CLOSE_DEPOSIT_PACKET;
16     memcpy(packet, &packet_type, SIZE_OF_PACKET_TYPE);
17     memcpy(packet + SIZE_OF_PACKET_TYPE, &index, SIZE_OF_PACKET_INDEX);
18     memcpy(packet + SIZE_OF_PACKET_TYPE + SIZE_OF_PACKET_INDEX, &deposit_id,
       ↪ SIZE_OF_ID_DEPOSIT);
19     return packet;
20 }
21
22 void* create_get_list_of_deposits_packet(uint32_t index) {
23     uint32_t packet_length = SIZE_OF_PACKET_TYPE + SIZE_OF_PACKET_INDEX;
24     void* packet = malloc(packet_length);
25     uint16_t packet_type = GET_LIST_OF_DEPOSITS_PACKET;
26     memcpy(packet, &packet_type, SIZE_OF_PACKET_TYPE);
27     memcpy(packet + SIZE_OF_PACKET_TYPE, &index, SIZE_OF_PACKET_INDEX);
28     return packet;
29 }
30
31 void* create_acknowledgment_packet(uint32_t* packet_size, uint32_t index,
       ↪ uint16_t ack_type, uint32_t number) {
```

```
32      *packet_size = SIZE_OF_PACKET_TYPE + SIZE_OF_PACKET_INDEX +
    ↪ SIZE_OF_PACKET_ACK_TYPE + SIZE_OF_ACK_NUMBER;
33      void* packet = malloc(*packet_size);
34      uint16_t packet_type = ACKNOWLEDGMENT_PACKET;
35      memcpy(packet, &packet_type, SIZE_OF_PACKET_TYPE);
36      memcpy(packet + SIZE_OF_PACKET_TYPE, &index, SIZE_OF_PACKET_INDEX);
37      memcpy(packet + SIZE_OF_PACKET_TYPE + SIZE_OF_PACKET_INDEX, &ack_type,
    ↪ SIZE_OF_PACKET_ACK_TYPE);
38      memcpy(packet + SIZE_OF_PACKET_TYPE + SIZE_OF_PACKET_INDEX +
    ↪ SIZE_OF_PACKET_ACK_TYPE, &number, SIZE_OF_ACK_NUMBER);
39      return packet;
40  }
41
42  void* create_error_packet(uint32_t* packet_length, uint32_t index, char*
    ↪ err_text, uint64_t msg_size) {
43      *packet_length = SIZE_OF_PACKET_TYPE + SIZE_OF_PACKET_INDEX + msg_size;
44      void* packet = malloc(*packet_length);
45      uint16_t packet_type = ERROR_PACKET;
46      memcpy(packet, &packet_type, SIZE_OF_PACKET_TYPE);
47      memcpy(packet + SIZE_OF_PACKET_TYPE, &index, SIZE_OF_PACKET_INDEX);
48      memcpy(packet + SIZE_OF_PACKET_TYPE + SIZE_OF_PACKET_INDEX, err_text,
    ↪ msg_size);
49      return packet;
50  }
51
52  void* create_refill_deposit_packet(uint32_t* packet_length, uint32_t index,
    ↪ uint32_t deposit_id, uint32_t amount) {
53      *packet_length = SIZE_OF_PACKET_TYPE + SIZE_OF_PACKET_INDEX +
    ↪ SIZE_OF_ID_DEPOSIT + SIZE_OF_PACKET_AMOUNT;
54      void* packet = malloc(*packet_length);
55      uint16_t packet_type = REFILL_DEPOSIT_PACKET;
56      memcpy(packet, &packet_type, SIZE_OF_PACKET_TYPE);
57      memcpy(packet + SIZE_OF_PACKET_TYPE, &index, SIZE_OF_PACKET_INDEX);
58      memcpy(packet + SIZE_OF_PACKET_TYPE + SIZE_OF_PACKET_INDEX, &deposit_id,
    ↪ SIZE_OF_ID_DEPOSIT);
59      memcpy(packet + SIZE_OF_PACKET_TYPE + SIZE_OF_PACKET_INDEX +
    ↪ SIZE_OF_ID_DEPOSIT, &amount, SIZE_OF_PACKET_AMOUNT);
60      return packet;
61  }
62
63  void* create_list_of_deposit_packet(uint32_t* packet_length, uint32_t index,
    ↪ uint32_t deposit_id, uint32_t block_number,
64          uint32_t initial_amount, double current_amount) {
65      *packet_length = SIZE_OF_PACKET_TYPE + SIZE_OF_PACKET_INDEX +
    ↪ SIZE_OF_PACKET_BLOCK_NUMBER +
66              SIZE_OF_ID_DEPOSIT + SIZE_OF_INITIAL_AMOUNT + sizeof(double);
67      void* packet = malloc(*packet_length);
68
69      uint16_t  packet_type = LIST_OF_DEPOSITS_PACKET;
70      memcpy(packet, &packet_type, SIZE_OF_PACKET_TYPE);
71      memcpy(packet + SIZE_OF_PACKET_TYPE, &index, SIZE_OF_PACKET_INDEX);
72      memcpy(packet + SIZE_OF_PACKET_TYPE + SIZE_OF_PACKET_INDEX, &block_number,
    ↪ SIZE_OF_PACKET_BLOCK_NUMBER);
73      memcpy(packet + SIZE_OF_PACKET_TYPE + SIZE_OF_PACKET_INDEX +
    ↪ SIZE_OF_PACKET_BLOCK_NUMBER, &deposit_id,
74              SIZE_OF_ID_DEPOSIT);
75      memcpy(packet + SIZE_OF_PACKET_TYPE + SIZE_OF_PACKET_INDEX +
    ↪ SIZE_OF_PACKET_BLOCK_NUMBER + SIZE_OF_ID_DEPOSIT,
76              &initial_amount, SIZE_OF_INITIAL_AMOUNT);
77      memcpy(packet + SIZE_OF_PACKET_TYPE + SIZE_OF_PACKET_INDEX +
```

```
    ↪ SIZE_OF_PACKET_BLOCK_NUMBER + SIZE_OF_ID_DEPOSIT +
78              SIZE_OF_INITIAL_AMOUNT, &current_amount, sizeof(double));
79      return packet;
80  }
81
82  void* create_show_bank_amount_packet(uint32_t* packet_length, uint32_t index,
    ↪ double bank_amount){
83      *packet_length = SIZE_OF_PACKET_TYPE + SIZE_OF_PACKET_INDEX + sizeof(double)
    ↪ ;
84      void* packet = malloc(*packet_length);
85      uint16_t packet_type = SHOW_BANK_AMOUNT_PACKET;
86      memcpy(packet, &packet_type, SIZE_OF_PACKET_TYPE);
87      memcpy(packet + SIZE_OF_PACKET_TYPE, &index, SIZE_OF_PACKET_INDEX);
88      memcpy(packet + SIZE_OF_PACKET_TYPE + SIZE_OF_PACKET_INDEX, &bank_amount,
    ↪ sizeof(double));
89      return packet;
90  }
```

Листинг 21: receiving_thread.c

```
1  #include "main.h"
2  void* receiving_thread(void* arg) {
3      struct sockaddr_in cliaddr;
4      int cliaddr_len = sizeof(struct sockaddr_in);
5      struct sockaddr_in servaddr;
6      int port = ((Receiving_thread_input*) arg)->port;
7      int *sockfd = ((Receiving_thread_input*) arg)-> initial_sockfd;
8
9      while (1) {
10         void* packet = malloc(MAX_PACKET_SIZE);
11         bzero(packet, MAX_PACKET_SIZE);
12         bzero(&cliaddr, sizeof(cliaddr));
13
14         if (recvfrom(*sockfd, packet, (size_t) MAX_PACKET_SIZE, MSG_WAITALL, (
    ↪ struct sockaddr *) &cliaddr, &cliaddr_len) <= 0) {
15             free(packet);
16             free(arg);
17             return NULL;
18         }
19
20         int new_sockfd;
21         if ((new_sockfd = socket(AF_INET, SOCK_DGRAM, 0)) < 0) {
22             perror("ERROR: socket creation failed.\n");
23             free(packet);
24             free(arg);
25             return NULL;
26         }
27
28         memset(&servaddr, 0, sizeof(servaddr));
29         servaddr.sin_family = AF_INET;
30         servaddr.sin_addr.s_addr = INADDR_ANY;
31
32         if (setsockopt(new_sockfd, SOL_SOCKET, SO_REUSEADDR, &(int) {1}, sizeof(
    ↪ int)) < 0) {
33             fprintf(stderr, "ERROR: setsockopt(SO_REUSEADDR) failed");
34         }
35
36         pthread_t listening_thread = create_listening_thread(new_sockfd, packet,
    ↪  cliaddr, cliaddr_len);
37         list_of_clients_add(
38                 make_new_client(new_sockfd, cliaddr.sin_port, strdup(inet_ntoa(
```

```
    ↪ cliaddr.sin_addr)), listening_thread));
39
40      }
41  }
42
43  Receiving_thread_input* init_receiving_thread_input_structure(int port, int *
    ↪ initial_sockfd) {
44      Receiving_thread_input* new_input_structure = (Receiving_thread_input*)
    ↪ malloc(sizeof(Receiving_thread_input));
45
46      new_input_structure->port = port;
47      new_input_structure->initial_sockfd=initial_sockfd;
48      return new_input_structure;
49  }
50
51  pthread_t create_receiving_thread(int port, int *initial_sockfd) {
52      pthread_t receiving_thread_;
53
54      Receiving_thread_input* receiving_thread_input =
    ↪ init_receiving_thread_input_structure(port, initial_sockfd);
55
56      if( pthread_create(&receiving_thread_, NULL, receiving_thread,
    ↪ receiving_thread_input)) {
57          return -1;
58      }
59      return receiving_thread_;
60  }
```