Полное руководство по сетевому программированию для разработчиков игр. Часть 2. (2 стр)

Автор: x84

Адреса и порты

В системе может быть открыто большое количество сокетов, причем необязательно разных, несколько сокетов могут использовать один и тот же протокол для передачи данных. А теперь представим себе такую ситуацию: запущено две программы, и обе используют UDP. Извне в систему поступает UDP-пакет с данными. Системе надо определить какой из двух программ адресован этот пакет (Ты же не хочешь, чтоб твою личную переписку читал кто-то другой?). Исходя из дескрипторов сокетов этого понять нельзя, потому что опять же в пакете нет данных о дескрипторах... Вот тут-то и приходится вводить такое понятие, как "порт". К реальным физическим портам это не имеет никакого отношения. Это просто абстракция, число, которое позволяет распределять пакеты по адресатам. Когда сетевое приложение начинает свою работу, оно просит систему выделить ей какой-то конкретный порт или назначить любой из пула свободных (эфемерных) портов. Это делается на обоих концах соединения. Копии программы (например, чат), запущенные на двух разных компьютерах, не обязательно должны получить от системы в распоряжение одинаковый порт (одно и то же число). Главное, чтобы они знали адрес удаленного компьютера в сети и номер удаленного порта, чтобы у них была возможность отсылать по адресу на конкретный порт данные. Однако, это распространенная практика - использовать для одной программы один и тот же порт, чтобы пользователи не терялись в догадках, на каком же там порту висит программа-адресат. Например, порт 80 - общепринятый стандарт для передачи HTTP страниц, 21 - это для передачи файлов по протоколу FTP (5 уровень OSI) и т.д.

Порт - это 16-битное неотрицательное число (слово). Соответственно, в каждой системе не может использоваться одновременно более 65535 портов. Этого вполне достаточно (когда ты в последний раз запускал 65 тысяч сетевых программ?? :)).

Например, если в системе несколько сетевых карт, а мы хотим получать пакеты только от одной из них, то надо привязать сокет к адресу соответствующей сетевой карты (интерфейса), (иногда говорят "назначить ему имя"). Или мы хотим, чтоб наше приложение работало через какой-то конкретный порт - мы опять же назначаем сокету "имя".

Ну-с... Пора взглянуть на код, который проделывает все вышеописанные операции. Напомню, что впредь я буду сознательно опускать код, отвечающий за обработку ошибок. Ты можешь посмотреть коды возможных ошибок и их описание либо в MSDN, либо в man page для нужной функции.

Для начала, мы посмотрим, как выглядит "имя" сокета:

```
struct sockaddr_in
{
  unsigned char  sin_len;  // (*присутствует только во FreeBSD)
  unsigned char  sin_family;
  unsigned short   sin_port;
  struct in_addr  sin_addr;
  char    sin_zero[8];  // (*отсутствует в Linux)
};

struct in_addr
{
  unsigned long int  s_addr;  // должно быть длиной 4 байта (int32)
};
```

Структура общего вида для привязки сокета имеет тип sockaddr. Но нас интересует только sockaddr_in. Суффикс _in означает, что мы собираемся использовать семейство адресов Интернет. Существует еще куча разных структур, все они имеют разные поля и определяются по-разному, предоставляют разные возможности и доступ к разным протоколам, но все они должны быть приведены к типу sockaddr, который их все объединяет, и с которым работают все сетевые вызовы. Это и есть тот самый механизм, который позволяет одним и тем же функциям работать с сокетами совершенно разного типа.

В общем случае нам надо создать структуру sockaddr_in, обнулить ее и заполнить только три ее поля: sin_family, sin_port и sin_addr. Это работает на интересующих нас системах. Вот как это делается:

```
// Listing 2.01 nix

#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

#define PORT 5000    // пусть пока будет равен 5000

struct sockaddr_in addr;
memset (&addr, 0, sizeof (struct sockaddr_in)); // обнуление нужно для
                                                // максимальной переносимости

addr.sin_family = AF_INET;
addr.sin_port = htons (PORT);
addr.sin_addr.s_addr = inet_addr ("192.168.0.1"); // для примера


// Listing 2.01 win

#include <string.h>
#include <winsock2.h>

#define PORT 5000

struct sockaddr_in addr;
memset (&addr, 0, sizeof (struct sockaddr_in));

addr.sin_family = AF_INET;
addr.sin_port = htons (PORT);
addr.sin_addr.s_addr = inet_addr ("192.168.0.1");
```

Конечно, незнакомые htons() и inet_addr() сразу же бросаются в глаза. Эти функции, которые кажутся "маленькими и незначительными" и иногда теряются в массе кода, на самом деле выполняют очень важные функции.

Дело в том, что компьютеры разной архитектуры хранят и обрабатывают данные по-разному. Различия заключаются в порядке следования байтов. Существуют два порядка: "от старшего к младшему" (fore-word == big-endian == network order) - сетевой порядок следования байтов; и "от младшего к старшему" (back-word == little-endian == host order) - узловой порядок следования байтов.

Чтобы все стало на свои места возьмем для примера два целых 32-битных числа - 0XFFC3B2A7 и 0x21F2CE07. Представим, что в памяти они записаны одно за другим.

В архитектурах, использующих сетевой порядок следования байтов "от старшего к младшему" содержимое памяти будет выглядеть так:

```
.. FF C3 B2 A7 21 F2 CE 07 ..
```

Мы видим, в младших адресах памяти хранятся старшие байты чисел, т.е. в начале идет старший байт, а заканчивается область памяти младшим байтом числа. Числа записываются в память, также как люди записывают их на бумагу (слева направо, сначала старшие разряды, потом младшие).

И наоборот, в архитектурах, где используется узловой порядок следования байтов "от младшего к старшему" содержимое памяти будет выглядеть так:

```
.. A7 B2 C3 FF 07 CE F2 21 ..
```

Опять же, очевидно, в данном случае числа хранятся "более привычным образом", по младшим адресам памяти записаны младшие байты чисел, по старшим - старшие байты чисел.

Зачем нам надо это знать? Ответ: потому что в сети всегда используется сетевой порядок байтов, а внутри компьютера числа представляются в зависимости от архитектуры. Например, процессоры Motorola 68000 работают с сетевым порядком следования байтов, а машины на базе Intel x86 - с узловым. Поэтому, при передаче данных, на машине Motorola никакого конвертирования делать не надо, а на x86 - придется конвертировать числа из прямого порядка следования байтов в сетевой. Конвертировать нужно обязательно, чтобы машины разной архитектуры могли не только слышать, но и понимать друг друга.

Чтобы мы справились с задачей конвертирования, умные дядьки написали для нас несколько функций, которые и берут все эти обязанности на себя. Вот эти вызовы:

```
unsigned long int  ntohl (unsigned long int netlong);
unsigned short int  ntohs (unsigned short int netshort);
```

"Зачем такие запутанные имена, и что конкретно каждая из этих функций делает?" - спросишь ты. Ответ опять же очевиден: они конвертируют числа разной длины из одного порядка следования байтов в другой. А имена на самом деле очень даже "говорящие", т.е.:

htonl - host to network long (длинное целое из узлового в сетевой)
htons - host to network short (короткое целое из узлового в сетевой)
ntohl - network to host long (длинное целое из сетевого в узловой)
ntohs - network to host short (короткое целое из сетевого в узловой)

Эти функции работают достаточно интеллектуально, то есть на машинах с сетевым порядком следования байтов, они немедленно возвращаются, ничего не конвертируя, что сохраняет нам драгоценные процессорные такты. Ну а если же вы на машине с процессором от Intel - что же поделать, функции выполнят нужные действия... Intel и Motorola уже давно ведут эту маленькую "религиозную" войну, а нам остается только подстраиваться под существующие условия. Мы можем заменить эти функции на свои собственные, написанные на ассемблере, но я думаю, что в каждой конкретной реализации на всех ОС они выполнены оптимальным образом, поэтому не факт, что в замене есть смысл.

Насчет inet_addr() - она конвертирует строку символов (IP-адрес в так называемой "точечной нотации", понятной человеку, например 192.168.0.1) в 32-битное число с сетевым порядком следования байтов. У нее есть "зеркальное отражение" - функция inet_ntoa() (ntoa == network to alpha), которая выполняет в точности обратное преобразование:

```
unsigned long int inet_addr (char * cp);
char * inet_ntoa (struct in_addr in);
```

Любопытный читатель спросит: "Почему мы не конвертируем константу AF_INET в сетевой порядок следования байтов?". Ответ: "Патамушта!!" :) Просто номер порта и IP-адрес записывается в заголовок пакета и отправляется через сеть адресату. А все, что творится в сети, записано в сетевом порядке. Что же касается Address Family - то это значение используется ядром операционной системы, для правильного формирования пакетов и обеспечения прослойки между программистом и сетью, и в саму сеть это число не попадает. Поэтому мы храним его в узловом порядке. Вот и "фся любоff". :)

```
// Listing 2.02
//
// Windows:
//  int bind (SOCKET s, const struct sockaddr * name, int namelen);
//
// Linux & FreeBSD:
//  int bind (int s, const struct sockaddr * addr, int addrlen);

int serror = bind (sd, (struct sockaddr *) &addr, sizeof (struct sockaddr_in));

// дальше идет проверка возврата функции bind() и обработка ошибок
```

Проверка ошибок для bind() осуществляется все теми же средствами. Для Linux и FreeBSD в случае ошибки bind() возвращает -1, в Windows - SOCKET_ERROR (SOCKET_ERROR == -1, однако мы привыкли пользоваться предопределенными константами, к тому же, ситуация может измениться в будущем). Типы ошибок и их описания смотри в MSDN и man pages.

Данный вызов свяжет наш сокет с определенным адресом и портом (они записаны в структуре addr) на данной конкретной машине, после чего другие приложения не смогут запросить этот порт у системы, пока он не освободится (то есть сокет не будет удален).

Обрати внимание, наша структура sockaddr_in приводится к более общему типу sockaddr. Таковы законы многофункционального интерфейса, и ничего с этим не поделаешь. Мы можем использовать приведение типов в любом стиле - как в стиле C, так и C++. Без приведения компилятор языка C выдаст предупреждение о несоответствии типов, но программу скомпилирует, и она даже будет работать. Это потому что интерфейс сокетов определяет тип структуры, исходя из ее содержимого. Если ты используешь C++, то без приведения типов ты даже не получишь бинарник.

В Windows есть возможность не указывать порт, а заставить систему связать сокет с любым доступным портом из списка эфемерных портов (в Windows - от 1024 до 5000, в Linux и FreeBSD устанавливается администратором). Для этого достаточно в качестве sin_port указать 0. При использовании неподключаемых сокетов (UDP) в Linux, FreeBSD и в Windows вызывать bind() необязательно. Тогда порт и адрес назначается системой при первом обращении к сокету (в дальнейшем мы рассмотрим этот случай).

адрес, присвоив sin_addr.s_addr = INADDR_ANY, тогда сокет сможет принимать данные с любых адресов, если он неподключаемый (UDP), или только с того адреса интерфейса, через который пришел самый первый запрос на подключение, в случае с TCP.

Если порт или адрес не был точно установлен при привязке, то узнать его можно после того, как система самостоятельно назначит сокету "имя", при помощи функции getsockname(). Мы рассмотрим ее более внимательно чуть позже.

Есть одна очень тонкая деталь, важная для программистов под Linux или FreeBSD. Пользователь, не имеющий прав root, не может создавать и использовать сокеты с портами в пределах 0-1024. Это связано с жесткими мерами безопасности и с борьбой против IP-spoofing'a и других изощренных видов сетевых атак (для этих систем это особенно актуально). Нам еще рано полностью вникать в это дело, мы займемся этим, когда основной материал будет изучен, а пока просто "Низя!!!". :)

Первый блин всегда похож на сферу (иногда бывает и куб)

Ну вот мы и подошли к тому моменту, когда можно попробовать создать объектно-ориентированный код, который будет осуществлять инициализацию, проверку ошибок и очистку. Сразу предупреждаю - ногами не бить, я щекотки боюсь. Возможно, этот код не идеален, но я же недаром такой подзаголовок выбрал... :) Код будет работать (эээ... ну хорошо... исправляюсь: может быть, будет работать) на любой из трех типов систем: Linux, FreeBSD или семейство Windows. Здесь я пока еще не использую Windows GUI, то есть код надо компилировать, как консольное приложение. Я обильно полил код комментариями (конечно, за исключением очевидных мест), так что все, думаю, будет ясно.

Код надо скачать, и в дальнейшем держать перед глазами, потому что дальнейшее изучение будет опираться на этот код.

Исходный код:
win: chapter02.zip
*nix: chapter02.tar.gz

Страницы: 1 **2**

#TCP, #TCP/IP, #UDP

9 июня 2003 (Обновление: 30 сен 2009)

Комментарии [1]