Полное руководство по сетевому программированию для разработчиков игр. Часть 3. UDP (2 стр)

Автор: x84

Теперь пора узнать, что происходит с дэйтаграммой, когда она достигает адресата... Первым делом система производит несколько тестов, которые определяют, являются ли служебные данные, включенные в дэйтаграмму, допустимыми для данной конкретной машины. Например, если в системе нет сокета, который бы прослушивал порт и адрес, указанные в дэйтаграмме, то система отбрасывает дэйтаграмму, как неверную (этот процесс называется "drop" - уронить, сбросить). Соответственно, получатель должен успеть открыть и привязать соответствующий сокет к правильному "имени" до того, как дэйтаграмма достигнет цели, иначе он не сможет прочитать данные, которых по сути-то нет. После того как система убедится, что дэйтаграмма легальна, она помещает ее в очередь входящих сообщений. После этого, наша программа сможет их оттуда прочитать.

Взглянем на код, который осуществляет чтение:

```
// Linux & FreeBSD

  int recvfrom ( int s, void * buf, int len, int flags,
                 struct sockaddr * from, int * fromlen);

// Windows

  int recvfrom ( SOCKET s, char * buf, int len, int flags,
                 struct sockaddr * from, int * fromlen);
```

Первый параметр функции recvfrom() идентичен соответствующему у функции sendto(). Второй - это буфер, в который будут записаны данные из дэйтаграммы. Третий - это длина ожидаемого сообщения. Четвертый - это опять

кого она пришла и куда отвечать (адресную структуру заполняет функция recvfrom()). Ну и шестой параметр - это длина адресной структуры, указанной в пятом параметре, длину устанавливает функция recvfrom(), исходя из данных в адресной структуре.

Хмм... Казалось бы все просто, но нет! С функцией recvfrom() все намного сложнее, чем с sendto().

Во-первых, многие часто неправильно толкуют значение пятого параметра (from). Кто-то заполняет структуру каким-то адресом, от которого он ожидает дэйтаграмму, и думает, что тем самым осуществляет некоторые функции файрволла. На самом деле, функция recvfrom() ожидает дэйтаграмму от того адреса, к которому привязан сокет, то есть от конкретного сетевого интерфейса в системе. И записывает адресную структуру сама, то есть помещает туда адрес и порт отправителя дэйтаграммы (recvfrom() берет их из служебных данных заголовка дэйтаграммы, к которому мы пока еще не умеем получать доступ). Этот механизм нужен для того, чтобы наша программа могла получить адрес отправителя и отослать туда ответ, если это требуется. Если в качестве пятого параметра передать 0, то адресная структура не будет заполнена (допустим, нас отправитель не интересует, мы просто принимаем дэйтаграммы от всех желающих, но никому не отсылаем ответ), но в этом случае мы не сможем узнать, кто отправил нам данные.

Вторая тонкость работы recvfrom() - это буфер и размер ожидаемого сообщения. Разумеется, мы никогда не можем знать точно, какого размера данные к нам придут. Если мы выделим слишком большой буфер и сообщение окажется меньше ожидаемого, то мы потеряем драгоценные байты памяти, без надобности простаивающие в ожидании заполнения. Если буфер будет меньше, чем фактическая длина пришедшей дэйтаграммы, то recvfrom() "отсечет" ту часть сообщения, которая не помещается в буфер. Как же быть?

Есть три варианта выхода из этой ситуации:

1. Использование только сообщений с фиксированной длиной, то есть и отправитель и адресат до начала отсылки должны знать, какого размера сообщениями они будут оперировать. Как правило, это число "жестко" зашивается в код (например, константой) - hard-coded value.

2. В первой отосланной и принятой датаграмме должна содержатся длина последующего значимого сообщения. Но этот вариант не всегда приемлем. Например, максимальная допустимая длина сообщения в стеке протоколов TCP/IP равна 65535, то есть значение длины самой большой дэйтаграммы не будет занимать больше двух байт. IP заголовок в лучшем случае содержит в себе 20 байт служебных данных (это число может варьироваться, в зависимости от самого IP заголовка), плюс UDP заголовок - еще 8 байт. Итого 28 байт. Получается, что для передачи двух байт, в которых будет записана длина последующей дэйтаграммы, которая будет содержать ценные пользовательские данные, нам необходимо отослать 28 байт. Соотношение служебные(бесполезные)/ пользовательские(полезные) данные составит 28:2 == 14:1. Нам такое вряд ли подходит...

соответствующего размера. Это опять же не самый выгодный вариант. Однако если мы знаем длину максимально возможного сообщения, то мы можем существенно сократить размеры промежуточного буфера.

Да... Не очень-то радужные перспективы, но на данный момент мы ничего не можем с этим поделать. Впоследствии мы научимся грамотно контролировать эту ситуацию, пока же ограничим наши исходящие и входящие буферы 100 байтами (в сумме будет ровно 128 - для ровного счета :)).

Еще один недостаток recvfrom() опять-таки связан с буфером и третьим параметром. Эта функция не славится интеллектом... Представим себе такую ситуацию:

1. Мы выделяем буфер размером 20 байт.

2. В качестве второго параметра мы передаем указатель на этот буфер, а в качестве третьего - целое значение 30 (например, ошиблись при расчете).

3. Приходит дэйтаграмма размером 25 байт.

4. Наша программа отправляется "в даун", вызывая ошибку сегментации памяти.

Почему? Очень просто... Зарезервированный размер буфера меньше, чем переданное в третьем параметре число, поэтому, пытаясь записать 21-й байт сообщения, recvfrom() выходит за пределы памяти, доступной конкретному приложению. Стек заполняется "мусором" и приложение "умирает"... В этом плане recvfrom() полностью полагается на благоразумность программиста, поэтому в наши обязанности входит контроль за фактическим размером буфера и значением, передаваемым в качестве третьего параметра...

Однако на этом перечисление всех тонкостей работы с recvfrom() не закончено... Дальше дело касается значений, возвращаемых recvfrom(). В случае ошибки она возвращает -1 или SOCKET_ERROR, как и все другие вышеперечисленные функции. А в функции успешного завершения - одного из двух значений: либо количество полученных байт, либо 0. 0 вовсе не означает, что было получено 0 байт, это значение означает, сокет был закрыт до вызова recvfrom(). Да. Такая ситуация считается нормальной, и чуть позже мы поймем почему... На данный момент мы будем считать это значение также ошибочным, но потом заставим наш каркас реагировать на это должным образом. Насчет количества полученных байт - это значение может быть меньше, чем указанное в третьем параметре (len), когда фактическая длина полезных данных, содержащихся в полученной дэйтаграмме, меньше, чем ожидаемая.

И, наконец, самое главное... Здесь мы впервые вводим понятие "блокирующие вызовы". Что это значит? А то, что во время выполнения блокирующей функции наша программа ждет окончания ее выполнения, а в случае с сетевыми приложениями такая ситуация встречается крайне часто. Функция recvfrom() - блокирующая. Например, если вся инициализация была проделана правильно, все параметры указаны верно, и мы вызываем recvfrom(), но

дэйтаграмма затеряется в пути (такое иногда бывает). Программу невозможно будет контролировать, и единственное средство, которое сможет помочь в такой ситуации - это принудительное завершение программы при помощи инструментов ОС (Диспетчер задач в Windows или команда kill в *nix). Конечно, не все так мрачно, есть средства, которые позволят нам обойти блокирование. Наша программа в дальнейшем сможет сама правильно обходить блокирование, но пока мы должны с этим смириться. :)

И еще об адресах

Существует как минимум два специальных адреса, которые нам интересны: 127.0.0.1 и x.x.x.255.

127.0.0.1 — это так называемый «закольцованный» адрес (loopback). Все пакеты, которые поступают от компьютера на этот адрес не покидают источник, а возвращаются обратно. То есть они не выходят за пределы одного компьютера, но обрабатываются сетевой подсистемой так, как будто это самый настоящий пакет, который прошел долгий путь от источника к адресату. Запустив программу-отправителя и программу-получателя на одном компьютере, мы можем из отправителя послать пакет на этот адрес и тогда получатель сможет его получить и обработать. Хм... "Для чего это нужно? Ведь есть же гораздо более удобные способы заставить две локально запущенные программы взаимодействовать между собой?!" - спросишь ты. Для тестинга! Для чего же еще? Ведь не у всех есть возможность создать специальную сеть компьютеров для тестирования сетевых приложений. То есть дает нам возможность проверить локально, насколько хорошо работает наша программа, и если тест пройден, то можно идти дальше на "реальный полигон" для испытания сетевого ПО - в Интернет.

Что же касается x.x.x.255 — то это так называемый «широковещательный» адрес (broadcast). Мы рассматриваем сети класса C. Если вместо x.x.x подставить адрес сети класса C, то получится широковещательный адрес, действительный для конкретной подсети. Например, 192.168.0.255. Что он нам дает? Очень многое. Пакет, отправленный по этому адресу будет получен всеми компьютерами находящимися в данном сегменте сети. То есть получается что-то типа массовой рассылки. Рассмотрим такую ситуацию: мы делаем сервер для игры в новую стрелялку по сети (локальной). Сервер запущен на какой-то из машин в этом сегменте сети. Клиенты хотят подключиться к нему и начать игру. Как им узнать адрес, на каком из компьютеров запущен сервер? Очень просто: они посылают широковещательный пакет в сеть. Его принимают все компьютеры, включая и тот, на котором запущена программа-сервер. Он отвечает каждому из клиентов, что, мол, "да! вот он я здесь вишу и жду подключений!". Ну а все остальные компьютеры просто откидывают (drop) этот пакет. Вот такая схема. Когда каждому из клиентов придет ответ от сервера - они посмотрят на адрес отправителя и тем самым узнают, на каком адресе висит сервер.

творится в игровом мире каждому из клиентов по отдельности... Ему достаточно лишь сформировать широковещательный пакет и выпустить его в сеть. Его получат все. Но обработан он будет только теми из компьютеров, которые заинтересованы в этой информации (слушают открытый сокет на определенном порту), а все остальные - просто откинут (drop) его за ненадобностью.

Об адресации и прочих интересных штуках мы поговорим подробно чуть позже, а пока нам нужно лишь знать заветный адрес 127.0.0.1 для того, чтобы тестить свой код. :)

Уффф. Ну вот. Теперь мы знаем, как отсылать и принимать текстовые строки по протоколу UDP. Я немного исправил код из второй главы, чтобы наш каркас смог отправлять и принимать сообщения. Пока мы ограничимся фиксированной длиной (100 байт, как уже говорилось). Взгляните на код и постарайтесь понять, что и как он делает.

Исходный код:
win: chapter03.zip
*nix: chapter03.tar.gz

Страницы: 1 **2**

#UDP, #сетевое программирование, #сокеты

31 июля 2003 (Обновление: 24 сен 2009)