Технология клиент-сервер

Эта технология описывает вариант "распределения ролей" в общей задаче сетевого общения. Обычно она используется при предоставлении некоторого сервиса. Мы говорим о сервере как о пассивной стороне общения, которая и предоставляет требуемый сервис по запросам. Клиент в этой схеме — это активная сторона, которая формирует запросы к серверу и получает от него ответы на эти запросы. В общем случае клиентов может быть много и они независимы друг от друга, а сервер — это одна сущность, которая отвечает за обработку запросов.

Схему клиент-сервер можно реализовать и на основе протокола UDP. Однако при этом сервер должен поддерживать регистрацию клиентов и соответственно проверку того кто и когда прислал ему запрос, хранить в некотором смысле историю запросов и т.п., чтобы поддерживать единый контекст общения с отдельно взятым клиентом. Поэтому схема клиент-сервер обычно реализуется на основе протокола TCP.

Напомним, что TCP работает в терминах соединения, и тогда клиенты могут устанавливать TCP соединения с сервером и в рамках этих соединений (пока они не закрыты) решать все интересующие их вопросы.

Иллюстрация: схема работы сервера с несколькими клиентами.

Клиент посылает запрос на установление соединения (connect) — сервер подтверждает и выделяет у себя отдельный канал (сокет) для работы с этим клиентом, клиент же работает со своим изначальным сокетом. Потом клиент или сервер могут послать запрос на закрытие соединения (close)

Пока соединение действует, клиент и сервер могут обмениваться сообщениями по сокетам, связанным с этим соединением.

Проблемы:

1. сервер должен как-то отслеживать с кем у него установлено соединение, регистрировать новые соединения и удалять закрытые

2. сервер должен как-то понимать кто из клиентов к нему обращается, т.е. обнаруживать сам факт входящего сообщения от клиента в рамках существующих соединений и выстраивать дисциплину обслуживания, если обращается несколько клиентов одновременно

3. обслуживание одного клиента не должно существенно влиять на ожидание обслуживания других клиентов (справедливое разделение времени).

Клиент-сервер на основе TCP

Базовые шаги по работе с протоколом TCP в рамках socket интерфейса состоят в следующем.

1. Определиться с адресатами, например, создать структуры

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

и заполнить их нужными значениями.

2. Создать канал ввода-вывода для сетевого общения

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

— ProtocolFamily Internet, потоковый протокол, 0 - для TCP.

Пока эти шаги полностью аналогичны работе с UDP. А вот дальше начинаются отличия. Клиент и сервер принципиально отличаются друг от друга тем, что у клиента только один абонент — сервер, а у сервера может быть много абонентов-клиентов. Поэтому системные вызовы будут разными для клиента и сервера.

На стороне клиента.

3. запросить TCP соединение

```
connect (sock, (struct sockaddr*)&addr_to, sizeof(addr_to));
```

В данном случае выполнять привязку сокета к своему сетевому интерфейсу вызовом bind не обязательно. Протокол TCP выполнит эту привязку по своему разумению. Но если вам надо выполнить привязку к какому-то конкретному интерфейсу вашей станции (в работаете на сетевом шлюзе у которого несколько сетевых карт), то можете предварительно вызвать bind с адресным параметром addr_this, в котором указан конкретный интефейс.

Зачем мы вызывали bind в UDP?. Затем, чтобы при посылке сообщения командой sendto в отправляемый пакет прописывался алрес отправителя, и на принимающей стороне было понятно куда посылать ответ. Здесь при успешном завершении connect будет установлено соединение, и протокол TCP сам позаботится, чтобы в рамках этого соединения данные передавались именно между нашими станциями.

4. Теперь можно отсылать сообщения или принимать сообщения через имеющийся сокет sock

```
send (sock, buf, buflen, 0);
```

— отправить сообщение TCP

```
recv(sock, buf, BUFLEN, 0);
```

— прочитать сообщение TCP

Также можно писать и читать через общие функции ввода/вывода

```
write (sock, buf, length);
read (sock, buf, BUFLEN);
```

Все эти функции возвращают количество переданных/прочитанных байтов или отрицательный код ошибки.

5. По окончании работы закрыть сокет

```
close (sock);
```

На стороне сервера.

Тут все немного по-другому, поскольку мы должны поддерживать соединения от нескольких клиентов.

3. Привязываем сокет к сетевому интерфейсу сервера (в точности как в UDP)

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

4. Организуем очередь необработанных запросов на входящие TCP соединения

```
listen (sock, n);
```

— установить длину очереди в `n` необработанных запросов.

5. Разгрузка очереди необработанных запросов на соединение

```
int new_sock = accept (sock,(struct sockaddr*)&addr_from, sizeof(addr_from));
```

— взять из обчерели запрос на соединение, установить соединие, `new_sock` — это теперь сокет, который отвечает на стороне сервера за данное соединение. Таким образом, на сервере появится несколько сокетов, каждый из которых будет отвечать за общение со своим клиентом, а исходный сокет `sock` будет принимать запросы на соединения от новых клиентов.

6. Завершение работы

```
close(new_sock);  // больше не работаем с этим клиентом
close(sock);      // больше не принимаем запросы на соединения
```

Итак, в простейшем виде работа клиента с сервером описыватся примерно такой схемой

```
клиент                                  сервер
       заполнение адресных структур
sock = socket(...)                      sock = socket(...)
                                        bind(sock,...)
                                        listen(...)
connect()  - запрос соединения ->       new_sock = accept(sock,...)
write(sock,...) ---  данные  --->       read(new_sock,...)
read(sock,...)  <--- данные  ----       write(new_sock,...)
.....................................................
          или  send  <---->  recv
.....................................................
close(sock)                             close(new_sock)
                                        close(sock)
```

В этой схеме много неясного. Во-первых, любая операция обмена данными (connect, accept, read, write, send, recv) может работать в блокирущем или неблокирующем режимах. В блокирующем режиме нам придется ждать на этих вызовах прихода данных. В неблокирующем режиме надо устраивать проверки, повторы и таймеры. Во-вторых, тут не видно как сервер разберется с множеством клиентов.

Проблема в том, что, сервер не должен блокироваться на вызове accept, так как при блокировке он больше ничего не станет делать, в том числе и обслуживать соединившихся клиентов. Если accept не блокировать, то возникает вопрос как отслеживать входящие запросы на новые соединения, т.е. надо регулярно возвращаться к этому вызову, чтобы не пропустить обработку таких запросов.

Решение этой проблемы основывается на двух основных принципах — контроль состояния каналов ввода-вывода и многопоточная работа сервера (каждому клиенту свой отдельный поток). Обе этих техники затрагивают системные функции управления ресурсами вычислительной системы. Рассмотрм их по отдельности.

TCP сервер на базе select

Вычислительная система предоставляет процессу некоторое количество каналов ввода-вывода. Системная функция select позволяет проверить состояние некоторого множества таких каналов. Проверка затрагивает состояния “есть что читать”, “можно писать”, “в канале ошибка”. Проверка может продолжаться некоторое время — мгновенное, в течении интервала времени, с блокировкой до появления события.

По пунктам.

1. Множество рабочих каналов ввода/вывода задается множеством типа `fd_set`. По сути оно работает как битовое множество:

```
fd_set  iod;            // input output descriptors
FD_SETSIZE              максимальное количество элементов в множестве
FD_ZERO( &iod );        обнулить множество
FD_SET( k, &iod );      добавить дескриптор k в множество (k-й элемент = 1)
FD_CLR( k, &iod );      удалить дескриптор k из множества (k-й элемент = 0)
FD_ISSET( k, &iod );    проверить установлен ли элемент k в 1
```

2. Временной интервал для проверки состояни каналов

```
struct timeval {
    long int tv_sec;  // seconds
    long int tv_usec; // microseconds
};
\begin{verbatim}


Функция select.
```

```
Создаем три множества для контроля указанных трех состояний (на самом деле --- только
Устанавливаем единицы для тех каналов, состояния которых хотим проверить по указанным

\begin{verbatim}
    fd_set  read_set, write_set, exc_set;
    struct timeval tv;    // секунды и микросекунды
    int ret = select ( size, &read_set, &write_set, &exc_set, &tv );
```

По завершении функции select структурные параметры сохраняют единицы у тех каналов, в которых возникли указанные состояния. Мы можем проверить где остались эти единицы (т.е. состояния этих каналов) и далее действовать в соответствии с логикой нащей работы. Так как структурные параметры функции select могут быть изменены в результате вызова, то перед следующим вызовом их надо обновлять.

Идея реализации сервера.

Все сокеты, обслуживающие активные соединения от клиентов, и начальный сокет, обслуживающий запросы на установление соединения, регистрируются в множестве типа `fd_set`.

Копия этого множества отправляется как параметр в вызов функции select, которая оставляет в этом множесте только те дескрипторы каналов, в которых есть данные для чтения.

Далее в этом полученном множестве ищутся ненулевые элементы (т.е. каналы, в который есть что читать) и обрабатываются данные этих каналов (либо accept, либо read|recv).

При закрытии какого-либо соединения соответствующий дескриптор сокета исключается из множества активных дескрипторов и более в проверке не участвует.

Пример. Предположим, что сервер способен ответить на запрос клиента сразу одним сообщением. Но это ответное сообщение может иметь разную длину. Для клиента есть несколько вариантов работы:

— сервер может первым делом сообщить клиенту длину своего сообщения. Тогда клиент создает буфер под эту длину и спокойно прочитает сообщение сервера.

— сервер не сообщает длину, а обозначает конец сообщения некоторым согласованным с клиентом условным кодом. Здесь клиент не может с гарантией предоставить для чтения буфер достаточного размера, но может читать сообщения порциями и проверять наличие специального кода конца.

В обоих случаях схема сервера будет выглядеть примерно так.

```
иициализация сетевых интерфейсов
sock = socket(...);
bind(sock ...);
listen(sock ...);
fd_set active;  // список всех активных сокетов
FD_ZERO( &active);
FD_SET(sock, &active);
fd_set ready;   // какие сокеты можно читать
бесконечный цикл
   ready = active;
   select (..., ready, ...);
   проверка ready на наличие элементов
   если ready содержит sock, то  // есть запрос на новое соединение
      newsock = accept(sock ...)
      добавить  newsock в active
   если ready содержит другой дескриптор fd, то  // пришли данные от клиента
      прочитать запрос клиента из fd
      обработать запрос клиента
      ответить клиенту через fd
      (а если клиент просит завершить соединение?)
```

```
        (тогла закрыть fd и исключить его из active)
    конец бесконечного цикла
```

Возможны случаи, когда обработка запроса конкретного клиента может занять существенное время. Во время этой обработки сервер не сможет реагировать на запросы других клиентов. В такой ситуации можно разбить ответ одному клиенту на несколько шагов, и между этими шагами пытаться обслуживать других клиентов, если они требуют внимания. Такая тактика потребует введения разных очередей, списков и т.п. по отслеживанию состояния ответов на запросы клиентов и может оказаться нетривиальной. Поэтому в рамках использования select мы не будем это рассматривать, так как подоные проблемы просто и естественно решаются в рамках использования fork.

Повторим этот псевдокод в чуть более конкретном, но упрощенном виде (без инклюдов, проверок успешности и т.п.)

```
#define PORT    5555
#define BUFLEN  512

// Две выделенные наши функции для непосредственного чтения/записи
int   readFromClient(int fd, char *buf);
void  writeToClient (int fd, char *buf);

int  main (void)
{
    int     i, err, opt=1;
    int     sock, new_sock;
    fd_set  active_set, read_set;
    struct  sockaddr_in  addr;
    struct  sockaddr_in  client;
    char    buf[BUFLEN];
    socklen_t  size;

    sock = socket (PF_INET, SOCK_STREAM, 0);
    // проверка sock

    setsockopt(sock,SOL_SOCKET,SO_REUSEADDR,(char*)&opt,sizeof(opt));

    addr.sin_family = AF_INET;
    addr.sin_port = htons(PORT);
    addr.sin_addr.s_addr = htonl(INADDR_ANY);
    err = bind(sock,(struct sockaddr*)&addr,sizeof(addr));
    // проверка err

    err = listen(sock,3);
    // проверка err

    FD_ZERO(&active_set);
    FD_SET(sock, &active_set);

    // Основной бесконечный цикл проверки состояния сокетов
    while (1) {
        read_set = active_set;
        if ( select(FD_SETSIZE,&read_set,NULL,NULL,NULL)<0 ) { /* отказ */ }

        for (i=0; i<FD_SETSIZE; i++) {
            if ( FD_ISSET(i,&read_set) ) {
                if ( i==sock ) {
                    size = sizeof(client);
```

```
                    new_sock = accept(sock,(struct sockaddr*)&client,&size);
                    // проверка new_sock
                    FD_SET(new_sock, &active_set);
                } else {
                    err = readFromClient(i,buf);
                    if ( err<0 ) {
                        close (i);
                        FD_CLR(i,&active_set);
                    } else {
                        // готовим ответ клиенту
                        writeToClient(i,buf);
                    }
                }
            }
        }
    }
}
```

Пример. cli_tcp, ser_tcp_select.cpp — клиент и сервер с использованием select.

TCP сервер на базе poll

Системный вызов poll работает идейно так же как и select, но в некоторых аспектах устроен более удобно. Основное отличие в том, что состояния сетевых каналов по каждому клиенту представлены отдельной структурой pollfd

```
struct pollfd {
   int fd;         // дескриптор файла
   short events;   // запрошенные события
   short revents;  // возникшие события
};
```

и для анализа используется массив таких структур. Мы можем назначить какие события мы хотим проверять для каждого отдельного клиента и также нам не нужно отслеживать максимальный номер сокета для просмотра множества fd_set.

```
pollfd  act_set[100];
act_set[0].fd = sock;
act_set[0].events = POLLIN;   // requested event
act_set[0].revents = 0;       // returned event
```

Здесь канал sock будет проверяться на событие POLLIN, соответствующее появлению данных для чтения. Остальные 99 элементов мы зарезервировали для отслеживания активных каналов (сщкдинений).

В плане реализации сценария работы с клиентами схема сервера здесь очень похожа на предыдущую. Поэтому просто повторим код предыдущих примеров с соответствующими изменениями.

```
#define PORT    5555
#define BUFLEN  512

int   readFromClient(int fd, char *buf);
void  writeToClient (int fd, char *buf);

int  main (void)
{
    int     i, err, opt=1;
    int     sock, new_sock;
```

```
struct  sockaddr_in  addr;
struct  sockaddr_in  client;
char    buf[BUFLEN];
socklen_t  size;

sock = socket (PF_INET, SOCK_STREAM, 0);
// проверка sock
setsockopt(sock,SOL_SOCKET,SO_REUSEADDR,(const void*)&opt,sizeof(opt));

addr.sin_family = AF_INET;
addr.sin_port = htons(PORT);
addr.sin_addr.s_addr = htonl(INADDR_ANY);
err = bind(sock,(struct sockaddr*)&addr,sizeof(addr));
// проверка err

// Создаем очередь на 3 входящих запроса соединения
err = listen(sock,3);
// проверка err

pollfd  act_set[100];
act_set[0].fd = sock;
act_set[0].events = POLLIN;   // requested event
act_set[0].revents = 0;       // returned event
int num_set = 1;

while (1) {
    int ret = poll (act_set, num_set, -1); // -1 блокирующий
    // ret < 0  -ошибка
    if (ret > 0) {
       for (i=0; i<num_set; i++) {
          if (act_set[i].revents & POLLIN) {
              act_set[i].revents &= ~POLLIN;
              if (i==0) { // это наш sock - запрос на соединение
                  size = sizeof(client);
                  new_sock = accept(act_set[i].fd,(struct sockaddr*)&client,&size
                  // проверка new sock
                  if (num_set < 100) { // ограничение на число клиентов!
                     act_set[num_set].fd = new_sock;
                     act_set[num_set].events = POLLIN;
                     act_set[num_set].revents = 0;
                     num_set++;
                  } else { // уже нет ресурсов
                     close(new_sock);
                  }
               } else { // пришли данные в существующем соединени
                  err = readFromClient(act_set[i].fd, buf);
                  // проверка err
                  // если ошибка, то закрывем соединение
                  // и удаляем из множества сокетов
                  } else { // данные прочитаны нормально - отвечаем
                      writeToClient(act_set[i].fd,buf);
                  }
              }
          }
       }
    }
```

```
    }
}
```

Пример. cli_tcp, ser_tcp_poll.cpp — клиент и сервер с использованием poll.