Лабораторная работа № 1
Клиент-серверное приложение для передачи сообщений с использованием протоколов TCP и UDP

1.1. Цель работы:

Целью первой лабораторной работы является ознакомление студентов с основной структурой клиент-серверного приложения, а также со способами реализации данных приложений с помощью библиотеки Boost::Asio.

1.2. Задание на лабораторную работу:

Разработать два клиент-серверных приложения, основанных на транспортных протоколах TCP и UDP. Каждое клиент-серверное приложение должно состоять из двух самостоятельных модулей: клиентской части и серверной части.

Клиент-серверное приложение, основанное на транспортном протоколе TCP. Клиент должен инициировать соединение с сервером и отправлять ему текстовые сообщения. Сервер должен принимать сообщения клиента и выводить их на экран, а также иметь возможность отвечать на принятые сообщения.

Клиент-серверное приложение, основанное на транспортном протоколе UDP. Клиент и сервер имеют одинаковое строение, они оба должны иметь возможность отправлять сообщения друг другу и выводить их на экран.

Все приложения должны иметь простейший графический интерфейс.

Для данной лабораторной работы необходимо использовать неблокируемые (асинхронные) сокеты.

1.3. Методические указания:

1.3.1. Понятие сокет:

Сокеты (sockets) - программный высокоуровневый интерфейс, предоставляющий возможность взаимодействовать с телекоммуникационными протоколами. Сокет представляет собой некую конечную точку коммутации, с помощью которой ваше приложение может обмениваться данными по локальной сети или глобальной сети интернет. Процесс обмена данными с помощью данного интерфейса достаточно прост: приложение записывает данные, которые необходимо отправить, в сокет; отправка данных, их транспортировка осуществляется используемым стеком протоколов и сетевой аппаратурой; при поступлении данных в сокет, приложение считывает пришедшие данные.

1.3.2. Обзор сокетов

Сокеты делятся на несколько видов.

В зависимости от используемого транспортного протокола сокеты делятся на потоковые и датаграмные.

Потоковые сокеты – сокеты, использующие, в качестве протокола транспортного уровня, протокол TCP. Данные сокеты отличаются тем, что используют процедуру установления соединения, в результате которой устанавливается некий логический канал. Использование потоковых сокетов гарантирует успешную доставку данных, что обеспечивается механизмом квитирования.

Датаграмные сокеты - сокеты, использующие, в качестве протокола транспортного уровня, протокол UDP. Данные сокеты не используют процедуру установления соединения и не гарантируют успешную доставку данных, однако, в связи с тем, что не происходит передачи служебной информации датаграмные сокеты являются более быстрыми.

Выбор между датаграмными и потоковыми сокетами определяется данными, которые будут передаваться через них. В случае, если необходимо передавать чувствительные к потерям данные (документы, важные сообщения) лучше использовать потоковые сокеты, когда же необходимо передавать чувствительные к задержкам данные (видео, аудиоинформацию в реальном времени) лучше использовать датаграмные сокеты.

В зависимости от процедуры управления сокеты делятся на синхронные и асинхронные.

Синхронные (блокируемые) – сокеты, задерживающие управление на время выполнения операции.

Асинхронные (неблокируемые) – сокеты, возвращающие управления незамедлительно, продолжая работы в фоновом режиме. В данной лабораторной работе будут рассмотрен именно этот тип сокетов.

1.4. Особенности работы с сокетами библиотеки Boost::Asio

В собрание библиотек Boost входит библиотека Boost::Asio, которая предоставляет набор классов для работы с сетью. Библиотека Boost:Asio содержит в себе: классы для работы с сокетами(asio::ip::tcp, asio::ip::udp, и asio::ip::icmp); классы для работы с устройствами(asio::io_service и asio::streambuf) в синхронном и асинхронном режиме; классы, имеющие дело с SSL(asio::ssl).

Boost.Asio это чисто заголовочная библиотека. Для подключения Boost::Asio к вашему проекту необходимо подключить следующие библиотеки.

- Boost::System - эта библиотека предоставляет поддержку операционной системе для библиотек Boost.
- OpenSSL - эта библиотека используется, если вы решите использовать SSL поддержку, предоставляемую Boost.Asio.

В Windows системах, в зависимости от компилятора, который вы используете, может понадобиться подключение системных библиотек для работы с сокетами - ws2_32 и wsock32.

В случае, если вы используете Qt Creator в качестве среды разработки, то необходимо в файл qMake (name_project.pro) дописать следующие строки, изменив при этом путь к библиотекам:

```
LIBS += -LC:\Qt\boost\lib -llibboost_system-mgw49-mt-d-1_57
LIBS += -LC:\Qt\boost\lib -llibboost_thread-mgw49-mt-d-1_57
LIBS += -lws2_32
LIBS += -lwsock32
INCLUDEPATH +=C:\Qt\boost\include\boost-1_57\
DEPENDPATH +=C:\Qt\boost\include\boost-1_57\
```

Также необходимо подключить следующие заголовочные файлы:

#include "boost/asio.hpp" – библиотека Boost::Asio

#include "boost/bind.hpp" – работа с привязками

#include "boost/thread.hpp" – работа с потоками

1.4.1. Обзор классов для работы с сокетами

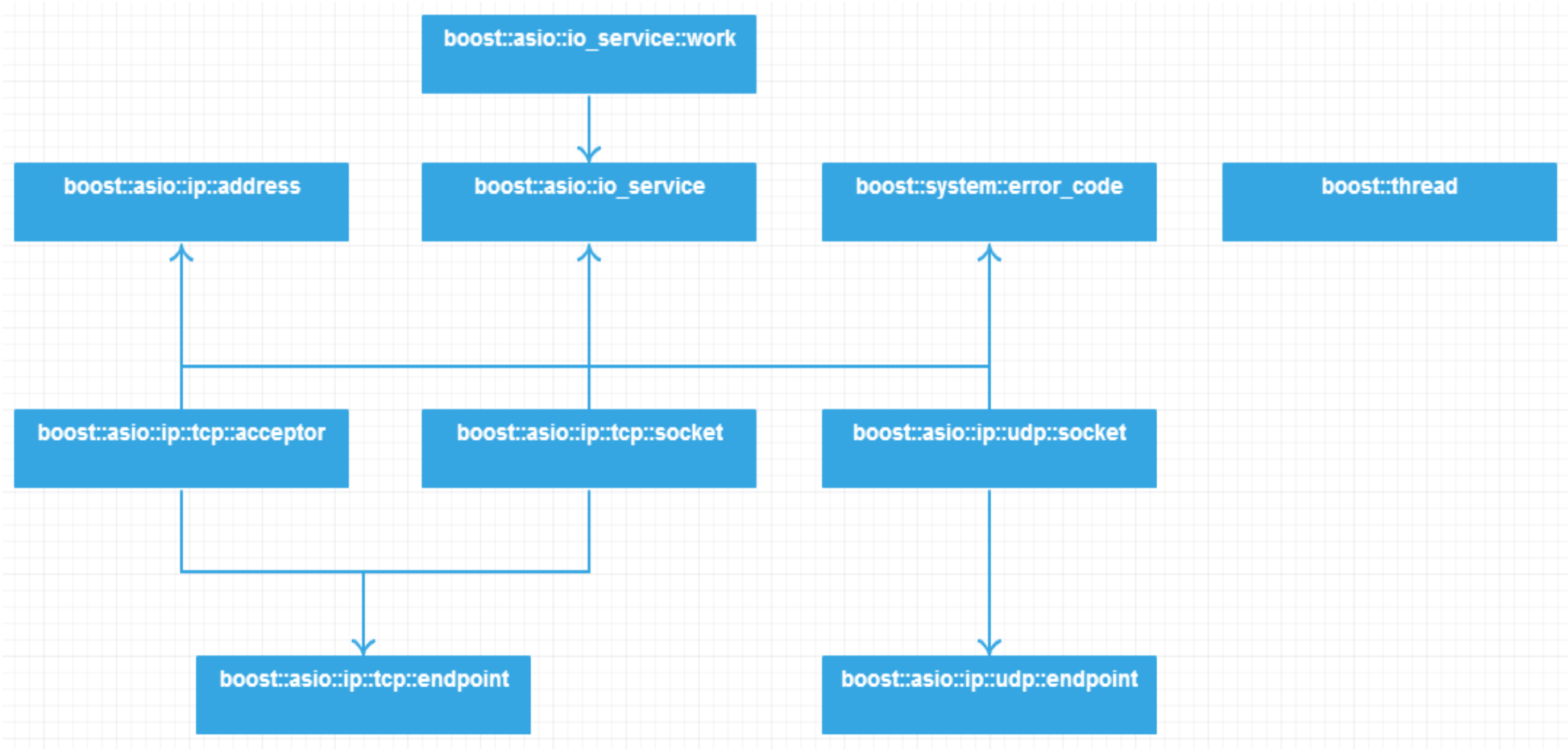

Рисунок 1.1 Диаграмма основных классов библиотеки Boost::Asio

Для работы с асинхронными сокетами необходимо изучить основы библиотеки Boost::Asio, а именно класс io_service. Данный класс предоставляет базовую функциональность для организации асинхронной работы, поэтому рассмотрим подробнее особенности работы с ним. Диаграмма классов представлена на рисунке1.1

1.4.1.1. Класс io_service

Io_service – класс, предоставляющий базовую функциональность асинхронного ввода/вывода для различных операционных систем. Основная деятельность объекта данного класса заключается в том, чтобы ожидать конца асинхронной операции и вызывать определённый обработчик.

Таблица 1.1

Конструктор	io_service();

После объявления объекта необходимо вызвать метод run(). При этом объект заберёт на себя управление, то есть заблокирует поток. Если в вашем приложении есть графический интерфейс, то необходимо вызывать метод run() в другом потоке.

Таблица 1.2

Функция	std::size_t run();
Описание функции	Забирает управление потоком, начинает ожидать конец асинхронных операций и вызывать их обработчики. Возвращает количество обработчиков, которые должны обрабатываться объектом.

В случае если асинхронных операций нет, вызывается метод stop(), который переводит объект в пассивное состояние и возвращает управление.

Таблица 1.3

Функция	void stop();
Описание функции	Переводит объект в пассивное состояние и возвращает управление.

1.4.1.2. Класс work

Чтобы объект продолжал выполнять обработку асинхронных операций, даже когда отложенных операций нет, необходимо нагрузить его работой, для этого используется класс io_service::work.

Таблица 1.4

Конструктор	io_service::work(boost::asio::io_service & io_service);
Описание параметров	io_service – объект, который будет выполнять обработку.

После создания объекта класса Work, объект io_service будет находиться в пассивном режиме. Далее необходимо вызвать метод run() для io_service. В этом случае io_service будет работать пока не будет вызван метод stop() или не уничтожен объект Work.

Все обработчики, которые будут вызваны, выполняется в том же потоке, где вызван метод run();

1.4.1.3. Класс thread

Для организации потоков используется класс boost::thread.

Таблица 1.5

Конструктор перемещения	thread(thread&& other) noexcept;
Описание параметров	other –функция, которая будет выполняться в другом потоке.

Рассмотрим пример:

У нас есть объект io_service. Для того, чтобы при запуске метода run() объект не забрал управление, мы хотим перенести запуск этого метода в другой поток.

```
boost::asio::io_service service ();
```

Для начала необходимо написать функцию, которая будет исполняться в другом потоке.

```
void run_io_service()
{
	boost::asio::io_service::work work(service);
	service.run();
}
```

Теперь необходимо создать поток и перенести исполнение вышеописанной функции в него.

```
boost::thread thread(run_io_service);
```

После этого функция run_io_service() начнёт выполняться в другом потоке.

1.4.1.4. Функция bind

Если необходимо использовать функцию с параметрами или функцию пользовательского класса, то воспользуемся boost::bind(), которая свяжет саму функцию и её параметры.

Пример:

1) Связка с параметрами

```
int number;
void run_io_service(int number);
boost::thread thread(boost::bind(run_io_service,number));
```

2) Связка с функцией объекта

```
int number;
MyClass * this;
void MyClass::run_io_service(int number);
boost::thread thread(boost::bind(MyClass::run_io_service,this,number));
```

1.4.1.5. Классы для непосредственной работы с сокетами

Ip::tcp::socket - класс для организации потокового сокета. Предоставляет унифицированный интерфейс для работы с потоковыми сокетами. В отличие от Socket API, Ip::tcp::socket не может прослушивать порт, для этого существует специализированный класс ip::tcp::acceptor.

Ip::tcp::acceptor – класс для организации TCP сервера. Класс позволяет принимать запросы на соединение по протоколу TCP. При поступлении запрос становиться в очередь запросов. При желании сервера установить соединение, создаётся новый ip::tcp::socket и происходит установление соединения.

Ip::udp::socket – класс для организации датаграмного сокета. Предоставляет унифицированный интерфейс для работы с датаграмными сокетами.

Подробную работу с этими классами рассмотрим в пункте 1.4.2. и 1.4.3.

1.4.1.6. Классы ip::udp::endpoint и ip::tcp::endpoint

Для работы с IP адресами в библиотека Boost::Asio используется класс endpoint. Endpoin представляет собой связку адреса и порта.

Таблица 1.6

Конструктор	basic_endpoint(const boost::asio::ip::address & addr, unsigned short port_num);
Описание параметров	addr –адрес конечной точки port_num – номер порта

Для переопределения параметров используются функции:

Таблица 1.7

Функция	void address(const boost::asio::ip::address & addr);
Описание функции	Устанавливает адрес конечной точки
Описание параметров	addr –адрес конечной точки

Таблица 1.8

Функция	boost::asio::ip::address address() const;
Описание функции	Возвращает адрес конечной точки

Таблица 1.9

Функция	void port(unsigned short port_num);
Описание функции	Устанавливает порт конечной точки

Описание параметров	port_num –порт конечной точки

Таблица 1.10

Функция	unsigned short port() const;
Описание функции	Возвращает порт конечной точки

1.4.2. Клиент-серверные приложения с установлением соединения

1.4.2.1. Установка соединения

Серверная часть

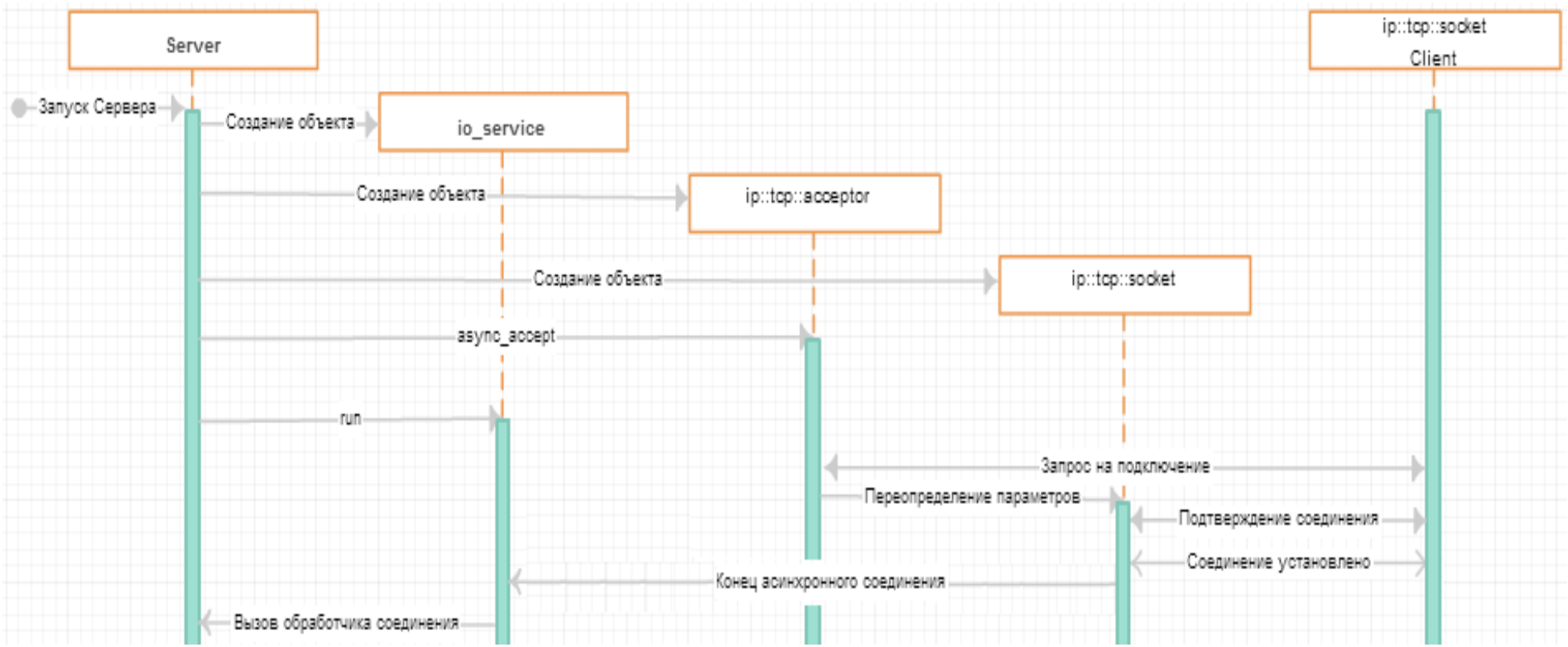

Рисунок 1.2 Диаграмма взаимодействия при установлении соединения (Серверная часть)

На рисунке 1.2 приведена диаграмма взаимодействия при установлении соединения на стороне сервера, рассмотрим каждый шаг.

a) Создание объекта ip::tcp::acceptor, привязка его к локальному адресу и порту, создание очереди запросов на подключение

Таблица 1.11

Конструктор	basic_socket_acceptor(boost::asio::io_service & io_service, const endpoint_type & endpoint, bool reuse_addr = true);
Описание параметров	io_service – объект для работы с асинхронными операциями endpoint – конечная точка к которой будет подключён acceptor(Локальный адрес и порт) reuse_addr – опция, которая определяет, будет ли разрешено использование уже задействованного адреса (по умолчанию разрешено)

Пример :

```
boost::asio::io_service service();
boost::asio::ip::tcp::endpoint ep(boost::asio::ip::tcp::v4(), 5000);
boost::asio::ip::tcp::acceptor acceptor(service,ep);
```

b) Перевод в режим асинхронного прослушивания запросов и назначение обработчика, вызывающегося при установлении соединения

Таблица 1.12

Функция	template< typename Protocol1, typename SocketService, typename AcceptHandler > void-or-deduced async_accept(basic_socket< Protocol1, SocketService > & peer, AcceptHandler handler,);
Описание функции	Переводит acceptor в режим асинхронного прослушивания запросов, назначает обработчик handler, который будет вызываться при установлении соединения
Описание параметров	peer – сокет, к которому будет совершено первое подключение. handler – обработчик

Пример:

```
boost::asio::ip::tcp::socket sock (service);
acceptor.async_accept(sock,boost::bind(handle,_1));
```

При поступлении запросов на соединение, происходит процедура установления соединения между сокетом peer и удалённым узлом. После этого объект io_service вызывает обработчик handler. Сигнатура функции обработчика должна быть:

```
void handler(const boost::system::error_code& error);
```

Если объект err существует, то произошла ошибка, в противном случае соединение успешно установлено.

В обработчике необходимо вновь вызвать функцию async_accept, если вы хотите принимать новые подключения или закрыть acceptor.

Клиентская часть

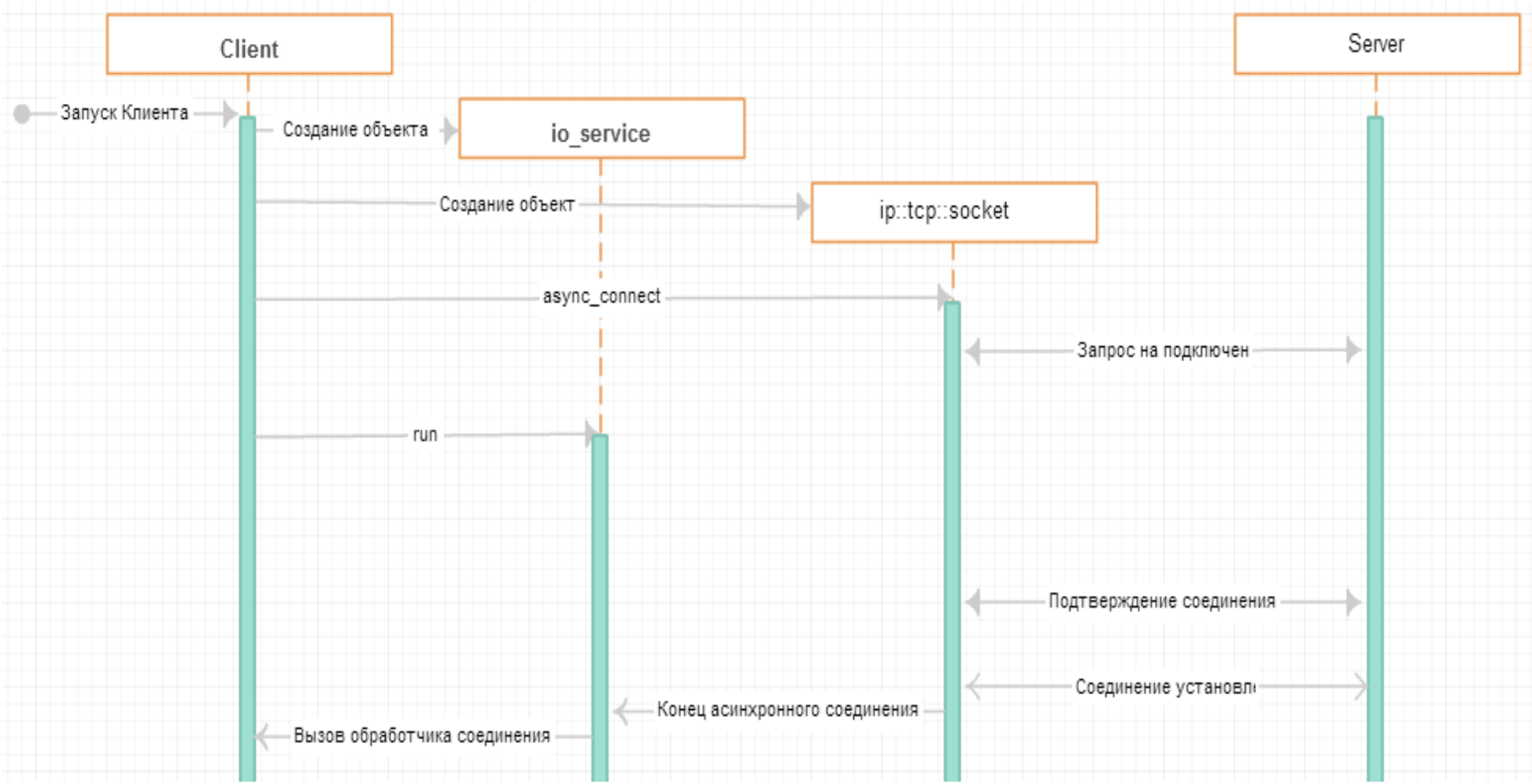

Рисунок 1.3 Диаграмма взаимодействия при установлении соединения (Клиентская часть)

На рисунке 1.3 приведена диаграмма взаимодействия при установлении соединения на стороне клиента, рассмотрим каждый шаг:

a) Создание объекта ip::tcp::socket

Таблица 1.13

Конструктор	basic_stream_socket(boost::asio::io_service & io_service);
Описание параметров	io_service – объект для работы с асинхронными операциями

b) В Boost::Asio в отличии от SOCKET API не нужно вручную привязывать сокет к определённому локальному адресу или порту, так как до вызова функции async_connect () сокет не создаётся. При вызове данной функции создаётся сокет, использующий произвольный свободный порт, и происходит попытка соединения.

Таблица 1.14

Функция	template< typename ConnectHandler > void-or-deduced async_connect(const endpoint_type & peer_endpoint, ConnectHandler handler);
Описание функции	Открывает системный сокет, использующий произвольный незанятый порт, пытается установить соединение
Описание параметров	peer_endpoint – удалённая конечная точка handler – обработчик соединения

Пример:

```
sock.async_connect(ep, boost::bind(handle,_1));
```

После успешного соединения объект io_service вызывает обработчик handler. Сигнатура функции обработчика должно быть:

```
void handler(const boost::system::error_code& error);
```

Если объект err существует, то произошла ошибка, в противном случае соединение успешно установлено.

1.4.2.2. Обмен данными

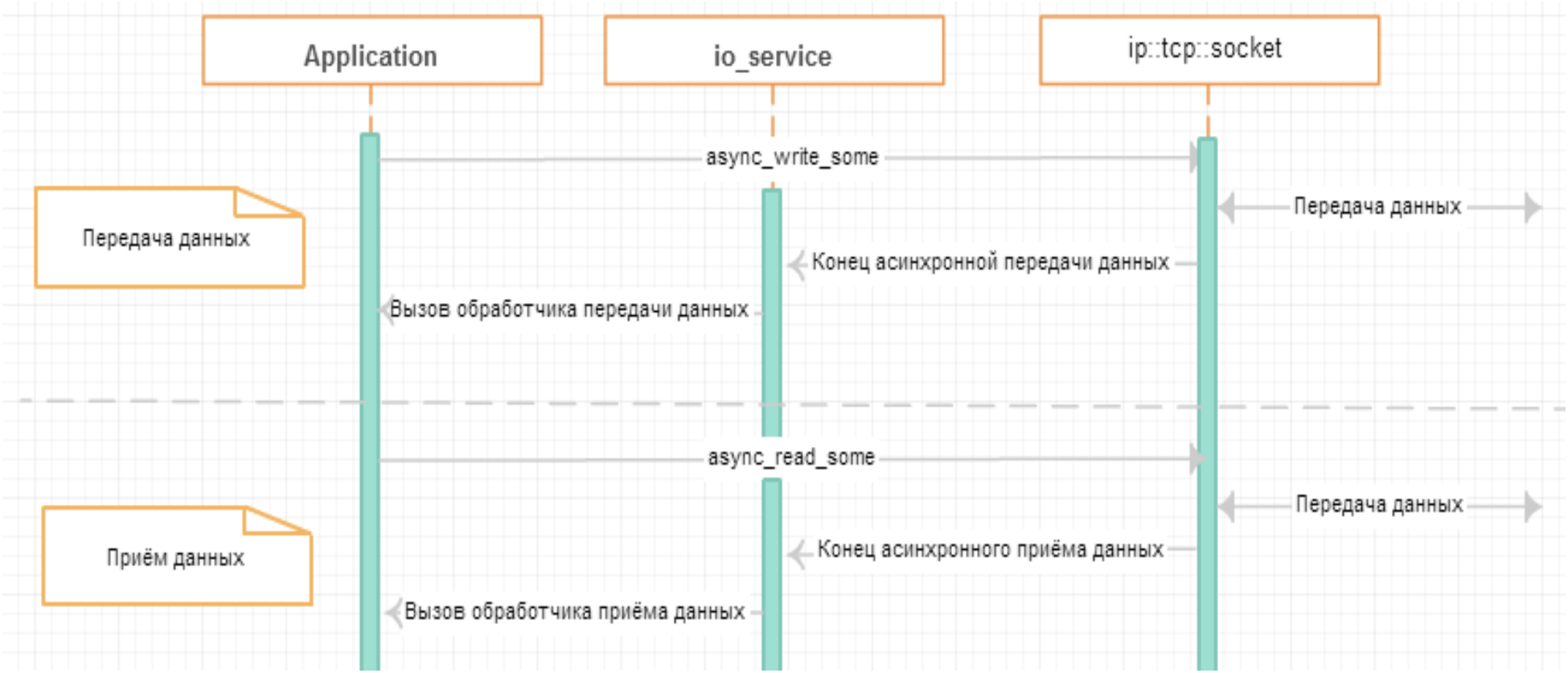

Рисунок 1.4 Диаграмма взаимодействия при передачи данных

На рисунке 1.4 приведена диаграмма взаимодействия при обмене данными, рассмотрим каждый шаг.

Для отправки данных используется функция:

Таблица 1.15

Функция	template< typename ConstBufferSequence, typename WriteHandler > void-or-deduced async_send(const ConstBufferSequence & buffers, WriteHandler handler);
Описание функции	Отправляет данные из буфера buffers, после выполнения вызывается обработчик передачи handler
Описание параметров	buffers – буфер handler – обработчик передачи

Пример:

```
int size = 1024;
char buff_write[size];
sock.async_send(boost::asio::buffer(buff_write,size),boost::bind(boost::bind(handle,_1,_2));
```

После записи данных в буфер для отправки, данные будут поделены на пакеты (если это необходимо) и отправлены получателю.

Для получения данных из буфера используется функция:

Таблица 1.16

Функция	template< typename MutableBufferSequence, typename ReadHandler > void-or-deduced async_receive(const MutableBufferSequence & buffers, ReadHandler handler);
Описание функции	При поступлении данных , читает данные в буфер buffers, после выполнения вызывается обработчик приёма handler
Описание параметров	buffers – буфер handler – обработчик приёма

Сигнатура функции обработчика для приёма и передачи должно быть:

```
void handler(const boost::system::error_code& error,  std::size_t bytes_transferred);
```

Если объект err существует, то произошла ошибка, в противном случае данные приняты/переданы. bytes_transferred – количество байт принятых/переданых.

1.4.2.3. Закрытие соединения

Для корректного завершения работы с socket и acceptor используется функция:

Таблица 1.17

Функция	boost::system::error_code close(boost::system::error_code & ec);
Описание функции	Закрывает сокет. Если произошла ошибка возвращает его в объект ec.

1.4.2.4. Обработка ошибок

Для обработки ошибок используется boost::system::error_code. Объект содержит в себе код произошедшей ошибки.

Таблица 1.18

Функция	int value() const noexcept;
Описание функции	Возвращает номер ошибки

1.4.3. Клиент-серверные приложения без установления соединения

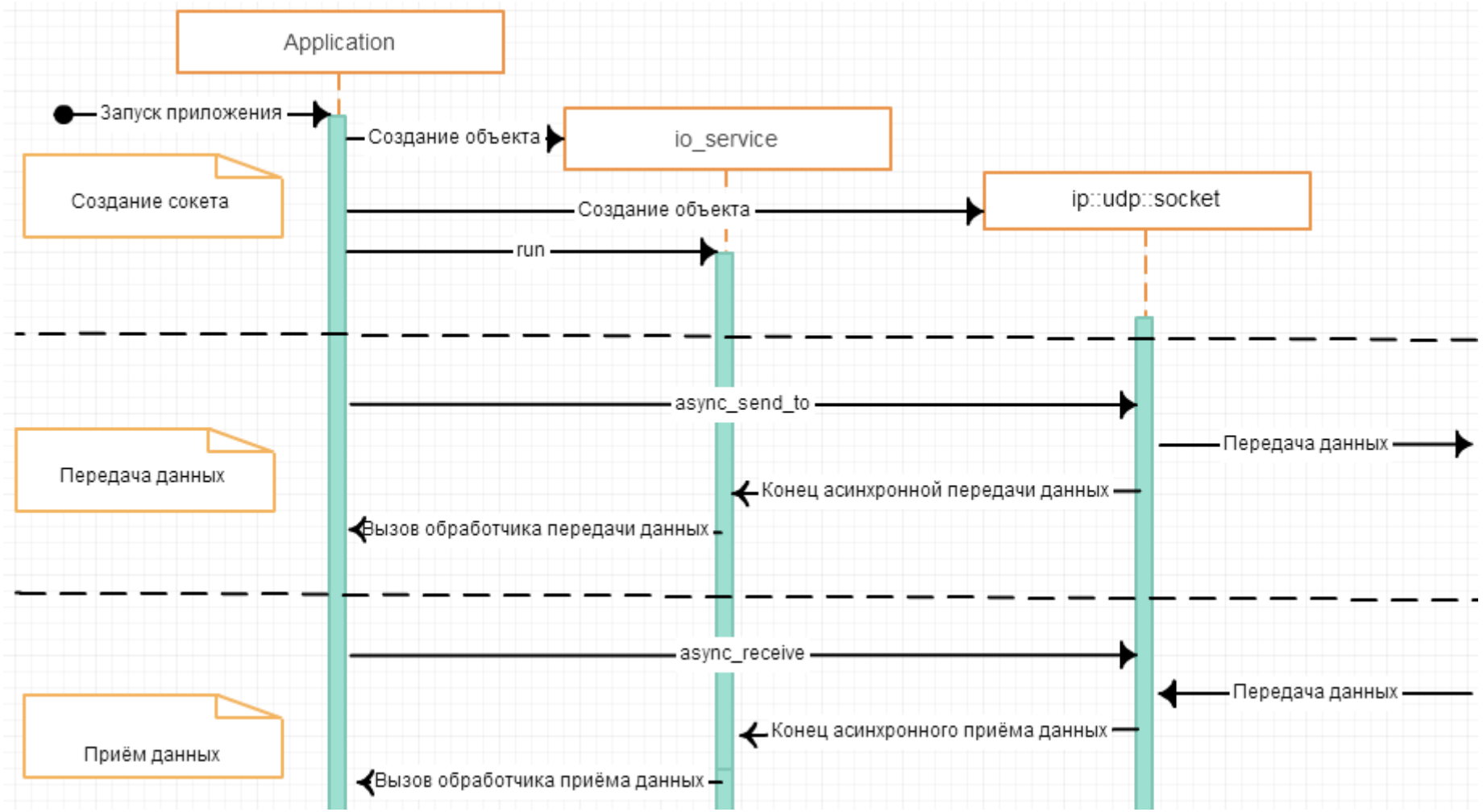

Рисунок 1.5 Диаграмма взаимодействия датаграмных сокетов.

При использовании датаграмных сокетов, не происходит процедуры установления соединения, поэтому их использование гораздо проще.

На рисунке 1.5 приведена диаграмма взаимодействия, при использовании датаграмных сокетов, рассмотрим каждый шаг:

a) Создание объекта ip::udp::socket, привязка его к локальному адресу и порту.

Таблица 1.19

Конструктор	basic_datagram_socket(boost::asio::io_service & io_service, const endpoint_type & endpoint);
Описание параметров	io_service – объект для работы с асинхронными операциями endpoint – конечная точка к которой будет подключён socket(Локальный адрес и порт)

Пример:

```
boost::asio::ip::udp::socket sock(service,ep);
```

b) Отправка сообщений осуществляется с помощью функции:

Таблица 1.20

Функция	template< typename ConstBufferSequence, typename WriteHandler > void-or-deduced async_send_to(const ConstBufferSequence & buffers, const endpoint_type & destination, WriteHandler handler);
Описание функции	Отправляет данные из буфера buffers на удалённый адрес конечной точки destination. После отправки данных вызывает обработчик handler
Описание параметров	buffers – буфер destination – удалённая конечная точка handler – обработчик передачи

c) Для приёма данных используется функция async_receive(), аналогичная функции ip::tcp::socket:: async_receive().Сигнатура функций обработчиков аналогична сигнатурам ip::tcp::socket.

Для закрытия сокета используется функция close(), аналогична функсии ip::tcp::socket:: close().

1.5. Контрольные вопросы и задания.

1) Описать процесс установления соединения на стороне клиента.
2) Описать процесс установления соединения на стороне сервера.
3) Объяснить различия между обменом пакетами в ориентированных на соединение протоколах и не ориентированных на соединение протоколах.
4) Объяснить различия блокируемых и неблокируемых сокетов.
5) Объяснить, как реализуются асинхронные сокеты, в выбранной вами библиотеке.