Сокеты

Андрей Боровский

В конкурсе на лучшую компьютерную идею всех времен и народов сокеты, без сомнения, могли бы рассчитывать на призовое место. Как и другие средства межпроцессного взаимодействия, рассмотренные в этой серии статей, сокеты впервые были реализованы именно платформе Unix (4.2BSD), однако, концепция сокетов, как универсального средства обмена данными между процессами, оказалась настолько удачна, что все современные системы поддерживают, по крайней мере, некоторое подмножество сокетов. Причины успеха сокетов заключаются в их простоте и универсальности. Программы, обменивающиеся данными с помощью сокетов, могут работать в одной системе и в разных, используя для обмена данными как специальные объекты системы, так и сетевой стек. Как и каналы, сокеты используют простой интерфейс, основанный на «файловых» функциях read(2) и write(2) (открывая сокет, программа Unix получает дескриптор файла, благодаря которому можно работать с сокетами, используя файловые функции), но, в отличие от каналов, сокеты позволяют передавать данные в обоих направлениях, как в синхронном, так и в асинхронном режиме. Большинство программистов используют для работы с сокетами различные библиотеки высокого уровня, однако, высокоуровневые библиотеки, как правило, не позволяют задействовать всю мощь и все многообразие сокетов. Наглядный пример многообразия – файловые сокеты. Программистам Windows должны быть знакомы сетевые сокеты, которые обычно организуют обмен данными с помощью протоколов семейства TCP/IP, однако в Unix есть и другие типы сокетов, специально предназначенные для обмена данными между локальными процессами.

Сокеты в файловом пространстве имен

Сокеты в файловом пространстве имен (file namespace, их еще называют «сокеты Unix») используют в качестве адресов имена файлов специального типа. Важной особенностью этих сокетов является то, что соединение с их помощью локального и удаленного приложений невозможно, даже если файловая система, в которой создан сокет, доступна удаленной операционной системе. В следующем фрагменте кода мы создаем сокет и связываем его с файлом socket.soc (это фрагмент программы сервера fsserver.c, которую вы найдете на диске):

```
sock = socket(AF_UNIX, SOCK_DGRAM, 0);

if (sock < 0) {
  perror("socket failed");
  return EXIT_FAILURE;
}

srvr_name.sa_family = AF_UNIX;
strcpy(srvr_name.sa_data, "socket.soc");
if (bind(sock, &srvr_name, strlen(srvr_name.sa_data) +
        sizeof(srvr_name.sa_family)) < 0) {
  perror("bind failed");
  return EXIT_FAILURE;

}
```

Константы и функции, необходимые для работы с сокетами в файловом пространстве имен, объявлены в файлах <sys/types.h> и <sys/socket.h>. Как и файлы, сокеты в программах представлены дескрипторами. Дескриптор сокета можно получить с помощью функции socket(2). Первый параметр этой функции – домен, к которому принадлежит сокет. Домен сокета обозначает тип соединения

(а не доменное имя Интернета, как вы могли бы подумать). Домен, обозначенный константой AF_UNIX, соответствует сокетам в файловом пространстве имен. Второй параметр socket() определяет тип сокета. значение SOCK_DGRAM указывает датаграммный сокет (я предпочитаю этот вариант написания используемому в [1] «дейтаграммный»). Датаграммные сокеты осуществляют ненадежные соединения при передаче данных по сети и допускают широковещательную передачу данных. Другой часто используемый тип сокетов – SOCK_STREAM соответствует потоковым сокетам, реализующим соединения «точка-точка» с надежной передачей данных. Впрочем, в пространстве файловых имен датаграммные сокеты также надежны, как и потоковые сокеты. Третий параметр функции socket() позволяет указать протокол, используемый для передачи данных. Мы оставляем значение этого параметра равным нулю. В случае ошибки функция socket() возвращает -1.

После получения дескриптора сокета мы вызываем функцию bind(2), которая связывает сокет с заданным адресом (связывать сокет с адресом необходимо в программе-сервере, но не в клиенте). Первым параметром функции является дескриптор, а вторым – указатель на структуру sockaddr (переменная srvr_name), содержащую адрес, на котором регистрируется сервер (третий параметр функции – дина структуры, содержащей адрес). Вместо общей структуры sockaddr для сокетов Unix (сокетов в файловом пространстве имен) можно использовать специализированную структуру sockaddr_un. Поле sockaddr.sa_family позволяет указать семейство адресов, которым мы будем пользоваться. В нашем случае это семейство адресов файловых сокетов Unix AF_UNIX. Сам адрес семейства AF_UNIX (поле sa_data) представляет собой обычное имя файла сокета. После вызова bind() наша программа-сервер становится доступна для соединения по заданному адресу (имени файла).

При обмене данными с датаграммными сокетами мы используем не функции write() и read(), а специальные функции recvfrom(2) и sendto(2). Эти же функции могут применяться и при работе с потоковыми сокетами, но в соответствующем примере мы воспользуемся «сладкой парочкой» read()/write(). Для чтения данных из датаграммного сокета мы используем функцию `recvfrom(2)`, которая по умолчанию блокирует программу до тех пор, пока на входе не появятся новые данные.

```
bytes = recvfrom(sock, buf, sizeof(buf),  0, &rcvr_name, &namelen);
```

Вызывая функцию `recvfrom()`, мы передаем ей указатель на еще одну структуру типа sockaddr, в которой функция возвращает данные об адресе клиента, запросившего соединение (в случае файловых сокетов этот параметр не несет полезной информации). Последний параметр функции `recvfrom()` – указатель на переменную, в которой будет возвращена длина структуры с адресом. Если информация об адресе клиента нас не интересует, мы можем передать значения NULL в предпоследнем и последнем параметрах. По завершении работы с сокетом мы закрываем его с помощью «файловой» функции close(). Перед выходом из программы-сервера следует удалить файл сокета, созданный в результате вызова socket(), что мы и делаем с помощью функции unlink().

Если программа-сервер показалась вам простой, то программа-клиент (fsclient.c) будет еще проще. Мы открываем сокет с помощью функции socket() и передаем данные (тестовую строку) серверу с помощью «напарника» recvfrom(), функции sendto(2):

```
srvr_name.sa_family = AF_UNIX;
strcpy(srvr_name.sa_data, SOCK_NAME);
strcpy(buf, "Hello, Unix sockets!");
sendto(sock, buf, strlen(buf), 0, &srvr_name,
    strlen(srvr_name.sa_data) + sizeof(srvr_name.sa_family));
```

Первый параметр функции sendto() – дескриптор сокета, второй и третий параметры позволяют указать адрес буфера для передачи данных и его длину. Четвертый параметр предназначен для передачи дополнительных флагов. Предпоследний и последний параметры несут информацию об адресе сервера и его длине, соответственно. Если при работе с датаграммными сокетами вызвать функцию connect(2) (см. ниже), то можно не указывать адрес назначения каждый раз (достаточно указать его один раз, как параметр функции connect()). Перед вызовом функции sendto() нам надо заполнить структуру sockaddr (переменную srvr_name) данными об адресе сервера. После окончания передачи данных мы закрываем сокет с помощью close(). Если вы запустите программу-сервер, а затем программу-клиент, то сервер распечатает тестовою строку, переданную клиентом.

Парные сокеты

Сокеты в файловом пространстве имен похожи на именованные каналы тем, что для идентификации сокетов используются файлы специального типа. В мире сокетов есть и аналог неименованных каналов – парные сокеты (socket pairs). Как и не именованные каналы, парные сокеты создаются парами и не имеют имен. Естественно, что область применения парных сокетов – та же, что и у неименованных каналов, - взаимодействие между родительским и дочерним процессом. Так же как и в случае неименованного канала, один из дескрипторов используется одним процессом, другой – другим. В качестве примера использования парных сокетов мы рассмотрим программу sockpair.c, создающую два процесса с помощью fork(). Дочерние процессы sockpair.c используют парные сокеты для обмена вежливым английским приветствием.

```
#include <sys/types.h>
#include <sys/socket.h>
#include <stdlib.h>
#include <stdio.h>
#include <errno.h>

#define STR1 "How are you?"
#define STR2 "I'm ok, thank you."
#define BUF_SIZE 1024

int main(int argc, char ** argv)
{
  int sockets[2];
  char buf[BUF_SIZE];
  int pid;

  if (socketpair(AF_UNIX, SOCK_STREAM, 0, sockets) < 0) {
      perror("socketpair() failed");
      return EXIT_FAILURE;
  }
  pid = fork();
  if (pid != 0) {
    close(sockets[1]);
    write(sockets[0], STR1, sizeof(STR1));
    read(sockets[0], buf, sizeof(buf));
    printf("%s\n", buf);
    close(sockets[0]);
  } else {
    close(sockets[0]);
    read(sockets[1], buf, sizeof(buf));
    printf("%s\n", buf);
    write(sockets[1], STR2, sizeof(STR2));
    close(sockets[1]);
  }
}
```

Парные сокеты создаются функцией socketpair(2). У функции socketpair() четыре параметра. Первые три параметра функции те же, что и у socket(), а четвертым параметром является массив из двух переменных, в которых возвращаются дескрипторы. Дескрипторы сокетов, возвращенные socketpair(), уже готовы к передаче данных, так что мы сразу можем применять к ним функции read()/write(). После вызова fork() каждый процесс получает оба дескриптора, один из которых он должен закрыть. Для закрытия сокета мы используем функцию close().

При взгляде на интерфейс программирования парных сокетов может возникнуть вопрос, а почему собственно эти функции относятся к сокетам? Ведь при работе с ними мы не используем ни адреса, ни модель клиент-сервер. Это верно, но заметьте, что функции socketpair() передаются значения домена и типа сокета, так что и формально, и с точки зрения реализации в системе мы используем настоящие сокеты. Следует отметить, что указание домена в функции socketpair() выглядит явно излишне, поскольку для этой функции система поддерживает только сокеты в домене AF_UNIX (вполне логичное ограничение, если учесть, что парные сокеты не имеют имен и предназначены для обмена данными между родственными процессами).

Сетевые сокеты

Мы переходим к рассмотрению самого важного и универсального типа сокетов – сетевых сокетов. Думаю, что о значении, которое имеют сетевые сокеты в Unix-системах, распространяться не нужно. Даже если вы пишете систему приложений, предназначенных для работы на одном компьютере, рассмотрите возможность использования сетевых сокетов для обмена данными между этими приложениями. Возможно, в будущем ваш программный комплекс наберет мощь и возникнет необходимость распределить его компоненты на нескольких машинах. Использование сетевых сокетов сделает процесс масштабирования проекта безболезненным. Впрочем, у сетевых сокетов есть и недостатки. Даже если сокеты используются для обмена данными на одной и той же машине, передаваемые данные должны пройти все уровни сетевого стека, что отрицательно сказывается на быстродействии и нагрузке на систему.

В качестве примера мы рассмотрим комплекс из двух приложений, клиента и сервера, использующих сетевые сокеты для обмена данными. Текст программы сервера вы найдете в файле netserver.c, ниже мы приводим некоторые фрагменты. Прежде всего, мы должны получить дескриптор сокета:

```
sock = socket(AF_INET, SOCK_STREAM, 0);
if (socket < 0) {
  printf("socket() failed: %d\n", errno);
  return EXIT_FAILURE;
}
```

В первом параметре функции socket() мы передаем константу AF_INET, указывающую на то, что открываемый сокет должен быть сетевым. Значение второго параметра требует, чтобы сокет был потоковым. Далее мы, как и в случае сокета в файловом пространстве имен, вызываем функцию bind():

```
serv_addr.sin_family = AF_INET;
serv_addr.sin_addr.s_addr = INADDR_ANY;
serv_addr.sin_port = htons(port);
if (bind(sock, (struct sockaddr *) &serv_addr, sizeof(serv_addr)) < 0) {
  printf("bind() failed: %d\n", errno);
  return EXIT_FAILURE;
}
```

Переменная serv_addr, - это структура типа sockaddr_in. Тип sockaddr_in специально предназначен для хранения адресов в формате Интернета. Самое

главное отличие sockaddr_in от sockaddr_un – наличие параметра sin_port, предназначенного для хранения значения порта. Функция htons() переписывает двухбайтовое значение порта так, чтобы порядок байтов соответствовал принятому в Интернете (см. врезку). В качестве семейства адресов мы указываем AF_INET (семейство адресов Интернета), а в качестве самого адреса – специальную константу INADDR_ANY. Благодаря этой константе наша программа сервер зарегистрируется на всех адресах той машины, на которой она выполняется.

Остроконечники и тупоконечники

Именно так переводятся на русский язык термины little-endian и big-endian. В компьютерной литературе эти термины обозначают порядок байтов, используемый процессором для представления простых многобайтовых типов (например, 32-битного целого). В оригинале же (то есть в сказочной повести Дж. Свифта «Гулливер в стране лилипутов») так именовались враждебные общественные течения, приверженцы которых придерживались противоположных взглядов на порядок очистки яйца от скорлупы. Разногласия между остроконечниками и тупоконечниками даже стали причинной войны между Лилипутией и враждебным государством Блефуску. Впрочем, в компьютерном мире проблемы порядка байтов могут достигать совсем не лилипутских размеров. На процессорах Intel порядок байтов остроконечный, а, например, в системах MacOS X – Power PC Sun SPARC - тупоконечный (если учесть, что Apple отказалась от PowerPC, а Sun заменяет RISC-архитектуры оптеронами, получится, что остроконечники побеждают). Однако протоколы Интернета используют «тупоконечный» порядок байтов. Для того чтобы избежать путаницы, во всех системах, включая «тупоконечников» рекомендуется использовать функцию htons(). Эта функция «знает» порядок байтов в системе и, если нужно, приводит его в соответствие с принятым в протоколах TCP/IP. В русскоязычном Интернете одно время кочевала статья, в которой утверждалась (впрочем, шутливо), что чуждый для Intel порядок байтов и вызванная этим необходимость в дополнительной операции перестановки являются результатом заговора со стороны гигантских софтверных компаний. В книге [1] используются термины «прямой порядок байтов» для little-endian и «обратный порядок байтов» для big-ndian.

Чтобы понять, что мы должны делать дальше, давайте вспомним, как работает сетевая подсистема Unix и, в данном случае, любой другой ОС. Сетевой сервер должен уметь выполнять запросы множества клиентов одновременно (наш сервер netserver.c фактически может обработать запрос только одного клиента, но речь сейчас идет об общем случае). При этом в соединениях «точка-точка», например, при использовании потоковых сокетов, для каждого клиента у сервера должен быть открыт отдельный сокет. Из этого следует, что мы не должны устанавливать соединение с клиентом через сам сокет sock, предназначенный для прослушивания входящих запросов (обычно, при использовании сетевых сокетов мы и не можем этого сделать), иначе все другие попытки соединиться с сервером по указанному адресу и порту будут заблокированы. Вместо этого мы вызываем функцию listen(2), которая переводит сервер в режим ожидания запроса на соединение:

```
listen(sock, 1);
```

Второй параметр listen() – максимальное число соединений, которые сервер может обрабатывать одновременно. Далее мы вызываем функцию accept(2), которая устанавливает соединение в ответ на запрос клиента:

```
newsock = accept(sock, (struct sockaddr *) &cli_addr, &clen);
if (newsock < 0) {
  printf("accept() failed: %d\n", errno);
  return EXIT_FAILURE;
}
```

Получив запрос на соединение, функция accept() возвращает новый сокет, открытый для обмена данными с клиентом, запросившим соединение. Сервер как бы перенаправляет запрошенное соединение на другой сокет, оставляя сокет sock свободным для прослушивания запросов на установку соединения. Второй параметр функции accept() содержит сведения об адресе клиента, запросившего соединение, а третий параметр указывает размер второго. Так же как и при вызове функции recvfom(), мы можем передать NULL в последнем и предпоследнем параметрах. Для чтения и записи данных сервер использует функции read() и write(), а для закрытия сокетов, естественно, close().

В программе-клиенте (netclient.c) нам, прежде всего, нужно решить задачу, с которой мы не сталкивались при написании сервера, а именно выполнить преобразование доменного имени сервера в его сетевой адрес. Разрешение доменных имен выполняет функция gethostbyname():

```
server = gethostbyname(argv[1]);
if (server == NULL) {
  printf("Host not found\n");
  return EXIT_FAILURE;
}
```

Функция получает указатель на строку с Интернет-именем сервера (например, www.unix.com или 192.168.1.16) и возвращает указатель на структуру hostent (переменная server), которая содержит имя сервера в приемлемом для дальнейшего использования виде. При этом, если необходимо, выполняется разрешение доменного имени в сетевой адрес. Далее мы заполняем поля переменной serv_addr (структуры sockaddr_in) значениями адреса и порта:

```
serv_addr.sin_family = AF_INET;
memcpy(&serv_addr.sin_addr.s_addr, server->h_addr, server->h_length);
serv_addr.sin_port = htons(port);
```

Программа-клиент открывает новый сокет с помощью вызова функции socket() аналогично тому, как это делает сервер (дескриптор сокета, который возвращает socket() мы сохраним в переменной sock), и вызывает функцию connect(2) для установки соединения:

```
if (connect(sock, &serv_addr, sizeof(serv_addr)) < 0) {
  printf("connect() failed: %d", errno);
  return EXIT_FAILURE;
}
```

Теперь сокет готов к передаче и приему данных. Программа-клиент считывает символы, вводимые пользователем в окне терминала. Когда пользователь нажимает <Ввод> программа передает данные серверу, ждет ответного сообщения сервера и распечатывает его.

На протяжении этой статьи мы несколько раз упоминали не-блокирующие сокеты. Остановимся на них немного подробнее. О не-блокирующих сокетах вам нужно знать, прежде всего, то, что ими можно не пользоваться. Благодаря многопоточному (многопрограммному) программированию мы можем использовать блокирующие сокеты во всех ситуациях (и тогда, когда нам нужно обрабатывать несколько сокетов одновременно, и тогда, когда нам требуется возможность прервать операцию, выполняемую над сокетом). Рассмотрим, тем не менее, две функции, необходимые для работы с не-блокирующими сокетами. По умолчанию функция socket() создает блокирующий сокет. Чтобы сделать его не-блокирующим, мы используем функцию fcntl(2):

```
sock = socket(PF_INET, SOCK_STREAM, 0);
fcntl(sock, F_SETFL, O_NONBLOCK);
```

Теперь любой вызов функции read() для сокета sock будет возвращать управление сразу же. Если на входе сокета нет данных для чтения, функция read() вернет значение EAGAIN. Для поверки состояния не-блокирующих сокетов можно воспользоваться функцией select(2). Функция select() способна проверять состояние нескольких дескрипторов сокетов (или файлов) сразу. Первый параметр функции – количество проверяемых дескрипторов. Второй, третий и четвертый параметры функции представляют собой наборы дескрипторов, которые следует проверять, соответственно, на готовность к чтению, записи и на наличие исключительных ситуаций. Сама функция select() – блокирующая, она возвращает управление, если хотя бы один из проверяемых сокетов готов к выполнению соответствующей операции. В качестве последнего параметра функции select() можно указать интервал времени, по прошествии которого она вернет управление в любом случае. Вызов select() для проверки наличия входящих данных на сокете sock может выглядеть так:

```
fd_set set;
struct timeval interval;
FD_SET(sock, &set);
tv.tv_sec = 1;
tv.tv_usec = 500000;

...

select(1, &set, NULL, NULL, &tv);
if (FD_ISSET(sock, &set) {

  // Есть данные для чтения

}
```

Все, что касается функции select() теперь объявляется в заголовочном файле <sys/select.h> (раньше объявления элементов функции select() были разбросаны по файлам <sys/types.h>, <sys/time.h> и <stdlib.h>). В приведенном фрагмента кода FD_SET и FD_ISSET – макросы, предназначенные для работы с набором дескрипторов fd_set.

На этом мы закончим знакомство с увлекательным миром межпроцессного взаимодействия Linux. Следующая статья будет посвящена управлению процессами, сигналам и потокам.

Литература

1. Стивенс У., **UNIX: Разработка сетевых приложений.** - СПб.: Питер, 2004
2. W. R. Stevens, S. A. Rago, **Advanced Programming in the UNIX® Environment: Second Edition,** Addison Wesley Professional, 2005