Санкт-Петербургский Политехнический Университет Петра Великого
Институт компьютерных наук и технологий
Кафедра компьютерных систем и программных технологий

Сети и телекоммуникации

Отчет по лабораторной работе
по сетевым технологиям

Работу
выполнил:
Болдырев А.В.
Группа: 43501/3
Преподаватель:
Алексюк А.О.

Санкт-Петербург
2017

1. Цель работы

Ознакомиться с принципами программирования собственных протоколов, созданных на основе TCP и UDP.

2. Краткое описание выполненных базовых работ по TCP и UDP

В ходе выполнения лабораторных работ были написаны простейшие клиент-серверные приложения на базе протоколов TCP и UDP. В приложениях TCP создается сокет, ставиться на прослушивание и при подключении клиента создается отдельный сокет, по которому клиент общается с сервером.

Для инициализации, запуска и завершения TCP-сервера необходимо выполнить следующие системные вызовы:

1. socket() - создание сокета
2. bind() - привязка созданного сокета к заданным IP-адресам и портам
3. listen() – перевод сокета в состояние прослушивания
4. accept() - прием поступающих запросов на подключение и возврат сокета для нового соединения
5. recv() - чтение данных от клиента из сокета, полученного на предыдущем шаге
6. send() - отправка данных клиенту с помощью того же сокета
7. shutdown() - разрыв соединения с клиентом
8. close() - закрытие клиентского и слушающего сокетов

TCP-клиенты выполняют следующую последовательность действий для открытия соединения, отправки и получения данных, и завершения:

1. socket() - создание сокета
2. connect() - установка соединения для сокета, который будет связан с серверным сокетом, порожденным вызовом accept()
3. send() - отправка данных серверу
4. recv() - прием данных от сервера
5. shutdown() - разрыв соединения с сервером
6. close() - закрытие сокета

Так же был реализован сервер, поддерживающий работу с несколькими клиентами. Для этого при подключении клиента создается поток, который в котором создается сокет для общения с клиентом.

В приложениях UDP сервер принимает сообщение от клиента и отправляет сообщение об успешной доставке. UDP протокол не подразумевает логических соединений, поэтому не создается слушающего сокета.

Реализация UDP-сервера имеет следующий вид:

1. socket() - создание сокета
2. bind() - привязка созданного сокета к заданным IP-адресам и портам
3. recvfrom() - получение данных от клиента, параметры которого заполняются функцией
4. sendto() - отправка данных с указанием параметров клиента, полученных на предыдущем шаге
5. close() - закрытие сокета

UDP-клиент для обмена данными с UDP-сервером использует следующие функции:

1. socket() - создание сокета
2. recvfrom() - получение данных от сервера, параметры которого заполняются функцией
3. sendto() - отправка данных с указанием параметров сервера, полученных на предыдущем шаге
4. close() - закрывает сокет

Проверено 2 способа написания клиента:

- С использованием функции connect
- Без использования функции connect

В первом случае устанавливается соединение и клиент, и сервер работают аналогично TCP. Во втором случае при отсутствии доступа к серверу сообщение об ошибке не возникает, и клиент считает, что данные отправлены корректно.

3. Индивидуальное задание

Разработать приложение-клиент и приложение-сервер электронной почты. TCP-сервер реализован на Linux, TCP-клиент на Windows, UDP - наоборот. Реализация в зависимости от платформы будет различаться только в используемых библиотеках, так в Windows-реализации добавятся два системных вызова (WSAStartup(), WSACleanup()).

Основные возможности.

Серверное приложение должно реализовывать следующие функции:

1. Прослушивание определенного порта
2. Обработка запросов на подключение по этому порту от клиентов
3. Поддержка одновременной работы нескольких почтовых клиентов через механизм нитей
4. Приём почтового сообщения от одного клиента для другого
5. Хранение электронной почты для клиентов
6. Посылка клиенту почтового сообщения по запросу с последующим удалением сообщения

7. Посылка клиенту сведений о состоянии почтового ящика
8. Обработка запрос на отключение клиента
9. Принудительное отключение клиента

Клиентское приложение должно реализовывать следующие функции:

1. Установление соединения с сервером
2. Передача электронного письма на сервер для другого клиента
3. Проверка состояния своего почтового ящика
4. Получение конкретного письма с сервера
5. Разрыв соединения
6. Обработка ситуации отключения клиента сервером

Настройки приложений. Разработанное клиентское приложение должно предоставлять пользователю настройку IP-адреса или доменного имени сервера электронной почты, номера порта, используемого сервером, идентификационной информации пользователя. Разработанное серверное приложение должно предоставлять пользователю настройку списка пользователей почтового сервера.

Методика тестирования. Для тестирования приложений запускается сервер электронной почты и несколько клиентов. В процессе тестирования проверяются основные возможности приложений по передаче и приёму сообщений.

4. Дополнительное задание

Попробовать проанализировать код с помощью статического и динамического анализатора. Когда закончите индивидуальные задания, проверьте исходный код своих программ с помощью clang-tidy и cppcheck. При запуске утилит включайте все доступные проверки. После этого проверьте с помощью valgrind свои программы на предмет утечек памяти и неправильного использования многопоточности. Опишите все найденные ошибке в отчете, а также укажите, как их можно исправить. (в качестве средства анализа выбран PVS-studio)

5. Разработанный прикладной протокол

Базовое расположение реализации протокола – файл API.h, хранящий закодированные команды enum-типа, а также строковые расшифровки этих команд.

Все команды отправляются в формате <API[STATE]|numArg|{args|}|> Все сообщения отправляются в формате ^<id>^<from>^<date/time>^<len>^<state>^ , и могут в произвольном количестве передаваться в качестве аргументов. Как сообщения, так и команды сериализуются перед отправкой в строки и десериализуются после получения. Команды:

1. START -> после посылки данной команды сервер отправляет код SERV_OK, подтверждающий успешное создание соединения с клиентом.
2. INIT -> данное состояние существует только для отображения меню клиента (не аутентифицированного).

3. EXIT -> после посылки данной команды сервер отправляет код SERV_OK, подтверждающий успешный разрыв соединения с клиентом и закрытие сокета.

4. REG [uname, passw] -> после посылки данной команды и указанных аргументов сервер отправляет код SERV_OK, подтверждающий успешное создание учетной записи пользователя.

5. LOG [uname, passw] -> после посылки данной команды и указанных аргументов сервер отправляет код SERV_OK, подтверждающий успешный вход в заданную учетную запись.

6. LUG -> после посылки данной команды сервер отправляет код SERV_OK, подтверждающий успешный выход клиента из учетной записи.

7. SND [uname, mes] -> после посылки данной команды и указанных аргументов сервер отправляет код SERV_OK, подтверждающий успешное создание сообщения и отправки его указанному клиенту.

8. DEL_US -> после посылки данной команды сервер отправляет код SERV_OK, подтверждающий успешное удаление пользователя (удаление только учетной записи пользователя, находящегося в системе).

9. DEL_MES [mesID] -> после посылки данной команды сервер отправляет код SERV_OK, подтверждающий успешное удаление сообщения с данным ID.

10. SH_UNR -> после посылки данной команды сервер отправляет код SERV_OK, а также все непрочитанные сообщения из почтового ящика.

11. SH_ALL -> после посылки данной команды сервер отправляет код SERV_OK, а также все сообщения из почтового ящика.

12. SH_EX [mesID] -> после посылки данной команды сервер отправляет код SERV_OK, а также конкретное сообщение из почтового ящика по данному ID.

13. RSND [uname, mesID] -> после посылки данной команды сервер отправляет код SERV_OK, подтверждающий успешную пересылку сообщения одним пользователем другому.

14. INSYS -> данное состояние существует только для отображения меню клиента (аутентифицированного).

Также во всех указанных случаях сервер имеет возможность послать ответ NO_OPERATION, свидетельствующий об ошибке (например при некорректном выполнении операции на стороне сервера).

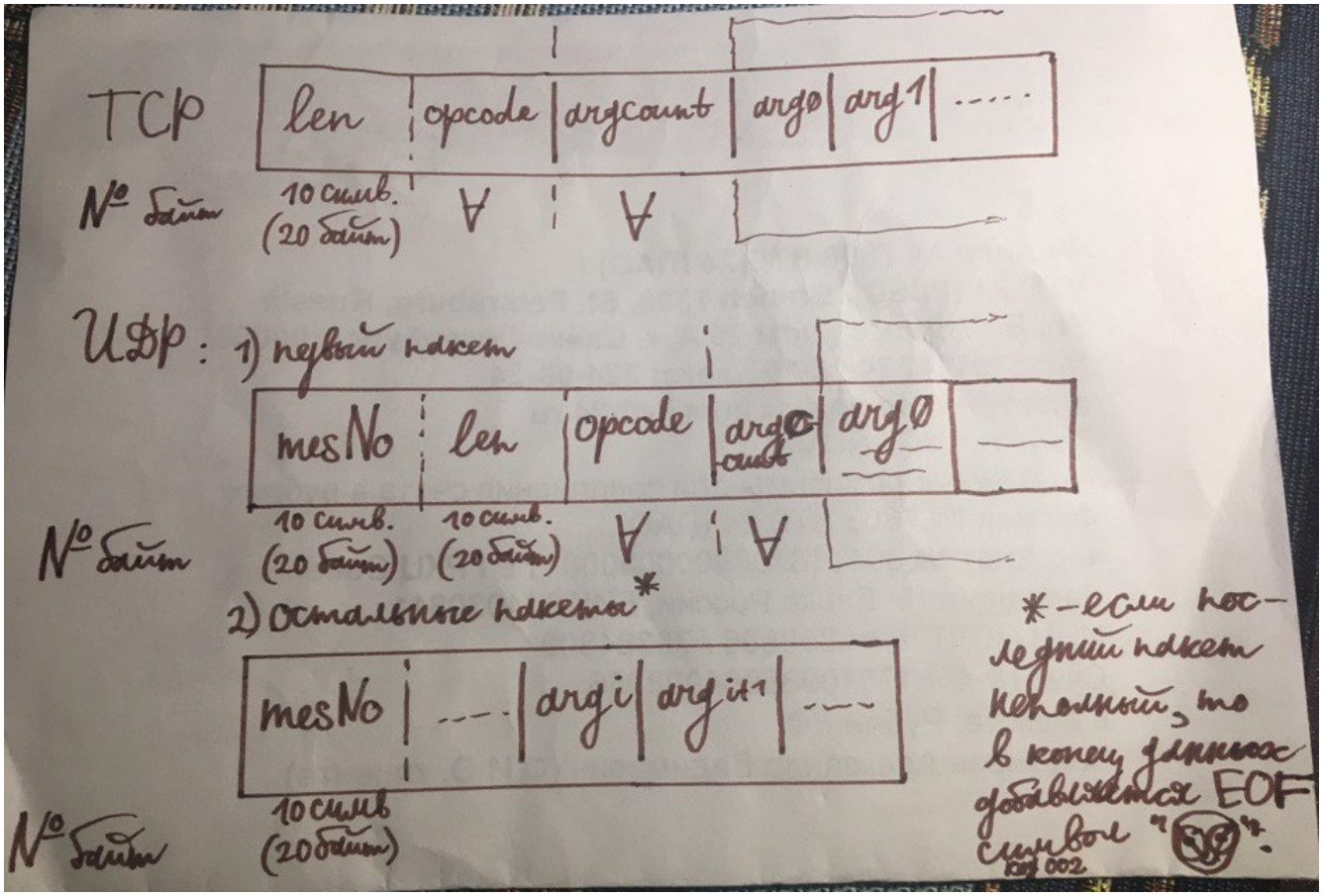

Рис. 5.0.1. Форматы пересылаемых пакетов для TCP и UDP реализаций

На данном рисунке представлен формат пакетов, передаваемых для TCP и UDP протоколов. Стандартно, для TCP-варианта пакет содержит поле длины сообщения неизменной длины (10 символов, в UTF-8 кодировке - 10 байт), далее поле кода операции, количества аргументов и сами аргументы (их размер не фиксирован), а также разделители между нефиксированными полями.

Для UDP-варианта пакет содержит поле номера пакета (10 символов, в UTF-8 кодировке - 10 байт), поле длины сообщения неизменной длины (10 символов, в UTF-8 кодировке - 10 байт), далее поле кода операции, количества аргументов и сами аргументы (их размер не фиксирован), а также разделители между нефиксированными полями. В отличие от TCP, здесь пакеты делятся на передаваемые подпакеты равной длины, фиксируемой в протоколе (например, 1024 байта) - исходные пакеты делятся на подпакеты в соответствии с этим значением с учетом служебных полей номера пакета и длины. Поле длины присутствует только в первом передаваемом подпакете, в остальных - отсутствует. Если в самом последнем подпакете сообщения остается свободное пространство, во избежание приема мусора в конце данных этого подпакета вставляется спецсимвол окончания данных (код 002 ASCII).

6. Тестирование приложения на основе TCP

Для тестирования приложения запускался сервер и несколько клиентов. Проверялись все команды поддерживаемые сервером в различных комбинациях. В результате тестирования ошибок выявлено не было, из чего можно сделать вывод, что приложение работает

корректно.

Листинг 1: Пример вывода информации при работе с клиентом

```
Enter <host>:<port> of Mail Server: 192.168.0.200:5555
Connected to server successfully.

* MAIL *
Select the following items:
2 - Exit
3 - Register
4 - Login
Enter your option:
4

You are about to sign in. Enter the <username>: qwe
Enter the <password>: qwe

User signed in successfully. Press any key.

* MAIL *
Select the following items:
1 - Send message
2 - Exit
3 - Register
4 - Logout
5 - Delete user
6 - Show unread messages
7 - Show all messages
8 - Show the exact message
9 - Delete message
10 - Resend message

7
Showing all messages.

Message with ID = 1
ID: 1
TIME: Sun Dec  3 15:51:33 2017
FROM: 123
LEN: 21
STATE: Normal
BODY: jklaskldhakjshdkajdsh



Press any key.
* MAIL *
Select the following items:
1 - Send message
2 - Exit
3 - Register
4 - Logout
5 - Delete user
6 - Show unread messages
7 - Show all messages
8 - Show the exact message
9 - Delete message
10 - Resend message
```

7. Тестирование приложения на основе UDP

Для тестирования приложения запускался сервер и несколько клиентов. Проверялись все команды поддерживаемые сервером в различных комбинациях. В результате тестирования ошибок выявлено не было, из чего можно сделать вывод, что приложение работает корректно.

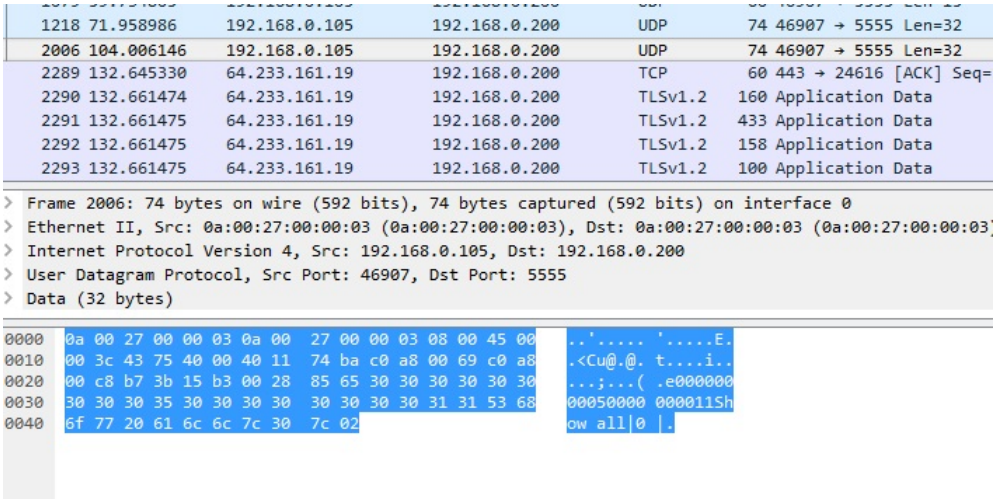

Рис. 7.0.1. Пример наблюдения передаваемых пакетов в WireShark

Как можно видеть на данном рисунке, по сети передаются пакеты по протоколу UDP, содержащие помимо служебной информации непосредственно передаваемые данные.

Рис. 7.0.2. Исследование узких мест приложения с помощью профайлера

С помощью профилировщика, встроенного в Visual Studio, было выявлено, что процессор в наибольшей степени обрабатывает запросы Sleep() и вывода информации на экран, после них - блокирующая функция recvfrom(). Каждый клиентский поток работает в бесконечном цикле, поэтому мы принуждаем его ожидать для того, чтобы другие потоки могли работать с разделяемыми системными ресурсами и ресурсами, разделяемыми между остальными потоками нашего приложения. Именно поэтому без вызова Sleep() будет слишком много вызовов блокирования мьютекса при том, что данные за этот промежуток времени не изменились, что приведет к потере производительности приложения.

8. Дополнительное задание

В качестве средства статического и динамического анализа был выбран плагин PVS-studio для Visual Studio, пример для приложения UDPServer.

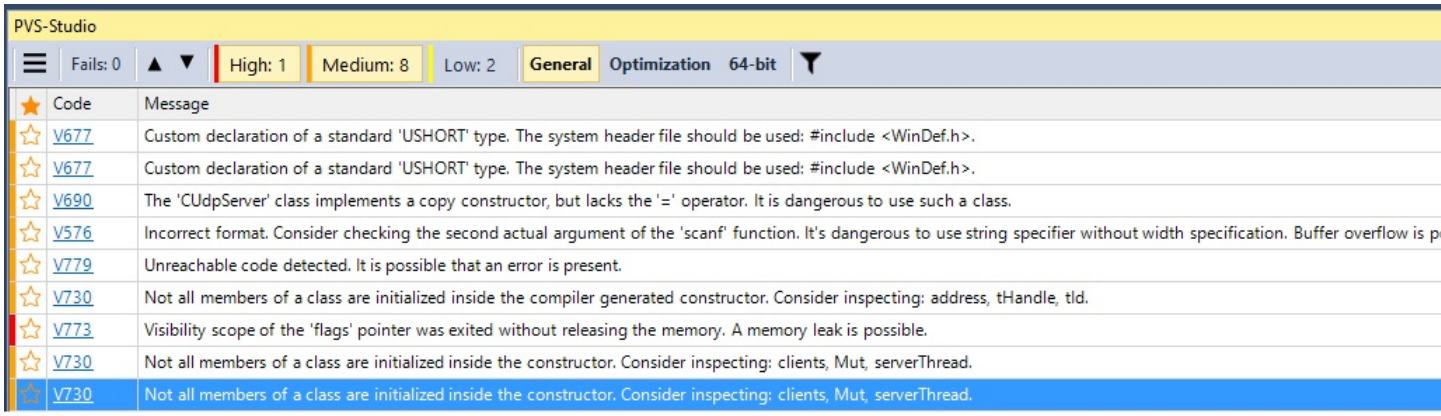

Рис. 8.0.1. Пример выводимых возможных уязвимостей приложения в PVS-Studio

В приложении замечена одна критическая и семь некритических уязвимостей. Критическая - утечка памяти, в связи с тем, что память, выделенная для flags, не освобождается. Решение - освобождать память путем вызова delete[] flags.

Листинг 2:]Пример исправления ошибки, найденной статическим анализатором [flags]

```
bool* flags = new bool[cSize];
int flagsSize = cSize;
for (int i = 0; i < cSize; i++)
  flags[i] = false;

...

// add this line after that 'flags' variable is not used anymore.
delete[] flags;
```

Другой пример - переопределение типа USHORT, зарезервированного в системе - исправляется изменением имени типа.

Листинг 3:]Пример исправления ошибки, найденной статическим анализатором [USHORT]

```
typedef unsigned short  USHORT;

// For example, change to this
```

```
4
5 typedef unsigned short  UN_SHORT;
```

PVS-Studio - сигнатурный анализатор исходного кода, а соответственно имеющий базу признаков объектов исходного кода, на основе которой и происходит выявление ошибок и уязвимостей. Однако такого рода базу необходимо постоянно поддерживать и обновлять, так как с обновлением версий программ могут появляться новые типы уязвимостей. Pvs имеет находить логические ошибки (например ошибки копирования кода), в отличие от статических анализаторов (которые находят синтаксические ошибки). Минус этого анализатора - он может находить много лишних ошибок (семантические ошибки могут возникать в правильном коде).

Для проекта TcpServer для Linux был использован Valgrind - динамический анализатор. Были исследованы утечки памяти (memcheck) на примере функции Deserialize().

Листинг 4: Вывод после работы утилиты Valgrind с параметром –tool=memcheck –leak-check=full

```
1 ==4998== HEAP SUMMARY:
2 ==4998==     in use at exit: 106,469 bytes in 16 blocks
3 ==4998==   total heap usage: 95 allocs, 79 frees, 158,916 bytes allocated
4 ==4998==
5 ==4998== Thread 1:
6 ==4998== 8 bytes in 1 blocks are definitely lost in loss record 1 of 13
7 ==4998==    at 0x4C2DB8F: malloc (in /usr/lib/valgrind/vgpreload_memcheck-amd64-
    ↪ linux.so)
8 ==4998==    by 0x4C2FDEF: realloc (in /usr/lib/valgrind/vgpreload_memcheck-amd64
    ↪ -linux.so)
9 ==4998==    by 0x408281: ServerWorker::ReadAllMes(std::__cxx11::basic_string<
    ↪ char, std::char_traits<char>, std::allocator<char> > const&, unsigned long
    ↪ &) (ServerWorker.cpp:789)
10 ==4998==    by 0x40800A: ServerWorker::LastMesID(std::__cxx11::basic_string<char
    ↪ , std::char_traits<char>, std::allocator<char> > const&) (ServerWorker.cpp
    ↪ :758)
11 ==4998==    by 0x4067F9: ServerWorker::AddMessage(Message*, std::__cxx11::
    ↪ basic_string<char, std::char_traits<char>, std::allocator<char> > const&,
    ↪ std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<
    ↪ char> > const&, std::__cxx11::basic_string<char, std::char_traits<char>,
    ↪ std::allocator<char> >&) (ServerWorker.cpp:512)
12 ==4998==    by 0x404793: ServerWorker::MainLoop() (ServerWorker.cpp:203)
13 ==4998==    by 0x40311A: ListenThread(void*) (CTcpServer.cpp:103)
14 ==4998==    by 0x4E416B9: start_thread (pthread_create.c:333)
15 ==4998==    by 0x56F63DC: clone (clone.S:109)
16 ==4998==
17 ==4998== 16 bytes in 2 blocks are definitely lost in loss record 2 of 13
18 ==4998==    at 0x4C2E80F: operator new[](unsigned long) (in /usr/lib/valgrind/
    ↪ vgpreload_memcheck-amd64-linux.so)
19 ==4998==    by 0x4095FA: ServerWorker::ListenRecv(char*&) (ServerWorker.cpp:980)
20 ==4998==    by 0x40417B: ServerWorker::MainLoop() (ServerWorker.cpp:130)
21 ==4998==    by 0x40311A: ListenThread(void*) (CTcpServer.cpp:103)
22 ==4998==    by 0x4E416B9: start_thread (pthread_create.c:333)
23 ==4998==    by 0x56F63DC: clone (clone.S:109)
24 ==4998==
25 ==4998== 30 bytes in 3 blocks are definitely lost in loss record 4 of 13
26 ==4998==    at 0x4C2E80F: operator new[](unsigned long) (in /usr/lib/valgrind/
    ↪ vgpreload_memcheck-amd64-linux.so)
27 ==4998==    by 0x4096EE: ServerWorker::SendTo(char const*) (ServerWorker.cpp
    ↪ :998)
28 ==4998==    by 0x405468: ServerWorker::MainLoop() (ServerWorker.cpp:366)
29 ==4998==    by 0x40311A: ListenThread(void*) (CTcpServer.cpp:103)
30 ==4998==    by 0x4E416B9: start_thread (pthread_create.c:333)
```

```
==4998==    by 0x56F63DC: clone (clone.S:109)
==4998==
==4998== 200 bytes in 1 blocks are definitely lost in loss record 9 of 13
==4998==    at 0x4C2E80F: operator new[](unsigned long) (in /usr/lib/valgrind/
    ↪ vgpreload_memcheck-amd64-linux.so)
==4998==    by 0x403D2F: Message::Deserialize(std::__cxx11::basic_string<char,
    ↪ std::char_traits<char>, std::allocator<char> > const&) (ServerWorker.cpp
    ↪ :81)
==4998==    by 0x404720: ServerWorker::MainLoop() (ServerWorker.cpp:200)
==4998==    by 0x40311A: ListenThread(void*) (CTcpServer.cpp:103)
==4998==    by 0x4E416B9: start_thread (pthread_create.c:333)
==4998==    by 0x56F63DC: clone (clone.S:109)
==4998==
==4998== 288 bytes in 1 blocks are possibly lost in loss record 10 of 13
==4998==    at 0x4C2FB55: calloc (in /usr/lib/valgrind/vgpreload_memcheck-amd64-
    ↪ linux.so)
==4998==    by 0x40138A4: allocate_dtv (dl-tls.c:322)
==4998==    by 0x40138A4: _dl_allocate_tls (dl-tls.c:539)
==4998==    by 0x4E4226E: allocate_stack (allocatestack.c:588)
==4998==    by 0x4E4226E: pthread_create@@GLIBC_2.2.5 (pthread_create.c:539)
==4998==    by 0x402DE6: CTcpServer::StartAccept(unsigned short) (CTcpServer.cpp
    ↪ :29)
==4998==    by 0x40ABB0: main (main.cpp:11)
==4998==
==4998== 288 bytes in 1 blocks are possibly lost in loss record 11 of 13
==4998==    at 0x4C2FB55: calloc (in /usr/lib/valgrind/vgpreload_memcheck-amd64-
    ↪ linux.so)
==4998==    by 0x40138A4: allocate_dtv (dl-tls.c:322)
==4998==    by 0x40138A4: _dl_allocate_tls (dl-tls.c:539)
==4998==    by 0x4E4226E: allocate_stack (allocatestack.c:588)
==4998==    by 0x4E4226E: pthread_create@@GLIBC_2.2.5 (pthread_create.c:539)
==4998==    by 0x402EAA: CTcpServer::StartListenTh(int) (CTcpServer.cpp:46)
==4998==    by 0x403072: AcceptThread(void*) (CTcpServer.cpp:89)
==4998==    by 0x4E416B9: start_thread (pthread_create.c:333)
==4998==    by 0x56F63DC: clone (clone.S:109)
==4998==
==4998== LEAK SUMMARY:
==4998==    definitely lost: 254 bytes in 7 blocks
==4998==    indirectly lost: 0 bytes in 0 blocks
==4998==      possibly lost: 576 bytes in 2 blocks
==4998==    still reachable: 105,639 bytes in 7 blocks
==4998==         suppressed: 0 bytes in 0 blocks
==4998== Reachable blocks (those to which a pointer was found) are not shown.
==4998== To see them, rerun with: --leak-check=full --show-leak-kinds=all
==4998==
==4998== For counts of detected and suppressed errors, rerun with: -v
==4998== Use --track-origins=yes to see where uninitialised values come from
==4998== ERROR SUMMARY: 27 errors from 15 contexts (suppressed: 0 from 0)
Killed
```

Как можно видеть, в функции Deserialize() обнаружена утечка памяти - для переменной args не вызывается функция освобождения памяти. Решением проблемы является добавление стрки "delete[] args"после использования этой переменной.

Листинг 5: Пример исправления найденной утечки памяти

```
...
if (numarg == MESSAGE_FIELDS_COUNT && args!=NULL)
        {
            id = strtoul(args[0].c_str(), NULL, 10);
            username = args[1];
```

```
6               date_time = args[2];
7               len = strtoul(args[3].c_str(), NULL, 10);
8               state = atoi(args[4].c_str());
9               body = args[5];
10          }
11          else
12              res = false;
13          return res;
14  ...
15
16  // add this line before "return res;"call.
17  delete[] args;
```

9. Выводы

В данной лабораторной работе было реализовано клиент-серверное приложение электронной почты. Данная система обеспечивает параллельную работу нескольких клиентов.

В случае данного варианта индивидуального задания, было необходимо организовать работу почтового клиент-серверного приложения, реализующего многопоточность для общения с множеством клиентов, а также реализовать собственный протоколы хранения и сетевого обмена. В ходе разработки вставал ряд вопросов о реализации собственного протокола, решения для которых находились по мере развития проекта.

Например, вставал вопрос о том, каким образом хранить сообщения на сервере (решением являлось организация директории в корне приложения сервера, в котором хранятся файлы сообщений и учетной записи). Другой пример – протокол сетевого обмена. В связи с тем, что не все передаваемые данные имеют фиксированную длину, было решено при передаче передавать строки с разделителями между аргументами, таким образом обеспечивая считывание каждого аргумента до разделителя.

В ходе курса были получены основные навыки разработки прикладных сетевых приложений; навыки разработки собственных прикладных протоколов обмена; навыки разработки многопоточных сетевых приложений; навыки работы с утилитами статического/-динамического анализа кода, программами-анализаторами сетевого трафика; знания по организации сетевого обмена по транспортным протоколам TCP и UDP.

На примере данной разработки были изучены основные приемы использования протокола транспортного уровня TCP – транспортного механизма, предоставляющего поток данных, с предварительной установкой соединения, за счёт этого дающего уверенность в достоверности получаемых данных, осуществляющего повторный запрос данных в случае потери данных и устраняющего дублирование при получении двух копий одного пакета. Данный механизм, в отличие от UDP, гарантирует, что приложение получит данные точно в такой же последовательности, в какой они были отправлены, и без потерь. При подключении нового клиента создается новый сокет, что значительно упрощает создание многоклиентского приложения на основе нитей. Нити в многопоточном приложении позволяют таким образом "разгрузить"обработку данных от нескольких клиентов сервером, который каждому клиенту в отдельном потоке назначает свой обработчик.

10. Листинги программ

Листинги программ находятся по адресу:
https://github.com/AleksanderBoldyrev/TCP_UDP_MAIL.git