Глава 18

Использование wxSocket

©Перевод сделан Митрофановым Николаем

Сокет является средством для передачи данных. Сокет не знает какие данные передаются через него, куда эти данные приходят или откуда приходят. Его целью является транспортировка данных из пункта А в пункт Б. Сокеты используются каждый раз, когда вы бродите по сети Интернет, проверяете электронную почту или входите в программу мгновенного обмена сообщениями. Важнейших особенностью сокетов является то, что они могут быть использованы для соединения любых двух устройств, поддерживающих сокеты, даже если одно из них является компьютером, а другое — холодильником!

API сокетов изначально было частью операционной системы BSD Unix, а далее проникло в другие операционные системы и постепенно стало стандартом. Все современные ОС предоставляют специальный слой сокетов, которые обеспечивают возможность передачи данных по сети (такой как, например, Интернет), используя распространенные протоколы, такие как TCP или UDP. Используя класс `wxSocket` из библиотеки wxWidgets, вы можете пересылать любые объемы данных с одного компьютера на другой. Данная глава предполагает наличие у читателя некоторых основных знаний по терминологии сокетов, но операции над ними в целом достаточно просты.

Хотя архитектура и функции сокетов очень похожи в Windows, Linux и Mac OS X, но каждая реализация API сокетов имеет свои особенности. Что более важно, события от сокетов имеют очень различные API на каждой из платформ, что часто создает сложности при их программировании. wxWidgets предоставляет удобные классы, которые позволяют легко использовать сокеты в современных приложениях без необходимости заботится о нюансах каждой платформы в отдельности.

Обратите внимание, что на момент написания данной книги wxWidgets не поддерживал отправку и получение данных по протоколу UDP. Будущие релизы wxWidgets возможно добавят такую возможность.

18.1 Классы сокетов и обзор их функциональности

Ядром системы является класс `wxSocketBase`, который реализует основные функции для отправки и получения данных, закрытия сокета, оповещения об ошибках и так далее. Создание слушающего сокета или соединения к

серверу потребует использования классов `wxSocketServer` или `wxSocketClient` соответственно. Класс `wxSocketEvent` используется для оповещения приложения о событиях, происходящих с сокетом. Абстрактный класс `wxSocketBase` и его потомки (такие как `wxIPV4address`) позволяют указать имя хоста и порт. Наконец, классы потоков, такие как `wxSocketInputStream` и `wxSocketOutputStream`, можно использовать в сочетании с другими классами потоков для перемещения и преобразования данных через сокет. Тема потоков обсуждалась в Главе 14 «Файлы и потоки».

Как будет показано в разделе «Флаги для сокетов» сокеты в библиотеке wxWidgets могут работать по-разному. Традиционный подход по организации потоков для сокетов состоит в отключении механизма событий для сокета и использовании блокирующих вызовов его методов. С другой стороны, вы можете включить события от сокета и устранить необходимость в использовании выделенного потока; wxWidgets пришлет событие в приложение, когда сокет потребует этого. Последний метод позволяет получать и обрабатывать данные в фоновом режиме и, таким образом, избегать блокировки GUI, а кроме того не требует помещать каждый сокет в свой отдельный поток.

В данной главе приводятся примеры использования обоих методов, а также описание программных интерфейсов для `wxSocket` и других родственных классов. Примеры и описания могут изучаться и использоваться независимо, хотя каждому примеру предшествует объяснение соответствующей части API.

18.2 Введение в сокеты и их простая обработка

В качестве введения в программирование сокетов с использованием wxWidgets давайте сразу напишем небольшое клиент-серверное приложение. Код достаточно простой и требует минимальных знаний о программировании сокетов. Для краткости, многие GUI-элементы программы опущены, чтобы сосредоточить внимание на основных приемах программирования сокетов. Полный код приложения можно найти на прилагаемом компакт-диске в папке examples/chap18.

Программа выполняет очень простую задачу. Сервер ждет соединение, а когда соединение установлено, читает строку из десяти символов, посланную клиентом, а затем отправляет эти же десять символов обратно клиенту. Аналогично, клиент создает соединение, посылает через него строку, а затем получает ее же в ответ. Строка, посылаемая клиентом, жестко задана в нашем примере и имеет вид «0123456789» (без кавычек). Интерфейс серверной и клиентской программ показан на рисунке 18.1.

18.2.1 Клиент

Вот код программы-клиента.

```
BEGIN_EVENT_TABLE(MyFrame, wxFrame)
    EVT_MENU(CLIENT_CONNECT, MyFrame::OnConnectToServer)
    EVT_SOCKET(SOCKET_ID,    MyFrame::OnSocketEvent)
END_EVENT_TABLE()
```

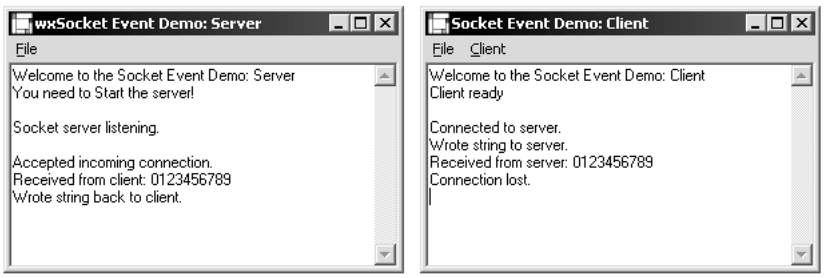

Рис. 18.1: Вид клиента и сервера

```
void MyFrame::OnConnectToServer(wxCommandEvent& WXUNUSED(event))
{
    wxIPV4address addr;
    addr.Hostname(wxT("localhost"));
    addr.Service(3000);

    // Создаем сокет
    wxSocketClient* Socket = new wxSocketClient();

    // Устанавливаем обработчик событий
    // и подписываемся на большинство событий
    Socket->SetEventHandler(*this, SOCKET_ID);
    Socket->SetNotify(wxSOCKET_CONNECTION_FLAG |
                                        wxSOCKET_INPUT_FLAG |
                                        wxSOCKET_LOST_FLAG);
    Socket->Notify(true);

    // Ждем события о подключении
    Socket->Connect(addr, false);
}

void MyFrame::OnSocketEvent(wxSocketEvent& event)
{
    // Получаем сокет, сгенерировавший событие
    wxSocketBase* sock = event.GetSocket();

    // Создаем буфер для передаваемой строки
    char buf[10];

    switch(event.GetSocketEvent())
    {
        case wxSOCKET_CONNECTION:
        {
            // Заполняем его символами цифр от '0' до '9'
            char mychar = '0';
            for (int i = 0; i < 10; i++)
```

```
            {
                buf[i] = mychar++;
            }

            // Посылаем строку серверу
            sock->Write(buf, sizeof(buf));

            break;
        }
        case wxSOCKET_INPUT:
        {
            sock->Read(buf, sizeof(buf));

            break;
        }

        // Сервер закрывает соединение после отправки данных
        case wxSOCKET_LOST:
        {
            sock->Destroy();

            break;
        }
    }
}
```

18.2.2 Сервер

Вот код программы-сервера.

```
BEGIN_EVENT_TABLE(MyFrame, wxFrame)
    EVT_MENU(SERVER_START, MyFrame::OnServerStart)
    EVT_SOCKET(SERVER_ID,  MyFrame::OnServerEvent)
    EVT_SOCKET(SOCKET_ID,  MyFrame::OnSocketEvent)
END_EVENT_TABLE()

void MyFrame::OnServerStart(wxCommandEvent& WXUNUSED(event))
{
    // Создаем адрес - по умолчанию localhost:3000
    wxIPV4address addr;
    addr.Service(3000);

    // Создаем сокет. Мы должны хранить данный указатель на класс,
    // чтобы иметь возможность его впоследствии корректно уничтожить
    m_server = new wxSocketServer(addr);

    // Используем метод Ok(), чтобы убедиться, что сервер работает
```

```
    if (! m_server->Ok())
    {
        return;
    }

    // Устанавливаем обработчик событий
    // и подписываемся на большинство событий
    m_server->SetEventHandler(*this, SERVER_ID);
    m_server->SetNotify(wxSOCKET_CONNECTION_FLAG);
    m_server->Notify(true);
}

void MyFrame::OnServerEvent(wxSocketEvent& WXUNUSED(event))
{
    // Принять новое соединение и получить указатель на сокет
    wxSocketBase* sock = m_server->Accept(false);

    // Сообщаем новому сокету как и где обрабатывать события
    sock->SetEventHandler(*this, SOCKET_ID);
    sock->SetNotify(wxSOCKET_INPUT_FLAG | wxSOCKET_LOST_FLAG);
    sock->Notify(true);
}

void MyFrame::OnSocketEvent(wxSocketEvent& event)
{
    wxSocketBase *sock = event.GetSocket();

    // Обработка события
    switch(event.GetSocketEvent())
    {
        case wxSOCKET_INPUT:
        {
            char buf[10];

            // Читаем данные в буфер
            sock->Read(buf, sizeof(buf));

            // Записываем их обратно
            sock->Write(buf, sizeof(buf));

            // Работа с сокетом завершена, уничтожаем его
            sock->Destroy();

            break;
        }
        case wxSOCKET_LOST:
        {
```

```
            sock->Destroy();
            break;
        }
    }
}
```

18.2.3 Подключение к серверу

В данном разделе объясняется, как инициировать подключение клиента к серверу, используя классы `wxSockAddress` и `wxSocketClient`.

Адресация сокета

Все классы, представляющие собой адрес сокета, являются производными от базового класса `wxSockAddress`, благодаря которому в методы можно легко передать адрес, независимо от используемого протокола. Класс `wxIPV4address` предоставляет все необходимые методы для задания удаленного хоста, используя текущую стандартную для Интернет схему адресации IPv4. Класс `wxIPV6address` частично реализован и будет завершен, когда IPv6 приобретет более широкое распространение.

Примечание: При представлении адреса в виде беззнакового целого числа необходимо учитывать порядок байтов в сети. Порядок байтов в сети соответствует порядку big endian (32- и 64-битные архитектуры Intel или AMD используют little endian, а архитектура Apple — big endian). В зависимости от того, как беззнаковые целые адреса хранятся или вводятся, вам, возможно, потребуется использовать макрос `wxINT32_SWAP_ON_LE`, который поменяет порядок байтов на правильный только для платформы little endian. Например:

```
IPV4addr.Hostname(wxINT32_SWAP_ON_LE(longAddress));
```

Метод `Hostname` может принимать строку адреса в формате `wxString` (например, www.wxwidgets.org) или IP-адрес в виде 4-байтового целого числа (как уже отмечалось ранее в big endian). Без каких-либо параметров, `Hostname` возвращает имя хоста в настоящий момент.

Метод `Service` устанавливает удаленный порт, который можно задать используя строковое (`wxString`) описание для указания известного порта или `unsigned short` для указания произвольного порта. Вызов метода без параметров возвращает номер порта, выбранный в настоящий момент.

`IPAddress` возвращает адрес удаленного хоста в виде IP-адреса в формате `wxString`.

`AnyAddress` устанавливает адрес равным любому адресу текущей машины. Эквивалентен использованию адреса `INADDR_ANY`.

Клиенты сокета

Класс `wxSocketClient` является производным от `wxSocketBase` и наследует все методы базового класса. Добавленные методы предназначены для инициализации и установки подключения к удаленному серверу.

Метод `Connect` принимает аргумент класса `wxSockAddress`, указывающий клиенту адрес и порт сервера для подключения. Как упоминалось ранее, вы должны использовать классы наподобие `wxIPV4address`, а не `wxSockAddress` напрямую. Второй параметр, по умолчанию равный `true`, указывает на то, что вызов `Connect` должен быть блокирующим. Если такой вызов делать в основном потоке, то на время подключения GUI будет заблокирован.

Метод `WaitOnConnect` можно использовать после вызова `Connect` в случаях, если вызов `Connect` был деблокирующим В первом параметре передается время ожидания в секундах, а во втором — время ожидания, но в миллисекундах Если соединение успешно установлено или запрос провалился (например, было задано неправильное имя хоста), то возвращается `true`. Если время ожидания истекло, то возвращается `false`. Если параметр задать равным `-1`, то будет использовано значение по умолчанию, которое составляет 10 минут, но может быть переопределено с помощью `SetTimeout`.

18.2.4 События от сокетов

Все события от сокетов можно отфильтровать с помощью одного макроса — `EVT_SOCKET`.

`EVT_SOCKET(идентификатор, функция)` посылает события от сокета с определенным идентификатором указанной функции. Функция должна принимать `wxSocketEvent` в качестве аргумента.

Хотя класс `wxSocketEvent` сам по себе очень простой, но содержит тип события и сокет, для которого данное событие имело место. В результате отпадает необходимость хранить указатель на сокет.

Типы событий от сокетов

Таблица 18.1 содержит список типов событий, которые возвращает метод `GetSocketEvent`.

Таблица 18.1: Типы событий от сокетов

Тип	Описание
`wxSOCKET_INPUT`	Создается всякий раз, когда имеются данные, доступные для чтения. Это происходит в том случае, если входная очередь была пуста, а потом поступили новые данные, или если приложение прочитало некоторые данные, но не все данные были прочитаны.
`wxSOCKET_OUTPUT`	Создается, если сокет подключился с помощью `Connect` или был принят с `Accept`. После этого, такое сообщение будет сгенерировано только в случае, когда операция вывода сначала прервется, а потом буфер вновь станет доступным.
`wxSOCKET_CONNECTION`	Создается, когда успешно завершается запрос на подключение (клиент), или когда новое соединение прибывает во входящую очередь (сервер).

Таблица 18.1: Типы событий от сокетов (продолжение)

Тип	Описание
`wxSOCKET_LOST`	Создается, когда происходит закрытие соединения. Это означает, что соединение утрачено или удаленный узел закрыл его. Такое событие также будет сгенерировано, если установление соединения провалилось.

Основные методы wxSocketEvent

`wxSocketEvent` используется в качестве параметра для обработчиков событий сокетов.

Метод `GetSocket` возвращает указатель на `wxSocketBase`, который является сокетом, породившим данное событие.

`GetSocketEvent` возвращает тип произошедшего события из таблицы 18.1.

Использование событий от сокетов

Для того чтобы использовать механизм событий сокетов, вам необходимо написать обработчик событий, а также указать сокету какие именно события он должен генерировать. Класс `wxSocketBase` содержит несколько методов для управления событиями, использование которых можно увидеть в коде нашей программы-сервера сразу после создания сокета. Заметим, что сокет генерирует только указанные ему события, поэтому для каждого создаваемого сокета необходимо явно указать события, которые вы хотите получать.

`SetEventHandler` принимает ссылку на обработчик событий и идентификатор события. Идентификатор события должен соответствовать записи в таблице событий для класса-обработчика событий.

`SetNotify` принимает в качестве параметра комбинацию битовых флагов, устанавливающих генерируемые события. Например, при установке `wxSOCKET_INPUT_FLAG | wxSOCKET_LOST_FLAG` событие будет сгенерировано, только если есть данные для чтения или когда сокет будет закрыт.

`Notify` принимает параметр типа `bool`, который включает/выключает генерацию событий сокетом. Данный метод позволяет включать или отключать события по мере надобности без необходимости переконфигурации событий, которые вы хотите получать.

18.2.5 Состояние сокета и оповещение об ошибках

До начала рассказа о передаче и приеме данных мы кратко опишем вспомогательные методы для уведомления о состоянии и ошибках, чтобы вы могли обращаться к ним при работе с данными.

`Close` закрывает сокет и отключает дальнейшую передачу данных. Удаленная сторона явно уведомляется о том, что вы закрыли соединение. Заметим, что когда вы закрываете сокет некоторые события от сокета могут уже находится в очереди

событий. Таким образом, необходимо быть готовым к тому, что возможно получить события от сокета даже после его закрытия.

`Destroy` используется вместо оператора `delete`, поскольку события могли «дойти» до сокета и после того, как он был удален с помощью оператора `delete`. `Destroy` закрывает сокет и добавляет его в список объектов, которые будут удалены после обработки всех событий.

`Error` возвращает `true`, если при выполнении последней операции произошла ошибка.

`GetPeer` возвращает объект класса `wxSockAddress`, содержащий информацию об удаленном компьютере, такую как IP-адрес и порт.

`IsConnected` возвращает `true`, если сокет подключен и `false` в противном случае.

`LastCount` возвращает количество прочитанных байтов при последней операции ввода/вывода.

`LastError` возвращает последнюю ошибку. Обратите внимание, что успешная операция не обновляет код ошибки, поэтому сначала необходимо вызвать метод `Error`, чтобы определить произошла ли ошибка. В таблице 18.2 приведены значения кода ошибки.

Таблица 18.2: Коды ошибок для сокетов

Тип	Описание
`wxSOCKET_INVOP`	Недопустимая операция. Например, используется недопустимый тип адреса.
`wxSOCKET_IOERR`	Ошибка ввода / вывода. Например, не удалось инициализировать сокет.
`wxSOCKET_INVADDR`	Неправильный адрес. Происходит при попытке подключения без указания адреса или если адрес указан неправильно.
`wxSOCKET_INVSOCK`	Сокет был использован в недопустимом контексте или не был должным образом инициализирован.
`wxSOCKET_NOHOST`	Указанный адрес не существует.
`wxSOCKET_INVPORT`	Был указан недопустимый порт.
`wxSOCKET_WOULDBLOCK`	Сокет не может быть заблокирован, хотя операция требует блокировки (смотрите обсуждение режимов сокета).
`wxSOCKET_TIMEDOUT`	Операция на сокете завершилась по тайм-ауту.
`wxSOCKET_MEMERR`	Операция затребовала слишком много памяти.

`Ok` возвращает для клиента `true` если он подключен к серверу, а для сервера — когда сервер подключен и ждет запроса.

`SetTimeout` определяет максимальное время (в секундах) блокировки сокета. По умолчанию данная величина равна 10 минутам.

18.2.6 Прием и передача данных через сокет

`wxSocketBase` поддерживает различные базовые и продвинутые методы для чтения и записи данных через сокет. Все операции чтения и записи сохраняют

результаты операции и позволяют через `LastCount` получить число прочитанных байт, а также последнюю ошибку, используя `LastError`.

Чтение

`Discard` удаляет все пришедшие данные из буфера сокета.

`Peek` позволяет получить доступ к данным внутри буфера, не удаляя их оттуда. Методу необходимо передать буфер для считывания данных, а также его размер.

`Read` извлекает данные из буфера сокета и копирует их в переданный буфер, вплоть до указанного размера.

`ReadMsg` читает данные, посылаемые `WriteMsg` в указанный буфер с определенным размером. Если буфер переполнится, то лишние данные будут отброшены. `ReadMsg` всегда пытается получить все сообщение, посланное с использованием `WriteMsg`, если не происходит ошибка.

`Unread` копирует данные из предоставленного буфера обратно в буфер сокета. Также необходимо указать количество данных для возвращения.

Запись

`Write` передает данные через сокет. Вам необходимо указать передаваемые данные и количество байт для отправки.

`WriteMsg` очень похож на `Write`, но при передаче также добавляет специальный заголовок. Заголовок используется для того, чтобы клиент точно знал, сколько данных необходимо прочитать. Обратите внимание, что данные, передаваемые с использованием `WriteMsg`, на удаленной стороне необходимо читать с помощью `ReadMsg`.

18.2.7 Создание сервера

Класс `wxSocketServer` добавляет к `wxSocketBase` несколько методов для создания слушателя и приема соединений. Чтобы создать сервер необходимо определить порт для приема входящих соединений. При конструировании `wxSocketServer` использует тот же класс `wxIPV4address`, что и для `wxSocketClient`, но без указания удаленного хоста. В большинстве случаев, после создания серверного сокета необходимо вызвать метод `Ok`, чтобы проверить, что сокет успешно создан и принимает входящие соединения.

Основные методы wxSocketServer

`wxSocketServer` принимает адрес объекта с указанием порта для прослушивания, а также необязательных флагов (смотрите раздел «Флаги для сокетов» далее в этой главе).

`Accept` принимает входящий запрос на соединение, если таковой имеется, и создает для него сокет. Опционально можно указать необходимость ожидать новое подключение или сразу возвращать `NULL`, если ожидающего входящего соединения нет. Если вы укажете необходимость ожидания, то GUI на это время будет заблокирован.

`AcceptWith` работает аналогично `Accept`, но ему необходимо передать ссылку на уже существующий объект класса `wxSocketBase`. Возвращается логическая переменная, указывающая было ли соединение принято.

`WaitForAccept` принимает на вход количество секунд и миллисекунд, означающее время в течении которого необходимо ожидать нового соединения. Метод возвращает `true`, если в указанный период времени был запрос на новое соединение и `false` в противном случае.

Обработка установления нового подключения

Когда слушающий сокет обнаруживает новое входящее соединение, то он сообщает об этом обработчику событий. Обработчик события может принять соединение и выполнить любые необходимые действия. Предполагая, что соединение будет некоторое время использоваться и сразу не закроется, вы можете назначить обработчик событий для нового сокета. Помните, что слушающий сокет продолжает принимать новые подключения, пока не будет закрыт, а новые сокеты создаются для каждого нового соединения. В течение жизни программы-сервера слушающий сокет может породить тысячи новых сокетов.

18.2.8 Краткие итоги рассказа о событиях сокетов

С точки зрения программиста использование сокетов с помощью событий являются наилучшим выбором для простой обработки данных от сокета, который устраняет необходимость в создании и удалении потоков. Наша тестовая программа не использует потоки, однако ее интерфейс не будет блокироваться во время ожидания данных. Так как команды чтения из сокета вызываются, если получены какие-нибудь данные, то чтение этих данных сразу успешно завершится и вернет доступные данные. Если необходимо прочитать большой объем данных, то он может быть разбит на небольшие кусочки, которые будут собираться в один буфер. А можно вызвать функцию `Peek`, чтобы узнать сколько данных доступно, и, если прибыли не все данные, то приложение может просто ждать следующего события чтения.

Далее мы рассмотрим, как различные флаги позволяют изменить поведения сокета.

18.3 Флаги для сокетов

Поведение сокета может существенно меняться в зависимости от переданных ему флагов. Флаги и их значения описаны в таблице 18.3 и более подробно далее.

Таблица 18.3: Флаги для сокетов

Тип	Описание
`wxSOCKET_NONE`	Обычная функциональность (поведение основано на нижележащих функциях `send` и `recv`).
`wxSOCKET_NOWAIT`	Читать и записывать максимально возможное количество данных и немедленно их вернуть.

Таблица 18.3: Флаги для сокетов (продолжение)

Тип	Описание
wxSOCKET_WAITALL	Ждать все затребованные для чтения или записи данные, пока не произойдет ошибка.
wxSOCKET_BLOCK	Блокировать GUI во время чтения или записи данных.

Если флаг не указан (по умолчанию wxSOCKET_NONE), то операции ввода/вывода завершаются после получения небольшого количества данных, даже если передача еще не завершена. Это эквивалентно одному блокирующему вызову низкоуровневых recv или send. Обратите внимание, что здесь «блокировка» относится к тому, когда завершиться вызов функции, а не к блокировке GUI в течение времени вызова.

Если указан флаг wxSOCKET_NOWAIT, то операции ввода/вывода будут немедленно возвращать управление. Операция чтения будут возвращать только уже имеющиеся данные, а операция записи будет записывать максимально возможное количество данных, зависящее от того, сколько свободного места доступно в выходном буфере. Такое поведение эквивалентно одному неблокирующему вызову низкоуровневых recv или send. Обратите внимание, что здесь «не блокируется» относится к тому, когда завершиться вызов функции, а не к блокировке GUI в течение времени вызова.

Если указан флаг wxSOCKET_WAITALL, то операции ввода/вывода не завершаться, пока все необходимые для чтения или записи данные не будут получены (или пока не возникнет ошибка), при необходимости выполняется несколько блокирующих низкоуровневых вызовов. Такое поведение эквивалентно циклу, который делает столько блокирующих низкоуровневых вызовов функций recv или send, чтобы передать все данные. Опять же, здесь «блокировка» относится к тому, когда завершиться вызов функции, а не к блокировке GUI в течение времени вызова. Обратите внимание, что использование ReadMsg и WriteMsg будет неявно использовать флаг wxSOCKET_WAITALL и игнорировать флаги wxSOCKET_NONE и wxSOCKET_NOWAIT.

Флаг wxSOCKET_BLOCK блокирует GUI во время операций ввода/вывода. Если данный флаг не указан, то сокет не посылает события об обновлении во время операций ввода/вывода. Таким образом GUI остается заблокированным до завершения операции. Если флаг не используется, то приложение должно само позаботиться о том, чтобы избежать нежелательных рекурсивных вызовов wxYield.

Подведем итог:

- с wxSOCKET_NONE сокет будет пытаться прочитать небольшое количество данных, независимо от того, сколько их на самом деле.
- с wxSOCKET_NOWAIT сокет всегда будет возвращать управление немедленно, даже если не сможет прочитать или записать какие-либо данные.
- с wxSOCKET_WAITALL сокет всегда будет возвращать управление только после прочтения или записи всех данных.
- флаг wxSOCKET_BLOCK не имеет ничего общего с предыдущими флагами. Он контролирует блокировку GUI во время операций с сокетом.

18.3.1 Блокирующие и неблокирующие сокеты в wxWidgets

Термин «блокировка» имеет в wxWidgets двойное значение. В обычном программировании сокетов блокировка означает, что текущий поток зависает (блокируется) во время вызова `recv` до возникновения тайм-аута или окончания чтения полного объема данных. Если заблокированным окажется главный поток выполнения, то и GUI будет тоже заблокирован.

Однако в wxWidgets блокировка может относиться к двум различным типам: блокировке сокета и блокировке GUI. Целью флага `wxSOCKET_BLOCK` является указание того, будет ли заблокирован GUI, если заблокирован вызов сокета. Как может оказаться, что вызов сокета будет заблокирован, а GUI — нет? Это возможно, так как события могут продолжать обрабатываться через вызов `wxYield` в то время как операция с сокетом ждет своего завершения. `wxYield` будет обрабатывать все события из очереди событий, включая события от графического интерфейса. Пока операции с сокетом не завершены, ваш код будет заблокирован функцией сокета, но события будут продолжать обрабатываться.

Для новичка в wxWidgets такое решение кажется панацеей для написания приложений с сокетами. Если вы первый раз знакомитесь с сокетами, то можете предположить, что вам никогда не потребуются другие потоки выполнения для обработки сокетов. Вы даже можете подумать, что сможете просто использовать события от сокетов, а для всех сокетов использовать флаг `wxSOCKET_WAITALL` без `wxSOCKET_BLOCK`. К сожалению, такая попытка может привести к фатальным последствиям и предупреждающим сообщениям, причина которых будет не ясна.

Рассмотрим случай сервера с двумя активными соединениями, каждое из которых использует флаг `wxSOCKET_WAITALL`. Также предположим, что необходимо получить большое количество данных по крайне медленному соединению. Сокет 1 не имеет всех необходимых данных в буфере чтения, поэтому он вызывает `wxYield`. В очереди событий существуют еще ожидающее событие от сокета 2, которое wxWidgets пытается обработать. Однако, это событие также не может быть завершено и оно вызывает `wxYield`. Это приводит к появлению печально известного сообщения «wxYield called recursively»[1]. В конце концов стек будет заполнен рекурсивными вызовами `wxYield`, пока не переполнится. Многие пользователи могут предположить, что это сообщение указывает на недостаток wxWidgets, хотя проблема кроется в коде приложения. Проще говоря, приложение должно быть запрограммировано так, чтобы такая ситуация не происходила. Если подобная ошибка присутствует в коде приложения, то она может и должна быть исправлена.

Побочным эффектом вызова сокета без блокировки GUI является то, что ваше приложение будет потреблять максимально возможное количество процессорного времени. Причина в том, что приложение остается доступным, а сокет должен постоянно проверять данные, чтобы немедленно вернуться, когда данные станут доступны. Единственный способ сделать это состоит в том, чтобы создать цикл, который постоянно обрабатывает не заблокированные события с последующим вызовом `wxYield`.

[1] wxYield вызывается рекурсивно (англ.)

Невозможный сокет

Не обманывайте себя, думая, что wxWidgets может создать чудесное средство работы с сокетами. Вы не можете одновременно получить следующие вещи (даже если вам кажется, что можете):

- `wxSOCKET_WAITALL`
- Без блокировки GUI
- Загрузка процессора не 100%
- Один поток выполнения

Вы *можете* указать флаг `wxSOCKET_WAITALL` без блокировки GUI, но это приведет к 100% загрузке процессора. Вы *можете* использовать `wxSOCKET_WAITALL` и получить 0% загрузки процессора только если также заблокируется графический интерфейс с помощью флага `wxSOCKET_BLOCK`. Вы *можете* использовать `wxSOCKET_WAITALL` без блокировки GUI и загрузки процессора в 100%, но только если будут использованы дочерние потоки. Вы *можете* использовать сокеты в одном потоке без 100% загрузки процессора и блокировки GUI, если используется флаг `wxSOCKET_NOWAIT`. Таким образом вы можете одновременно получить три любых результата, но не все четыре.

18.3.2 Как флаги влияют на поведение сокета

Так как флаги `wxSOCKET_NONE`, `wxSOCKET_NOWAIT` и `wxSOCKET_WAITALL` являются взаимно исключающими, а `wxSOCKET_BLOCK` не имеет смысла в сочетании с `wxSOCKET_NOWAIT` (если функция возвращает результат немедленно, то как она может блокировать графический интерфейс?), получаем, что всего существует пять разумных комбинаций флагов:

- `wxSOCKET_NONE | wxSOCKET_BLOCK`: Получаются стандартные блокирующие вызовы (используются `recv` и `send`).
- `wxSOCKET_NOWAIT`: Получаются стандартные неблокирующие вызовы.
- `wxSOCKET_WAITALL | wxSOCKET_BLOCK`: Получаются стандартные блокирующие вызовы, но `recv` или `send` вызываются пока не будут получены/отправлены все необходимые данные.
- `wxSOCKET_NONE`: Получаются стандартные вызовы методов сокетов за исключением того, что графический интерфейс не будет блокироваться, так как постоянно вызывается `wxYield`, пока операция не будет завершена (например, не все данные получены).
- `wxSOCKET_WAITALL`: Ведет себя подобно `wxSOCKET_WAITALL | wxSOCKET_BLOCK`, но графический интерфейс не блокируется.

Два последних флага могут привести к рекурсивному вызову `wxYield`, хотя они также являются и наиболее полезными при использовании в основном потоке (так как они блокируют сокет, но не блокируют GUI). Использование этих двух вариантов должно происходить с крайней осторожностью, так как они могут являться источником множества проблем и падений, часто происходящих из-за непонимания принципов работы данных методов.

18.3.3 Использование wxSocket в качестве стандартного сокета

Использование сокета с флагами `wxSOCKET_NONE | wxSOCKET_BLOCK` или `wxSOCKET_NOWAIT` ничем не отличается от использования стандартных сокетов языка C, кроме того, что вы вызываете методы `wxSocket` вместо функций языка C. Но у класса `wxSocket` по-прежнему будет множество преимуществ по сравнению с использованием C API напрямую, в том числе объектно-ориентированный интерфейс, отсутствие множества платформо-зависимого кода инициализации, обеспечивающего согласованное поведение на различных платформах (особенно актуально в отношении кодов ошибок, которые отличаются друг от друга на большинстве платформ), а также наличие некоторых методов более высокого уровня, таких как `WriteMsg` и `ReadMsg`. Как мы увидим далее, класс `wxSocket` также поддерживает использование потоков ввода/вывода.

18.4 Использование потоков для сокетов

Используя потоки ввода/вывода в wxWidgets, можно легко перемещать и преобразовывать большое количество данных, написав всего несколько строчек кода. Рассмотрим задачу отправки некоторого файла через сокет. Один из вариантов ее решения — открыть файл, прочитать содержимое файла в память, а затем отправить блок памяти в сокет. Такой подход хорошо работает для небольших файлов, но чтение мегабайтных или гигабайтных файлов в память будет очень долгим на компьютерах с небольшим количеством памяти, а в большинстве случаев — просто невозможным. Кроме того, что если вы хотите сжать файл перед отправкой, чтобы уменьшить сетевой трафик? Чтение больших файлов в память, последующее сжатие и запись в сокет будет просто не эффективным и не практичным.

Второй подход подразумевает чтение файла небольшими кусками (по несколько килобайт), сжатие этих кусочков и их посылку через сокет. К сожалению, сжатие отдельных кусочков будет не таким эффективным, как сжатие всего файла целиком. Однако можно воспользоваться непрерывным сжатием (когда для сжатия одной части используется информацию из предыдущей, что помогает избежать необходимости иметь свой собственный заголовок для каждой части), что приведет к необходимости написать десятки строк кода для синхронизации чтения из файла, сжатия и передачи. Библиотека wxWidgets предоставляет лучший путь решения данной проблемы.

Так как в wxWidgets реализованы классы `wxSocketInputStream` и `wxSocketOutputStream`, то появляется возможность очень легко пропустить данные из других потоков через сокеты. Учитывая, что wxWidgets также содержит

потоки для файлов, строк, текста, памяти и сжатия Zlib, то это дает возможность использования сокетов в уникальных и эффективных решениях. Вернувшись к проблеме передачи файла со сжатием, становится очевидным новое решение. Чтобы отправить файл необходимо файловый поток перенаправить потоку сжатия, а далее в сокет. В итоге получается непрерывное сжатие файла, которое читает не более нескольких килобайт за раз. На принимающей стороне мы посылаем поток данных от сокета в поток декомпрессии Zlib и, наконец, в выходной файл. Все это может быть сделано с помощью нескольких строк кода.

Если делать все операции в отдельном потоке выполнения, то можно не беспокоится о блокировке GUI или нагрузке процессора в 100%, как могло произойти в первых двух решениях при передаче крупных файлов.

Полные исходники использования потоков сокетов можно найти на прилагаемом CD-ROM в каталоге examples/chap18.

18.4.1 Поток для пересылки файла

Пересылающий поток демонстрирует использование потоков ввода/вывода, располагающихся в куче. FileSendThread является наследником от wxThread.

```
FileSendThread::FileSendThread(wxString Filename, wxSocketBase* Socket)
{
     m_Filename = Filename;
     m_Socket = Socket;
     Create();
     Run();
}

void* FileSendThread::Entry()
{
     // Устанавливаем тайм-аут
     m_Socket->SetTimeout(10);
     // Ждем, когда все данные будут записаны.
     // На это время блокируем сокет
     m_Socket->SetFlags(wxSOCKET_WAITALL | wxSOCKET_BLOCK);
     // Читаем указанный файл
     wxFileInputStream* FileInputStream
          = new wxFileInputStream(m_Filename);
     // Выходной поток для записи в сокет
     wxSocketOutputStream* SocketOutputStream
          = new wxSocketOutputStream(*m_Socket);
     // Результаты сжатия будут записаны в поток сокета
     wxZlibOutputStream* ZlibOutputStream
          = new wxZlibOutputStream(*SocketOutputStream);
     // Записываем результатов сжатия в файловый поток
     ZlibOutputStream->Write(*FileInputStream);
     // Передача всех данных
     ZlibOutputStream->Sync();
```

```
    // delete пошлет EOF в поток сжатия
    delete ZlibOutputStream;
    // Удаляем не нужные объекты
    delete SocketOutputStream;
    delete FileInputStream;
    return NULL;
}
```

18.4.2 Поток для получения файла

Принимающий поток демонстрирует использование потоков ввода/вывода, располагающихся на стеке. FileReceiveThread является наследником от wxThread.

```
FileReceiveThread::FileReceiveThread(
    wxString Filename, wxSocketBase* Socket)
{
    m_Filename = Filename;
    m_Socket = Socket;
    Create();
    Run();
}

void* FileReceiveThread::Entry()
{
    // Устанавливаем тайм-аут
    m_Socket->SetTimeout(10);
    // Ждем поступающие данные или ошибку. На время вызова
    // блокируем сокет
    m_Socket->SetFlags(wxSOCKET_WAITALL | wxSOCKET_BLOCK);
    // Пишем результат в указанный файл
    wxFileOutputStream FileOutputStream(m_Filename);
    // Получаем поток данных от сокета
    wxSocketInputStream SocketInputStream(*m_Socket);
    // Этот объект будет распаковывать данные из сокета в поток
    wxZlibInputStream ZlibInputStream(SocketInputStream);
    // Запись в файловый поток результатов чтения
    // из потока распаковщика
    FileOutputStream.Write(ZlibInputStream);
    return NULL;
}
```

18.5 Альтернативы использованию wxSocket

Хотя `wxSocket` обеспечивает невероятную гибкость и хорошо интегрирован в wxWidgets, это не единственный метод, который можно использовать для связи с другими узлами. Если вам просто необходимо выполнить FTP- или HTTP-операции, то можно использовать классы `wxFTP` или `wxHTTP`, которые написаны на основе

wxSocket. Однако эти классы являются неполными, и вам, возможно, захочется использовать CURL — популярную библиотеку, которая дает очень простой API для передачи файлов с использованием множества распространенных протоколов. Существует оболочка над CURL для wxWidgets, которая называется wxCURL.

wxWidgets также обеспечивает высокоуровневый способ межпроцессного взаимодействия, посредством использования классов wxServer, wxClient и wxConnection, а также API, построенного на базе протокола Microsoft DDE (Dynamic Data Exchange). Фактически, в системе Windows эти классы построены на базе DDE, а на других платформах — на сокетах. Основным преимуществом использования данного высокоуровнего API является простота его использовании по сравнению с использованием wxSocket. Еще одним преимуществом является то, что под Windows, ваше приложение станет доступны через DDE, позволяя другим приложениям (не обязательно написанным на wxWidgets) получить к нему доступ. Недостаток в том, что на платформах, отличных от Windows, данный протокол является нестандартным, поэтому не-wxWidgets приложениям сложно по нему взаимодействовать Однако, если вам просто необходимо, чтобы два приложения на wxWidgets могли взаимодействовать друг с другом, то данный протокол полностью решит эту задачу. Очень простой пример такого взаимодействия будет показан в разделе «Single Instance or Multiple Instances?» главы 20 «Perfecting Your Application».

Для получения дополнительной информации вы можете обратится к теме «Interprocess Communication Overview» в справочной документации wxWidgets и к исходникам samples/ipc дистрибутива wxWidgets. Также можно посмотреть указанные классы в действии в стандартном просмотрщике помощи utils/helpview/src, также находящимся в пакете wxWidgets.

18.6 Итоги

Вы увидели, что класс wxSocket обеспечивает переносимость и многочисленные усовершенствования сокетов операционной системы. Чтобы сделать программирование сокетов еще проще, в wxWidgets также реализованы классы потоков для сокетов, которые можно сочетать с другими подобными классами. Вдумчиво изучив материал данной главы вы сможете осуществлять переносимые и надежные операции над сокетами в системах Windows, Linux и Mac OS X, используя wxSocket и связанные с ним классы.

Далее будет рассказано как можно упростить написание приложения с помощью использования технологии документ/вид.