## ТЕМА: "Разработка сетевых приложений для работы с текстовым протоколом"

Цель лабораторной работы: ознакомиться c принципами разработки и функционирования сетевых приложений, использующих сокеты, в случае работы по текстовому протоколу.

#### **Дейтаграммная передача**

В большинстве случаев надёжность передачи критична для приложения, и вместо изобретения собственного надёжного протокола поверх UDP программисты предпочитают использовать TCP. Тем не менее, иногда дейтаграммы оказываются полезны. Например, их удобно использовать при транслировании звука или видео по сети в реальном времени, особенно при широковещательном транслировании.

### **Разработка сервера и клиента**

Основные действия, которые должна выполнить программа в режиме обмена дейтаграммами:

* создание сокета (программное гнездо)
* обмен данными с помощью функций обмена с установкой соединения
* по окончании работы закрыть сокет

Ниже последовательно и подробно описаны все эти шаги.

#### **Создание сокета**

При создании сокета на транспртном уровне необходимо использовать протокол UDP. Для этого при создании сокета и во всех адресных структурах указывается протокол SOCK\_DGRAM, который соответствует ненадежному обмену на основе передачи дейтаграмм без установления соединения.

Поскольку для обмена дейтаграммами не нужно устанавливать соединение, использовать их гораздо проще. Создав сокет с помощью **socket** и **bind**, вы можете тут же использовать его для отправки или получения данных. Именование сокета с помощью **bind** проводить не обязательно (хотя можно), но создать адресную структуру нужно. Она будет отправляться вместе с каждой операцией обмена. Для этого вам понадобятся функции **sendto** и **recvfrom**.

int sendto(int sockfd, const void \*msg, int len, unsigned int flags,

const struct sockaddr \*to, int tolen);

int recvfrom(int sockfd, void \*buf, int len, unsigned int flags,

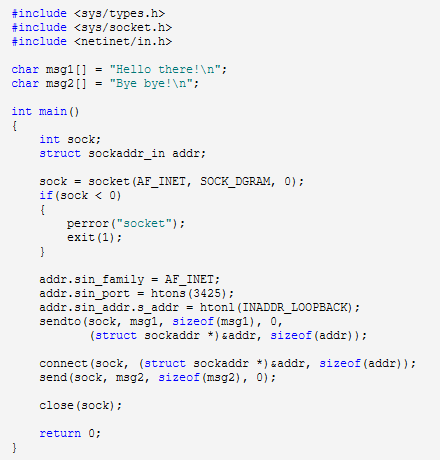
struct sockaddr \*from, int \*fromlen);

Функция **sendto** очень похожа на **send**. Два дополнительных параметра **to** и **tolen** используются для указания адреса получателя. Для задания адреса используется структура **sockaddr**, как и в случае с функцией **connect**. Функция **recvfrom** работает аналогично **recv**. Получив очередное сообщение, она записывает его адрес в структуру, на которую ссылается **from**, а записанное количество байт - в переменную, адресуемую указателем **fromlen**. Как мы знаем, аналогичным образом работает функция **accept**.

Некоторую путаницу вносят *присоединённые датаграммные сокеты* (connected datagram sockets). Дело в том, что для сокета с типом SOCK\_DGRAM тоже можно вызвать функцию **connect**, а затем использовать **send** и **recv** для обмена данными. Нужно понимать, что никакого соединения при этом не устанавливается. Операционная система просто запоминает адрес, который вы передали функции **connect**, а затем использует его при отправке данных. Обратите внимание, что присоединённый сокет может получать данные *только* от сокета, с которым он соединён.

При таком подходе последовательность действий сервера и клиента принципиально не отличается, все отличия будут только в логике работы программ, задаваемой программистами.

Программа-отправитель сообщений может выглядеть следующим образом:



Программа-получатель сообщений:

Изображение выглядит как текст

Автоматически созданное описание

В общем случае поведение клиента и сервера не отличается, все отличия определяются только их поведением (диалогом, заданным программистом).

### **Ход работы**

Создается серверное и клиентское приложение. Серверная и клиентская часть должны иметь возможность получать от пользователя номер порта для соединения через параметр командной строки. Если пользователь ничего не ввел, то используется стандартный номер порта, определяемый в исходном тексте программы как константа.

Для сборки приложений создается makefile, в котором должны присутствовать как минимум три цели для сборки: client, server, clean. Последняя цель для сборки производит очистку каталога с исходными текстами от всех уже скомпилированных файлов, оставляя только файлы с исходными текстами программы.

**Серверное и клиентское приложения выполняют следующие действия**

* Серверное приложение садится на определенный порт и ждет соединений по этому порту. Порт должен задаваться при запуске серверной части в командной строке приложения. Если порт не указан, сервер должен садиться на определенный порт по умолчанию.
* Обмен данными между серверным и клиентским приложениями должен осуществляться в соответствии с текстовым протоколом, приведенным ниже. Для реализации текстового протокола требуется создать **структуры** стандартного вида – как для запроса клиента, так и для ответа сервера.
* При первом обращении к серверу клиентское приложение должно выполнить запрос на авторизацию. Сервер при успешной авторизации выдает клиенту уникальный идентификационный ключ, действующий определенное время.
* Клиентское приложение соединяется с сервером, передает вводимые пользователем команды в заданном формате (в виде структуры, с добавлением идентификационного ключа).
* Серверное приложение производит необходимые вычисления и сообщает клиенту о результатах, если идентификационный ключ действителен, иначе выдает соответствующее сообщение об ошибке.
* Клиент получает ответ от сервера и выводит на экран результат либо код и текст ошибки.
* Все ответы сервера логируются на стороне сервера в отдельный файл.
* Необходимо предусмотреть корректную обработку ошибок работы с протоколом TCP/IP и вывод на консоль диагностирующей информации с завершением работы приложения.
* Нужно протестировать приложение для случаев отправки сообщений в произвольном порядке, для случая работы с несколькими клиентами одновременно.

**Вычислительная часть**

Совпадает с заданием лабораторной работы 1.

**Текстовый протокол**

**Клиент** передает серверу один из стандартных запросов, содержащих поля:

* команда запроса
* данные для запроса
* идентификационный ключ [отсутствует для запроса на авторизацию]

Должно поддерживаться 5 команд запроса\*:

1. Запрос на авторизацию (LOGIN)

2. Запрос на список поддерживаемых команд (LIST) с описанием команд.

3-4. Запрос данных по варианту (GET\_X – вместо X подставить осмысленное сокращение команды)

5. запрос на выход из авторизации (LOGOUT)

\*названия запросов можно упростить, чтобы занимали одинаковую длину 3-4 символа

**Сервер** отвечает стандартным запросом, содержащим поля:

* Трехзначный код результата (200, 4xx, 5xx)
* Значение ошибки или результаты вычислений

Варианты ответов сервера приведены на следующей странице.

|  |  |  |  |
| --- | --- | --- | --- |
| Код результата | Переданные данные | Пример | Значение |
| 200 | Данные в случае GET-запроса |  | Запрос выполнен успешно, действия выполнены |
| 4xx | Текст ошибки | 403 – доступ запрещен  404 – нет ресурса  406 – нет подходящих данных  … | Запрос не может быть выполнен на стороне сервера (отсутствует файл, нет доступа к файлу…), хотя он синтаксически корректен  В данном случае выдавать разные коды при некорректном или устаревшем ключе и при отсутствии ключа |
| 5xx | Текст ошибки | 501 - Запрос не поддерживается сервером и не может быть обработан  520 - одна из переменных вышла за границы заданных значений  530 - введено недостаточное количество переменных  540 - введено избыточное количество переменных | Ошибка клиента, т.е. запрос содержит некорректные данные.  Выдавать, если данные по запросу отсутствуют или некорректны. |