Министерство науки и высшего образования Российской Федерации

Федеральное государственное бюджетное образовательное учреждение

высшего образования

«Ульяновский государственный технический университет»

Кафедра «Вычислительная техника»

Сети ЭВМ и телекоммуникации

(название дисциплины)

Лабораторная работа №2-3

Вариант №5

«Разработка многопоточного сервера с использованием механизма сокетов»

(название (тема) работы)

Выполнил

Студент группы \_\_\_ИВТАСбд-42\_\_\_\_

\_\_\_\_\_\_\_Сулейманов М.З. (Фамилия И.О.)

Проверил:

преподаватель кафедры «ВТ»

(должность)

\_\_\_\_\_\_Мартынов А.И.\_\_\_\_\_\_\_\_\_\_\_\_\_

(Фамилия И.О.)

Ульяновск

2024

**Цель работы**

Научиться работать с сокетами и строить многопоточные серверные приложения на примере сервера и клиента шифрования

**Задание**

Разработать серверное приложение, которое осуществляет взаимодействие при помощи механизма сокетов TCP/IP. Сервер работает в локальной сети по определенному IP-адресу и порту, которые настраиваются при запуске сервера, и отвечает на запросы клиента. Формат команд, которые должен обрабатывать сервер определяется согласно варианту задания. Основное требование к серверу, это обеспечение многопоточности, т.е. серверное приложение должно быть разработано таким образом, чтобы одновременно могло обрабатывать запросы нескольких пользователей и возможность работы со стандартными клиентами.

Клиентская часть должна реализовывать функции, соответствующие номеру варианта. Обмен информацией с сервером осуществляется в текстовом режиме посредством определенных команд, каждая из которых выполняет определенное действие. Каждая команда сервера состоит из служебного слова и параметров. Сервер не должен быть чувствителен к регистру команд. Запрос должен завершаться символом конца строки (0x10) для обеспечения совместимости сервера со стандартными клиентами, такими как Telnet.

**Теоретическая часть**

Какие бы замечательные идеи в области телекоммуникаций, распределенных баз знаний или поисковых систем вам не пришли в голову, реализовать их на практике можно, лишь написав соответствующую программу. Основные операционные среды (Unix или Windows) базируются в настоящее время на идеологии сокетов (socket). Эта технология была разработана в университете г. Беркли (США) для системы Unix, поэтому соединители (сокеты) иногда называют сокетами Беркли (berkeley sockets). Сокеты реализуют механизм взаимодействия не только партнеров по телекоммуникациям, но и процессов в ЭВМ вообще. Технология сокетов лежит в основе современного сетевого программирования.

Работа с сокетами содержит ряд этапов: сокет создается, настраивается на заданный режим работы, применяется для организации обмена и, наконец, ликвидируется. Технология сокетов поддерживает работу с любыми стеками протоколов, совмещенные процедуры ввода/вывода, использование большого числа сервиспровайдеров (серверов услуг), возможность группирования сокетов, что позволяет реализовать их приоритетное обслуживание, и многое другое. Набор операторов, поддерживающих интерфейс сервис провайдера, образует отдельную динамическую библиотеку.

Для общей синхронизации работы сервиспровайдеров и приложений в winsock введено понятие объектов событий. Объекты событий служат, в частности, для организации работы совмещенных по времени процессов информационного обмена. Здесь уместно замечание об использовании стандартных номеров портов. В многозадачных, многопользовательских системах стандартные номера портов используются при инициализации процесса. Так как допускается несколько идентичных соединений (например, несколько одновременных сессий FTP) между клиентом и сервером, стандартными номерами портов здесь не обойтись. Ведь PsIPs сервера могут соответствовать несколько PcIPc клиента.

В системах, ориентированных на соединение, пара комбинаций IP-адресов и номеров портов однозначно определяет канал связи между двумя процессами в ЭВМ. Такая комбинация называется сокетом (socket). Номера портов могут и совпадать, так как относятся к разным машинам, но IP-адреса должны быть обязательно разными. Впервые идея сокета была использована в системе BSD4.3 Unix для организации сетевого ввода/вывода. В Unix внешнее устройство и файл с точки зрения системного программиста эквивалентны. Сетевые процедуры несколько сложнее и не укладываются в такую простую схему. Из этой схемы выпадают, прежде всего, операции, при которых Сокет сервер пассивно ожидает обращения, особенно операции обмена, не ориентированные на соединение. является пограничным понятием между протоколами телекоммуникаций и операционной системой ЭВМ. Сокеты играют важную роль при написании прикладных программ (API).

Сокет отправителя = IP-адрес отправителя + номер порта отправителя

Сокет адресата = IP-адрес адресата + номер порта адресата

При обменах, ориентированных на соединение, формируется ансамбль ( IPSPS + IPDPD ), где IPSPS — адрес и порт отправителя, а IPDPD– адрес и порт места назначения. Межкомпьютерные коммуникации не сводятся к знакомству с соседским депозитарием, к выполнению операций Telnet/ssh, FTP/scp и т.д. Одной из важнейших задач является удаленный контроль за процессами в больших распределенных системах, когда обмен информацией активизируется не человеком, а ЭВМ. Примерами таких задач могут служить управление современными высокотехнологичными производствами, сбор метео- или другой геофизической информации в реальном масштабе времени, эксперименты в области физики высоких энергий, где для контроля установки и сбора экспериментальных данных используются десятки (а иногда и сотни) вычислительных машин, которые обмениваются диагностической информацией и данными. Именно для решения таких задач и применяются идеи сокетов, "труб" и т.д.. Понятие сокета в прикладных программах — это не просто комбинация IP-адресов и номеров портов, это указатель на структуру данных, где хранятся параметры виртуального канала. Прежде чем воспользоваться сокетом, его нужно сформировать. Оператор формирования сокета имеет вид:

s=socket(INTAF, INT type, INT protocol);

где все параметры целочисленные, AF (address\_family) характеризует набор протоколов, соответствующий данному сокету (это может быть набор Internet, Unix, Appletalk и т.д.). Для Интернет AF может принимать только значение PF\_INET, для Unix PF\_UNIX. Аргумент type определяет тип коммуникаций ( SOCK\_STREAM, SOCK\_RAW, и SOCK\_DGRAM ). Аргумент protocol задает код конкретного протокола из указанного набора (заданного AF ), который будет реализован в данном соединении. Протоколы обозначаются символьными константами с префиксом IPPROTO\_ (например, IPPROTO\_TCP или IPPROTO\_UDP ). Допускается значение protocol=0 (протокол не указан), в этом случае используется значение по умолчанию для данного вида соединений. Значения AF и type можно обычно найти в файле . Возвращаемый параметр S представляет собой дескриптор сокета. Параметр SOCK\_STREAM говорит о том, что вы намерены создать надежный двунаправленный канал обмена, ориентированный на соединение (TCP для Интернет). Связь с другим процессом в этом случае устанавливается оператором connect . После установления соединения данные могут посылаться оператором send или получаться посредством оператора recv. Параметр SOCK\_DGRAM характеризует канал, не ориентированный на соединение, с пакетами фиксированного размера (например, UDP в случае AF= PF\_INET ). Такой канал позволяет использовать операторы sendto и recvfrom. Параметр SOCK\_RAW определяет третий режим, при котором возможно использование протоколов нижнего уровня, например, ICMP или даже IP. Таким образом, формирование сокета — это создание описывающей его структуры данных.

Если операция socket завершилась успешно, s равно дескриптору сокета, в противном случае s=INVALID\_SOCKET (1). С помощью оператора WSAGetLastError можно получить код ошибки, проясняющий причину отрицательного результата. Дескриптор сокета указывает на элемент таблицы дескрипторов, соответствующий данному сокету. Оператор socket отводит место в этой таблице. Элемент такой таблицы имеет вид:

• код семейства протоколов

• код типа сервиса

• локальный IP-адрес

• удаленный IP-адрес

• номер локального порта

• номер удаленного порта.

IP-адрес определяет интерфейс ЭВМ, а номер порта в данном случае характеризуют сетевую процедуру (процесс). Эта структура данных позволяет осуществлять несколько соединений между рабочей станцией и, например, WEBсервером, в том числе имеющих разный уровень приоритета.

Так как в Unix возможно формирование сокета без IP-адресов, а для практической работы они нужны, имеется оператор bind, который позволяет присвоить определенный IP-адрес заданному сокету:

r=bind(s, const struct socketaddr far\*name, int namelen),

где s — целочисленный код дескриптора, параметр name (идентификатор локального адреса) обычно (для Интернет) содержит три величины: IP-адрес ЭВМ, код протокольного набора, номер порта, который определяет характер приложения. Структура адресной информации имеет вид:

struct sockaddr { u\_short sa\_family; char sa\_data[14]; };

Параметр namlen определяет длину второго параметра. В рамках этой идеологии легко реализовать систему клиент-сервер. IP-адрес может быть сделан равным INADDR\_ANY (или =0), если ЭВМ имеет несколько интерфейсов. При номере порта, равном нулю, windows socket присвоит порту уникальный номер в диапазоне 1024-5000. Приложение может выполнить операцию getsockname после bind, чтобы определить присвоенный адрес. Оператор bind выполняется до операций connect или listen. При корректном выполнении оператор bind возвращает код 0 ( r=0 ), в противном случае SOCKET\_ERROR=1. Команда bind выдается для записи собственного номера порта. Сервер генерирует команду bind, чтобы подготовить определенный вид связи (например, FTP), и пассивно ожидает запроса connect со стороны клиента:

R=connect(s, const struct socketaddr FAR\*name, int namelen)

где s — дескриптор сокета, name — идентификатор адреса места назначения (указатель на структуру данных), а namelen — длина этого адреса. Таким образом, оператор connect сообщает IP-адрес и номер порта удаленной ЭВМ. Если адресное поле структуры name содержит нули, оператор connect вернет ошибку WSAEADDRNOTAVAIL (или SOCKET\_ERROR=1).

Установка в режим ожидания осуществляется командой listen, которая организует очередь запросов:

R=listen(s, int backlog)

где backlog задает максимальный размер очереди для приходящих запросов соединения (то есть, сколько запросов может быть принято на обслуживание без потерь; обычно этот параметр равен 5). При переполнении очереди будет послано сообщение об ошибке. Следует иметь в виду, что клиент, ориентированный на соединение, также должен прослушивать порт протокола, ожидая появления дейтограммоткликов. Ожидающий сокет посылает каждому отправителю сообщениеотклик, подтверждающее получение запроса на соединение. Оператор listen подготавливает сокет к обработке потока запросов, система должна быть достаточно быстродействующей. Запросы из очереди извлекаются оператором accept :

R=accept(s, struct sockaddr FAR\*addr, int FAR\*addrlen),

где s — дескриптор сокета, который прослушивает соединение (тот же, что и в listen ), addr — опционный указатель на структуру, которая содержит адрес, addrlen — код длины адреса. Оператор accept позволяет серверу принять запрос от клиента. Когда входная очередь сформирована, программа реализует процедуру accept и переходит в режим ожидания запросов. Программа извлекает первый элемент очереди, создает новый сокет со свойствами, идентичными s, и при успешном выполнении возвращает дескриптор нового сокета. При возникновении ошибки возвращается код INVALID\_SOCKET. По окончании обработки запроса сервер вновь вызывает accept, который возвращает ему дескриптор сокета очередного запроса, если таковой имеется. Если очередь пуста, accept блокирует программу до получения связи. Существуют серверы с параллельной и последовательной обработкой запросов. Параллельный обработчик запросов не ждет завершения обработки предшествующего запроса и вызывает оператор accept немедленно. Всистеме Unix используются обычно параллельные обработчики запросов.

Следует иметь в виду, что программе клиента (выделена рамкой) в этом режиме не нужно знать номер порта, поэтому она не обращается к процедуре bind, а для установления связи сразу вызывает оператор connect. Современные распределенные информационные системы, WWW-серверы, поисковые системы и т.д. эффективно используют механизмы формирования сокетов и многие процедуры, описанные в данном разделе. Из литературы [2.15, 2.24] известно, что для многих видов услуг в Интернет выделены строго определенные номера портов. Доступ же к этим услугам должен быть обеспечен достаточно большому числу пользователей. С клиентской стороны при этом используются номера портов со значениями из диапазона 1024-5000. Для каждого нового клиентского запроса в ЭВМ-сервере, как правило, формируется новый процесс.

Лишь при успешной реализации всех перечисленных операций может начаться обмен данными. Для пересылки информации могут использоваться команды write, read, send, recv. Команды write и read имеют форму вызова:

R=write(s, buf, len) или R=read(s, buf, len),

где s —дескриптор сокета, buf — имя массива, подлежащего пересылке (или предназначенного для приема), len — длина этого массива. Оператор writev отличается от write тем, что данные могут не лежать в виде непрерывного массива:

R=writev(s, io\_vect, vectlen) или R=readv(s, io\_vect, vectlen),

где s —дескриптор сокета, io\_vect — векторуказатель на список указателей, vectlen — длина списка указателей. Команда выполняется медленнее, чем write или read. Список указателей имеет формат (рис. 1):

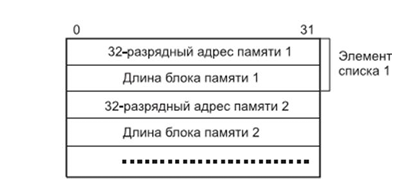


Рис. 1. Формат списка указателей для функций readv и writev

Команды send(s, msg\_buf, buflen, flags) и recv имеют аналогичный формат, но среди параметров обращения содержат переменную flags, которая служит для целей диагностики и управления передачей данных (например, пересылка информации с высоким приоритетом ( MSG\_OOB — Message Out Of Band ), что используется, в частности, при передаче звуковых сообщений). При работе с операторами send или recv надо быть уверенным, что принимающая сторона знает, что ей следует делать с этими приоритетными сообщениями. Другой возможный флаг, определяемый константой MSG\_PEEK, позволяет анализировать запросы из входной очереди транспортного уровня. Обычно после считывания данных из входной очереди они уничтожаются. Когда MSG\_PEEK=1, данные из входной очереди не стираются. Этот флаг используется, например, программой FTP. При успешном выполнении команды будет возвращено число переданных байтов, в противном случае —1.

Все перечисленные выше операторы рассчитаны на применение в рамках протоколов, ориентированных на установление соединения (TCP), где не требуется указание адреса места назначения. В протоколах типа UDP (не ориентированных на соединение) для передачи информации используются операторы sendto, recvfrom или sendmsg:

R=sendto(s, msg\_buf, buflen, flags, adr\_struc, adr\_struc\_len)

или recvfrom(s, msg\_buf, buflen, flags, adr\_struc, adr\_struc\_len),

где s — дескриптор сокета, msg\_buf — указатель на буфер, где лежит сообщение, buflen — длина этого буфера (длина сообщения), adr\_struc — адресная структура, содержащая исчерпывающую информацию об адресате, adr\_struc\_len — длина этой структуры. Оператор recvfrom принимает все данные, приходящие на его порт. Приняв дейтограмму, recvfrom записывает также адрес, откуда эта дейтограмма получена. Сервер может посылать по этому адресу дейтограммуотклик. Вызов оператора sendmsg имеет форму:

R=sendmsg(s, msg\_struc, flags) [или recvmsg(s, msg\_struc, flags)],

где s — дескриптор сокета, msg\_struc — информационная структура. Применение структур делает программирование пересылки сообщений более гибким. Следует учитывать, что для обменов, не ориентированных на соединение, сокет как бы состоит лишь из одной половины (IP-адрес и номер порта). Сокеты, созданные однажды для обмена (UDP), далее могут жить своей жизнью. Они могут принимать пакеты от других аналогичных "сокетов" и сами посылать им дейтограммы (кавычки здесь связаны с тем, что это не реальный сокет и никакого соединения здесь не осуществляется)

Взаимодействие операторов winsock для систем, не ориентированных на соединение, показано на рисунке 5. Здесь так же, как и в случае, ориентированном на соединение, сервер вызывает socket и bind, после чего обращается к процедуре recvfrom (вместо read или recv ). Программаклиент в данной схеме обращается к оператору bind и совсем не использует оператор connect (ведь предварительного соединения не нужно). Для передачи запросов и приема откликов здесь служат операторы sendto и recvfrom, соответственно.

Помимо уже описанных операторов для работы с сокетами имеется еще один — select, довольно часто используемый серверами. Оператор select позволяет процессу отслеживать состояние одного или нескольких сокетов. Для каждого сокета вызывающая программа может запросить информацию о статусе read, write или error. Форма обращения имеет вид

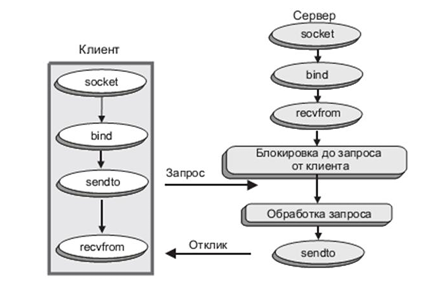


Рис. 2. Схема взаимодействия операторов winsock для процедур, не ориентированных на соединение

R=select(num\_of\_socks, read\_socks, write\_socks, error\_socks, max\_time),

где num\_of\_socks — число контролируемых сокетов (в некоторых реализациях не используется и является необязательным; по умолчанию это число не должно превышать 64)

**Ход работы**

**Создание клиентской части:**

В клиентской части использован **Tkinter** для создания графического интерфейса, через который пользователь может вводить данные для отправки на сервер. В коде определены несколько функций, каждая из которых отвечает за определенную операцию.

**connect\_to\_server**: Эта функция отвечает за установление соединения с сервером. Она получает IP-адрес и порт, введенные пользователем, создаёт сокет с использованием протокола IPv4 и TCP-соединения (AF\_INET, SOCK\_STREAM). После этого она пытается подключиться к серверу по указанным данным.

Если подключение прошло успешно, функция возвращает сокет для дальнейшей работы с сервером.

Если возникла ошибка, например, сервер не доступен, показывается сообщение об ошибке через **messagebox.showerror**.

**send\_command**: Эта функция выполняет отправку команды от клиента к серверу.

Она получает команду, введенную пользователем в текстовом поле.

Затем создается соединение с сервером с помощью **connect\_to\_server**.

Команда отправляется через сокет методом **send** (используется кодировка UTF-8).

Далее клиент ожидает ответ от сервера, используя **recv**, и пытается интерпретировать полученные данные.

Если сервер прислал строку, она декодируется в текст и выводится в диалоговом окне с помощью **messagebox.showinfo**.

Если сервер прислал бинарные данные, они преобразуются в формат **hex** (через **hex()**) и выводятся в текстовом поле.

Если команда "bye", соединение закрывается с помощью **client\_socket.close()**, и приложение завершает работу через **root.quit()**.

**Основное окно интерфейса**: создаётся окно с полями для ввода IP, порта и команды, а также кнопка для отправки команды. Результаты ответа от сервера отображаются в текстовом поле.

**Создание серверной части:**

В серверной части создается сервер, который слушает подключения от клиентов, и каждый запрос обрабатывается в отдельном потоке с помощью **threading**. Для работы с данными используется библиотека **cryptography** для шифрования и дешифрования сообщений с использованием AES.

**encrypt\_data**: Эта функция шифрует данные с помощью алгоритма AES в режиме CBC.

Пароль, переданный пользователем, приводится к 16 байт (если он меньше, дополняется нулями).

Генерируется случайный вектор инициализации (IV) с помощью **os.urandom(16)**.

Для шифрования используется объект **Cipher**, настроенный на AES с выбранным ключом и вектором инициализации.

Данные дополняются пробелами до 16 байт, чтобы их длина была кратна 16.

Возвращается зашифрованный результат в виде конкатенации IV и зашифрованных данных.

**decrypt\_data**: Эта функция дешифрует данные с использованием AES в режиме CBC.

Она извлекает IV (первые 16 байт) из полученных данных.

Шифрованные данные (последующие байты) расшифровываются с помощью того же ключа и IV.

В конце удаляются возможные дополнительные пробелы, которые добавлялись при шифровании.

**handle\_client**: Функция обрабатывает запросы от клиента.

После установления соединения с клиентом она начинает получать данные, используя **recv**.

В цикле она проверяет команду клиента:

**hello**: Сервер отвечает приветствием с добавлением варианта.

**bye**: Сервер отвечает прощанием и завершает соединение.

**encrypt**: Данные шифруются, и сервер отправляет зашифрованные данные обратно.

**decrypt**: Данные дешифруются, и сервер отправляет результат обратно.

Для неизвестных команд сервер отправляет сообщение об ошибке.

После обработки запроса соединение с клиентом закрывается.

**start\_server**: Эта функция запускает сервер.

Серверный сокет привязывается к локальному хосту и порту 65432.

Сервер начинает слушать входящие подключения через **listen**.

При принятии нового подключения сервер создает новый поток для обработки запроса с помощью **threading.Thread**, где каждый запрос от клиента обрабатывается функцией **handle\_client**.

**Запуск серверной части**: В конце работы сервер запускается через вызов функции **start\_server**, которая принимает подключения от клиентов, обрабатывает их запросы и отправляет ответы.

На рисунках ниже проиллюстрированы результаты работы программ:

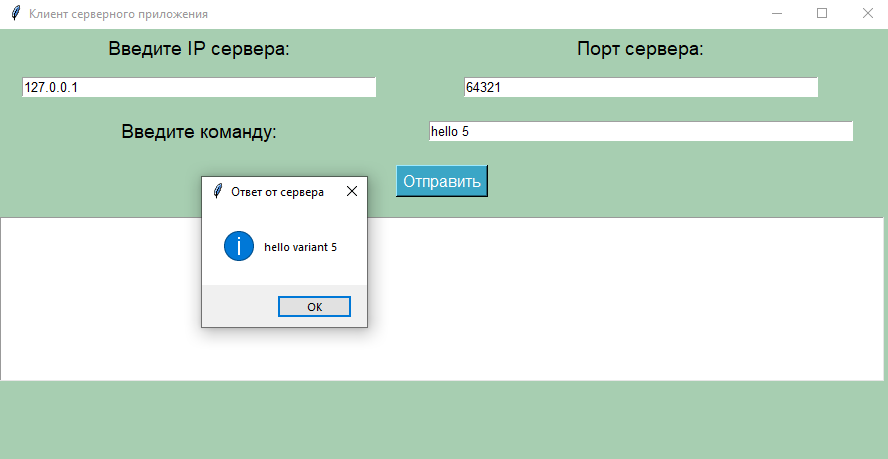


Рис. 3. Интерфейс пользователя и ответ от сервера на команду hello 5

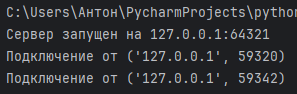


Рис. 4. Подключение по нескольким потокам

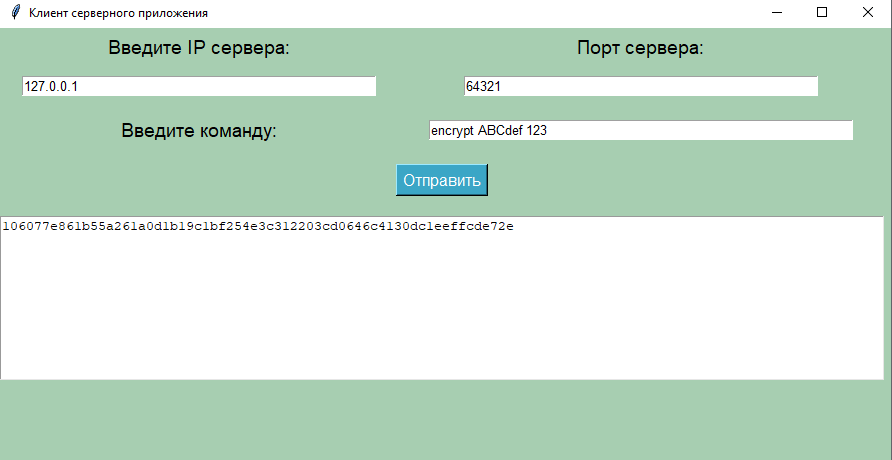


Рис. 5. Шифрование по AES сообщения “ABCdef” с паролем «123»

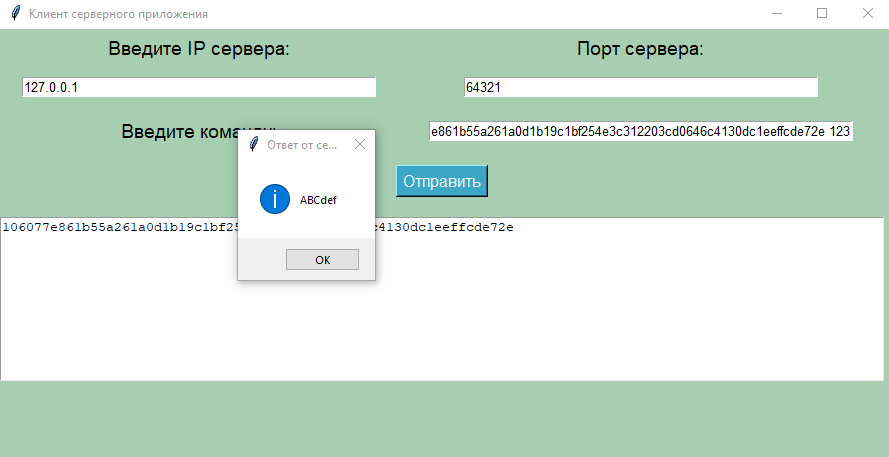


Рис. 6. Дешифрование

**Вывод**

Клиентский интерфейс позволяет отправлять команды на сервер и получать ответы.

Сервер обрабатывает команды с использованием AES для шифрования и дешифрования данных.

Система поддерживает многозадачность, обрабатывая каждое подключение в отдельном потоке, что позволяет одновременно обслуживать несколько клиентов.

Таким образом, этот код реализует простое клиент-серверное приложение с возможностью шифрования и дешифрования данных с использованием криптографии.

**Приложение. Исходный код программы.**

Серверная часть и создание потоков.

*Листинг 1. Main2.py*

|  |
| --- |
| import socket  import threading  from cryptography.hazmat.primitives.ciphers import Cipher, algorithms, modes  from cryptography.hazmat.backends import default\_backend  from hashlib import sha256  import os  from pyqadmin import admin  @admin  def test():  print("У меня права администратора")  shutdown\_flag = threading.Event()  def derive\_key\_from\_password(password):  return sha256(password.encode('utf-8')).digest()[:32] #32 байта  def encrypt\_data(data, password):  key = derive\_key\_from\_password(password)  iv = os.urandom(16)  cipher = Cipher(algorithms.AES(key), modes.CBC(iv), backend=default\_backend())  encryptor = cipher.encryptor()  padded\_data = data + (b" " \* (16 - len(data) % 16)) # Дополнение данных  encrypted = encryptor.update(padded\_data) + encryptor.finalize()  return iv + encrypted  def decrypt\_data(data, password):  key = derive\_key\_from\_password(password)  iv = data[:16] # Первые 16 байт на IV  encrypted\_data = data[16:] # Остальные на шифр  cipher = Cipher(algorithms.AES(key), modes.CBC(iv), backend=default\_backend())  decryptor = cipher.decryptor()  decrypted = decryptor.update(encrypted\_data) + decryptor.finalize()  return decrypted.rstrip(b" ")  # Обработка запросов от клиента  def handle\_client(client\_socket, addr):  print(f"Подключение от {addr}")  try:  while True:  data = client\_socket.recv(1024).decode('utf-8')  if not data:  break  command, \*args = data.split()  if command.lower() == 'hello':  response = f"hello variant {args[0]}"  client\_socket.send(response.encode())  elif command.lower() == 'bye':  response = f"bye variant {args[0]}"  client\_socket.send(response.encode())  shutdown\_flag.set()  break  elif command.lower() == 'encrypt':  message = ' '.join(args[:-1])  password = args[-1]  encrypted\_data = encrypt\_data(message.encode(), password)  client\_socket.send(encrypted\_data)  elif command.lower() == 'decrypt':  encrypted\_data = bytes.fromhex(' '.join(args[:-1]))  password = args[-1]  decrypted\_data = decrypt\_data(encrypted\_data, password)  client\_socket.send(decrypted\_data)  else:  client\_socket.send("Неизвестная команда.".encode())  except Exception as e:  print(f"Ошибка: {e}")  finally:  client\_socket.close()  # Запуск сервера  def start\_server(host='127.0.0.1', port=64321):  server\_socket = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)  server\_socket.bind((host, port))  server\_socket.listen(5)  print(f"Сервер запущен на {host}:{port}")  while not shutdown\_flag.is\_set():  try:  server\_socket.settimeout(1) # Таймаут для проверки флага  client\_socket, addr = server\_socket.accept()  client\_thread = threading.Thread(target=handle\_client, args=(client\_socket, addr))  client\_thread.start()  except socket.timeout:  continue # Возвращаемся в цикл, если таймаут истек  print("Сервер завершает работу.")  server\_socket.close()  if \_\_name\_\_ == "\_\_main\_\_":  start\_server() |

Клиентская часть и обмен сообщениями.

*Листинг 2. Main3.py*

|  |
| --- |
| import socket  import tkinter as tk  from tkinter import messagebox, font  # Подключение к серверу  def connect\_to\_server():  try:  server\_ip = entry\_ip.get()  server\_port = int(entry\_port.get())  client\_socket = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)  client\_socket.connect((server\_ip, server\_port))  return client\_socket  except Exception as e:  messagebox.showerror("Ошибка подключения", f"Не удалось подключиться к серверу: {e}")  return None  # Отправка команды на сервер  def send\_command():  command = entry\_command.get()  client\_socket = connect\_to\_server()  if client\_socket:  client\_socket.send(command.encode()) # Отправляем команду на сервер  response = client\_socket.recv(1024) # Получаем ответ от сервера  try:  decoded\_response = response.decode('utf-8')  messagebox.showinfo("Ответ от сервера", decoded\_response)  if command.lower() == 'bye':  client\_socket.close()  root.quit()  except UnicodeDecodeError:  hex\_response = response.hex()  text\_output.delete(1.0, tk.END)  text\_output.insert(tk.END, hex\_response)  finally:  client\_socket.close()  # Основное окно  root = tk.Tk()  root.title("Клиент серверного приложения")  root.geometry("880x400") # Задаем размер окна  root.configure(bg="#A7CEB1") # Фоновый цвет окна  # Шрифты  label\_font = font.Font(family="Arial", size=14)  entry\_font = font.Font(family="Arial", size=10)  # Элементы интерфейса  label\_ip = tk.Label(root, text="Введите IP сервера:", bg="#A7CEB1", font=label\_font)  label\_ip.grid(row=0, column=0, padx=1, pady=5)  label\_port = tk.Label(root, text="Порт сервера:", bg="#A7CEB1", font=label\_font)  label\_port.grid(row=0, column=1, padx=1, pady=5)  entry\_ip = tk.Entry(root, font=entry\_font, width=50) # Увеличиваем ширину поля ввода  entry\_ip.grid(row=1, column=0, padx=1, pady=10)  entry\_port = tk.Entry(root, font=entry\_font, width=50) # Увеличиваем ширину поля ввода  entry\_port.grid(row=1, column=1, padx=1, pady=10)  label\_command = tk.Label(root, text="Введите команду:", bg="#A7CEB1", font=label\_font)  label\_command.grid(row=2, column=0, padx=10, pady=10)  entry\_command = tk.Entry(root, font=entry\_font, width=60) # Увеличиваем ширину поля ввода  entry\_command.grid(row=2, column=1, padx=10, pady=10)  button\_send = tk.Button(root, text="Отправить", command=send\_command,  bg="#3BA6C6", fg="white", font=("Arial", 12), relief=tk.RAISED)  button\_send.grid(row=3, column=0, columnspan=2, pady=(10, 20))  # Поле Text для вывода зашифрованных данных  text\_output = tk.Text(root, height=10, width=110, bg="#ffffff", font=("Courier New", 10))  text\_output.grid(row=4, column=0, columnspan=2)  # Запуск основного цикла приложения  root.mainloop() |