Полное руководство по сетевому программированию для разработчиков игр. Часть 4. TCP (3 стр)

Автор: x84

Задача клиента и реакция сервера

Теперь посмотрим, что делает в это время клиент... Его задача – отослать запрос на сервер и ждать решения. Ответ может быть одним из двух: либо мы подключились, либо нам отказано в подключении. Как отослать такой запрос? Давай посмотрим!

```
// Linux & FreeBSD

  int connect (int s, const struct sockaddr * server_addr, int namelen);

// Windows

  int connect (SOCKET s, const struct sockaddr * server_addr, int namelen);
```

Ну вот... Ничего сложного. В случае успеха функция connect() вернет 0. Это будет для нас флагом о том, что мы успешно подключились к серверу и теперь можем начинать обмен данными. В случае невозможности подключения будет возвращено либо –1 (*nix), либо SOCKET_ERROR (Windows). Как доставать коды ошибок и их описания ты уже знаешь.

s – это сокет клиента, через который мы хотим подключиться к серверу.

должен быть приведен к типу sockaddr *. Такова судьба.

namelen – третий параметр, в нашем случае всегда равен int namelen == sizeof (struct sockaddr_in); Это аксиома.

Вот пример кода для отправки запроса:

```
#define SERVER_ADDRESS    "192.168.0.1"
#define SERVER_PORT    10000

struct sockaddr_in addr;
addr.sin_family = AF_INET;
addr.sin_port = htons (SERVER_PORT);
addr.sin_addr.s_addr = inet_addr (SERVER_ADDRESS);

int namelen = sizeof (struct sockaddr_in); //можно написать sizeof(addr) - без разницы

int error = connect (sd, (struct sockaddr *) &addr, namelen);
if (error == -1) // использовать SOCKET_ERROR в WINDOWS, в остальном - все то же самое
  // не получилось...
else
// принимаем-отправляем данные
```

Думаю, все ясно без слов. Единственное добавление – connect() тоже блокирует программу. И мы опять-таки должны пока с этим смириться.

Вернемся к нашему серверу. Так-с... Мы создали очередь, пора продвигаться дальше. Допустим, в эту очередь поступил запрос (клиент вызвал connect()). Нам надо его принять и обслужить. Для этого мы должны вызвать функцию accept():

```
// Linux & FreeBSD

  int accept (int s, struct sockaddr * addr, int * addrlen);

// Windows

  SOCKET accept (int s, struct sockaddr * addr, int * addrlen);
```

Ок, на первый взгляд все понятно, хотя, ничего не понятно! :) Сейчас разберемся. Функция accept() берет из очереди запросов на подключение первый из них и удаляет запрос из очереди (в общем случае очередь смещается или, если больше запросов нет, то очередь становится пустой и способна принимать новые запросы). Далее эта функция создает новый сокет (!!!), через который мы сможем общаться с тем, кто подал заявку на подключение. Как видно, именно этот свежевыпеченный сокет она нам и возвращает.

s - это дескриптор того сокета, который был создан в начале. Функции accept() необходимо знать, через какой канал связи приходят запросы. В параметре s мы и указываем ей наш сокет. Причем это не оказывает никакого воздействия на главный (прослушивающий) сокет (s). Тот продолжает находится в ожидающем состоянии, и его не волнует дальнейшая судьба запроса на подключение - его задача - принимать запросы и помещать их в очередь. Короче, s - это дескриптор того сокета, который был создан с помощью socket(), привязан с помощью bind() к определенному адресу (мы сказали "Эй, сокет открыт на таком-то адресе, заходи, не стесняйся!") и который был переведен в прослушивающий режим с помощью listen().

addr - это указатель на адресную структуру, которая после возврата будет содержать адрес, порт и всю прочую информацию о клиенте, которого мы будем обслуживать.

addrlen - это указатель на число. В нем мы передаем длину адресной структуры (исходя из семейства адресов, с которым мы работаем). В нашем случае мы должны передать туда sizeof(struct sockaddr_in). После возврата из accept() это число будет содержать фактическую длину адресной структуры, описывающей клиента, чей запрос был принят.

Функция accept() блокирует программу. Это значит, что если в очередь не поступило ни одного запроса на подключение, или все запросы уже были обработаны, то accept() будет ждать до тех пор, пока не поступит очередной запрос. То есть она не возвратит ничего до тех пор, пока не будет что возвращать. В это время мы не сможем влиять на программу изнутри. Придется ждать. Если такого запроса не поступит, то ждать придется вечно. Мы, конечно, сможем "убить" нашу программу, но вряд ли это то, чего мы хотим. Пока нам придется смириться с таким положением веще й, позже мы узнаем, как обходить такие ситуации.

о клиенте записывается в addr. Мы просто смотрим его адрес и порт. Ну а дальше нам решать - либо продолжать сеанс и начать обмен данными с ним, либо просто закрыть сокет, который создала функция accept(), при помощи closesocket() (Windows) или close() (*nix). Посмотрим, как это выглядит на языке C:

```
// Listing 4.01 win & nix

#ifdef _WINDOWS_
  SOCKET client;
#else
  int client;
#endif

struct sockaddr_in client_addr;
int client_addrlen = sizeof (struct sockaddr_in);

client = accept (sd, (struct sockaddr *) &client_addr, &client_addrlen);

if (client_is_valid()) // client_is_valid() должна проверять тот ли это клиент, или нет
  transmit_data();  // transmit_data() - отослать/принять данные
else
#ifdef _WINDOWS_
  closesocket (client);
#else
  close (client);
#endif
```

Все проще пареной репы! :)

Что касается возвращаемого значения функции accept() - то тут мы снова наблюдаем расхождение в философии двух семейств операционных систем. В *nix возвращается либо целое НЕОТРИЦАТЕЛЬНОЕ число в случае успеха, либо -1 в случае ошибки. А в Windows - либо целое ПОЛОЖИТЕЛЬНОЕ число в случае успеха, либо INVALID_SOCKET (равно 0) в случае ошибки. Думаю, разницу между положительными и неотрицательными числами объяснять не надо?! :)

Опять же, особое внимание стоит обратить на второй параметр. Он обязательно должен быть приведен к типу sockaddr *. Так работает OSI - и это нам только на руку.

#OSI, #TCP, #UDP, #клиент, #сервер, #сокеты

1 ноября 2003 (Обновление: 18 ноя 2009)

Комментарии [40]