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

Автор: x84

Передача по TCP

Итак, создавать TCP сокет научились, создавать очереди тоже умеем, принимать запросы - тоже можем, закрывать сокет умеем уже давно... Вроде бы и все... Ах да! Совсем забыл! Осталось "всего лишь" научиться передавать данные по TCP соединению! :))

Для отсылки/приема данных через подключенный сокет (TCP-сокеты называются подключаемыми) используется пара функции send()/recv() соответственно. Эта пара функций используется как на клиенте, так и на сервере. Но есть одно различие... Рассмотрим их по порядку...

Объявление send() выглядит так:

```
// Linux & FreeBSD

  int send (int s, const void * msg, int len, int flags);

// Windows

  int send (SOCKET s, const char * buf, int len, int flags);
```

Но на сервере все обстоит по-другому. После вызова accept() у нас на сервере получилось как минимум два сокета (почему я говорю "как минимум"? потому что мы вскоре научимся обслуживать клиентов целым скопом, а не по отдельности). Первый сокет – это тот, который мы создали с помощью socket(). Мы его перевели в прослушивающий режим. И теперь через него мы не можем слать данные. Он теперь стоит "на воротах" и ждет, когда кто-нибудь придет, чтоб сообщить нам об этом. Его мы больше не трогаем. Но у нас есть еще и второй. Тот, который нам вернула функция accept(). Вспомните, мы записали его дескриптор в переменную client. Вот как раз при помощи него мы и будем общаться с нашим клиентом. Поэтому в качестве первого параметра функции send() на стороне сервера мы должны передать наш новый дескриптор client. В дальнейшем мы будем называть client "клиентским дескриптором".

msg и buf – это указатели на последовательность байтов, которую нужно отправить.

len – это длина этой последовательности (учти, что при передаче текстовых строк для их правильного отображения необходимо также передавать и завершающий нулевой символ, или добавлять к принятой на другой стороне строке ноль в конце, иначе ты рискуешь замусорить свой экран в лучшем случае, а в худшем – отправить в даун свою программу).

flags – это тот же самый набор флагов, как и в sendto().

Аналогия между send() и sendto() очевидна. И действительно, первые четыре параметра этих функций (за исключением серверной части кода и фокуса с клиентским дескриптором) почти идентичны. Однако, это всего лишь внешнее сходство...

Самый первый вопрос, который обычно приходит на ум на этом этапе изучения сетевого программирования: "Но как send() узнает, кому надо слать данные?! Ведь мы не указываем адресата!" Да, действительно, адресата нет. А все потому, что send() оперирует с подключаемыми сокетами. То есть, эта функция уже знает, что s – это дескриптор подключенного сокета. Канал между двумя компьютерами налажен. Соответственно этот же канал является каналом по умолчанию. Функции send() достаточно дескриптора сокета для пересылки данных. Просто потому, что на данный момент этот дескриптор связывает нас со второй стороной соединения.

Теперь надо вернуться к тонкостям потоковой передачи данных, которые мы обсуждали в самом начале этой части. send() полностью возлагает на нас обязанности по контролю за потоком. Как уже было сказано, send() не гарантирует, что все данные из указанного буфера были поставлены в исходящий поток. Все зависит от состояния внутренних буферов сокета (они скрыты от наших глаз) и от других факторов. send() возвращает одно из двух значений... Как обычно это –1 или SOCKET_ERROR в случае ошибки, или количество байт, выставленных на конвейер в случае успеха. Как правило, количество байт, выставленных на конвейер равно len, но иногда (особенно при слишком большом объеме данных) оно может варьироваться от единицы до len. Если случилось так, что send() вернула число меньшее, чем len, то нам надо снова вызвать send() передав в качестве второго параметра указатель на оставшуюся часть данных и соответствующим образом изменив len. И так до тех пор, пока все данные не будут

```
char * cstr == "Hello, Networking Universe!"; // наша строка
int len = strlen (cstr) + 1;     // единицу мы добавили для концевого нулевого символа

int n = 0;
while (n < len)
{
  int sent = send (client, cstr + n, len - n, 0); // client? допустим, мы на сервере! :)
  if (sent < 0)
    // если произошла ошибка, здесь мы должны прервать выполнение цикла
  n += sent;
}
```

После прохождения такого цикла мы гарантированно получили одно из двух: либо произошла ошибка, либо данные были отправлены так, как мы этого хотели. Конечно, это не самый лучший способ написания такого кода, но мы пока не берем в расчет абсолютную рациональность и максимально упрощаем все для облегчения понимания.

Хмм... вроде бы все, что касается send() мы расписали... Остались только флаги (четвертый параметр) но мы отложим их на потом...

Пора взяться за recv()... :)

Объявление recv() выглядит следующим образом:

```
// Linux & FreeBSD

  int recv (int s, void * buf, int len, int flags);

// Windows

  int recv (SOCKET s, char * buf, int len, int flags);
```

accept()). Второй параметр – это указатель на буфер, куда будут записаны принятые данные. Третий – ожидаемый объем данных. И четвертый – это набор флагов, регулирующих деятельность recv() (их мы отложим на потом).

Как всегда, все выглядит просто и понятно, но есть такое правило: самое сложное содержимое скрывается в самой простой форме. Поэтому мы рассмотрим recv() поподробней...

Самый первый подвох – это значение, возвращаемое recv(). Вариантов значений может быть три, пройдемся по ним по порядку... Если произошла ошибка – recv() сигнализирует нам об этом также, как и все рассмотренные нами ранее функции. Второй вариант развития событий – это когда recv() возвращает 0... Это вовсе не значит, что было принято 0 байт. Это означает, что соединение закрыто. Поэтому мы должны отслеживать такие ситуации и правильно на них реагировать. И, наконец, третий вариант – это когда возврат recv() является целым положительным числом. Оно равно количеству полученных байт. Как уже говорилось выше, из-за особенностей потоковой передачи данных это значение может быть меньше, чем значение переданное в третьем параметре. Это тоже должно быть принято нами к сведению. Для того, чтобы наша программа могла должным образом реагировать на все три варианта событий мы можем написать «обертку» для recv() примерно такого вида:

```
#define BUFLEN 512

char cstr[BUFLEN]; // наш буфер приема

int n = 0;
while (n < BUFLEN)
{
  int recvd = recv (client, cstr + n, BUFLEN - n, 0); //client? допустим, мы на сервере!
  if (recvd < 0)
    // если произошла ошибка, здесь мы должны прервать выполнение цикла
  else if (recvd == 0)
    // соединение было закрыто
  else n += recvd;
}
```

Эта «обертка» гарантирует, что если количество данных известно наперед, они будут получены полностью, и она также следит за правильностью выполнения.

фактическая длина буфера, то скорее всего мы получим Segmentation Fault. Даже если этого не случится, то у нас в программе появится «бомба замедленного действия», которая рано или поздно «бабахнет».

Уфф... Вроде бы все что лежит на поверхности знаний о передаче по TCP мы освоили... Писать реализацию framework'а классов для работы с TCP-сокетами пока не имеет смысла, т.к. без полных знаний о функционировании сетей сделать это в полной мере мы все равно не сможем, поэтому в следующей части предлагаю приступить к более глубокому изучению принципов взаимодействия между компьютерами...

Страницы: 1 2 3 **4**

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

1 ноября 2003 (Обновление: 18 ноя 2009)

Комментарии [40]

Контакт
Сообщества
Участники
Каталог сайтов
Категории
Архив новостей