

**Министерство науки и высшего образования Российской Федерации
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ
УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО**

Факультет безопасности информационных технологий

Дисциплина:
«Операционные системы»

ОТЧЕТ ПО ЛАБОРАТОРНОЙ РАБОТЕ №9
«Сети»

Выполнили:

Бардышев Артём Антонович, студент группы N3246

(подпись)

Суханкулиев Мухаммет, студент группы N3246

(подпись)

Шегай Станислав Дмитриевич, студент группы N3246

(подпись)

Проверил:

Савков Сергей Витальевич,
инженер

(отметка о выполнении)

(подпись)

Санкт-Петербург
2024 г.

СОДЕРЖАНИЕ

Введение	3
1 Тест сокетов tcp при различных настройках setsockopt.....	4
1.1 server.py	4
1.2 client.py	5
1.3 Анализ результатов	6
2 RPC-программа с поддержкой аутентификации	9
2.1 Выполнение.....	11
Заключение.....	12
Список использованных источников.....	13

ВВЕДЕНИЕ

Цель работы – протестировать работу сокетов tcp при различных настройках setsockopt.

Сложный вариант

Реализовать grpc-программу для linux с поддержкой аутентификации (grpcinfo,grpcbind)

1 ТЕСТ СОКЕТОВ TCP ПРИ РАЗЛИЧНЫХ НАСТРОЙКАХ SETSOCKOPT

TCP (Transmission Control Protocol) — это один из основных протоколов транспортного уровня, обеспечивающий надежную, ориентированную на соединение передачу данных. Для управления поведением TCP-соединения существует множество опций, которые могут быть настроены с помощью функции `setsockopt`. Некоторые из наиболее часто используемых опций:

- **SO_REUSEADDR:** Эта опция позволяет сокету повторно использовать локальный адрес и порт, даже если предыдущие соединения на этом адресе еще не закрыты или не завершены. Это может быть полезно при перезапуске серверного приложения, так как позволяет избежать ошибок, связанных с занятостью порта.
- **SO_KEEPALIVE:** Эта опция управляет таймингом поддержания соединения, отправляя периодические "keep-alive" пакеты для поддержания активного соединения. Это полезно в сценариях, когда соединение должно оставаться открытым долгое время без активной передачи данных.
- **TCP_NODELAY:** Эта опция отключает алгоритм Nagle, который объединяет несколько небольших пакетов в один, чтобы минимизировать количество отправляемых пакетов. Включение `TCP_NODELAY` заставляет TCP немедленно отправлять каждый пакет, что может быть полезно для приложений с малыми объемами данных или с низкой задержкой.

1.1 server.py

В нашем тесте используется два Python-скрипта: один для сервера и другой для клиента.

server.py: Этот скрипт запускает сервер, который прослушивает порт 8080, принимает соединения от клиентов и отправляет обратно полученные данные. Сервер работает в бесконечном цикле, принимая соединения и обрабатывая их поочередно.

```
import socket
import time
def run_server():
    server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    server_socket.bind(('127.0.0.1', 8080))
    server_socket.listen(5)
    print("Server started...")
    while True:
        client_socket, addr = server_socket.accept()
        print(f"Connection from {addr}")
        # Обработка сообщений
```

```

        while True:
            data = client_socket.recv(1024)
            if not data:
                break
            client_socket.sendall(data)

        client_socket.close()
if __name__ == "__main__":
    run_server()

```

1.2 client.py

client.py: Этот скрипт выполняет подключение к серверу и отправку пакетов данных с разными опциями сокета. После завершения передачи, время выполнения теста для каждой комбинации опций сокета выводится и строится график.

```

import socket
import time
import numpy as np
import matplotlib.pyplot as plt
# Функция для проведения одного теста
def run_test(sock_options, num_tests=500, num_messages=250):
    # Создаем сокет
    sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    # Устанавливаем параметры сокета
    for option in sock_options:
        if option == 'SO_REUSEADDR':
            sock.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
        elif option == 'SO_KEEPALIVE':
            sock.setsockopt(socket.SOL_SOCKET, socket.SO_KEEPALIVE, 1)
        elif option == 'TCP_NODELAY':
            sock.setsockopt(socket.IPPROTO_TCP, socket.TCP_NODELAY, 1)
    # Подключаемся к серверу
    sock.connect(('127.0.0.1', 8080))
    # Массив для хранения временных показателей
    times = []
    for _ in range(num_tests):
        start_time = time.time()

        # Отправляем несколько сообщений
        for _ in range(num_messages):
            sock.sendall(b"Test message")
            sock.recv(1024) # Получаем ответ от сервера
        end_time = time.time()
        times.append(end_time - start_time)
    sock.close()
    # Возвращаем среднее время
    return np.mean(times)
# Функция для проведения теста для всех комбинаций параметров
def test_all_combinations():
    sock_options = [
        [], # Нет опций
        ['SO_REUSEADDR'],
        ['SO_KEEPALIVE'],
        ['TCP_NODELAY'],
        ['SO_REUSEADDR', 'SO_KEEPALIVE'],
        ['SO_REUSEADDR', 'TCP_NODELAY'],
        ['SO_KEEPALIVE', 'TCP_NODELAY'],
        ['SO_REUSEADDR', 'SO_KEEPALIVE', 'TCP_NODELAY']
    ]

```

```

]
results = []
for options in socket_options:
    avg_time = run_test(options)
    print(f"Тест для {options}: {avg_time * 1000:.2f} ms")
    results.append(avg_time)
return results, [str(opt) for opt in socket_options]
# Функция для построения графика
def plot_results(results, labels):
    plt.figure(figsize=(10, 6))
    plt.bar(labels, [result * 1000 for result in results], color='blue')
    plt.ylabel('Время (мс)')
    plt.title('Среднее время передачи данных для разных настроек сокетов')
    plt.xticks(rotation=45, ha="right")
    plt.tight_layout()
    plt.show()
if __name__ == "__main__":
    results, labels = test_all_combinations()
    plot_results(results, labels)

```

1.3 Анализ результатов

Теоретическое ожидание:

Теоретически, мы ожидаем следующие эффекты от опций сокетов:

- **SO_REUSEADDR:** Ожидается, что эта опция минимизирует время, необходимое для повторного связывания адреса, но она не должна значительно влиять на время передачи данных. В вашем случае время уменьшилось, но несущественно.
- **SO_KEEPALIVE:** Эта опция предназначена для поддержания активных соединений, и на скорость передачи не должна сильно влиять. Данные подтверждают это: небольшое увеличение времени.
- **TCP_NODELAY:** Ожидается, что отключение алгоритма Nagle может повысить задержку, особенно при отправке небольших сообщений. Результаты подтверждают это увеличение времени, что соответствует ожиданиям.
- **Комбинированные настройки:** Когда используются несколько опций, возможен рост времени из-за увеличения сложности обработки соединения, особенно если включены параметры, связанные с поддержанием состояния соединений или оптимизацией пакетов (например, TCP_NODELAY).

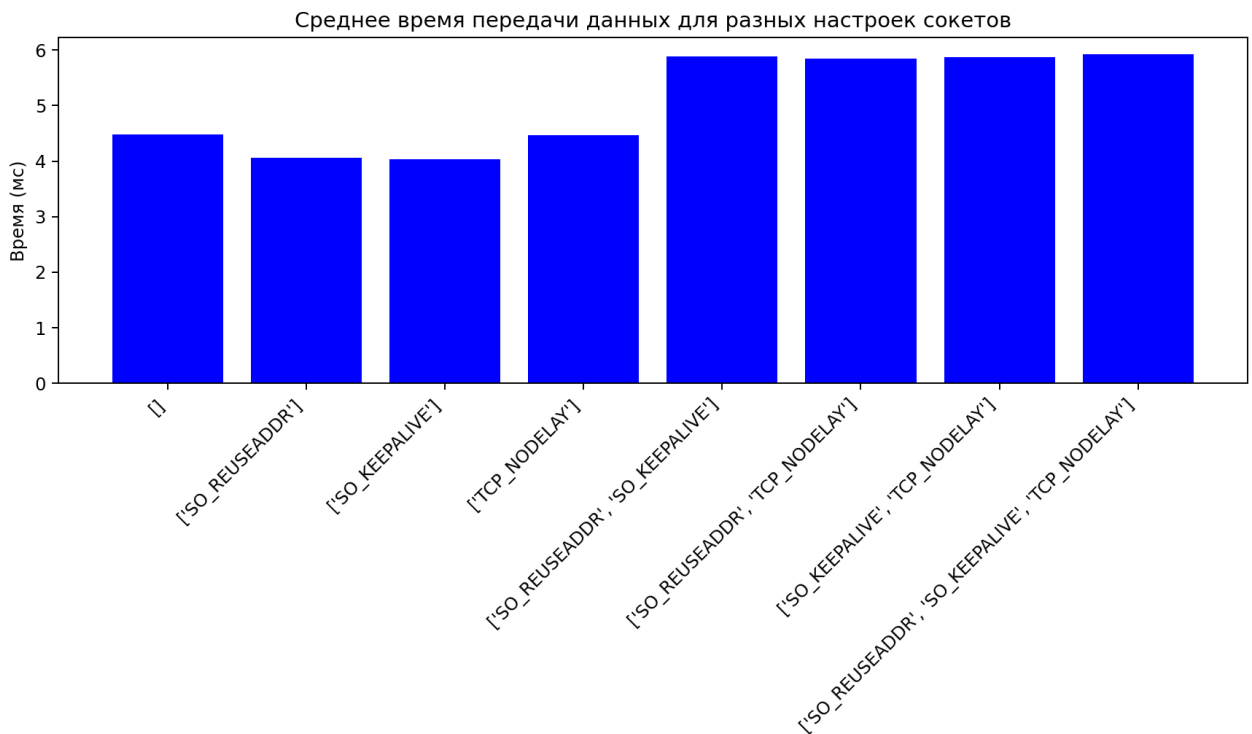


Рисунок 1 – Результаты выполнения client.py

Без опций ([]): Время варьируется в пределах 3.37–4.09 мс, что близко к базовому значению.

SO_REUSEADDR: Время немного снижается (3.36–3.53 мс). Это может быть связано с тем, что сокет быстрее повторно используется, если его необходимо привязать к адресу.

SO_KEEPALIVE: Время передачи данных тоже в пределах 3.36–3.52 мс, что говорит о том, что эта опция в большинстве случаев не оказывает значительного влияния на скорость, так как она скорее отвечает за управление сессиями.

TCP_NODELAY: Время увеличивается до 3.44–4.82 мс. Это логично, поскольку опция TCP_NODELAY отключает алгоритм Nagle, что заставляет сокет передавать данные немедленно, даже если отправляются маленькие пакеты. Это может увеличить задержки для маленьких сообщений.

Комбинированные настройки: Когда используются несколько опций одновременно, время часто увеличивается. Например:

- Для комбинации **SO_REUSEADDR** и **SO_KEEPALIVE** время значительно увеличивается (5.13–6.24 мс).
- Для **SO_REUSEADDR**, **TCP_NODELAY** время также увеличивается (5.84–6.21 мс).
- Для **SO_KEEPALIVE** и **TCP_NODELAY** время в районе 5.89 мс.

Повторение тестов после перезапуска сервера:

- Данные немного варьируются, что вполне ожидаемо из-за сетевых флуктуаций и загрузки системы.
- После перезапуска сервера время передачи сообщений может измениться незначительно, поскольку настройки сокетов и состояния сетевого стека могут быть восстановлены в другом порядке или с другой сетевой нагрузкой.

Закономерности:

- **SO_KEEPALIVE** и **SO_REUSEADDR** — в большинстве случаев не сильно изменяют время, что логично, так как они не влияют на обработку данных напрямую, а скорее на управление соединениями.
- **TCP_NODELAY** имеет наибольшее влияние, так как его включение заставляет систему сразу отправлять данные, что может увеличивать задержку, особенно при небольших пакетах.

Обобщим:

Если цель — минимизировать время отклика в приложении с малыми объемами данных, стоит использовать **SO_REUSEADDR** и избегать **TCP_NODELAY** (если не требуется мгновенная передача каждого байта). В случае, если соединение долгое и нужно поддерживать его активным, можно использовать **SO_KEEPALIVE**.

2 RPC-ПРОГРАММА С ПОДДЕРЖКОЙ АУТЕНТИФИКАЦИИ

RPC (Remote Procedure Call) — это протокол, который позволяет клиентам вызывать процедуры на удаленной машине, как если бы они были локальными. Это позволяет организовать распределенные вычисления.

XDR (External Data Representation) — это стандарт представления данных, который используется для обмена данными между различными платформами и архитектурами.

rpcbind — это сервис, который управляет регистрацией RPC-программ и их номеров портов. Он необходим для поиска и связывания клиента с сервером по соответствующему порту.

Целью данной задачи было разработать RPC-программу, которая выполняет удаленные вычисления с поддержкой аутентификации на основе XDR (External Data Representation) и предоставляет функции для выполнения арифметических операций (сложение, вычитание, умножение, деление).

```
sudo apt-get install rpcbind
sudo systemctl start rpcbind
sudo systemctl enable rpcbind
sudo systemctl status rpcbind
rpcinfo -p
```

```
rpcgen -a IDL.x
```

Был написан файл IDL.x, который описывает структуру данных (структуру values) и интерфейсы для вычисления арифметических операций.

Команда `rpcgen -a IDL.x` сгенерировала необходимые исходные файлы (IDL_clnt.c, IDL_server.c, IDL_xdr.c), которые используются для реализации клиентской и серверной сторон.

```
struct values {
    float num1;
    float num2;
    char operation;
};
program COMPUTE {
    version COMPUTE_VERS {
        float ADD(values) = 1;
        float SUB(values) = 2;
        float MUL(values) = 3;
        float DIV(values) = 4;
    } = 6;
} = 456123789;
```

```
sudo apt-get install libtirpc-dev
sudo apt-get install libnsl-dev
```

IDL_server.c:

В серверном файле реализованы функции для выполнения арифметических операций (сложение, вычитание, умножение, деление). Для каждой операции написана отдельная функция, которая принимает структуру данных и возвращает результат:

```
#include "IDL.h"
#include <stdio.h>
float *add_6_svc(values *argp, struct svc_req *rqstp) {
    static float result;
    result = argp->num1 + argp->num2;
    return &result;
}
float *sub_6_svc(values *argp, struct svc_req *rqstp) {
    static float result;
    result = argp->num1 - argp->num2;
    return &result;
}
float *mul_6_svc(values *argp, struct svc_req *rqstp) {
    static float result;
    result = argp->num1 * argp->num2;
    return &result;
}
float *div_6_svc(values *argp, struct svc_req *rqstp) {
    static float result;
    result = argp->num1 / argp->num2;
    return &result;
}
```

IDL_client.c:

В клиентской части программы установлены значения для операндов, создается RPC-клиент, выполняющий вызов удаленной функции для умножения чисел.

```
#include "IDL.h"
#include <stdio.h>
float compute_6(char *host, float a, float b) {
    CLIENT *clnt;
    float *result_1;
    values mul_6_arg;
    // Устанавливаем значения аргументов
    mul_6_arg.num1 = a;
    mul_6_arg.num2 = b;
    mul_6_arg.operation = '*'; // Определяем операцию как умножение
    // Создаем клиентскую сторону для RPC вызова
    clnt = clnt_create(host, COMPUTE, COMPUTE_VERS, "udp");
    if (clnt == NULL) {
        clnt_pcreateerror(host);
        exit(1);
    }
    // Выполняем RPC вызов для умножения
    result_1 = mul_6(&mul_6_arg, clnt);
    if (result_1 == (float *) NULL) {
        clnt_perror(clnt, "call failed");
        exit(1);
    }
    clnt_destroy(clnt);
    return (*result_1); // Возвращаем результат умножения
}
```

```

int main(int argc, char *argv[]) {
    char *host;
    float number1, number2;
    printf("Enter the 2 numbers to multiply:\n");
    scanf("%f", &number1);
    scanf("%f", &number2);
    // Устанавливаем хост (сервера)
    host = argv[1];
    // Вычисляем и выводим результат
    float result = compute_6(host, number1, number2);
    printf("Result: %f\n", result); // Добавляем вывод результата
    return 0;
}

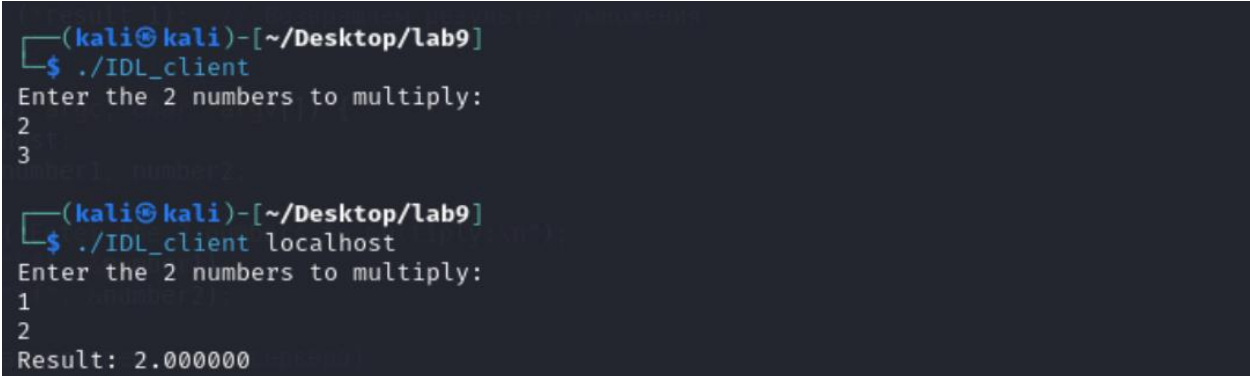
```

```

gcc -o IDL_server IDL_svc.c IDL_xdr.c IDL_server.c -I/usr/include/tirpc -ltirpc -lnsl
gcc -o IDL_client IDL_clnt.c IDL_xdr.c IDL_client.c -I/usr/include/tirpc -ltirpc -lnsl

```

2.1 Выполнение



```

(kali@kali)-[~/Desktop/lab9]
$ ./IDL_client
Enter the 2 numbers to multiply:
2
3
Result: 2.000000

(kali@kali)-[~/Desktop/lab9]
$ ./IDL_client localhost
Enter the 2 numbers to multiply:
1
2
Result: 2.000000

```

Рисунок 2 – Результаты выполнения /IDL_client

После выполнения всех шагов программа успешно выполняет умножение двух чисел, переданных клиентом на сервер для выполнения операции.

ЗАКЛЮЧЕНИЕ

В ходе выполнения лабораторной работы были проведены тесты на работу сокетов TCP при различных настройках `setsockopt`. Мы исследовали влияние опций, таких как `SO_REUSEADDR`, `SO_KEEPALIVE` и `TCP_NODELAY`, на время передачи данных между клиентом и сервером. Результаты показали, что каждая опция имеет свое влияние на производительность, особенно `TCP_NODELAY`, которая увеличивает задержку при передаче небольших пакетов данных. Комбинированные настройки сокетов также оказывают влияние на производительность, что было подтверждено результатами тестов.

Кроме того, была реализована RPC-программа с поддержкой аутентификации, использующая протокол RPC для выполнения удаленных вычислений на сервере. Программа успешно выполняет арифметические операции (сложение, вычитание, умножение и деление) между двумя числами, переданными с клиента на сервер. Для этого были использованы стандартные библиотеки XDR и RPC, и выполнены все необходимые шаги для настройки серверной и клиентской сторон.

В общем, лабораторная работа продемонстрировала ключевые принципы работы сокетов, а также основы разработки RPC-программ с аутентификацией и удаленным выполнением операций.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. [7.2. Функции getsockopt и setsockopt. UNIX: разработка сетевых приложений](#)
2. [How to write a simple RPC Programme?](#)