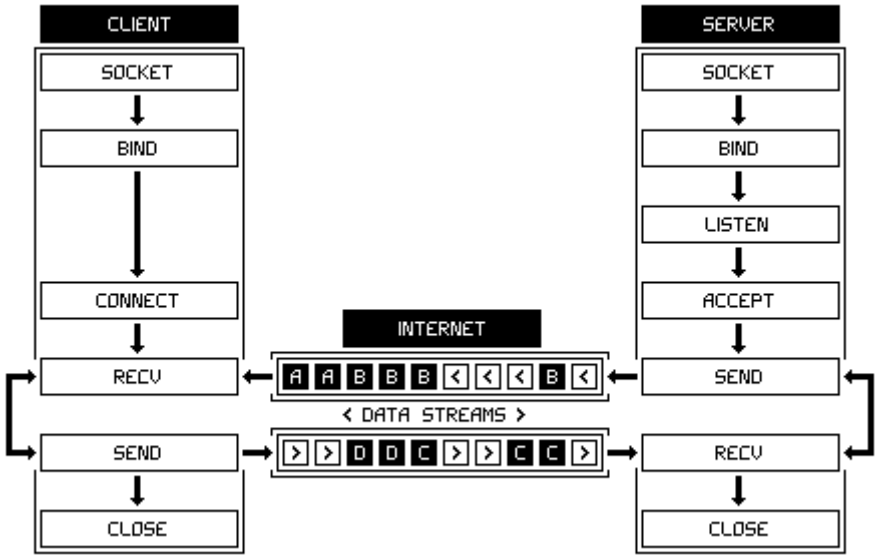
Полное руководство по сетевому программированию для разработчиков игр. Часть 4. TCP (2 стр)

Автор: x84

Схема кода клиента и сервера

Давай посмотрим на схему кода на обеих частях соединения - на клиенте и на сервере. А также попробуем сравнить ее с аналогичной для UDP.



заключается все-таки не в том, как устроен код на обоих концах соединения, а в том, как данные идут по сети.

Для начала необходимо понять разницу между пакетной и потоковой передачей данных. Когда мы имеем дело с пакетами, то данные передаются как бы обрывками – один пакет == одна порция. С потоками все несколько иначе. Когда мы создаем TCP-сокет (SOCK_STREAM, STREAM – поток) то сетевая подсистема нашей ОС создает два конвейера по которым данные передаются в обе стороны. Мы снова рассмотрим ситуацию с пересылкой четырех строк. Сервер должен послать клиенту строки "AA" и "BBBB", а клиент должен передать серверу "CCC" и "DD".

Итак, на иллюстрации мы видим два потока, соединяющих клиента с сервером. Эти потоки работают, как конвейеры. Представим, что скорость каждого из конвейеров варьируется (постоянной скорости нет, она меняется). Получается, что если мы равномерно (с постоянной скоростью) будем "выкладывать" данные на конвейер, а конвейер движется неравномерно, то данные окажутся хаотично разбросанными по всей ленте конвейера. На иллюстрации четко видно, что данные разорваны, но их общий порядок следования не изменен. То есть они все равно сохраняют структуру, "кто за кем идет". Это и есть принцип потоковой передачи данных. То есть в идеале данные будут сгруппированы именно в такие логические группы, какие определил программист (строки не будут разорваны). Но в реальности же на соединение влияет множество факторов, начиная от физического состояния сетевых коммуникаций и заканчивая степенью загрузки конечных и промежуточных станций, а также пропускной способностью сети. Ты когда-нибудь смотрел потоковое видео в Интернет? Например, какой-нибудь очередной трейлер супер-блокбастерной игры? Видел, наверное, что видеопоток поступает с задержками, величина которых постоянно скачет. То есть, как только видепроигрыватель получил очередную порцию данных – он их проигрывает. Если все, что поступило на данный момент уже показано пользователю, то проигрыватель притормаживает, ожидая новую порцию потока. Протокол TCP гарантирует, что все переданные данные будут доставлены полностью и том же порядке, в котором они были выложены на конвейер, но он не гарантирует, что поток будет непрерывен, потому что в этом смысле состояние потока зависит не от протокола. Кстати, официального термина "конвейер" нет, я просто использовал его для удобства объяснения. Официально все это называется "поток" – stream.

Данные в TCP не делятся на группы. Это значит, все, что поступает в поток – интерпретируется как единое целое. Протоколу TCP нет дела до того, какого рода данные мы выпускаем на поток. Он работает с последовательностью байтов. Поэтому он не может гарантировать, что если мы попытаемся выложить строку "BBBB" на поток, то она будет выложена вся сразу. Возможно, она будет выложена в два захода, или же в три. Количество "попыток" не определено. То же самое касается не только отправителя, но и принимающего. То есть принимающий в конкретный момент времени может не принять ничего, может принять сразу обе строки "AA" и "BBBB" или же может принять "AABBB", а оставшуюся часть строки – только "со второго захода". Однако мы можем это контролировать. Но пока еще мы ничего не знаем о том, как выглядит программная часть на обоих концах соединения... Ну так чего же мы медлим?!

"Захады, дарагой! Гостэм будэш!"

использовать SOCK_STREAM (TCP - потоковая передача данных). То есть вызов socket() должен выглядеть так:

```
int error = socket (PF_INET, SOCK_STREAM, 0);
```

После этого сервер может (по желанию) опубликовать свой адрес в Интернет при помощи вызова bind(), тем самым привязав сокет к определенному интерфейсу/порту. В качестве адреса он может использовать любой из тех, которые существуют в системе. Или применить INADDR_ANY для задания адреса – тогда он сможет принимать запросы на подключение и данные с любого адреса системы (имеются ввиду не IP-адреса клиентов, а адреса сетевых интерфейсов системы (например сетевых карт или модемных соединений), если их несколько). Вызов bind() для клиента проводится в добровольном порядке, но, как правило, это лишнее. В любом случае, и сервер и клиенты должны заранее знать, номер порта, через который они будут взаимодействовать. Обычно, это число (номер порта) либо "жестко" зашивается в код (hard-coded value), либо публикуется в общедоступном для клиентов месте (например, на www-сайте проекта), а в сами программы клиентов должны быть встроены средства, позволяющие сменить порт, назначенный по умолчанию... В клиентах привязывать сокет к адресу/порту вовсе необязательно. Дело в том, что клиент, как правило, первый инициирует соединение. Он отправляет запрос на подключение серверу. А в этом запросе содержится "обратный адрес" клиента. Таким образом, сервер всегда знает, кто к нему подключился. Позже мы увидим, как все это работает.

Имейте ввиду, что несмотря на то, что на схеме все действия как на стороне клиента, так и на сервере находятся на одних и тех же уровнях, это вовсе не означает, что эти действия должны проделываться одновременно.

Далее на сервере мы должны создать очередь, куда будут помещаться запросы на подключение. То есть мы должны заявить миру о своем гостеприимстве, как бы говоря: "Ок, все желающие - добро пожаловать! Наш сокет всегда открыт для вас!". Мы должны вызвать функцию listen():

```
// Linux & FreeBSD

  int listen (int s, int backlog);

// Windows

  int listen (SOCKET s, int backlog);
```

подключение, одновременно находящихся в очереди. То есть, вызывая listen() мы говорим сетевой подсистеме нашей ОС: "Ок, дорогая, я хочу, чтоб к этому сокету не могло подключиться народу больше, чем указано в backlog, то есть не хочу, чтобы количество запросов в очереди в каждый конкретный момент времени превышало число backlog". Хммм... Казалось бы все просто, ан нет! :) На самом деле мы не можем создать очередь длиной 4294967295 единиц. Это слишком много. Если мы попытаемся это сделать, то система скажет нам: "Да ты что, родной?! Совсем свихнулся?! Хочешь в могилу меня свести?!". Поэтому в операционных системах существует ограничение на количество вхождений в очереди backlog. Если мы укажем слишком большое число, или вообще передадим отрицательное, то система втихаря (она нам об этом не скажет) ограничит эту очередь до приемлемой длины (до максимума, допустимого в системе). Этот максимум можно установить вручную, но сейчас мы пока не будем этим заниматься, нам пока хватит очереди длиной в 10 вхождений максимум...

Как это работает? Клиент отсылает запрос на подключение и сетевая подсистема сервера помещает этот запрос в очередь. Если в очереди запросов нет места, то клиент получит отказ в подключении. Задача сервера - рассмотреть каждый из запросов, принять его и обработать (обслужить) клиента должным образом, после чего он может переходить к следующему запросу. То есть все это происходит в цикле. Когда очередь пуста, сервер находится в ожидании следующих клиентов. Когда в очереди стоит кто-то из клиентов, сервер принимает его, удаляет его запрос из очереди и обслуживает клиента.

Вызов listen() не блокирует программу. Он просто переводит сокет в "пассивный" режим и немедленно возвращается. listen() возвращает либо 0 в случае успеха, либо сигнализирует об ошибке, так же, как это делают и другие функции... После listen() сокет окажется переведенным в прослушивающий режим и он будет принимать запросы от клиентов.

Страницы: 1 **2** 3 4 Следующая »

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

1 ноября 2003 (Обновление: 18 ноя 2009)

Комментарии [40]