



Vasilui 26 сентября 2013 в 16:54

## «Boost.Asio C++ Network Programming». Глава 3: Эхо сервер/клиент

C++, API

Tutorial

Всем привет!

Продолжаю перевод книги John Torjo «Boost.Asio C++ Network Programming».

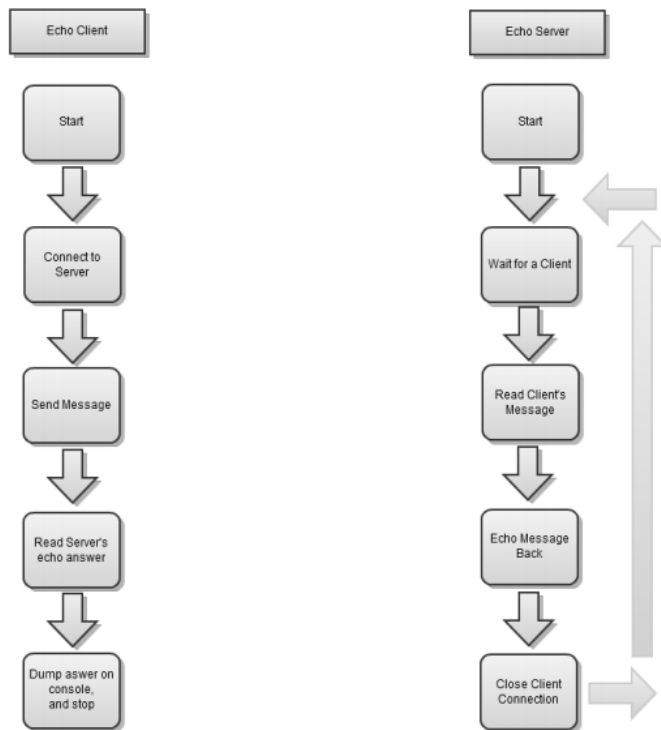
Содержание:

- [Глава 1: Приступая к работе с Boost.Asio](#)
- Глава 2: Основы Boost.Asio
  - [Часть 1: Основы Boost.Asio](#)
  - [Часть 2: Асинхронное программирование](#)
- **Глава 3: Echo Сервер/Клиент**
- Глава 4: Клиент и Сервер
- Глава 5: Синхронное против асинхронного
- Глава 6: Boost.Asio – другие особенности
- Глава 7: Boost.Asio – дополнительные темы

В этой главе мы реализуем небольшое клиент/серверное приложение, которое, вероятно, будет самым простым клиент/серверным приложением. Это приложение эхо-сервер, который возвращает клиенту то, что тот ему написал, а затем закрывает соединение клиента. Сервер может работать с любым числом клиентов. Когда подключается новый клиент, он шлет сообщение. Сервер получает сообщение целиком и посылает его обратно. После этого он закрывает соединение.

Таким образом, каждый эхо-клиент подключается к серверу, посылает сообщение и читает то, что ответил сервер, убедившись, что это то же сообщение, которое он послал, заканчивает общение с сервером.

Сначала мы будем реализовывать синхронное приложение, а затем асинхронное, так что вы можете легко их сравнить:



Здесь будет приводиться не весь код целиком, а только его части, весь код можно посмотреть по ссылке в конце статьи.

## ТСР эхо сервер/клиент

Для ТСР мы можем иметь дополнительное преимущество, каждое сообщение заканчивается символом '\n'. Написание синхронного эхо сервер/клиента очень просто.

Мы приведем примеры программ таких как синхронный клиент, синхронный сервер, асинхронный клиент и асинхронный сервер.

## ТСР синхронный клиент

В большинстве нетривиальных примеров обычно код клиента гораздо проще, чем сервера (так как сервер должен иметь дело с несколькими клиентами).

Следующий пример это исключение из правил:

```

ip::tcp::endpoint ep( ip::address::from_string("127.0.0.1"), 8001);
size_t read_complete(char * buf, const error_code & err, size_t bytes)
{
    if ( err ) return 0;
    bool found = std::find(buf, buf + bytes, '\n') < buf + bytes;
    // we read one-by-one until we get to enter, no buffering
    return found ? 0 : 1;
}

void sync_echo(std::string msg)
{
    msg += "\n";
    ip::tcp::socket sock(service);
    sock.connect(ep);

```

```

        sock.write_some(buffer(msg));
        char buf[1024];
        int bytes = read(sock, buffer(buf), boost::bind(read_complete, buf, _1, _2));
        std::string copy(buf, bytes - 1);
        msg = msg.substr(0, msg.size() - 1);
        std::cout << "server echoed our " << msg << ": " << (copy == msg ? "OK" : "FAIL") << std::endl;
        sock.close();
    }
    int main(int argc, char* argv[])
    {
        char* messages[] = { "John says hi", "so does James", "Lucy just got home", "Boost.Asio is Fun!", 0 };
        boost::thread_group threads;
        for ( char ** message = messages; *message; ++message)
        {
            threads.create_thread( boost::bind(sync_echo, *message));
            boost::this_thread::sleep( boost::posix_time::millisec(100));
        }
        threads.join_all();
    }
}

```

Обратите внимание на функцию `sync_echo`. Она содержит всю логику для подключения к серверу, отправляет ему сообщение и ждет обратного ответа.

Вы заметили, что для чтения мы используем свободную функцию `read()`, потому что мы хотим получать сообщение все целиком до символа `'\n'`. Функции `sock.read_some()` будет недостаточно, так как она будет читать только то, что доступно, но совсем не обязательно все сообщение целиком.

Третий аргумент функции `read()` это завершающий обработчик. Она вернет 0, если сообщение прочитано полностью. В противном случае возвращается максимальный размер буфера, которые может быть прочитан на следующем шаге (до завершения `read`). В нашем случае всегда будет возвращаться 1, потому что мы не хотим ошибочно читать больше чем нам необходимо.

В `main()` мы создаем несколько потоков; по одному потоку для каждого сообщения, которое отправляет клиент, и ждем, пока они завершатся. Если вы запустите программу, то увидите следующий вывод:

```

server echoed our John says hi: OK
server echoed our so does James: OK
server echoed our Lucy just got home: OK
server echoed our Boost.Asio is Fun!: OK

```

Обратите внимание, что, так как мы имеем дело с синхронным клиентом, то нет никакой необходимости вызывать `service.run()`.

## ТСР синхронный сервер

Синхронный эхо сервер написать довольно просто, как показано в следующем фрагменте кода:

```

io_service service;
size_t read_complete(char * buff, const error_code & err, size_t bytes)
{

```

```

        if ( err) return 0;
        bool found = std::find(buff, buff + bytes, '\n') < buff + bytes;
        // we read one-by-one until we get to enter, no buffering
        return found ? 0 : 1;
    }

    void handle_connections()
    {
        ip::tcp::acceptor acceptor(service, ip::tcp::endpoint(ip::tcp::v4(),8001));
        char buff[1024];
        while ( true)
        {
            ip::tcp::socket sock(service);
            acceptor.accept(sock);
            int bytes = read(sock, buffer(buff), boost::bind(read_complete,buff,_1,_2));
            std::string msg(buff, bytes);
            sock.write_some(buffer(msg));
            sock.close();
        }
    }

    int main(int argc, char* argv[])
    {
        handle_connections();
    }

```

Вся логика сервера заключена в `handle_connections()`. Поскольку он однопоточный, то мы принимаем нового клиента, читаем сообщение, которое он прислал, посылаем его обратно, а затем ждем следующего клиента. Скажем, если подключаться сразу два клиента, то второму придется ждать, пока сервер обслуживает первого клиента. Еще раз обратите внимание, что, так как мы работаем синхронно, то нет никакой необходимости вызывать `service.run()`.

## ТСР асинхронный клиент

Как только мы начинаем работать асинхронно, код становится немного сложнее. Мы будем моделировать класс `connection`, как показано во [второй главе](#).

Глядя на следующие фрагменты кода в этом разделе, вы заметите, что каждая асинхронная операция запускает новую асинхронную операцию, сохраняя `service.run()` в работе.

Первое, основные функции:

```

#define MEM_FN(x) boost::bind(&self_type::x, shared_from_this())
#define MEM_FN1(x,y) boost::bind(&self_type::x, shared_from_this(),y)
#define MEM_FN2(x,y,z) boost::bind(&self_type::x, shared_from_this(),y,z)
class talk_to_svr : public boost::enable_shared_from_this<talk_to_svr>,boost::noncopyable
{
    typedef talk_to_svr self_type;
    talk_to_svr(const std::string & message) : sock_(service), started_(true),message_(message) {}
    void start(ip::tcp::endpoint ep)
    {
        sock_.async_connect(ep, MEM_FN1(on_connect,_1));
    }
}

```

```

public:
    typedef boost::system::error_code error_code;
    typedef boost::shared_ptr<talk_to_svr> ptr;
    static ptr start(ip::tcp::endpoint ep, const std::string & message)
    {
        ptr new_(new talk_to_svr(message));
        new_->start(ep);
        return new_;
    }
    void stop()
    {
        if ( !started_ ) return;
        started_ = false;
        sock_.close();
    }
    bool started() { return started_; }
    ...
private:
    ip::tcp::socket sock_;
    enum { max_msg = 1024 };
    char read_buffer_[max_msg];
    char write_buffer_[max_msg];
    bool started_;
    std::string message_;
};

```

Мы хотим всегда использовать shared pointers на `talk_to_svr`, для того чтобы пока есть асинхронные операции в экземпляре `talk_to_svr`, этот экземпляр оставался жив. Для того чтобы избежать таких ошибок как создание экземпляров `talk_to_svr` в стеке, мы сделали конструктор приватным и запретили конструктор копирования (наследовались от `boost::noncopyable`).

У нас есть основные функции, такие как `start()`, `stop()`, и `started()`, которые делают только то, о чем говорят их названия. Для создания соединения просто вызовите `talk_to_svr::start(endpoint, message)`. Так же у нас имеются буферы для чтения и записи (`read_buffer_` и `write_buffer_`).

Как объяснялось ранее, следующие строки сильно отличаются:

```

// equivalent to "sock_.async_connect(ep, MEM_FN1(on_connect, _1));"
sock_.async_connect(ep, boost::bind(&talk_to_svr::on_connect, shared_ptr_from_this(), _1));
sock_.async_connect(ep, boost::bind(&talk_to_svr::on_connect, this, _1));

```

В первом случае мы правильно создаем завершающий обработчик `async_connect`, он будет сохранять shared pointer на экземпляр `talk_to_server` пока он не вызовет завершающий обработчик, тем самым, убедившись, что мы все еще живы, когда это произойдет.

В последнем случае мы неправильно создаем завершающий обработчик. К тому времени, когда вызывается экземпляр `talk_to_server`, он может быть уже удален!

Для чтения и записи в сокет мы будем использовать следующий фрагмент кода:

```

void do_read()
{
    async_read(sock_, buffer(read_buffer_),

```

```

        MEM_FN2(read_complete, _1, _2), MEM_FN2(on_read, _1, _2));
    }
    void do_write(const std::string & msg)
    {
        if ( !started() ) return;
        std::copy(msg.begin(), msg.end(), write_buffer_);
        sock_.async_write_some( buffer(write_buffer_, msg.size()),
            MEM_FN2(on_write, _1, _2));
    }
    size_t read_complete(const boost::system::error_code & err, size_t bytes)
    {
        // similar to the one shown in TCP Synchronous Client
    }

```

Функция `do_read()` сначала убеждается, что мы читаем сообщение от сервера, после чего вызывается `on_read()`. Функция `do_write()` сначала копирует сообщение в буфер (имеет место вероятность того, что `msg` может выйти за область видимости и со временем разрушится), а затем убеждается в том, что вызов `on_write()` происходит после реальной записи. И наиболее важные функции, которые содержат основную логику класса:

```

void on_connect(const error_code & err)
{
    if ( !err) do_write(message_ + "\n");
    else stop();
}
void on_read(const error_code & err, size_t bytes)
{
    if ( !err)
    {
        std::string copy(read_buffer_, bytes - 1);
        std::cout << "server echoed our " << message_ << ": " << (copy == message_ ? "OK"
: "FAIL") << std::endl;
    }
    stop();
}
void on_write(const error_code & err, size_t bytes)
{
    do_read();
}

```

После этого мы подключаемся и посылаем сообщение на сервер, `do_write()`. Когда операция записи завершена, вызывается `on_write()`, которая инициирует функцию `do_read()`. Когда завершается `do_read()`, вызывается `on_read()`, здесь мы просто проверяем, что сообщение от сервера то же самое, что мы ему посылали и выходим из нее. Мы отправим три сообщения на сервер только чтобы сделать это все более интересным:

```

int main(int argc, char* argv[])
{
    ip::tcp::endpoint ep( ip::address::from_string("127.0.0.1"), 8001);
    char* messages[] = { "John says hi", "so does James", "Lucy got home", 0 };
    for ( char ** message = messages; *message; ++message)

```

```

{
    talk_to_svr::start( ep, *message);
    boost::this_thread::sleep( boost::posix_time::millisec(100));
}
service.run();
}

```

В предыдущем фрагменте кода будет генерироваться следующий код:

```

server echoed our John says hi: OK
server echoed our so does James: OK
server echoed our Lucy just got home: OK

```

## ТСР асинхронный сервер

Как показано ниже, основные функции очень похожи на функции асинхронного клиента:

```

class talk_to_client : public boost::enable_shared_from_this<talk_to_client>, boost::noncopyable
{
    typedef talk_to_client self_type;
    talk_to_client() : sock_(service), started_(false) {}
public:
    typedef boost::system::error_code error_code;
    typedef boost::shared_ptr<talk_to_client> ptr;
    void start()
    {
        started_ = true;
        do_read();
    }
    static ptr new_()
    {
        ptr new_(new talk_to_client);
        return new_;
    }
    void stop()
    {
        if ( !started_) return;
        started_ = false;
        vsock_.close();
    }
    ip::tcp::socket & sock() { return sock_;}
    ...
private:
    ip::tcp::socket sock_;
    enum { max_msg = 1024 };
    char read_buffer_[max_msg];
    char write_buffer_[max_msg];
    bool started_;
};

```

Это очень простой эхо-сервер, нет никакой необходимости в функции `is_started()`. Для каждого клиента мы просто читаем сообщение, которое он прислал, посылаем это же сообщение обратно и закрываем соединение. Функции `do_read()`, `do_write()` и `read_complete()` точно такие же как в асинхронном TCP клиенте. Основная логика класса заключена в функциях `on_read()` и `on_write()`:

```
void on_read(const error_code & err, size_t bytes)
{
    if ( !err)
    {
        std::string msg(read_buffer_, bytes);
        do_write(msg + "\n");
    }
    stop();
}
void on_write(const error_code & err, size_t bytes)
{
    do_read();
}
```

Работа с клиентами происходит следующим образом:

```
ip::tcp::acceptor acceptor(service, ip::tcp::endpoint(ip::tcp::v4(), 8001));
void handle_accept(talk_to_client::ptr client, const error_code & err)
{
    client->start();
    talk_to_client::ptr new_client = talk_to_client::new_();
    acceptor.async_accept(new_client->sock(), boost::bind(handle_accept, new_client, _1));
}
int main(int argc, char* argv[])
{
    talk_to_client::ptr client = talk_to_client::new_();
    acceptor.async_accept(client->sock(), boost::bind(handle_accept, client, _1));
    service.run();
}
```

Каждый раз, когда клиент подключается к серверу, вызывается `handle_accept`, который начинает асинхронно читать от этого клиента, а так же асинхронно ждет нового клиента.

## UDP эхо сервер/клиент

Поскольку в UDP не все сообщения доходят до получателя, то у нас нет гарантии, что сообщение пришло полностью. Так как мы работаем по UDP, то каждое сообщение, которое мы получаем, мы просто выводим, не закрывая сокет (на стороне сервера).

## UDP синхронный эхо клиент

UDP эхо-клиент немного проще, чем TCP эхо-клиент:

```
ip::udp::endpoint ep( ip::address::from_string("127.0.0.1"), 8001);
```



```

void sync_echo(std::string msg)
{
    ip::udp::socket sock(service, ip::udp::endpoint(ip::udp::v4(), 0) );
    sock.send_to(buffer(msg), ep);
    char buff[1024];
    ip::udp::endpoint sender_ep;
    int bytes = sock.receive_from(buffer(buff), sender_ep);
    std::string copy(buff, bytes);
    std::cout << "server echoed our " << msg << ": " << (copy == msg ? "OK" : "FAIL") << std::endl;
    sock.close();
}

int main(int argc, char* argv[])
{
    char* messages[] = { "John says hi", "so does James", "Lucy got home", 0 };
    boost::thread_group threads;
    for ( char ** message = messages; *message; ++message)
    {
        threads.create_thread( boost::bind(sync_echo, *message));
        boost::this_thread::sleep( boost::posix_time::millisec(100));
    }
    threads.join_all();
}

```

Вся логика заключена в функции `sync_echo()` ; подключение к серверу, отправка сообщения, получение ответного сообщения от сервера и закрытие соединения.

## UDP синхронный эхо-сервер

UDP эхо-сервер это самый простой сервер, который только можно написать:

```

io_service service;
void handle_connections()
{
    char buff[1024];
    ip::udp::socket sock(service, ip::udp::endpoint(ip::udp::v4(), 8001));
    while ( true)
    {
        ip::udp::endpoint sender_ep;
        int bytes = sock.receive_from(buffer(buff), sender_ep);
        std::string msg(buff, bytes);
        sock.send_to(buffer(msg), sender_ep);
    }
}

int main(int argc, char* argv[])
{
    handle_connections();
}

```

Здесь все очень просто и говорит само за себя.

Оставим написание асинхронных UDP сервера и клиента читателю в качестве упражнения.

## Резюме

Мы написали несколько приложений и, наконец, приступили к работе с Boost.Asio. Эти приложения очень хороши для начала работы с этой библиотекой.

В следующей главе мы будем создавать более сложные клиент/серверные приложения, будем учиться избегать таких ошибок как утечка памяти, дедлоки и так далее.

Всем большое спасибо! +29 166 51,3k 1 Поделиться

Ресурсы к этой статье: [ссылка](#)



68,5

Карма

0,0

Рейтинг



Подписаться

Кузьминых Василий @Vasilui

Пользователь

☐ NumPy Beginner's Guide

### ПОХОЖИЕ ПУБЛИКАЦИИ

8 апреля 2019 в 12:03

#### Operating Systems: Three Easy Pieces. Part 3: Process API (перевод)

5 1,7k 16 1 +1

28 января 2014 в 12:17

#### User Timing API

18 12,9k 111 11 +11

16 мая 2013 в 10:56

#### Новые функции Google для разработчиков: игровые API, перевод и тестирование

31 13,5k 72 15 +15

### ВАКАНСИИ

Senior Backend Engineer (Rest API, English, NodeJS, Python)  
от 3 500 до 4 500 \$ • DataDirect Networks Inc. (DDN) • Можно удаленно

Системный аналитик (интеграционные решения)  
от 150 000 Р • Voxberry • Москва • Можно удаленно

Тестировщик ПО  
от 80 000 Р • Онлайн школа Тетрика • Можно удаленно


Программист C++  
от 140 000 Р • НТЦ «Элинс» • Зеленоград

Разработчик C++  
до 150 000 Р • НТЦ ПРОТЕЙ • Санкт-Петербург • Можно удаленно

[Больше вакансий на Хабр Карьере](#)

Комментарии 1

Отслеживать новые в ☐ почте ☐ трекере

 **gotu4** 26 сентября 2013 в 18:48 +1

Далее про ZeroMQ — он грамотно построен, поэтому разбор по составляющим было бы интересно послушать. Вдруг что-то упустил из вида.

[Ответить](#)

Написать комментарий

**B** / U       

Предпросмотр

Отправить

☐ Markdown (?)

САМОЕ ЧИТАЕМОЕ

Сутки

Неделя

Месяц

Собеседование в Яндекс: театр абсурда :/

+564 182k 449 1103 +1076

Дата-центр возле Амстердама называют «выгребной ямой интернета», но он продолжает работу

+42 23k 45 80 +54

Люди подозревают, что технологии — отстой, потому что они на самом деле отстой

+57 25k 40 256 +256

Листая старые подшивки. Взгляд изнутри на компьютерную прессу 90-х

+101 14,6k 55 92 +92

Когда старый подход хуже новых двух: коллекция про нетривиальную разработку

Мерапост

Ваш аккаунт	Разделы	Информация	Услуги
Профиль	Публикации	Устройство сайта	Реклама

<a href="#">Трекер</a>	<a href="#">Новости</a>	<a href="#">Для авторов</a>	<a href="#">Тарифы</a>
<a href="#">Диалоги</a>	<a href="#">Хабы</a>	<a href="#">Для компаний</a>	<a href="#">Контент</a>
<a href="#">Настройки</a>	<a href="#">Компании</a>	<a href="#">Документы</a>	<a href="#">Семинары</a>
<a href="#">ППА</a>	<a href="#">Пользователи</a>	<a href="#">Соглашение</a>	<a href="#">Мегапроекты</a>
	<a href="#">Песочница</a>	<a href="#">Конфиденциальность</a>	<a href="#">Мерч</a>

Если нашли опечатку в посте, выделите ее и нажмите Ctrl+Enter, чтобы сообщить автору.

© 2006 – 2021 «Habr»

[Настройка языка](#)

[О сайте](#)

[Служба поддержки](#)

[Мобильная версия](#)

