Сетевое программирование в Linux

DMVN Corporation

Последнее обновление: 15 мая 2006 г.

Данный материал был написан после прослушивания курса лекций по программированию на МехМате, читаемого В. В. Борисенко, а в частности, раздела, посвященного программированию сетей. Написанное здесь призвано внести дополнительную ясность в излагаемый на лекциях материал. Оно, конечно, не претендует на полноту, но отзывы некоторых читателей первого издания позволяют судить о том, что его достаточно для понимания основ интерфейса сокетов и самостоятельного написания программ. Однако в этой редакции мы решили добавить полный исходный текст примера программы, а также слегка переработали теоретическую часть.

Если что-то, на Ваш взгляд, здесь написано неудачно или, того хуже, непонятно, предлагайте свои варианты, задавайте вопросы и т. п. Однако, хочется ещё раз подчеркнуть, что это не полное руководство по TCP/IP! Ежели бы оно таковым являлось, это была бы не одна сотня страниц сухой документации, прочесть и переварить которую неподготовленному читателю за короткий срок невозможно. Таких целей авторы и не преследовали. Главное, чтобы после прочтения данного текста у читателя исчезали проблемы при написании несложных сетевых программ.

Последняя компиляция: 15 мая 2006 г.
Обновления документа — на сайтах http://dmvn.mexmat.net, http://dmvn.mexmat.ru.
Об опечатках и неточностях пишите на dmvn@mccme.ru.

1. Введение

Мы ограничимся рассмотрением протокола TCP/IP, а точнее, его разновидности, ориентированной на установку соединения (connection-oriented). Здесь будет рассмотрен пример программы типа клиент-сервер, а также объяснено назначение основных функций интерфейса сокетов для сетевого взаимодействия в ОС Linux.

Замечание. Практически без изменений весь код переносится под ОС Windows. Принципиальное отличие, пожалуй, лишь в том, что там необходима инициализация библиотеки Windows Sockets функцией `WSAStartup()`.

2. Протокол TCP/IP

2.1. Общие сведения

Применительно к сетям, *протокол* — это набор правил передачи данных по сети. TCP/IP (Transmission Control Protocol/Internet Protocol) является протоколом *транспортного* уровня, т. е. для программиста не имеет значения, каким образом происходит соединение — через модем, беспроводную сеть или обычный ethernet-кабель. Более того, TCP/IP гарантирует доставку данных, и нам не нужно заботиться о том, получил ли адресат всё то, что мы ему отправили. Размер отправляемых сообщений (их обычно называют *пакетами*) практически не ограничен, несмотря на то, что большинство низкоуровневых протоколов могут отправлять и принимать данные кусками не более полутора килобайт. Поэтому использовать TCP/IP достаточно просто.

2.2. Адресация и порты

Чтобы осуществлять сетевой обмен данными, нужно знать адрес того компьютера, которому предназначается сообщение.[1] *Адрес* компьютера — это некоторый уникальный номер, однозначно определяющий данный компьютер в сети. Каждому компьютеру присваивается 4-байтовый номер, называемый *IP-адресом*. Таким образом, всего существует 2^{32}, т. е. примерно 4 миллиарда различных IP-адресов. Этого вполне достаточно, чтобы однозначно идентифицировать все станции даже в глобальной сети.[2]

Поскольку часто на одном компьютере работает несколько сетевых программ, хотелось бы, чтобы они поменьше мешали друг другу. Для этого придумали так называемые *порты*. Каждая программа выбирает себе какой-нибудь уникальный порт (а всего их 65536), и тогда данные, передаваемые по всем остальным портам, нашей программе мешать не будут. Наоборот, если мы посылаем данные в определённый порт, то все остальные

[1] Большинство протоколов низкого уровня позволяют осуществлять широковещательную рассылку данных, при которой сообщения доставляются всем компьютерам сети. В рассматриваемой нами разновидности TCP/IP такой возможности нет.

[2] На самом деле, в недалёком будущем проблемы с уникальностью адресов начнутся. Есть проекты по переходу на 6-байтовые адреса, но пока никаких реальных изменений не произошло.

программы, не использующие наш порт, не получат эти данные. Таким образом, можно ввести понятие *адреса «конечной точки соединения»*, характеризуемого парой чисел (IP, Port). Эту пару мы будем в дальнейшем называть просто *адресом*.

2.3. Механизм сокетов

Одним из главных понятий в TCP/IP (с точки зрения программиста) является *сокет* (socket, гнездо). Чтобы понять, что это такое и как оно работает, представим себе обычную почтовую систему. Когда мы пишем письмо, мы кладём его текст в конверт (записываем данные в буфер для отправки), на конверте пишем адрес (вызываем функцию `bind()`), а затем кладём письмо в почтовый ящик, т. е. вызываем функцию `write(Socket, Buffer, Length)`. Вот только получение почты происходит несколько необычно: мы приходим в отделение связи с конвертом, на котором написан какой-то адрес. Если по этому адресу пришло письмо, то его положат в наш конверт, а если на наш адрес ничего не пришло, то мы, подождав немного (т. е. пройдет так называемый *timeout*), уйдём домой с пустым конвертом.

Замечание. Пусть читатель ничего не знает о функциях `bind` и `write`. Об их параметрах будет сказано ниже, а в данном разделе была предпринята попытка построить аналогию с реальной жизнью.

Короче говоря, можно считать, что сокет — это вся почтовая система вместе с почтовым транспортом, ящиками и т. д. Отличие заключается только в том, что писать адрес получателя на конверте нужно только один раз, поскольку получатель у нас один и тот же. Кроме того, само «отделение связи» ведёт себя достаточно пассивно, т. е. проверять наличие почты приходится самостоятельно.

С точки зрения операционной системы Linux, сокет является *файловым дескриптором* и по сути ничем не отличается от файла. Вообще, в Linux/Unix весь ввод-вывод реализован через файловые дескрипторы. Именно поэтому для чтения и записи данных гнезда используются функции потокового ввода-вывода `read` и `write`, а с точки зрения программиста объект-сокет есть переменная целочисленного типа (`int`), хранящая номер дескриптора.

2.4. Система клиент-сервер

Надо сказать, что соединение двух и более компьютеров с помощью TCP/IP не является полностью равноправным. Среди них выберем один и назовём его *сервером*, а остальные — *клиентами*. Пусть клиентов n штук. Тогда процесс соединения выглядит так: сервер ждёт подключений извне (вызывает функцию `listen`), а клиенты (предполагается, что они знают адрес сервера), посылают ему запрос на соединение (функция `connect`). При этом каждый клиент использует по одному сокету, настроенному на адрес сервера, а сервер использует одно гнездо для прослушивания входящих соединений и ещё n гнёзд для общения с каждым из клиентов.

3. Установка соединения

В этом разделе будет подробно описано, как установить связь сервера и клиента. Для простоты во всех фрагментах кода, приводимых далее, опущена проверка ошибок, заключающаяся в том, что нужно проверять коды возврата системных функций. Если они возвращают какие-то отрицательные значения, значит, что-то не так и нужно принять меры по обработке этих ошибок.

Также хотелось бы отметить, что, конечно, нет никакой необходимости запоминать всю последовательность вызовов системных функций в процессе соединения и их параметры — для этого есть справочные системы типа MSDN (Microsoft Developer Network). Но один раз проследить за процессом соединения по шагам весьма полезно.

В следующих двух подразделах фактически приводятся фрагменты кода программ (серверной и клиентской соответственно), снабжённые развернутыми комментариями. Если собрать все эти строчки воедино, не меняя их порядка, поместить в функцию `main()` и объявить все переменные, должно заработать. Тут, правда, ничего не сказано о заголовочных файлах; по этому поводу лучше посмотреть пример.

Замечание. В ОС Windows с `include`'ами всё проще. Там один-единственный файл `winsock.h`.

Для начала нужно выбрать номер порта, на котором мы будем работать. Он является общим для клиента и сервера. Поэтому запасём переменную (точнее, константу) `Port`, назначив ей какое-то ненулевое значение, например, 5678.

3.1. Серверная часть

3.1.1. Установка обработчика сетевых ошибок

Прежде всего, нужно помнить, что несмотря на всякие гарантии доставки пакетов любая сеть очень ненадёжна по своей природе. В самом деле, что будет, если злоумышленник перережет провод, по которому передается

информация государственной важности, а мы об этом ничего не узнаем? Нам нужно уметь контролировать сетевые ошибки. Для этого существует функция `signal`, устанавливающая функцию-обработчик на определённый тип события (в нашем случае разрыв соединения). Это делается в самом начале программы так:

```
if (signal(SIGPIPE, &SigHandler) == SIG_ERR)
{ fprintf(stderr, "Error: cannot set signal handler!\n"); return 0; }
```

Здесь `SigHandler` — функция такого вида: `void SigHandler(int SigID)`, а системная константа `SIGPIPE` указывает на тип обрабатываемой ошибки. Внутри обработчика нужно как-то сообщить пользователю о том, что соединение накрылось, и завершить программу.

3.1.2. Прослушивающее гнездо

Теперь нужно создать гнездо, которое будет принимать входящие подключения. Это делается при помощи функции `socket`. Объявим переменную `int s0;` и создадим гнездо:

```
s0 = socket(AF_INET, SOCK_STREAM, 0);
```

Здесь `AF_INET` указывает на тип используемой адресации, `SOCK_STREAM` говорит о том, что создаётся гнездо для двустороннего соединения и предназначенное для потоковой передачи данных, т. е. без ограничения размера пакетов (помните ругательство «connection-oriented»?). При этом сокет напоминает 2 трубы, в одну из которых заливают воду (данные), а из другой трубы данные выливаются. Эти потоки воды могут, разумеется, приостанавливаться, но в целом передача данных имеет непрерывный характер. Последний нулевой параметр функции указывает на использование протокола по умолчанию, т. е. того, который является стандартным для данного типа гнезда.

Сейчас мы сделаем так, чтобы наше гнездо «прослушивало» сеть и ловило все входящие подключения (от любых адресов). Для этого настроим гнездо на **произвольный** адрес, определяемый специальной константой `INADDR_ANY`. Адрес определяется структурой типа `sockaddr_in`. Вот кусок кода, привязывающий сокет к произвольному адресу:

```
sockaddr_in MyAddr;
memset(&MyAddr, 0, sizeof(sockaddr_in));
MyAddr.sin_family = AF_INET;
MyAddr.sin_port = htons(Port);
MyAddr.sin_addr.s_addr = htonl(INADDR_ANY);
bind(s0, (sockaddr*) &MyAddr, sizeof(MyAddr));
```

Здесь мы объявляем переменную `MyAddr`, чистим память, которую эта переменная занимает (на всякий случай), после чего устанавливаем уже знакомый тип адреса `AF_INET`, выбираем какой-нибудь порт и устанавливаем «произвольный» адрес. Затем вызовом функции `bind` прикрепляем адрес к сокету. Вспоминая аналогию с почтовой системой, мы получаем следующую картину: когда мы придём на почтовый узел и увидим, что нам пришло письмо, то нам его отдадут независимо от того, кто его отправил.

Следующие строки настраивают гнездо так, что если вызывается функция закрытия гнезда (прекращения связи), а у него есть ещё какое-то количество неотправленных данных, то система сперва постарается их отправить в течение некоторого времени и только потом закроет гнездо. По большому счёту, это излишняя предосторожность, поэтому на эти две строки можно смело забить (по крайней мере не нужно пытаться понять, как оно там работает).

```
linger linger_opt = { 1, 0 };
setsockopt(s0, SOL_SOCKET, SO_LINGER, &linger_opt, sizeof(linger_opt));
```

3.1.3. Приём входящих соединений и функция `accept`

Наконец-то наступает кульминационный момент в жизни гнезда `s0`. Всё, что было написано выше, было нужно ради следующих двух команд:

```
listen(s0, 1);
sockaddr_in CliAddr;
socklen_t AddrLen;
int s1 = accept(s0, (struct sockaddr*) &CliAddr, &AddrLen);
```

Вызвав функцию `listen`, мы заставляем систему ждать прихода запроса на соединение от клиента. Второй параметр указывает на требуемое количество подключений — в нашем случае нужен только один клиент. Следует помнить, что эта функция будет ждать вечно, если никто не захочет подключаться к нашему серверу. Но

будем считать, что хоть один юзер догадался выполнить функцию `connect`, да ещё к тому же угадал номер порта и IP-адрес нашей станции. Тогда `listen` завершится, и можно будет выяснить, что же это за юзер — по крайней мере, мы сможем узнать IP-адрес его компьютера и номер порта, на котором он сидит. А делать это мы будем так: заведём ещё одну переменную типа `sockaddr_in`, назвав её `CliAddr`, и вызовем функцию `accept`, передав в качестве первого аргумента дескриптор того гнезда, на которое пришло сообщение, а в качестве второго — указатель на структуру для адреса клиента. Функция `accept` занесёт туда все данные о нашем новом клиенте, а её возвращаемое значение — дескриптор гнезда, по которому можно будет предавать клиенту данные.

С этого момента установка соединения со стороны сервера считается успешной. Далее про всякие `accept`'ы, `bind`'ы и `listen`'ы можно забыть. Они относятся к инициализационной части программы и далее нам нужно будет только посылать и принимать данные с помощью функций `read`/`write`. Более того, гнездо `s0` теперь можно даже закрыть — оно больше не пригодится. Закрытие гнезда производится функцией `close`.

3.2. Клиентская часть

Теперь посмотрим, что в это время происходит на другом конце провода.

3.2.1. Получение адреса сервера

Тот вид соединений, который мы здесь используем (т. е. connection-oriented), предполагает, что сервер сидит тихо и только «слушает». Клиент же, напротив, должен послать запрос на соединение по вполне конкретному адресу. Но откуда, позвольте спросить, он возьмёт этот адрес? Конечно, может быть так: два человека захотели сыграть по сети, и один из них знает IP-адрес компьютера другого. Тогда всё тривиально: указываем нужный адрес и подключаемся туда, где нас уже ждут. Но как быть, когда мы не знаем адреса сервера? Тут разумно воспользоваться функцией `gethostbyname`, позволяющей по имени сервера определить его адрес. Это эквивалентно тому, как если бы мы стали искать в адресной книге адрес требуемого учреждения (естественно, если мы знаем, как оно называется).

```
hostent * Host = gethostbyname(HostName);
```

Здесь `HostName` есть строка с именем сервера. Среди всех допустимых имён зарезервировано специальное имя `localhost`. Если выполняется подключение к `localhost`, то подключение происходит «к самому себе», т. е. клиент и сервер должны работать **на одном и том же** компьютере. С тем же успехом можно вместо слова `localhost` указать IP-адрес 127.0.0.1, также означающий себя самого. Функция `gethostbyname` возвращает адрес некоторой структуры типа `hostent`, у которой есть поле `h_addr_list`, представляющее собой массив адресов, соответствующих данному серверу. Мы выберем из этого массива первый элемент и засунем его в структуру... нет, совсем забыл, сначала надо же ещё объявить переменную `sockaddr_in SrvAddr`, почистить её `memset`'ом, и только потом скопировать адрес. Ну, а про настройку типа адреса и номера порта всё уже было сказано выше — тут ничего нового нет.

```
sockaddr_in SrvAddr;
memset(&SrvAddr, 0, sizeof(SrvAddr));
SrvAddr.sin_family = AF_INET;
SrvAddr.sin_port = htons(Port);
memmove(&(SrvAddr.sin_addr.s_addr), Host->h_addr_list[0], 4);
```

3.2.2. Подключение к серверу

Тут всё предельно просто. Для начала создадим гнездо — точно также, как и на сервере, а затем вызовем долгожданный `connect`:

```
int s0 = socket(AF_INET, SOCK_STREAM, 0);
connect(s0, (sockaddr*) &SrvAddr, sizeof(SrvAddr));
```

Вторым параметром передаётся тот самый адрес, которой был получен столь долгим и мучительным путём. Если на другом конце провода нас не ждут, то `connect` вернёт отрицательное значение. В противном случае можно считать, что соединение установлено.

4. Сетевой обмен данными

Настало время разобраться с тем, ради чего мы заварили всю эту кашу с сокетами (надеюсь, пока у читателя в голове подобной каши ещё не возникло). С посылкой данных всё совсем просто: делаем массив, ну, скажем, на 1024 байта (назовём его `SendBuf`), запихиваем данные в этот массив и вызываем функцию `write`:

```
write(s1, SendBuf, 1024 * sizeof(char));
```

Естественно, так выглядит команда посылки со стороны сервера. Для клиента — почти так же, только вместо `s1` нужно написать `s0`. А вот с приёмом пакетов дела обстоят несколько хитрее. Часто в сетевых программах (например в играх) нельзя ждать, пока придет очередное сообщение, а нужно выполнять какие-то другие действия (например, перерисовывать экран). Поэтому имеет смысл воспользоваться функцией `select`. Она позволяет определить, есть ли у гнезда новые данные, или нет. Кроме того, эта функция умеет ждать прихода сообщения в течение определённого промежутка времени, и если он проходит, а сообщения всё нет, то функция завершается. Эта функция позволяет отслеживать очень маленькие промежутки времени — с точностью до одной микросекунды. Напишем функцию `bool CheckMessage(int Socket, char * Buffer)`, которая будет проверять, пришло ли сообщение на сокете `Socket`. Если да, то она возвращает `true` и «сливает 1024 байта из входной трубы сокета» в массив `Buffer`, а если нет, то просто возвращает `false`.

```
bool CheckMessage(int Socket, int * Buffer)
{
    fd_set DSet;
    FD_ZERO(&DSet);
    FD_SET(Socket, &DSet);
    timeval tv;
    tv.tv_sec = 0;
    tv.tv_usec = 10000;
    if (select(Socket+1, &DSet, 0, 0, &tv) > 0)
    {
        read(Socket, Buffer, 1024);
        return true;
    }
    else return false;
}
```

Максимальный интервал ожидания установлен в 10000 микросекунд, т. е. если в течение $\frac{1}{100}$ секунды данные не появятся, будет выдано значение `false`. Использовать функцию `CheckMessage` надо примерно так: после установки соединения делаем бесконечный цикл, в котором вызываем эту функцию, и если результат положительный, то обрабатываем данные, иначе — продолжаем цикл.

Скажем пару слов о том, что такое `fd_set` и `select`. Как уже было сказано, сокет — это дескриптор потока ввода-вывода. В недрах ОС содержится информация по каждому дескриптору, и разработчики предусмотрели механизм получения этой информации. Структура `fd_set` описывает битовое множество всех дескрипторов. Существует три типа событий, сигнализирующих о состоянии каждого дескриптора:

- Есть данные для чтения
- Гнездо готово к записи данных
- Произошла ошибка

За эти три типа событий отвечают второй, третий и четвёртый параметры функции `select` соответственно. Чтобы получить информацию о каком-либо дескрипторе, нужно установить в соответствующем битовом множестве бит этого дескриптора и вызвать функцию `select`. Первый параметр ограничивает максимальный номер интересующего нас дескриптора плюс один.

Нас интересует только информация о данных на чтение, поэтому третий и четвёртый параметры установлены в нули. Последний параметр — это величина таймаута в микросекундах. Функция `select` будет «прослушивать» указанные дескрипторы в течение этого времени, и, если в каком-либо из выбранных дескрипторов произошло событие требуемого типа, вернёт положительное число. Чтобы узнать, в каком именно, вообще говоря, нужно проверить битовое множество `DSet`, но в нашем случае дескриптор всего один, и проверять ничего не нужно. Если бы таковая проверка была бы необходима, её следовало бы выполнять с помощью макроса `FD_ISSET`. Более подробную информацию о `select` всегда можно посмотреть в справочной системе Linux.

5. Пример программы

Строго говоря, теория изложена в полном объёме. Остаётся немного попрактиковаться и написать какую-нибудь программу. Но чтобы читателю не было скучно, мы кратко изложим на словах её алгоритм, а в качестве работоспособного примера приведём другую программу.

5.1. Упражнение: пересылка файла по сети

Напишите самостоятельно программу, пересылающую двоичный файл с одного компьютера на другой. Для чтения и записи двоичных (т. е. не обязательно текстовых) файлов есть функции

```
int fread(void * Buffer, int Size, int Count, FILE * F)
int fwrite(void * Buffer, int Size, int Count, FILE * F)
```

Пусть файл передаётся от сервера к клиенту. Тогда алгоритм работы будет примерно следующий: сервер в цикле считывает файл кусками (например, по 1024 байта) и отправляет клиенту сообщения такого формата:

Размер i-го куска (4 байта)	i-й кусок файла ($\leqslant 1024$ байт)

Клиент же записывает эти куски последовательно себе на диск в какой-нибудь файл, а после записи очередного куска отправляет подтверждение серверу. Сервер же в это время ждёт подтверждения, и когда оно приходит, посылает очередной кусок файла. Подтверждения нужны не для того, чтобы гарантировать доставку данных, а для того, чтобы в сети не произошло «затора» из пакетов. Запись на диск — медленная операция (по крайней мере, медленнее, чем чтение), и может получиться так, что клиент не будет успевать «сливать воду из трубы-сокета». Для маленьких файлов это будет незаметно, а вот при попытке переслать файл мегабайт на десять могут начаться всякие глюки...

5.2. Пример: игра «Быки и коровы»

Напомним, что игра состоит в угадывании четырёхзначного числа, загаданного противником, который информирует игрока о количестве точных совпадений, или «быков» (правильная цифра на правильном месте) и количестве неточных совпадений, или «коров» (правильная цифра не на своём месте).

В данном случае клиент и сервер фактически равноправны, поэтому код сервера и клиента отличается только механизмом установки соединения.

5.2.1. Серверная часть

```
#include <signal.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/socket.h>
#include <netinet/in.h>

const int MAX = 64;

void SigHandler(int SigID);                          // Обработчик ошибок гнезда
bool WaitMessage(int Socket, int * Buffer);          // Функция ожидания данных
bool GetAnswer(int N, int GN, int * B, int * C);     // Вычисляет количество быков и коров

int main(int argc, char * argv[])
{
    printf("Welcome to the game 'Bulls & Cows'. This is a SERVER.\n");
    printf("Command line syntax: BCSrv [port number]\n");

    // Устанавливаем обработчик
    if (signal(SIGPIPE, &SigHandler) == SIG_ERR)
    {
        fprintf(stderr, "Error: cannot set signal handler!\n");
        return 0;
    }

    // Порт по умолчанию
    int Port = 1234;
    if (argc > 1) Port = atoi(argv[1]);
    printf("Using port #%d.\n", Port);

    // Создаём слушающее гнездо
    int s0 = socket(AF_INET, SOCK_STREAM, 0);
    if (s0 < 0)
    {
        fprintf(stderr, "Error: cannot create a socket.\n");
        return 0;
    }
```

```
sockaddr_in MyAddr;
memset(&MyAddr, 0, sizeof(sockaddr_in));
MyAddr.sin_family = AF_INET;
MyAddr.sin_port = htons(Port);
MyAddr.sin_addr.s_addr = htonl(INADDR_ANY);

// Привязываем его к адресу
if (bind(s0, (sockaddr*) &MyAddr, sizeof(MyAddr)) < 0)
{
    fprintf(stderr, "Error: cannot bind a socket.\n");
    close(s0);
    return 0;
}

printf("Waiting for the client...\n");

// Настраиваем гнездо
linger linger_opt = { 1, 0 };
setsockopt(s0, SOL_SOCKET, SO_LINGER, &linger_opt, sizeof(linger_opt));

// Слушаем...
if (listen(s0, 1) < 0)
{
    fprintf(stderr, "Error: cannot listen on the socket.\n");
    close(s0);
    return 0;
}

// Кто-то подключился, принимаем соединение
sockaddr_in CliAddr;
socklen_t AddrLen;
int s1 = accept(s0, (struct sockaddr*) &CliAddr, &AddrLen);
if (s1 < 0)
{
    fprintf(stderr, "Error: cannot accept connection.\n");
    return 0;
}

printf("Connection accepted\n");

// Закрываем слуховое гнездо
close(s0);

int MyNumber;              // Number of server
int MyGuess;               // Guessed number of client

do
{
    printf("Enter number: ");
    scanf("%d", &MyNumber);
} while (MyNumber < 1023 || MyNumber > 9876);


int SendBuffer[MAX];
int RecvBuffer[MAX];
int TN = 0;

// Основной цикл программы
while (true)
{
    printf("============================\nGuess opponent's number: ");
    scanf("%d", &MyGuess); MyGuess = MyGuess % 10000; if (!MyGuess) break;
    TN++;
    SendBuffer[0] = MyNumber;  // Our RIGHT number (Server)
    SendBuffer[1] = MyGuess; // Guessed number of client
    write(s1, SendBuffer, MAX * sizeof(int));

    printf("Waiting for opponent's reply...\n");
    if (!WaitMessage(s1, RecvBuffer)) break;

    int HisNumber = RecvBuffer[0]; // Right client's number
    int HisGuess = RecvBuffer[1];
    int B = 0; int C = 0;
    if (!GetAnswer(HisNumber, MyGuess, &B, &C)) { printf("Wrong number!!!\n"); break; }
```

```
        printf("Opponent's guess: %d. Your try: %d bulls, %d cows.\n", HisGuess, B, C);
        if (HisGuess == MyNumber) printf("GAME OVER: Your opponent wins at the %d-th turn.", TN);
        if (HisNumber == MyGuess) printf("CONGRATULATIONS: You win at the %d-th turn!.", TN);
    }

    // Закрываем клиентское гнездо
    close(s1);
    return 0;
}

bool WaitMessage(int Socket, int * Buffer)
{
    fd_set DSet;
    FD_ZERO(&DSet);
    FD_SET(Socket, &DSet);
    timeval tv;
    tv.tv_sec = 100;
    tv.tv_usec = 0;
    if (select(Socket+1, &DSet, 0, 0, &tv) > 0)
    {
        read(Socket, Buffer, 1024);
        return true;
    }
    else
    {
        // Они что там, заснули, что-ли?
        printf("Connection timed out or other player fell asleep.\n");
        return false;
    }
}

bool GetAnswer(int N, int GN, int * B, int * C)
{
    int d0 = N % 10;
    N /= 10;
    int d1 = N % 10;
    N /= 10;
    int d2 = N % 10;
    N /= 10;
    int d3 = N % 10;

    if (d0 == d1 || d0 == d2 || d0 == d3 ||
        d1 == d2 || d1 == d3 || d2 == d3) return false;

    int gd0 = GN % 10;
    GN /= 10;
    int gd1 = GN % 10;
    GN /= 10;
    int gd2 = GN % 10;
    GN /= 10;
    int gd3 = GN % 10;

    if (gd0 == gd1 || gd0 == gd2 || gd0 == gd3 ||
        gd1 == gd2 || gd1 == gd3 || d2 == d3) return false;

    int b = 0; int c = 0;
    if (gd0 == d0) b++;
    if (gd1 == d1) b++;
    if (gd2 == d2) b++;
    if (gd3 == d3) b++;
    if ((gd0 != d0) && (gd0 == d1 || gd0 == d2 || gd0 == d3)) c++;
    if ((gd1 != d1) && (gd1 == d0 || gd1 == d2 || gd1 == d3)) c++;
    if ((gd2 != d2) && (gd2 == d0 || gd2 == d1 || gd2 == d3)) c++;
    if ((gd3 != d3) && (gd3 == d0 || gd3 == d1 || gd3 == d2)) c++;

    *B = b; *C = c; return true;
}

// Могучий обработчик сетевого облома
void SigHandler(int SigID)
{
    printf("Broken pipe. SID: %d", SigID);
    exit(-1);
}
```

5.2.2. Клиентская часть

Здесь опущены комментарии для фрагментов кода, совпадающих с серверной частью.

```
#include <signal.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>

const int MAX = 64;

void SigHandler(int);
bool WaitMessage(int Socket, int * Buffer);
bool GetAnswer(int N, int GN, int * B, int * C);

int main(int argc, char *argv[])
{
    printf("Welcome to the game 'Bulls & Cows'. This is a CLIENT.\n");
    printf("Command line syntax: BCCli [host address] [port number]\n");

    if (signal(SIGPIPE, &SigHandler) == SIG_ERR)
    {
        fprintf(stderr, "Error: cannot set signal handler!\n");
        return 0;
    }

    int s0 = socket(AF_INET, SOCK_STREAM, 0);
    if (s0 < 0)
    {
        fprintf(stderr, "Error: cannot create a socket.\n");
        return 0;
    }

    sockaddr_in SrvAddr;
    memset(&SrvAddr, 0, sizeof(SrvAddr));

    // Получение адреса сервера
    char * HostName = "localhost";
    if (argc >= 2) HostName = argv[1];

    hostent * Host = gethostbyname(HostName);
    if (!Host)
    {
        fprintf(stderr, "Error: cannot get host address.\n");
        close(s0);
        return 0;
    }

    SrvAddr.sin_family = AF_INET;
    int Port = 1234;
    if (argc >= 3) Port = atoi(argv[2]);
    SrvAddr.sin_port = htons(Port);

    // Заполняем адрес сервера
    memmove(&(SrvAddr.sin_addr.s_addr), Host->h_addr_list[0], 4);
    printf("Connecting to %s...\n", HostName);

    // Подключаемся...
    if (connect(s0, (struct sockaddr*) &SrvAddr, sizeof(SrvAddr)) < 0)
    {
        fprintf(stderr, "Error: cannot connect to remote host.\n");
        close(0);
        return 0;
    }

    printf("You are connected to server!");

    int MyNumber; // Number of server
    int MyGuess; // Guessed number of client

    do
```

```
    {
        printf("Enter number:");
        scanf("%d", &MyNumber);
    } while (MyNumber < 1023 || MyNumber > 9876);

    int SendBuffer[MAX];
    int RecvBuffer[MAX];
    int TN = 0;

    // Основной цикл программы
    while (true)
    {
        printf("============================\nGuess opponent's number: ");
        scanf("%d", &MyGuess); MyGuess = MyGuess % 10000; if (!MyGuess) break;
        TN++;
        SendBuffer[0] = MyNumber;  // Our RIGHT number (Server)
        SendBuffer[1] = MyGuess; // Guessed number of client
        write(s1, SendBuffer, MAX * sizeof(int));

        printf("Waiting for opponent's reply...\n");
        if (!WaitMessage(s1, RecvBuffer)) break;

        int HisNumber = RecvBuffer[0]; // Right client's number
        int HisGuess = RecvBuffer[1];
        int B = 0; int C = 0;
        if (!GetAnswer(HisNumber, MyGuess, &B, &C)) { printf("Wrong number!!!\n"); break; }
        printf("Opponent's guess: %d. Your try: %d bulls, %d cows.\n", HisGuess, B, C);
        if (HisGuess == MyNumber) printf("GAME OVER: Your opponent wins at the %d-th turn.", TN);
        if (HisNumber == MyGuess) printf("CONGRATULATIONS: You win at the %d-th turn!.", TN);
    }

    // Закрываем клиентское гнездо
    close(s1);
    return 0;
}

// Здесь ещё должны быть функции WaitMessage, GetAnswer и SigHandler,
// которые можно позаимствовать из серверной части, поскольку
// они ничем не отличаются.
```