Сетевое программирование

Мы рассматриваем сетевое программирование как использование некоторых стандартных библиотек с целью обеспечить обмен данными между отдельными "абонентами", т.е. независимыми программами, работающими в общем случае на разных машинах и объединенных сетевыми линими связи.

В целом о принципах и устройстве сетевых взаиможействи можно говорить очень долго, и для этого недостаточно даже годового специального курса. Но в некоторых аспектах мы можем познакомиться с простейшими решениями, не проникая в глубины.

Определим несколько необходимых понятий, которые нам потребуются, чтобы описывать логику работы. Однако опишем их не совсем строго, чтобы по возможности избежать погружения в технические детали.

Станция — компьютер, подключенный к сети (имеющий сетевой интерфейс).

Протокол — правила, регламентирующие представление данных и алгоритмы их обработки при решении определенного круга задач сетевого взаимодействия.

Пакет — обособленная единица передачи информации по сети, состоящая обычно из заголовка и вложенного набора данных.

Сетевой адрес — система идентификации конкретных станций в рамках глобальной или локальной сети.

Порт — условный канал ввода/вывода, привязанный к сетевому интерфейсу на конкретной станции; идентифицируется своим номером.

Традиционно выделяют несколько уровней сетевого взаимодействия — физический, канальный, сетевой, транспортный, сеансовый, представления, приложения.

Мы сейчас не будем вникать в их специфику, поскольку в простейшем случае работа приклалдного программиста сосредоточена в рамках транспортного уровня.

Задачей транспортного уровня является передача данных от программы, работающей на одной станции, к программе, работающей на другой станции. Эта задача решается двумя базовыми сценариями работы.

Датаграммный протокол (например, UDP) — аналог почтовой службы. Подготовили письмо, написали адреса получателя и отправителя, отправили письмо (и больше к нему отношения не имеем). Для получения данных проверили "почтовый ящих", извлекли оттуда письмо, если оно там есть, прочитали его содержимое и адрес отправителя (пригодится, чтобы ответить).

Потоковый протокол (например, TCP) — аналог общения по телефону. Выяснили адрес другого "абонента", запросили соединение с данным абонентом, получили соединение. После этого у нас открыт двусторонний канал обмена данными — можем писать в этот канал и читать из этого канала без дополнительного указания адресной информации, она уже зафиксирована в соединении. Когла надоест — запрашиваем разрыв соединения, получаем подтверждение разрыва и теперь свободны.

В первом случае говорят о доставке данных без гарантии, поскольку протокол не следит за тем дошло сообщение или нет. Во втором случае наличие соединения гарантирует доставку, т.е. пока соединение в силе, протокол проверяет дошли ли данные до другого абонента и в случае их потери дублирует передачу без нашего вмешательства. Естественно, датаграммная передача работает значительно быстрее, чем потоковая, поскольку нет необходимости следить за корректностью передачи данных. То есть при выборе протокола мы имеем альтернативу скорость–надежность и должны ее разрешать, исходя из конкретных ситуаций и задач.

Идентификация абонентов

Реализация сетевого программирования может поддерживаться разными библиотеками. Кроме того, само устройство сетей может быть разным. Поэтому здесь мы будем говорить только о сети Internet (или как еще говорят IP-сеть), что сразу ограничивает сетевую идентификацию так называемым IP-адресом и интернет протоколом (IP).

В настоящее время параллельно используются две версии IP, это версии 4 и 6. Для наших простейших нужд достаточно знать, что они отличаются форматом IP адреса. В четвертой версии IP адрес занимает 4 байта, в 6 версии — 16 байт. Пока что мы здесь для простоты рассмотрим интерфейс относительно 4 версии IP протокола. IP адрес принято записывать в так называемой точечной нотации, где каждый байт записвается десятичным числом с разделением точками, например, 192.168.143.101. Сейчас для нас не важно какой смысл имеет каждое из этих чисел. Просто пока считаем, что мы знаем нужные нам адреса.

IP адрес определяет станцию (точнее, ее конкретный сетевой интерфейс, станция может иметь несколько сетевых интерфейсов и соответственно несколько IP адресов).

На конкретной станции может функционировать несколько процессов (программ), которым нужна сеть. Чтобы исключить двусмысленность, каждый процесс захватывает отдельные каналы ввода-вывода для работы с сетью (порты). Порты идентифицируются натуральным числом, и мы должны позаботиться о том, чтобы захватить себе свободные порты (которые не заняты другими программами). Зарегистрированные сетевые службы занимают порты с зарегистрированными номерами. Использование занятых портов вашей программой может нарушить работоспособность системы. Поэтому номер порта для своих жкспериментов надо выбирать осторожно. В поисковых системах можно найти информацию о том какие порты зарегистрированы за какими службами. Информацию о том какие порты в данный момент задействованы на вашей станции можно получить по консольной команде `netstat -an`. Однако на практике почти всегда можно брать номер порта больше 5000 и все будет (как правило) работать номально.

Мы рассмотрим один из вариантов так называемого socket интерфейса, который позволяет пользователю создать канал сетевого ввода-вывода, привязать этот канал к сетевым интерфейсам, выполнять ввод-вывод данных через этот канал и анализировать успешность выполняемых операций. В рамках этого интерфейса за идентификацию отвечает структура

struct sockaddr_in

Эта структура содержит несколько полей, нас будут интересовать те, которые отвечают за семейство адресов, порт , IP адрес . Они так и называются

```
short    sin_family;
u_short  sin_port;
struct in_addr sin_addr;
```

При отправлении сообщений мы должны заполнить эту структуру необходимыми данными. При получении сообщения мы также имеем возможность получить такую структуру с данными, касающимися отправителя сообщения, и тем самым извлечь и использовать эти данные в дальнейшей работе приложения.

Семейство адресов у нас всегда будет AF_INET — интернет адреса. Порт — это просто целое число. А вот с адресом ситуация чуть сложнее, и чтобы не напрягаться, можно воспользоваться специальной функцией, которая даст нам представление адреса в нужной форме.

```
unsigned short port = 12345;
struct sockaddr_in addr;
struct hostent *hostinfo;
hostinfo = gethostbyname ("127.0.0.1");  // DNS сервис
if (hostinfo == nullptr) что-то пошло не так ...
addr.sin_family = AF_INET;
addr.sin_port = htons (port);   // объясним потом
addr.sin_addr = *(struct in_addr*) hostinfo->h_addr;
```

Вообще говоря, заполнить поле IP адреса можно разными способами. В учебных примерах из Интернета можно их увидеть.

Работа с протоколом UDP

UDP — User Datagram Protocol.

Как уже говорилось выше, этот протокол поддерживает передачу данных без гарантии. Тем не менее, в хорошо работающей сети шансы потерять данные при работе с UDP стремятся к нулю. Проблема может проявиться в случае очень сильно загруженных линий связи либо при очень слабой станции, которая просто не успевает обработать все все входящие пакеты.

Для работы с протоколом UDP в рамках socket итерфейса нужно выполнить следующие действия вызовы библиотечных функций.

1. Определиться с адресатами, например, создать структуры

```
struct sockaddr_in addr_to;    // адрес получателя
struct sockaddr_in addr_this;  // мой адрес (отправителя)
```

и заполнить из нужными значениями.

2. Создать канал ввода-вывода для сетевого общения

```
int sock = socket(PF_INET,SOCK_DGRAM,0);
```

— ProtocolFamily Internet, датаграмный протокол, 0 - для UDP.

3. Привязать канал (сокет) к сетевому интерфесу нашей станции (addr_this)

```
bind(sock, (struct sockaddr*)&addr_this, sizeof(addr_this));
```

— преобразование укаателя нуюно так как в описании этого параметра стоит другой тип (так исторически сложилось).

4. Теперь можно отсылать сообщения или принимать сообщения

```
int nbytes = sendto(sock, buf, buflength, 0, (struct sockaddr*)&addr_to, sizeof(addr_t
```

— отсылается байтовый буфер указанной длины с указанием адресной информации куда, nbytes покажет сколько на самом деле удалось отослать.

```
int nbytes = recvfrom(sock, buf, BUFLEN, 0, (struct sockaddr*)&addr_from, &size);
```

— попытка читать в байтовый буфер с указанной максимальной длиной, addr_from будет содержать алресную информацию отправителя этого сообщения, nbytes покажет сколько на самом деле удалось прочитать.

5. Закрыть канал, если он стал уже не нужен

```
close(sock);
```

Проблема асинхронности

В сетевом взаимодействии в полной мере проявляется проблема асинхронности, состоящая в том, что станции работают независимо, и практически невозможно предугадать к какой момент станция отправит означаетсвои данные и в какой момент они придут на принимающую станцию.

По умолчанию процедуры ввода-вывода работают в блокирующем режиме. Это означает, что если канал ввода-вывода не готов выполнить требуемую операцию, то выполнение запроса на ввод-вывод блокируется до появления в канале возможности выполнить эту операцию. При вводе блокировка естественно возникает, если в канале еще нет данных для чтения, при выводе блокировка может наступить если канал еще не закончил выполнять предыдущий вывод.

При попытке чтения из пустого канала ввод будет заблокирован, и наш процесс будет ожидать появления данных. Но придут ли эти данные вообще? Вдруг отправляющая станция перестала работать? В этом случае принимающая сторона будет бесконечно долго и безуспешно ждать.

Для решения этого вопроса принимающая сторона должна иметь возможность ожидать ответа некоторое время, после чего принимать решение нужно ли продолжать это ожидание или стоит закончить работу (или произвести новый запрос). Подобный механизм обычно называется таймером, устанавливающим определенный интервал времени на выполнение некоторых действий (или бездействий как при ожидании). Таймер можно реализовать на базе раличных системных вызовов. Здесь мы рассмотрим простейший частный случай таймера на ожидание ввода данных. Для этого используются два системных механизма

— отмена блокировки на вводе,

— блокирование (засыпание) процесса на заданный период времени.

Для отмены блокировки на вводе можно использовать системный вызов fcntl (установка режимов для канала обмена).

```
fcntl(fd, F_SETFL, O_NONBLOCK);
```

— установить каналу fd флаг отсутствия блокировки.

Если по умолчанию операция чтения данных из канала fd блокировалась до появляния возможности эти данные прочитать, то теперь каналу установлен флаг отмены блокировки, и вызов функции чтения завершится сразу, либо прочитав имеющиеся к моменту вызова данные, либо с указанием, что никаких данных не было прочитано. Теперь можно в цикле повторять попытки чтения до тех пор пока либо не придут данные для чтения, либо не истечет необходимое время. А вот необходимый интервал время можно отмерять, периодически "засыпая" на некоторое время.

Для засыпания в Linux можно использовать две функции

```
sleep (int num_seconds);
usleep (int num_micro_seconds);
```

При вызове этих функции процесс выключается на указанное время и не занимает системные ресурсы (процессор и т.п.), а потом возобнавляется и продолжает работу с кода, следующего за данным вызовом.

Специфические тонкости

Порядок байтов.

Так исторически сложилось, что на разных архитектурах вычислительных систем приняты разные решения по поводу хранения чисел. Например, представление чисел с плавающей точкой обычно следует стандарту IEEE-754, но также изредка встречаются представления по стандартам HFP, BFP, DFP. Сейчас нас это не особо волнует, поэтому здесь ограничимся только упоминанием.

С представлением целых чисел ситуация более запутанная. Проблема в порядке байтов, т.е. по каким адресам располагаются в памяти младшие и старшие байты в представлении целого числа. При размещении целого числа в памяти в качестве адреса ему сопоставляется начальный (наименьший) адрес байта среди всех байтов составляющих представление этог числа, т.е., если int x; char *p = (char*)&x; то байты числа x будут размещаться на позициях `p, p+1, p+2, p+3`. Но в каком порядке? В архитектуре x86, с котороый мы в основном работаем, принят порядок от младшего к старшему, т.е. сначала в памяти располагается младщй байт, потом следующий, и в самом конце старший. Эта система называется "little-endian".

В других архитектурах может быть принят порядок от старшего к младшему ("big-endian"), когда по адресу `p` лежит старший байт числа, а младший будет по адресу `p+3`. Именно такой порядок принят в стандартах обработки данных сетевыми протоколами.

Работая с сетевыми протоколами, мы должны учитывать эту особенность представления чисел, поскольку мы с своих программах формируем данные (адреса, номера портов) для сетевых функций и должны их передавать в правильной форме. То есть в нашем случае нужно преобразовать числа из "little-endian" в "big-endian" и наоборот.

Но с порядком байтов в представлении целого числа ситуация еще сложнее. Во-первых, этот порядок иногда можно переключать настройками операционной системы и аппаратуры, во-вторых, существуют системы со смашанными порядком — частино big, частично little endian. Чтобы не заморачиваться с выяснением что именно творится в вашей системе, библиотека предоставляет несколько функций для преобразования целых чисел между такими представлениями. Мнемоника названия этих функции опирается на слова “host-to-network” и “network-to-host”, т.е. преобразование из представления, принятого на вашей машине в сетевое и наоборот.

```
uint32_t htonl(uint32_t hostlong);   // long  - 4-байтовое число
uint16_t htons(uint16_t hostshort);  // short - 2-байтовое число
uint32_t ntohl(uint32_t netlong);
uint16_t ntohs(uint16_t netshort);
```

Так как речь идет только о порядке байтов, то числа интерпретируются как беззнаковые.

В нашем случае мы задаем номер порта для дальнейшего использования его в сетевых протоколах. Поэтому мы и писали, преобразовывая значение переменной `port` в сетевое представлние

```
addr.sin_port = htons (port);
```

Если мы захотим, например, распечатать номер порта отправителя после получения от него сообщения, то придется делать так

```
recvfrom(sock, buf, BUFLEN, 0, (struct sockaddr*)&addr_from, &size);
printf("port %d\n", ntohs(addr_from.sin_port));
```

Нужно ли перекодировать любые массивы данных, которые мы пересылаем по сети? Ответ — нет. Данные будут получены именно в том виде, в котором они были подготовлены, т.е. каждый байт данных останется на том же месте, которое он занимал перед передачей. Переколировка касается только тех данных, которые имеют управляющий смысл для работы сетевых программ. В нашем случае — это только номера портов и только при заполнении адресных структур. Все остальные данные — это зона нашей ответсвтенности и мы сами с ними разбираемся как при передаче, так и после приема.

Точечная нотация IP адреса.

Точечная нотация IP адреса — это удобная общепринятая форма его записи. Однако с точки зрения сетевых протоколов — это просто 4-байтовое число. Поэтому тут также возникает задача преобразования записи адреса из одной формы в другую. В сетевых библиотеках есть множество функций для выполнения подобных преобразований. Мы использовали только один из возможных вариантов.

```
struct hostent *hostinfo;
hostinfo = gethostbyname ("127.0.0.1");
addr.sin_addr = *(struct in_addr*) hostinfo->h_addr;
```

В данном варианте выполняется гораздо более общее преоразование. Функция gethostbyname на самом деле обращается к сервису регистрации доменных имен, и по указанному имени возвращает структуру с описанием свойств определенной сетевой станции, в частности, ее IP адрес. В данном примере мы запросили просто преобразование адреса из точеченой нотации в его сетевое представление, но могли бы запросить адрес любого другого абонента по его доменному имени, например,

```
struct hostent *hostinfo;
hostinfo = gethostbyname ("www.yandex.ru");
```

Другой вопрос, что бы мы с этим адресом стали делать?

Обратное преоразования сетевой записи адреса в точечную нотацию можно выполнить, например, так

```
recvfrom(sock, buf, BUFLEN, 0, (struct sockaddr*)&addr_from, &size);
printf("ip addr %s\n", inet_ntoa(addr_from.sin_addr));
}
```

Простые примеры

Примитивный пример.

Одна станция отправляет сообщение. Другая его принимает и отправляет обратно добавив в конце слово Echo. Первая станция распечатывает этот ответ. По ходу дела станции также печатают на экран всякую служебную информацию.

Пример для протокола UDP — cli0.cpp ser0.cpp.

Компилируем и собираем два отдельных исполняемых файла и запускаем их в разных консолях — сначала сервер, потом клиент.

Менее примитивный пример.

Станция занимается рассылкой сообщений всем, кто у нее зарегистрируется. Сообщеним является информация о текущем времени. Эти сообщения возникают в случайные моменты времени. Регистрация получателей выполняется отсылкой запроса на эту станцию со стороны клиента, который хочет эти сообщения получать. Будем для простоты считать, что таких клиентов может быть не более 8 и они уже не выходят из обслуживания до конца работы (станция будет просто сохранять их адреса в массиве по простейшей схеме).

В этой схеме у нас есть два источника асинхронных событий — запрос на обслуживание от клиента и сообщение о времени от “сервера”. Сообщения и запросы возникают в случайные моменты времени, и обе стороны взаимодействия не могут предугадать когда это произойдет. Основой реализации сервера является бесконечный цикл, в котором сервер проверяет наличие запросов от клиентов на регистрацию (и соответственно регистрирует их), а в некоторые случайные моменты этого цикла рассылает всем зарегистрированным клиентам сообщение — текущую дату и время.

Мы модифицируем предыдущий вариант UDP взаимодействия.

Пример для протокола UDP — cli2.cpp ser2.cpp.

Компилируем и собираем отдельно сервер и отдельно (несколько) клиентов. У разных клиентов надо задавать разные номера портов, чтобы они не пересекались с другими клиентами. В отдельных консолях запускаем сначала сервер и потом несколько клиентов.