21

Работа с сетью и сокеты

В этой главе описываются модули, используемые для низкоуровневой реализации сетевых серверов и клиентов. Стандартная библиотека языка Python обеспечивает широкую поддержку сетевых операций, начиная от операций с сокетами и заканчивая функциями для работы с прикладными протоколами высокого уровня, такими как HTTP. Для начала будет дано краткое (действительно, очень краткое) введение в разработку сетевых приложений. За дополнительной информацией читателям рекомендуется обращаться к специализированным книгам, таким как «UNIX Network Programming, Volume 1: Networking APIs: Sockets and XTI» У. Ричарда Стивенса (W. Richard Stevens) (издательство Prentice Hall, 1997, ISBN 0-13-490012-X).[1] Модули, предназначенные для работы с протоколами прикладного уровня, описываются в главе 22 «Разработка интернет-приложений».

Основы разработки сетевых приложений

Модули, входящие в стандартную библиотеку Python и предназначенные для разработки сетевых приложений, главным образом поддерживают два протокола Интернета: TCP и UDP. *Протокол TCP* – это надежный протокол с созданием логического соединения, используемый для создания между компьютерами двустороннего канала обмена данными. *Протокол UDP* – это низкоуровневый протокол, обеспечивающий возможность обмена пакетами, с помощью которого компьютеры могут отправлять и получать информацию в виде отдельных пакетов, без создания логического соединения. В отличие от TCP, взаимодействия по протоколу UDP не отличаются надежностью, что усложняет управление ими в приложениях, в которых необходимо гарантировать надежность обмена информацией. По этой причине большинство интернет-приложений используют протокол TCP.

---

[1] Стивенс У. «UNIX. Разработка сетевых приложений». – Пер. с англ. – СПб.: Питер, 2003.

Работа с обоими протоколами осуществляется с помощью программной абстракции, известной как сокет. *Сокет* – это объект, напоминающий файл, позволяющий программе принимать входящие соединения, устанавливать исходящие соединения, а также отправлять и принимать данные. Прежде чем два компьютера смогут обмениваться информацией, на каждом из них должен быть создан объект сокета.

Компьютер, принимающий соединение, (сервер) должен присвоить своему объекту сокета определенный номер порта. *Порт* – это 16-битное число в диапазоне 0 – 65 535, которое используется клиентами для уникальной идентификации серверов. Порты с номерами 0 – 1023 зарезервированы для нужд системы и используются наиболее распространенными сетевыми протоколами. Ниже перечислены некоторые распространенные протоколы с присвоенными им номерами портов (более полный список можно найти по адресу: *http://www.iana.org/assignments/port-numbers*):

Служба	**Номер порта**
FTP-Data	20
FTP-Control	21
SSH	22
Telnet	23
SMTP (электронная почта)	25
HTTP (WWW)	80
IMAP	143
HTTPS (безопасная WWW)	443

Процедура установки TCP-соединения между клиентом и сервером определяется точной последовательностью операций, как показано на рис. 21.1.

На стороне сервера, работающего по протоколу TCP, объект сокета, используемый для приема запросов на соединение, – это не тот же самый сокет, что в дальнейшем используется для обмена данными с клиентом. В частности, системный вызов `accept()` возвращает новый объект сокета, который фактически будет использоваться для обслуживания соединения. Это позволяет серверу одновременно обслуживать соединения с большим количеством клиентов.

Взаимодействия по протоколу UDP выполняются похожим способом, за исключением того, что клиенты и серверы не устанавливают логическое соединение друг с другом, как показано на рис. 21.2.

Следующий пример иллюстрирует применение протокола TCP клиентом и сервером, использующими модуль `socket`. В этом примере сервер просто возвращает клиенту текущее время в виде строки.

```
# Программа сервера времени
from socket import *
import time
```

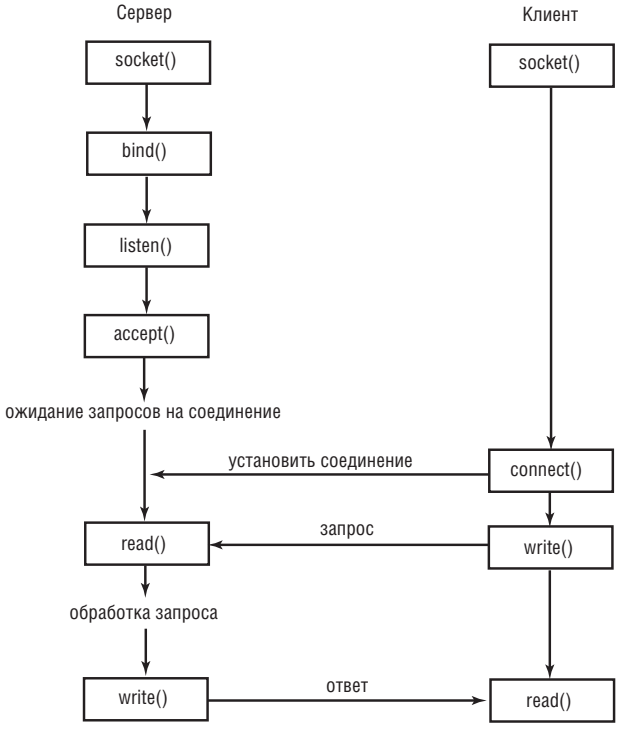

Рис. 21.1. *Процедура установки TCP-соединения*

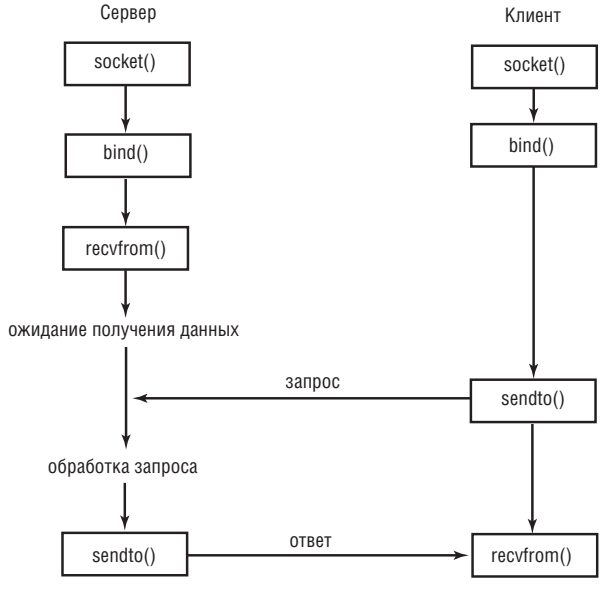

Рис. 21.2. *Обмен данными по протоколу UDP*

```
s = socket(AF_INET, SOCK_STREAM)  # Создает сокет TCP
s.bind(('',8888))                 # Присваивает порт 8888
s.listen(5)                       # Переходит в режим ожидания запросов;
                                  # одновременно обслуживает не более
                                  # 5 запросов.
while True:
    client,addr = s.accept()      # Принять запрос на соединение
    print("Получен запрос на соединение с %s" % str(addr))
    timestr = time.ctime(time.time()) + "\r\n"
    client.send(timestr.encode('ascii'))
    client.close()
```

Ниже приводится клиентская программа:

```
# Программа клиента, запрашивающего текущее время
from socket import *
s = socket(AF_INET,SOCK_STREAM)  # Создать сокет TCP
s.connect(('localhost', 8888))   # Соединиться с сервером
tm = s.recv(1024)                # Принять не более 1024 байтов данных
s.close()
print("Текущее время: %s" % tm.decode('ascii'))
```

Пример реализации обмена данными по протоколу UDP приводится в разделе с описанием модуля `socket` ниже, в этой главе.

В сетевых протоколах обмен данными часто выполняется в текстовой форме. Поэтому особое внимание необходимо уделять кодировке текста. В Python 3 все строки состоят из символов Юникода, что влечет за собой необходимость кодировать строки, передаваемые через сеть. Именно по этой причине в программе сервера к отправляемым данным применяется метод `encode('ascii')`. Точно так же, когда клиент принимает данные из сети, эти данные поступают в виде простой последовательности кодированных байтов. Если вывести эту последовательность на экран или попробовать интерпретировать ее как текст, результат, скорее всего, получится совсем не тот, какого вы ожидали. Поэтому прежде чем работать с данными, их необходимо декодировать. Для этого в программе клиента применяется метод `decode('ascii')` к принимаемым данным.

В оставшейся части главы описываются модули, имеющие отношение к программированию с применением сокетов. В главе 22 описываются высокоуровневые модули, обеспечивающие поддержку различных интернет-приложений, таких как электронная почта и Веб.

Модуль asynchat

Модуль `asynchat` упрощает реализацию приложений, в которых используются асинхронные сетевые операции, поддерживаемые модулем `asyncore`. Это достигается за счет обертывания низкоуровневых операций ввода-вывода, реализованных в модуле `asyncore`, высокоуровневым программным интерфейсом, предназначенным для работы с сетевыми протоколами, основанными на простых механизмах типа «запрос-ответ» (например, HTTP).

Для работы с этим модулем необходимо определить класс, производный от класса `async_chat`. Внутри этого класса необходимо определить два метода: `collect_incoming_data()` и `found_terminator()`. Первый метод вызывается всякий раз, когда через сетевое соединение поступают какие-либо данные. Обычно этот метод просто принимает данные и сохраняет их. Метод `found_terminator()` вызывается, когда будет обнаружен конец запроса. Например, при использовании протокола HTTP признаком конца запроса является пустая строка.

Для вывода данных объекты класса `async_chat` поддерживают выходную очередь FIFO. Если программе потребуется отправить данные, ей достаточно просто добавить их в очередь. Когда операция записи в сетевое соединение станет возможной, данные из этой очереди будут отправлены автоматически.

`async_chat([`*`sock`*`])`

Базовый класс. Используется для создания новых обработчиков. Класс `async_chat` является производным от класса `asyncore.dispatcher` и предоставляет те же методы. В аргументе *`sock`* передается объект сокета, который будет использоваться для обмена данными.

Экземпляр *`a`* класса `async_chat` обладает следующими методами помимо тех, что уже предоставляются базовым классом `asyncore.dispatcher`:

`a``.close_when_done()`

Сообщает об окончании последовательности исходящих данных, добавляя `None` в выходную очередь FIFO. Когда процедура записи обнаружит это значение, она закроет канал.

`a``.collect_incoming_data(`*`data`*`)`

Вызывается всякий раз при поступлении очередной порции данных. В аргументе *`data`* передаются полученные данные, которые обычно сохраняются для последующего использования. Этот метод должен быть реализован пользователем.

`a``.discard_buffers()`

Уничтожает все данные, хранящиеся в буферах ввода-вывода и в выходной очереди FIFO.

`a``.found_terminator()`

Вызывается, когда обнаруживается признак конца данных, установленный методом `set_terminator()`. Этот метод должен быть реализован пользователем. Обычно в этом методе выполняется обработка данных, собранных методом `collect_incoming_data()`.

`a``.get_terminator()`

Возвращает признак окончания последовательности данных.

`a``.push(`*`data`*`)`

Добавляет данные в выходную очередь FIFO. В аргументе *`data`* передается строка с исходящими данными.

a.push_with_producer(*producer*)

Добавляет объект поставщика *producer* в выходную очередь FIFO. Объект *producer* может быть любым объектом, имеющим метод more(). Метод more() должен возвращать строку при каждом вызове. Пустая строка служит признаком окончания данных. За кулисами объект класса async_chat в цикле будет вызывать метод more(), чтобы получить данные для записи в канал вывода. В очередь FIFO можно добавить несколько объектов поставщиков, многократно вызвав метод push_with_producer().

s.set_terminator(*term*)

Устанавливает признак окончания данных. Аргумент *term* может быть строкой, целым числом или объектом None. Если в аргументе *term* передается строка, всякий раз, когда она будет встречаться во входных данных, будет вызываться метод found_terminator(). Если в аргументе *term* передается целое число, оно интерпретируется, как счетчик байтов. Всякий раз, когда будет принято указанное количество байтов, будет вызываться метод found_terminator(). Если в аргументе *term* передается None, прием данных будет осуществляться до бесконечности.

В модуле имеется класс, который может передаваться методу *a*.push_with_producer() для воспроизводства данных.

simple_producer(*data* [, *buffer_size*])

Создает простой объект поставщика, который делит строку байтов *data* на фрагменты. Аргумент *buffer_size* определяет размер фрагмента и по умолчанию принимает значение 512.

Модуль asynchat всегда используется совместно с модулем asyncore. Например, модуль asyncore может использоваться для создания сервера, принимающего входящие соединения, а модуль asynchat — для реализации обработчиков соединений. В следующем примере демонстрируется, как реализовать минимально возможный веб-сервер, обрабатывающий запросы GET. В примере почти полностью отсутствуют проверки на наличие ошибок и другие особенности, но его должно быть достаточно для начала. Сравните этот пример с примером, который приводится ниже, в разделе с описанием модуля asyncore.

```
# Асинхронный сервер HTTP, реализованный на основе модуля asynchat
import asynchat, asyncore, socket
import os
import mimetypes
try:
    from http.client import responses     # Python 3
except ImportError:
    from httplib import responses         # Python 2

# Следующий класс включает модуль asyncore
# и просто принимает входящие соединения
class async_http(asyncore.dispatcher):
    def __init__(self,port):
        asyncore.dispatcher.__init__(self)
        self.create_socket(socket.AF_INET,socket.SOCK_STREAM)
```

```
        self.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
        self.bind((‘’,port))
        self.listen(5)

    def handle_accept(self):
        client,addr = self.accept()
        return async_http_handler(client)

# Следующий класс обслуживает асинхронные запросы HTTP.
class async_http_handler(asynchat.async_chat):
    def __init__(self,conn=None):
        asynchat.async_chat.__init__(self,conn)
        self.data = []
        self.got_header = False
        self.set_terminator(b"\r\n\r\n")

    # Принимает входящие данные и добавляет их в буфер
    def collect_incoming_data(self,data):
        if not self.got_header:
            self.data.append(data)

    # Обрабатывает признак конца данных (пустая строка)
    def found_terminator(self):
        self.got_header  = True
        header_data      = b"".join(self.data)
        # Преобразовать данные заголовка (двоичные) в текст
        # для последующей обработки
        header_text      = header_data.decode(‘latin-1’)
        header_lines     = header_text.splitlines()
        request          = header_lines[0].split()
        op               = request[0]
        url              = request[1][1:]
        self.process_request(op,url)

    # Добавляет текст в исходящий поток, предварительно кодируя его
    def push_text(self,text):
        self.push(text.encode(‘latin-1’))

    # Обрабатывает запрос
    def process_request(self, op, url):
        if op == “GET”:
            if not os.path.exists(url):
                self.send_error(404,"File %s not found\r\n")
            else:
                type, encoding = mimetypes.guess_type(url)
                size = os.path.getsize(url)
                self.push_text(“HTTP/1.0 200 OK\r\n")
                self.push_text(“Content-length: %s\r\n” % size)
                self.push_text(“Content-type: %s\r\n” % type)
                self.push_text(“\r\n")
                self.push_with_producer(file_producer(url))
        else:
            self.send_error(501,"%s method not implemented” % op)
        self.close_when_done()
```

```
        # Обработка ошибок
        def send_error(self,code,message):
            self.push_text("HTTP/1.0 %s %s\r\n" % (code, responses[code]))
            self.push_text("Content-type: text/plain\r\n")
            self.push_text("\r\n")
            self.push_text(message)

    class file_producer(object):
        def __init__(self,filename,buffer_size=512):
            self.f = open(filename,"rb")
            self.buffer_size = buffer_size
        def more(self):
            data = self.f.read(self.buffer_size)
            if not data:
                self.f.close()
            return data

    a = async_http(8080)
    asyncore.loop()
```

Чтобы опробовать этот пример, необходимо указывать адреса URL, соответствующие файлам, находящимся в том же каталоге, где будет запущен сервер.

Модуль asyncore

Модуль `asyncore` используется для разработки сетевых приложений, в которых события, связанные с сетью, обрабатываются асинхронно, как последовательность событий, рассылаемых циклом событий, построенным на основе системного вызова `select()`. Такой подход удобно использовать в сетевых программах, реализующих многозадачность без использования потоков управления или процессов. Этот прием способен обеспечить высокую производительность при коротких операциях. Все функциональные возможности этого модуля представлены классом `dispatcher`, который является тонкой оберткой вокруг обычного объекта сокета.

`dispatcher([`*sock*`])`

Базовый класс, определяющий неблокирующий объект сокета, управляемый событиями. В аргументе *sock* передается существующий объект сокета. Если конструктор вызывается без аргумента, позднее необходимо будет создать сокет вызовом метода `create_socket()` (описывается ниже). После его создания сетевые события будут обрабатываться специальными методами-обработчиками. Кроме того, все созданные объекты класса `dispatcher` сохраняются во внутреннем списке, который используется некоторыми функциями опроса.

Для обработки сетевых событий вызываются следующие методы объектов класса `dispatcher`. Они должны быть определены в классах, производных от класса `dispatcher`.

d`.handle_accept()`

Вызывается объектом сокета, принимающим соединения, когда поступает новый запрос на соединение.

d.handle_close()

Вызывается при закрытии сокета.

d.handle_connect()

Вызывается после того, как соединение будет установлено.

d.handle_error()

Вызывается, когда появляется необработанное исключение.

d.handle_expt()

Вызывается, когда сокет получает срочные данные.

d.handle_read()

Вызывается, когда появляются новые данные, доступные для чтения из сокета.

d.handle_write()

Вызывается, когда выполняется операция записи данных.

d.readable()

Этот метод используется циклом select(), чтобы посмотреть, готов ли объект записывать данные. Возвращает True, если готов, и False – в противном случае. Этот метод вызывается, чтобы определить, должен ли вызываться метод handle_read(), чтобы сообщить о получении новых данных.

d.writable()

Вызывается циклом select(), чтобы посмотреть, готов ли объект записывать данные. Возвращает True, если готов, и False – в противном случае. Этот метод всегда вызывается, чтобы определить, должен ли вызываться метод handle_write(), чтобы произвести операцию записи.

Вдобавок к предыдущим методам для выполнения низкоуровневых операций над сокетом могут использоваться следующие методы. Они похожи на методы объекта сокета.

d.accept()

Принимает соединение. Возвращает кортеж (*client*, *addr*), где в поле *client* возвращается объект сокета, используемый для обмена данными через соединение, а в поле *addr* – адрес клиента.

d.bind(*address*)

Присваивает сокету указанный адрес *address*. В аргументе *address* обычно передается кортеж (*host*, *port*), однако точное представление адреса зависит от используемого семейства адресов.

d.close()

Закрывает сокет.

d.connect(*address*)

Устанавливает соединение. В аргументе *address* передается кортеж (*host*, *port*).

d.create_socket(*family*, *type*)

Создает новый сокет. Аргументы имеют тот же смысл, что и в функции socket.socket().

d.listen([*backlog*])

Принимает входящие соединения. В аргументе *backlog* передается целое число, которое передается функции socket.listen(), на основе которой реализован этот метод.

d.recv(*size*)

Принимает до *size* байтов. Пустая строка служит признаком того, что клиент закрыл канал.

d.send(*data*)

Отправляет данные *data*. В аргументе *data* передается строка байтов.

Следующая функция используется для запуска цикла приема и обработки событий:

loop([*timeout* [, *use_poll* [, *map* [, *count*]]]])

Запускает бесконечный цикл опроса. Если в аргументе *use_poll* передается значение False, опрос выполняется с помощью функции select(), в противном случае используется функция poll(). Аргумент *timeout* определяет предельное время ожидания и по умолчанию принимает значение 30 секунд. В аргументе *map* передается словарь со всеми каналами для мониторинга. Аргумент *count* определяет количество операций опроса, которые должны быть выполнены перед тем, как функция вернет управление. Если в аргументе *count* передать None (по умолчанию), функция loop() выполняет бесконечный цикл опроса, пока все каналы не будут закрыты. Если в аргументе *count* передать 1, функция выполнит одну проверку на наличие событий и вернет управление.

Пример

Следующий пример реализует веб-сервер с минимальными возможностями на основе модуля asyncore. В примере определяются два класса: asynhttp, принимающий соединения, и asynclient, обрабатывающий запросы клиентов. Сравните этот пример с примером, который приводился в разделе с описанием модуля asynchat. Основное отличие этого примера состоит в том, что он реализован на более низком уровне, из-за чего потребовалось побеспокоиться о таких проблемах, как разбиение потока входных данных на строки, буферизация данных и идентификация пустых строк, завершающих заголовки запросов.

```
# Асинхронный сервер HTTP
import asyncore, socket
import os
import mimetypes
import collections
try:
    from http.client import responses     # Python 3
```

```
except ImportError:
    from httplib import responses         # Python 2

# Следующий класс просто принимает входящие соединения
class async_http(asyncore.dispatcher):
    def __init__(self,port):
        asyncore.dispatcher.__init__(self)
        self.create_socket(socket.AF_INET,socket.SOCK_STREAM)
        self.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
        self.bind((‘’,port))
        self.listen(5)

    def handle_accept(self):
        client,addr = self.accept()
        return async_http_handler(client)

# Класс, обслуживающий клиентов
class async_http_handler(asyncore.dispatcher):
    def __init__(self, sock = None):
        asyncore.dispatcher.__init__(self,sock)
        self.got_request    = False               # Запрос HTTP прочитан?
        self.request_data   = b””
        self.write_queue    = collections.deque()
        self.responding     = False

    # Может использоваться для чтения, только если запрос еще не был прочитан
    def readable(self):
        return not self.got_request

    # Читает входящие данные запроса
    def handle_read(self):
        chunk = self.recv(8192)
        self.request_data += chunk
        if b’\r\n\r\n’ in self.request_data:
            self.handle_request()

    # Обрабатывает входящий запрос
    def handle_request(self):
        self.got_request = True
        header_data  = self.request_data[:self.request_data.find(b’\r\n\r\n’)]
        header_text  = header_data.decode(‘latin-1’)
        header_lines = header_text.splitlines()
        request      = header_lines[0].split()
        op           = request[0]
        url          = request[1][1:]
        self.process_request(op,url)

    # Обрабатывает запрос
    def process_request(self,op,url):
        self.responding = True
        if op == “GET”:
            if not os.path.exists(url):
                self.send_error(404,”File %s not found\r\n” % url)
            else:
                type, encoding = mimetypes.guess_type(url)
```

```
                size = os.path.getsize(url)
                self.push_text('HTTP/1.0 200 OK\r\n')
                self.push_text('Content-length: %d\r\n' % size)
                self.push_text('Content-type: %s\r\n' % type)
                self.push_text('\r\n')
                self.push(open(url,"rb").read())
            else:
                self.send_error(501,"%s method not implemented" % self.op)

        # Обработка ошибок
        def send_error(self,code,message):
            self.push_text('HTTP/1.0 %s %s\r\n' % (code, responses[code]))
            self.push_text('Content-type: text/plain\r\n')
            self.push_text('\r\n')
            self.push_text(message)

        # Добавляет двоичные данные в выходную очередь
        def push(self,data):
            self.write_queue.append(data)

        # Добавляет текстовые данные в выходную очередь
        def push_text(self,text):
            self.push(text.encode('latin-1'))

        # Готов для записи, только если ответ готов
        def writable(self):
            return self.responding and self.write_queue

        # Записывает данные ответа
        def handle_write(self):
            chunk = self.write_queue.popleft()
            bytes_sent = self.send(chunk)
            if bytes_sent != len(chunk):
                self.write_queue.appendleft(chunk[bytes_sent:])
            if not self.write_queue:
                self.close()

    # Создать сервер
    a = async_http(8080)
    # Запустить бесконечный цикл опроса
    asyncore.loop()
```

См. также

Описание модулей socket (стр. 586), select (стр. 572), SocketServer (стр. 611), пакета http (стр. 623).

Модуль select

Модуль `select` предоставляет доступ к системным вызовам `select()` и `poll()`. Системный вызов `select()` обычно используется для реализации опроса, или мультиплексирования, обработки нескольких потоков ввода-вывода, без использования потоков управления или дочерних процессов. В системах UNIX эти вызовы можно использовать для работы с файлами, сокета-

ми, каналами и со многими другими типами файлов. В Windows их можно использовать только для работы с сокетами.

`select(`*`iwtd`*`, `*`owtd`*`, `*`ewtd`*` [, `*`timeout`*`])`

Запрашивает информацию о готовности к вводу, выводу и о наличии исключений для группы дескрипторов файлов. В первых трех аргументах передаются списки с целочисленными дескрипторами файлов или с объектами, обладающими методом `fileno()`, который возвращает дескриптор файла. Аргумент *`iwtd`* определяет список объектов, которые проверяются на готовность к вводу, *`owtd`* – список объектов, которые проверяются на готовность к выводу, и *`ewtd`* – список объектов, которые проверяются на наличие исключительных ситуаций. В любом из аргументов допускается передавать пустой список. В аргументе *`timeout`* передается число с плавающей точкой, определяющее предельное время ожидания в секундах. При вызове без аргумента *`timeout`* функция ожидает, пока хотя бы один из дескрипторов не окажется в требуемом состоянии. Если в этом аргументе передать число `0`, функция просто выполнит опрос и тут же вернет управление. Возвращает кортеж списков с объектами, находящимися в требуемом состоянии. Эти списки включают подмножества объектов в первых трех аргументах. Если к моменту истечения предельного времени ожидания ни один из дескрипторов не находится в требуемом состоянии, возвращается три пустых списка. В случае ошибки возбуждается исключение `select.error`. В качестве значения исключения возвращается та же информация, что и в исключениях `IOError` и `OSError`.

`poll()`

Создает объект, выполняющий опрос с помощью системного вызова `poll()`. Эта функция доступна только в системах, поддерживающих системный вызов `poll()`.

Объект *`p`*, возвращаемый функцией `poll()`, поддерживает следующие методы:

`p``.register(`*`fd`*` [, `*`eventmask`*`])`

Регистрирует новый дескриптор файла *`fd`*. В аргументе *`fd`* может передаваться целочисленный дескриптор или объект, обладающий методом `fileno()`, с помощью которого можно получить дескриптор. В аргументе *`eventmask`* передается битная маска, составленная с помощью битовой операции ИЛИ из следующих флагов, которая определяет интересующие события:

Константа	Описание
`POLLIN`	Имеются данные, доступные для чтения.
`POLLPRI`	Имеются срочные данные, доступные для чтения.
`POLLOUT`	Готов к записи.
`POLLERR`	Ошибка.
`POLLHUP`	Разрыв соединения.
`POLLNVAL`	Недопустимый запрос.

При вызове функции без аргумента *eventmask* проверяются события POLLIN, POLLPRI и POLLOUT.

p.unregister(*fd*)

Удаляет дескриптор файла *fd* из объекта, выполняющего опрос. Возбуждает исключение KeyError, если дескриптор не был зарегистрирован ранее.

p.poll([*timeout*])

Проверяет наступление событий для всех зарегистрированных дескрипторов. Необязательный аргумент *timeout* определяет предельное время ожидания в миллисекундах. Возвращает список кортежей (*fd, event*), где поле *fd* является дескриптором файла, а поле *event* – битной маской, определяющей события. Поля этой битовой маски соответствуют константам POLLIN, POLLOUT и так далее. Например, чтобы проверить наличие события POLLIN, достаточно просто проверить значение выражение *event* & POLLIN на неравенство нулю. Если возвращается пустой список, это означает, что в течение указанного времени ожидания не возникло ни одного события.

Дополнительные возможности модуля

Функции select() и poll(), объявленные в этом модуле, совместимы со многими операционными системами. Кроме того, в системах Linux модуль select предоставляет интерфейс к механизму определения состояния по перепаду и по значению (epoll), который может обеспечить значительно более высокую производительность. В системах BSD предоставляется возможность доступа к очереди ядра и к объектам событий. Описание этих программных интерфейсов можно найти в электронной документации к модулю select, по адресу *http://docs.python.org/library/select*.

Усложненный пример использования асинхронного ввода-вывода

Иногда модуль select используется для реализации серверов, основанных на тасклетах и сопрограммах – механизмах многозадачности, в которых не используются потоки управления или процессы. Следующий пример иллюстрирует данную концепцию, реализуя диспетчер задач для сопрограмм, основанный на операциях ввода-вывода. Предупреждаю, что это самый сложный пример в книге и вам может потребоваться внимательно исследовать его, чтобы понять, как он работает. Возможно, вам также потребуется ознакомиться с моим учебным руководством «A Curious Course on Coroutines and Concurrency» (*http://www.dabeaz.com/coroutines*), где можно найти дополнительный справочный материал.

```
import select
import types
import collections

# Объект, представляющий запущенную задачу
class Task(object):
    def __init__(self,target):
        self.target  = target # Сопрограмма
```

```
        self.sendval = None   # Значение, которое передается при возобновлении
        self.stack   = []     # Стек вызовов

    def run(self):
        try:
            result = self.target.send(self.sendval)
            if isinstance(result,SystemCall):
                return result
            if isinstance(result,types.GeneratorType):
                self.stack.append(self.target)
                self.sendval = None
                self.target = result
            else:
                if not self.stack: return
                self.sendval = result
                self.target = self.stack.pop()
        except StopIteration:
            if not self.stack: raise
            self.sendval = None
            self.target = self.stack.pop()

# Объект, представляющий "системный вызов"
class SystemCall(object):
    def handle(self,sched,task):
        pass

# Объект диспетчера задач
class Scheduler(object):
    def __init__(self):
        self.task_queue    = collections.deque()
        self.read_waiting  = {}
        self.write_waiting = {}
        self.numtasks      = 0

    # Создает новую задачу из сопрограммы
    def new(self,target):
        newtask = Task(target)
        self.schedule(newtask)
        self.numtasks += 1

    # Добавляет задачу в очередь задач
    def schedule(self,task):
        self.task_queue.append(task)

    # Приостанавливает задачу, пока дескриптор файла не станет
    # доступным для чтения
    def readwait(self,task,fd):
        self.read_waiting[fd] = task

    # Приостанавливает задачу, пока дескриптор файла не станет
    # доступным для записи
    def writewait(self,task,fd):
        self.write_waiting[fd] = task

    # Главный цикл диспетчера задач
    def mainloop(self,count=-1,timeout=None):
```

```
        while self.numtasks:
            # Проверить наличие событий ввода-вывода
            if self.read_waiting or self.write_waiting:
                wait = 0 if self.task_queue else timeout
                r,w,e = select.select(self.read_waiting, self.write_waiting,
                                      [], wait)
                for fileno in r:
                    self.schedule(self.read_waiting.pop(fileno))
                for fileno in w:
                    self.schedule(self.write_waiting.pop(fileno))

            # Запустить все задачи, имеющиеся в очереди,
            # которые готовы к запуску
            while self.task_queue:
                task = self.task_queue.popleft()
                try:
                    result = task.run()
                    if isinstance(result,SystemCall):
                        result.handle(self,task)
                    else:
                        self.schedule(task)
                except StopIteration:
                    self.numtasks -= 1

            # Если нет задач, готовых для запуска,
            # требуется решить - продолжать или выйти
            else:
                if count > 0: count -= 1
                if count == 0:
                    return

# Реализация различных системных вызовов
class ReadWait(SystemCall):
    def __init__(self,f):
        self.f = f
    def handle(self,sched,task):
        fileno = self.f.fileno()
        sched.readwait(task,fileno)

class WriteWait(SystemCall):
    def __init__(self,f):
        self.f = f
    def handle(self,sched,task):
        fileno = self.f.fileno()
        sched.writewait(task,fileno)

class NewTask(SystemCall):
    def __init__(self,target):
        self.target = target
    def handle(self,sched,task):
        sched.new(self.target)
        sched.schedule(task)
```

Программный код этого примера реализует «операционную систему» в миниатюре. Ниже коротко описывается, как он действует:

- Вся основная работа выполняется сопрограммами. Сопрограммы, как и генераторы, используют инструкцию `yield`, но в отличие от генераторов, в сопрограммах она используется не только чтобы возвращать значения, но и чтобы принимать значения, которые передаются с помощью метода `send(`*`value`*`)`.
- Класс `Task` представляет готовую к запуску задачу и является всего лишь тонкой оберткой вокруг сопрограммы. Объект *`task`* класса `Task` может выполнять единственную операцию — *`task`*`.run()`. Этот метод возобновляет выполнение задачи, которая продолжает работать, пока не встретит следующую инструкцию `yield`, после чего выполнение задачи приостанавливается. После запуска задачи атрибут *`task`*`.sendval` содержит значение, которое должно быть послано соответствующему выражению `yield` в задаче. Задачи продолжают выполняться, пока не будет встречена следующая инструкция `yield`. Значение, которое воспроизводится этой инструкцией, определяет, что будет происходить дальше внутри задачи:
  - Если возвращаемым значением является другая сопрограмма (`type.GeneratorType`), это означает, что задача временно передает управление другой сопрограмме. Атрибут `stack` объекта класса `Task` представляет стек вызовов сопрограмм, в котором сохраняется прежняя сопрограмма. При следующем запуске задачи управление будет передано новой сопрограмме.
  - Если возвращаемым значением является экземпляр класса `SystemCall`, это означает, что задаче требуется, чтобы диспетчер выполнил некоторую операцию от своего имени (например, запустил бы новую задачу, приостановил бы выполнение задачи, пока не появится возможность выполнить операцию ввода-вывода, и так далее). Назначение этого объекта описывается чуть ниже.
  - Если возвращается какое-либо другое значение, это может означать одно из двух: либо управление вернула сопрограмма, запущенная из задачи, либо управление вернула сама задача. Если стек вызовов не пуст, это означает, что произошел возврат из сопрограммы, запущенной из задачи, поэтому вызывающая задача выталкивается из стека вызовов, а возвращаемое значение сохраняется, чтобы его можно было передать вызывающей сопрограмме. Вызывающая сопрограмма получит это значение, когда будет запущена в следующий раз. Если стек вызовов пуст, возвращаемое значение просто уничтожается.
  - Исключение `StopIteration` означает, что сопрограмма завершила свою работу. Когда появляется это исключение, управление передается предыдущей сопрограмме, сохраненной в стеке вызовов (если таковая имеется), либо исключение будет передано диспетчеру и в этом случае оно будет интерпретироваться, как признак завершения работы сопрограммы.
- Объект класса `SystemCall` представляет системный вызов. Когда работающей задаче требуется сообщить диспетчеру, что он должен выполнить операцию от своего имени, она с помощью инструкции `yield` возвращает

экземпляр класса SystemCall. Этот объект называется «системным вызовом», потому что он имитирует способ обращения к системным службам в настоящих многозадачных операционных системах, таких как UNIX или Windows. В частности, когда программе требуется обратиться к службе операционной системы, она передает управление системному вызову и предоставляет некоторую дополнительную информацию, необходимую для выполнения операции. В этом смысле возврат объекта класса SystemCall напоминает вызов системной «ловушки».

- Объект класса Scheduler представляет коллекцию объектов класса Task, которыми он управляет. Вся основная работа диспетчера построена вокруг очереди задач (атрибут task_queue), в которой хранятся задачи, готовые к запуску. Над очередью задач выполняются четыре основных операции. Метод new() принимает новую задачу, заворачивает ее в объект класса Task и помещает созданный объект в очередь. Метод schedule() получает существующий объект класса Task и вставляет его обратно в очередь. Метод mainloop() запускает цикл диспетчера, который обрабатывает задачи одну за другой, пока в очереди не останется задач. Методы readwait() и writewait() помещают объект класса Task во временную область ожидания, где он остается до появления ожидаемого события ввода-вывода. В этом случае работа задачи приостанавливается, но она не уничтожается, а просто бездействует в ожидании.
- Метод mainloop() является основой диспетчера задач. В самом начале этот метод проверяет, имеются ли задачи, ожидающие событий ввода-вывода. Если такие задачи имеются, диспетчер подготавливает и вызывает функцию select(), чтобы проверить наличие событий ввода-вывода. Если имеются какие-либо события, представляющие интерес, соответствующие задачи возвращаются обратно в очередь задач, чтобы появилась возможность запустить их. Затем метод mainloop() выталкивает очередную задачу из очереди и вызывает ее метод run(). Если какая-либо задача завершает работу (возбуждает исключение StopIteration), она уничтожается. Если задача просто возвращает управление, она опять добавляется в очередь задач, чтобы обеспечить возможность ее запуска в следующем цикле. Так продолжается до тех пор, пока либо не опустеет очередь задач, либо все задачи не окажутся приостановленными в ожидании событий ввода-вывода. Метод mainloop() может принимать дополнительный аргумент count, позволяющий обеспечить завершение работы метода после указанного количества операций опроса. Это может пригодиться, когда диспетчер встраивается в другой цикл обработки событий.
- Самым сложным аспектом диспетчера является обработка экземпляров класса SystemCall в методе mainloop(). Когда задача возвращает экземпляр класса SystemCall, диспетчер вызывает его метод handle(), передавая в качестве аргументов соответствующие экземпляры классов Scheduler и Task. Назначение системного вызова состоит в том, чтобы выполнить некоторую внутреннюю операцию, необходимую задаче или диспетчеру. Классы ReadWait(), WriteWait() и NewTask() являются примерами системных вызовов, которые приостанавливают выполнение задачи на время операции ввода-вывода или создают новую задачу. Например,

конструктор `ReadWait()` принимает задачу и вызывает метод `readwait()` диспетчера. После этого диспетчер принимает задачу и помещает ее в соответствующую область ожидания. Здесь можно видеть соблюдение очень важного принципа разделения объектов. Задачи возвращают объекты класса `SystemCall` службе обработки запросов, но они не вступают в прямое взаимодействие с диспетчером. Объекты класса `SystemCall`, в свою очередь, могут выполнять операции над задачами или диспетчерами, но они никак не привязаны к конкретной реализации диспетчера или задачи. Благодаря этому теоретически имеется возможность создавать совершенно разные реализации диспетчера (возможно даже с использование потоков управления), которые могли бы просто вставляться в существующий фреймворк и он при этом продолжал бы работать.

Ниже приводится пример простого сетевого сервера времени, реализованного с помощью данного диспетчера задач. Он поможет понять многое из того, о чем говорилось в предыдущем списке:

```
from socket import socket, AF_INET, SOCK_STREAM
def time_server(address):
    import time
    s = socket(AF_INET,SOCK_STREAM)
    s.bind(address)
    s.listen(5)
    while True:
        yield ReadWait(s)
        conn,addr = s.accept()
        print("Получен запрос на соединение с %s" % str(addr))
        yield WriteWait(conn)
        resp = time.ctime() + "\r\n"
        conn.send(resp.encode('latin-1'))
        conn.close()

sched = Scheduler()
sched.new(time_server(('',10000)))      # Сервер на порту 10000
sched.new(time_server(('',11000)))      # Сервер на порту 11000
sched.run()
```

В этом примере параллельно запускаются два сервера – каждый из них использует свой номер порта для приема соединений (это легко проверить с помощью утилиты `telnet`). Инструкции `yield ReadWait()` и `yield WriteWait()` вызывают приостановку сопрограммы каждого из серверов до момента, пока не появится возможность выполнить операцию ввода-вывода над соответствующим сокетом. Когда эти инструкции возвращают управление сопрограмме, немедленно выполняется соответствующая операция ввода-вывода, такая как `accept()` или `send()`.

Конструкции `ReadWait` и `WriteWait` могут показаться слишком низкоуровневыми. К счастью, архитектура приложения позволяет скрыть эти операции за кулисами библиотечных функций и методов, при условии, что они также будут являться сопрограммами. Взгляните на следующий объект, который служит оберткой вокруг объекта и имитирует его интерфейс:

```
class CoSocket(object):
    def __init__(self,sock):
```

```
        self.sock = sock
    def close(self):
        yield self.sock.close()
    def bind(self,addr):
        yield self.sock.bind(addr)
    def listen(self,backlog):
        yield self.sock.listen(backlog)
    def connect(self,addr):
        yield WriteWait(self.sock)
        yield self.sock.connect(addr)
    def accept(self):
        yield ReadWait(self.sock)
        conn, addr = self.sock.accept()
        yield CoSocket(conn), addr
    def send(self,bytes):
        while bytes:
            evt = yield WriteWait(self.sock)
            nsent = self.sock.send(bytes)
            bytes = bytes[nsent:]
    def recv(self,maxsize):
        yield ReadWait(self.sock)
        yield self.sock.recv(maxsize)
```

Ниже приводится реализация сервера времени, основанная на применении класса `CoSocket`:

```
from socket import socket, AF_INET, SOCK_STREAM
def time_server(address):
    import time
    s = CoSocket(socket(AF_INET,SOCK_STREAM))
    yield s.bind(address)
    yield s.listen(5)
    while True:
        conn,addr = yield s.accept()
        print(conn)
        print("Получен запрос на соединение с %s" % str(addr))
        resp = time.ctime()+"\r\n"
        yield conn.send(resp.encode('latin-1'))
        yield conn.close()

sched = Scheduler()
sched.new(time_server(('',10000)))   # Сервер на порту 10000
sched.new(time_server(('',11000)))   # Сервер на порту 11000
sched.run()
```

В этом примере программный интерфейс объекта класса `CoSocket` выглядит, как интерфейс обычного сокета. Единственное отличие состоит в том, что каждая операция должна предваряться инструкцией `yield` (так как все методы определены, как сопрограммы). На первый взгляд все это выглядит так странно, что заставляет задаться вопросом: а есть ли в этом какой-то смысл? Если запустить сервер, который приводится выше, можно заметить, что одновременно может выполняться несколько его копий без применения потоков управления или дочерних процессов. При этом про-

грамма выглядит, как «обычный» поток управления, если игнорировать все инструкции `yield`.

Ниже приводится пример асинхронного веб-сервера, который одновременно обслуживает несколько соединений с клиентами, но не использует при этом функции обратного вызова, потоки управления или процессы. Сравните его с примерами, реализованными на основе применения модулей `asynchat` и `asyncore`.

```
import os
import mimetypes
try:
    from http.client import responses   # Python 3
except ImportError:
    from httplib import responses       # Python 2
from socket import *

def http_server(address):
    s = CoSocket(socket(AF_INET,SOCK_STREAM))
    yield s.bind(address)
    yield s.listen(50)

    while True:
        conn,addr = yield s.accept()
        yield NewTask(http_request(conn,addr))
        del conn, addr

def http_request(conn,addr):
    request = b""
    while True:
        data = yield conn.recv(8192)
        request += data
        if b'\r\n\r\n' in request: break

    header_data    = request[:request.find(b'\r\n\r\n')]
    header_text    = header_data.decode('latin-1')
    header_lines   = header_text.splitlines()
    method, url, proto = header_lines[0].split()
    if method == 'GET':
        if os.path.exists(url[1:]):
            yield serve_file(conn,url[1:])
        else:
            yield error_response(conn,404,"File %s not found" % url)
    else:
        yield error_response(conn,501,"%s method not implemented" % method)
    yield conn.close()

def serve_file(conn,filename):
    content,encoding = mimetypes.guess_type(filename)
    yield conn.send(b"HTTP/1.0 200 OK\r\n")
    yield conn.send(("Content-type: %s\r\n" % content).encode('latin-1'))
    yield conn.send(("Content-length: %d\r\n" %
                     os.path.getsize(filename)).encode('latin-1'))
    yield conn.send(b"\r\n")
    f = open(filename,"rb")
```

```
        while True:
            data = f.read(8192)
            if not data: break
            yield conn.send(data)

    def error_response(conn,code,message):
        yield conn.send(("HTTP/1.0 %d %s\r\n" %
                         (code, responses[code])).encode('latin-1'))
        yield conn.send(b"Content-type: text/plain\r\n")
        yield conn.send(b"\r\n")
        yield conn.send(message.encode('latin-1'))

    sched = Scheduler()
    sched.new(http_server(('',8080)))
    sched.mainloop()
```

Внимательное изучение этого примера позволит надежно усвоить особенности приемов разработки многозадачных программ с применением сопрограмм, которые используются некоторыми весьма сложными сторонними модулями. Однако чрезмерное употребление этих приемов может привести к тому, что вас уволят с работы после очередной инспекции вашего программного кода.

Когда имеет смысл использовать асинхронные операции при работе с сетью

Использование асинхронных операций ввода-вывода (модули `asyncore` и `asynchat`), операций опроса и сопрограмм, как это демонстрировалось в предыдущих примерах, остается одним из самых таинственных аспектов программирования на языке Python. И все же эти приемы используются намного чаще, чем можно было бы подумать. Одной из часто упоминаемых причин использования асинхронных операций ввода-вывода является минимизация нагрузки, связанной с большим количеством потоков управления, особенно когда возникает необходимость управлять множеством клиентов при наличии ограничений, связанных с глобальной блокировкой интерпретатора (см. главу 20 «Потоки и многозадачность»).

Исторически модуль `asyncore` был одним из первых модулей в стандартной библиотеке, обеспечившим поддержку асинхронного ввода-вывода. Модуль `asynchat` появился немного позже с целью упростить использование модуля `asyncore`. Оба эти модуля используют подход, основанный на обработке событий ввода-вывода. Например, когда возникает событие ввода-вывода, вызывается функция обратного вызова. Функция обратного вызова реагирует на событие ввода-вывода и выполняет некоторую обработку данных. Если вам придется создавать крупные приложения в таком стиле, вы обнаружите, что обработка событий пронизывает практически все части приложения (например, события ввода-вывода приводят к вызову функций-обработчиков, которые в свою очередь вызывают другие функции-обработчики, и так до бесконечности). Этот подход используется в пакете Twisted (*http://twistedmatrix.com*), одном из наиболее популярных пакетов, предназначенных для разработки сетевых приложений.

Более современные решения опираются на применение сопрограмм, но они сложнее и используются реже, так как сопрограммы впервые появились в версии Python 2.5. Одна из важнейших особенностей сопрограмм состоит в том, что они позволяют писать приложения, которые больше походят на многопоточные программы. Например, в примерах реализации вебсервера не используются функции обратного вызова, и они выглядят очень похожими на программы, основанные на использовании потоков управления, – нужно просто привыкнуть к использованию инструкции `yield`. В интерпретаторе Python, не использующем стек вызовов (*http://www.stackless.com*), эта идея развита еще дальше.

Вообще говоря, в большинстве сетевых приложений не следует стремиться использовать приемы асинхронного ввода-вывода. Например, если потребуется создать сервер, который постоянно отправляет данные через сотни или даже тысячи соединений одновременно, применение методики, основанной на создании множества потоков управления, может обеспечить более высокую производительность. Это обусловлено тем, что производительность функции `select()` снижается тем больше, чем больше число соединений, которые нужно отслеживать. В операционной системе Linux этот недостаток можно устранить за счет использования специальных функций, таких как `epoll()`, но это ограничивает переносимость программного кода. Пожалуй, самую большую выгоду от использования асинхронного ввода-вывода можно получить в приложениях, где сетевые операции необходимо интегрировать в другие циклы событий (например, в цикл событий графического интерфейса) или где сетевые операции приходится добавлять в программный код, который выполняет массивные вычисления. В подобных ситуациях использование асинхронного ввода-вывода способно уменьшить время отклика.

Исключительно в целях демонстрации ниже приводится программа, выполняющая задачу, о которой поется в песне «10 миллионов бутылок пива на стене»:

```
bottles = 10000000

def drink_beer():
    remaining = 12.0
    while remaining > 0.0:
        remaining -= 0.1

def drink_bottles():
    global bottles
    while bottles > 0:
        drink_beer()
        bottles -= 1
```

Теперь предположим, что необходимо добавить возможность удаленного мониторинга, которая позволяла бы клиентам подключаться и следить за тем, сколько бутылок осталось. Один из вариантов состоит в том, чтобы запустить сервер в отдельном потоке управления, заставив его выполняться параллельно с основным приложением, как показано ниже:

```
def server(port):
    s = socket.socket(socket.AF_INET,socket.SOCK_STREAM)
    s.bind(('',port))
    s.listen(5)
    while True:
        client,addr = s.accept()
        client.send(("%d bottles\r\n" % bottles).encode('latin-1')
        client.close()

# Запуск сервера мониторинга
thr = threading.Thread(target=server,args=(10000,))
thr.daemon=True
thr.start()
drink_bottles()
```

Другой вариант состоит в том, чтобы реализовать сервер на основе опроса каналов ввода-вывода и внедрить операцию опроса непосредственно в главный цикл вычислений. Ниже приводится пример, в котором используется диспетчер сопрограмм, разработанный выше:

```
def drink_bottles():
    global bottles
    while bottles > 0:
        drink_beer()
        bottles -= 1
        scheduler.mainloop(count=1,timeout=0) # Проверить наличие соединений

# Асинхронный сервер, основанный на сопрограммах.
def server(port):
    s = CoSocket(socket.socket(socket.AF_INET,socket.SOCK_STREAM))
    yield s.bind(('',port))
    yield s.listen(5)
    while True:
        client,addr = yield s.accept()
        yield client.send(("%d bottles\r\n" % bottles).encode('latin-1')
        yield client.close()

scheduler = Scheduler()
scheduler.new(server(10000))
drink_bottles()
```

Если написать отдельную программу, которая периодически будет выполнять подключение к программе, имитирующей опустошение бутылок пива, и измерять время, необходимое на получение информации о количестве оставшихся бутылок, результаты могут оказаться весьма удивительными. На компьютере автора (макбук с двухъядерным процессором, работающим на частоте 2 ГГц) среднее время отклика сервера (измерения проводились по 1000 запросов), основанного на сопрограмме, составило примерно 1 миллисекунду, против 5 миллисекунд для сервера, основанного на потоках управления. Такая разница объясняется тем, что реализация, основанная на сопрограмме, способна отвечать сразу же, как только обнаруживает попытку установить соединение, тогда как многопоточный сервер не может быть запущен, пока не будет запланирован на выполнение

операционной системой. Учитывая наличие потока управления, выполняющего массивные вычисления, и глобальной блокировки интерпретатора, сервер вынужден простаивать, пока вычислительный поток управления не исчерпает выделенный ему квант времени. Во многих системах величина кванта времени составляет примерно 10 миллисекунд, то есть вышеупомянутое время отклика многопоточного сервера точно соответствует среднему арифметическому значению времени ожидания переключения вычислительного потока операционной системой.

Недостатком периодического опроса является чрезмерная нагрузка, если его выполнять слишком часто. Например, хотя время отклика в примере с опросом оказалось меньше, общее время выполнения программы увеличилось более чем на 50%. Если изменить реализацию так, что опрос будет проводиться только через каждые шесть бутылок пива, время отклика увеличится весьма незначительно и составит 1,2 миллисекунды, тогда как время выполнения программы будет всего на 3% больше, чем время выполнения программы без опроса. К сожалению, часто не бывает иного способа четко определить, как часто следует выполнять опрос, кроме как выполнив измерение производительности приложения.

Несмотря на то что уменьшение времени отклика кажется победой, реализация собственного механизма многозадачности может породить неприятные проблемы. Например, в задачах необходимо с особой осторожностью подходить к выполнению любых операций, которые могут вызывать приостановку процесса. В примере с веб-сервером имеется фрагмент программного кода, который открывает файл и читает из него данные. Эта операция может приостановить выполнение всей программы на достаточно продолжительный промежуток времени, если доступ к файлу будет сопряжен с необходимостью перемещения головок жесткого диска. Единственный способ устранить эту проблему состоит в том, чтобы дополнительно реализовать асинхронный доступ к файлу и добавить эту особенность в диспетчер задач. Для более сложных операций, таких как выполнение запросов к базе данных, определить, как реализовать асинхронный доступ, может оказаться намного сложнее. Один из возможных способов реализации таких операций – вынести их в отдельный поток управления и организовать обмен результатами с диспетчером задач по мере их поступления, что можно осуществить с помощью очередей сообщений. В некоторых системах существуют низкоуровневые системные вызовы, выполняющие асинхронные операции ввода-вывода (например, семейство функций `aio_*` в UNIX). К моменту написания этих строк в стандартной библиотеке языка Python еще не был реализован доступ к этим функциям, однако вы можете попробовать поискать сторонние модули, обеспечивающие такую возможность. По опыту автора, использование подобных функциональных возможностей намного сложнее, чем выглядит, а выигрыш от их добавления в программу на языке Python не стоит затрачиваемых усилий – часто в таких ситуациях более предпочтительно бывает использовать библиотеку для работы с потоками управления.