

КЛИЕНТ/СЕРВЕРНЫЕ СЕТЕВЫЕ ПРИЛОЖЕНИЯ НА PYTHON — ЗАПИСКИ ДИЛЕТАНТА (ЧАСТЬ I)

Иванов А.В., ©19 мая 2010 г.¹
aivanov(злая собака)keldysh(жирная точка)ru
Институт прикладной математики им. М.В. Келдыша РАН

Содержание

1	Введение	1
2	Простейший пример клиент/серверного приложения	2
3	Протокол высокого уровня для пересылки «консервированных» данных и вывод логов	4
4	Передача исключений	7
5	Многопоточный сервер и удалённый доступ к экземплярам класса пользователя	10
6	Заключение	20

Программист в тире расстреливает магазин за магазином, но никак не может попасть в мишень. Тогда он прижимает ствол к ладони и нажимает на спуск — выстрел, летят клочья мяса. Задумчиво глядя через обильно кровоточащую дыру в ладони на мишень программист говорит: «Отсюда все уходит, проблема с той стороны...»

Анекдот.

1 Введение

Для меня, как человека не получившего специализированного программистского образования, сети и клиент/серверные приложения казались раньше чем то архисложным. Но начав писать на Python, и прочитав Марка Лутца [2], я обнаружил, что это не такие уж и страшные вещи.

¹Последняя правка 29 Июнь 2011 11:48

Через некоторое время у меня накопилось несколько решённых задач (чаты, система тестирования учащихся, система управления расчётами на кластере и проч.), в которых с самого начала прослеживалось определённое сходство, некоторые куски кода просто повторялись. В итоге оглянувшись назад, я вдруг понял, что эпизодически (новое приложение с клиент/серверной архитектурой мне приходится писать примерно раз в год—два) делаю одно и то же, и решил оформить повторяющиеся куски кода в библиотеку.

Параллельно родился этот текст. Он задумывался не столько как описание библиотеки (лучшее описание любой библиотеки это ИМНО её исходный код снабжённый краткими комментариями), сколько как описание некоторых интересных, отчасти специфичных для Python, приёмов работы с сокетами, клиентами и серверами, и как некоторое учебное пособие. Насколько эта попытка была удачна судить Вам.

2 Простейший пример клиент/серверного приложения

Не будем далеко ходить и повторяться. Что такое сокеты и с чем их едят вообще написано масса книг и статей. Как с ними работают конкретно на Python описано у Гвидо Ван Россума [1], более подробно у Марка Лутца [2] и совсем-совсем подробно у Дэвида Бизли [3]. Я приведу лишь краткий пример и сделаю пару замечаний.

Итак, мы сейчас напишем эхо-сервер. Это такое серверное приложение, которое считывает данные у клиента (строку) и возвращает такую же строку (или такую же строку + такую же строку). Вещь на первый взгляд бессмысленная, но наглядная. Сервер выглядит так:

file examples/server0.py:

```
-----  
1 #!/usr/bin/python  
2 from socket import *  
  
3 s = socket(AF_INET,SOCK_STREAM)  
4 s.setsockopt(SOL_SOCKET, SO_REUSEADDR, 1)  
5 s.bind(('127.0.0.1',1234))  
6 s.listen(5)  
  
7 while 1 :  
8     conn, addr = s.accept()  
9     data = conn.recv(1024)  
10    conn.send(data+data)  
-----
```

В первой строке мы указываем, что данный файл должен запускаться интерпретатором Python. Во второй импортируем стандартный модуль `socket`. В третьей создаем объект сетевого соединения.

В четвертой устанавливаем параметр соединения `REUSEADDR` (см. [4] стр. 236 или [3] стр. 598). Это не обязательно, но если запускать сервер несколько раз друг за другом, то возможно придется ждать освобождения порта между запусками (хотя в нашем примере сокет

и закрывается корректно по деструктору, но порт может быть занят после закрытия еще некоторое время).

В пятой строке связываем сокет с интерфейсом 127.0.0.1, порт 1234 и настраиваем в шестой.

Дальше в бесконечном цикле мы ждем соединения в строке 7; читаем из сокета данные (не более килобайта) в строке 8; в строке 9 возвращаем то что прочитали и снова то что прочитали; закрываем сокет и снова ждем.

Клиент выглядит следующим образом:

file examples/client0.py:

```
-----  
1 #!/usr/bin/python  
2 from socket import *  
3 import sys  
4 s = socket(AF_INET,SOCK_STREAM)  
5 s.connect(('127.0.0.1',1234))  
6 s.send( ' '.join(sys.argv[1:]) )  
7 print s.recv(2048)  
-----
```

В четвертой строке мы создаем объект соединения, в пятой открываем сокет. В шестой строке прописываем в соединение аргументы командной строки клиента. В седьмой строке читаем ответ (не более двух килобайт) и печатаем то что прочитали. Соединение можно не закрывать — оно будет закрыто при вызове деструктора экземпляра класса `socket`, который будет вызван при выходе из программы.

Попробуем запустить:

```
-----  
1 examples> ./server0.py &  
2 examples> ./client0.py qwerty  
3 qwertyqwerty  
4 examples> fg  
5 ^C  
-----
```

При запуске сначала запускаем сервер, символ `'&'` указывает, что сервер будет запущен в фоновом режиме. Затем запускаем клиент с какими то аргументами командной строки. Затем вызываем сервер из фонового режима при помощи команды `fg` и закрываем при помощи комбинации `Ctrl-C`.

Следует отметить, что в подавляющем большинстве клиент/серверных приложений ведущим является клиент, т.е. сервер лишь реагирует на клиентские запросы. Для обслуживания нескольких клиентов обычно сервер делается многопоточным.

3 Протокол высокого уровня для пересылки «консервированных» данных и вывод логов

Итак, мы имеем возможность передавать данные (строки) с клиента на сервер и обратно. Но есть две проблемы. Во первых, желательно знать длину передаваемого сообщения (если сообщение не влезет в буфер часть данных будет потерянна). Во вторых, как правило необходимо передавать структурированные данные.

Первая проблема может быть решена при помощи отправки сообщения состоящего из двух частей. Первая часть заранее известной длины (например восемь байт) содержит в текстовом виде длину сообщения, вторая содержит собственно само сообщение. При чтении данных следует учитывать, что длинная строка может быть прочитана за один раз не полностью (это зависит от деталей реализации модуля `socket` на различных платформах, проблемы как обычно начинаются при переходе под `MS Windows`), поэтому читать надо до тех пор, пока не будет прочитано нужной количество байт. Для отправки/приёма строк напомним функции `send_string` и `recv_string`.

Для перевода структурированных данных в строку (т.н. сериализации) и обратно существует много различных решений. Мы используем стандартный для `Python` модуль `pickle`, точнее `cPickle` (реализацию на `C` с производительностью в некоторых случаях на три порядка выше, [1] раздел 20.1 стр. 258). Этот модуль использует свой собственный формат, который тем не менее является стандартным для `Python`. Модуль корректно сериализует любые конструкции из встроенных типов, в том числе конструкции содержащие циклические ссылки. Модуль **не** сериализует файловые объекты, лямбда-функции, и должен с большой осторожностью использоваться для сериализации функций и пользовательских экземпляров класса — при обратной сериализации происходит автоматическая загрузка пользовательских модулей, содержащих сериализованные функции и классы (причём модуль идентифицируется исключительно по имени), что может приводить к ошибкам и в некоторых случаях даже создавать угрозу безопасности.

Для отправки/приёма структурированных данных создадим функции `send` и `recv`.

file `examples/mysocket1.py`:

```
-----
1 from socket import *
2 import sys, cPickle, pprint
3 #-----
4 class SocketClosed(Exception) : pass
5 LOG = [ 0, sys.stderr ] #loglevel, logfile
6 #-----
7 def send_string( connect, string ) :
8     connect.send( '%08i'%len(string) )
9     connect.send( string )
10    if LOG[0] : print>>LOG[1], 'SEND %s BYTES TO %s'%( len(string),
connect.getpeername() )
11    if LOG[0] == 2 : print>>LOG[1], string
12    if LOG[0] : LOG[1].flush()
13 #-----
```

```

14 def recv_string( connect ) :
15     try : length = int( connect.recv(8) )
16     except : raise SocketClosed()
17     string = connect.recv( length )
18     while len(string) < length : string += connect.recv( length -
len(string) )
19     if LOG[0] : print>>LOG[1], 'RECV %s BYTES FROM %s'%( length,
connect.getpeername() )
20     if LOG[0] == 2 : print>>LOG[1], string
21     if LOG[0] : LOG[1].flush()
22     return string
23 #-----
24 def send( connect, data ) :
25     if data.__class__ == str : send_string( connect, 'S'+data )
26     else : send_string( connect, 'P'+cPickle.dumps( data ) )
27     if LOG[0] == 3 : pprint.pprint( data, LOG[1] ); LOG[1].flush()
28 #-----
29 def recv( connect ) :
30     string = recv_string( connect )
31     if string.startswith('S') : data = string[1:]
32     elif string.startswith('P') : data = cPickle.loads( string[1:] )
33     if LOG[0] == 3 : pprint.pprint( data, LOG[1] ); LOG[1].flush()
34     return data

```

Заодно, раз уж мы стали создавать библиотеку, добавим возможность выводить логи. Для этого нам послужит переменная `LOG` — список из двух элементов, содержащий уровень детализации логов (0 — не выводить логи, 1 — выводить лишь адреса и длины пересылаемых сообщений, 2 — выводить содержимое сообщений, 3 — выводить структурированные данные) и файловый объект, в который производится вывод.

Для того, что бы не «консервировать» строки, будем предварять их при передаче буквой `S`, структурированные данные будем предварять буквой `P`.

Этот скромный модуль существенно расширяет наши возможности и решает целую кучу проблем.² Давайте напишем сервер, умножающий все на два:

file examples/server1.py:

```

1 #!/usr/bin/python
2 from mysocket1 import *
3 #-----
4 s = socket(AF_INET, SOCK_STREAM)

```

²До конца оценить всю прелесть возможности пересылать произвольные структурированные данные может лишь человек, пытавшийся решить эту проблему традиционным путём — т.е. придумавший протоколы обмена для конкретной задачи, реализовавший эти протоколы, и потом месяцами вылавливавший ошибки реализации.

```

5 s.setsockopt(SOL_SOCKET, SO_REUSEADDR, 1)
6 s.bind(('127.0.0.1',1234))
7 s.listen(5)
8 #-----
9 while 1 :
10     conn, addr = s.accept()
11     t = recv( conn )
12     send( conn, t*2 )
13     conn.close()

```

Напишем простой клиент (для наглядности будем выводить перед данными, полученными с сервера, несколько звездочек):

file examples/client1.py:

```

1 #!/usr/bin/python -u
2 from mysocket1 import *
3 LOG[0]=3
4 #-----
5 s = socket(AF_INET,SOCK_STREAM)
6 s.connect(('127.0.0.1',1234))
7 send( s, 'qwe' )
8 print '***** %s'%recv(s)
9 s.close()
10 #-----
11 s = socket(AF_INET,SOCK_STREAM)
12 s.connect(('127.0.0.1',1234))
13 send( s, 23 )
14 print '***** %s'%recv(s)
15 s.close()

```

Если Вы будете запускать пример из терминала так, как это было показано выше, Вы должны увидеть следующий вывод:

```

1 examples> ./server1.py &
2 examples> ./client1.py
3 SEND 4 BYTES TO ('127.0.0.1', 1234)
4 'qwe'
5 RECV 7 BYTES FROM ('127.0.0.1', 1234)
6 'qweqwe'
7 ***** qweqwe

```

```
8 SEND 6 BYTES TO ('127.0.0.1', 1234)
9 23
10 RECV 6 BYTES FROM ('127.0.0.1', 1234)
11 46
12 ***** 46
13 examples> fg
14 ^C
```

4 Передача исключений

Отладка клиент–серверных приложений обычно является настоящим кошмаром, особенно если сервер многопоточный. Представьте себе, что сервер «умножающий все на два» получил какие то данные, которые не могут быть умножены (например словарь). В этом случае на сервере будет возбуждено исключение, и клиент просто повиснет в ожидании данных. До некоторой степени здесь может помочь просмотр логов, но если сервер многопоточный это мало что дает. Поскольку ведущим как правило является клиент, идеальным решением является передача возникшего на сервере исключения клиенту вместе со стеком исключения, и раз мы можем передавать любые структурированные данные, это для нас не является большой проблемой.

Во первых необходимо сформировать отчёт об исключении на сервере. Для этого достаточно временно подсунуть в качестве `sys.stderr` экземпляр класса `_mybuf`, эмулирующий файловый объект открытый для записи, и вызвать функцию `sys.excepthook()`. Функция `send_exception()` передает законсервированный кортеж, состоящий из экземпляра класса исключения и стека исключения полученного через экземпляр `_mybuf`, полученная строка предваряется буквой `E`.

Функция `recv` при получении исключения сохраняет стек исключения в `sys.server_stack`, и возбуждает исключение повторно при помощи `raise`. Функция `sys.excepthook` переопределена так, что при выводе стека исключения добавляется стек, содержащийся в `sys.server_stack`:

file examples/mysocket2.py:

```
1 from socket import *
2 import sys, cPickle, pprint
3 #-----
4 class SocketClosed(Exception) : pass
5 LOG = [ 0, sys.stderr ] #loglevel, logfile
6 #-----
7 def send_string( connect, string ) :
8     connect.send( '%08i'%len(string) )
9     connect.send( string )
10    if LOG[0] : print>>LOG[1], 'SEND %s BYTES TO %s'%( len(string),
connect.getpeername() )
11    if LOG[0] == 2 : print>>LOG[1], string
12    if LOG[0] : LOG[1].flush()
```

```

13 #-----
14 def recv_string( connect ) :
15     try : length = int( connect.recv(8) )
16     except : raise SocketClosed()
17     string = connect.recv( length )
18     while len(string) < length : string += connect.recv( length -
len(string) )
19     if LOG[0] : print>>LOG[1], 'RECV %s BYTES FROM %s'%( length,
connect.getpeername() )
20     if LOG[0] == 2 : print>>LOG[1], string
21     if LOG[0] : LOG[1].flush()
22     return string
23 #-----
24 def send( connect, data ) :
25     if data.__class__ == str : send_string( connect, 'S'+data )
26     else : send_string( connect, 'P'+cPickle.dumps( data ) )
27     if LOG[0] == 3 : pprint.pprint( data, LOG[1] ); LOG[1].flush()
28 #-----
29 def recv( connect ) :
30     string, data = recv_string( connect ), None
31     if string.startswith('S') : data = string[1:]
32     elif string.startswith('P') : data = cPickle.loads( string[1:] )
33     elif string.startswith('E') :
34         exc, sys.server_stack = cPickle.loads( string[1:] )
35         if LOG[0] == 3 : print>>LOG[1], 'RECV EXCEPTION %s\n'%exc,
sys.server_stack; LOG[1].flush()
36         if exc != None : raise exc
37         if LOG[0] == 3 : pprint.pprint( data, LOG[1] ); LOG[1].flush()
38     return data
39 #-----
40 class _mybuf :
41     def __init__( self ) : self.L = []
42     def write( self, s ) : self.L.append( s )
43 #-----
44 def _excepthook( exctype, value, traceback ) :
45     if hasattr( sys, 'server_stack' ) :
46         old_stderr, sys.stderr = sys.stderr, _mybuf()
47         _sys_excepthook( exctype, value, traceback )
48         print>>old_stderr, ''.join( sys.stderr.L[:-4] ), sys.server_stack
49         del sys.server_stack; sys.stderr = old_stderr
50     else : _sys_excepthook( exctype, value, traceback )
51 #-----
52 def send_exception( connect ) :
53     old_stderr, sys.stderr, exc = sys.stderr, _mybuf(), sys.exc_info()[1]
54     _sys_excepthook( *sys.exc_info() )

```

```

55     sys.stderr, server_stack = old_stderr, ''.join( sys.stderr.L[1:-1] )
56     send_string( connect, 'E'+ cPickle.dumps(( exc, server_stack )) )
57     if LOG[0] == 3 : print>>LOG[1], 'SEND EXCEPTION %s\n'%exc,
server_stack; LOG[1].flush()
58 #-----
59 _sys_excepthook, sys.excepthook = sys.excepthook, _excepthook

```

Пример использования:

file examples/server2.py:

```

1 #!/usr/bin/python
2 from mysocket2 import *
3 #-----
4 s = socket(AF_INET,SOCK_STREAM)
5 s.setsockopt(SOL_SOCKET, SO_REUSEADDR, 1)
6 s.bind(('127.0.0.1',1234))
7 s.listen(5)
8 #-----
9 while 1 :
10     try :
11         conn, addr = s.accept()
12         t = recv( conn )
13         send( conn, t*2 )
14         conn.close()
15     except Exception, e : send_exception( conn )

```

file examples/client2.py:

```

1 #!/usr/bin/python -u
2 from mysocket2 import *
3 #-----
4 s = socket(AF_INET,SOCK_STREAM)
5 s.connect(('127.0.0.1',1234))
6 send( s, 'qwe' )
7 print '***** %s'%recv(s)
8 s.close()
9 #-----
10 s = socket(AF_INET,SOCK_STREAM)
11 s.connect(('127.0.0.1',1234))
12 send( s, {} )
13 print '***** %s'%recv(s)

```

```
14 s.close()
```

Запускаем:

```
-----  
1 examples> ./server2.py &  
2 examples> ./client2.py  
3 ***** qweqwe  
4 Traceback (most recent call last):  
5   File "./client2.py", line 13, in <module>  
6     print '***** %s'%recv(s)  
7   File "/home/aiv/Projects/PyArt/examples/mysocket2.py", line 36, in recv  
8     if exc != None : raise exc  
9   File "./server2.py", line 13, in <module>  
10    send( conn, t*2 )  
11 TypeError: unsupported operand type(s) for *: 'dict' and 'int'  
12 examples> fg  
13 ^C  
-----
```

5 Многопоточный сервер и удалённый доступ к экземплярам класса пользователя

Создание многопоточного сервера достаточно примитивно — при установлении соединения сервер (функция `server`) запускает функцию `_thread_func` в отдельной нити, передает ей установленное соединение и ждет нового соединения. Аргумент `entryIPs` функции `server` должен содержать список паттернов определяющих IP, с которых разрешено соединение с сервером (для разрешения всех IP необходимо указать `['*']`). Интерес представляет устройство функции `_thread_func` — именно оно определяет всю функциональность сервера (модуль `mysocket3.py`, строки 75–101):

file `examples/mysocket3.py`:

```
-----  
1 from socket import *  
2 import sys, cPickle, pprint, thread, fnmatch #, struct  
3 #-----  
4 class SocketClosed(Exception) : pass  
5 class SocketError(Exception) : pass  
6 LOG = [ 0, sys.stderr ] #loglevel, logfile  
7 #-----  
8 def send_string( connect, string ) :  
9     connect.send( '%08i'%len(string) )  
10    connect.send( string )  
-----
```

```

11     if LOG[0] : print>>LOG[1], 'SEND %s BYTES TO %s'%( len(string),
connect.getpeername() )
12     if LOG[0] == 2 : print>>LOG[1], string
13     if LOG[0] : LOG[1].flush()
14 #-----
15 def recv_string( connect ) :
16     try : length = int( connect.recv(8) )
17     except : raise SocketClosed()
18     string = connect.recv( length )
19     while len(string) < length : string += connect.recv( length -
len(string) )
20     if LOG[0] : print>>LOG[1], 'RECV %s BYTES FROM %s'%( length,
connect.getpeername() )
21     if LOG[0] == 2 : print>>LOG[1], string
22     if LOG[0] : LOG[1].flush()
23     return string
24 #-----
25 def send( connect, data ) :
26     if data.__class__ == str : send_string( connect, 'S'+data )
27     else : send_string( connect, 'P'+cPickle.dumps( data ) )
28     if LOG[0] == 3 : pprint.pprint( data, LOG[1] ); LOG[1].flush()
29 #-----
30 def send_wrap( connect, ID, doc, class_name, module_name ) :
31     send_string( connect, 'W'+cPickle.dumps( ( ID, doc, class_name,
module_name ) ) )
32     if LOG[0] == 3 : pprint.pprint( ( ID, doc, class_name, module_name ),
LOG[1] ); LOG[1].flush()
33 #-----
34 def recv( connect ) :
35     string, data = recv_string( connect ), None
36     if string.startswith('S') : data = string[1:]
37     elif string.startswith('P') : data = cPickle.loads( string[1:] )
38     elif string.startswith('W') : data = _wrap( connect, *cPickle.loads(
string[1:] ) )
39     elif string.startswith('E') :
40         exc, sys.server_stack = cPickle.loads( string[1:] )
41         if LOG[0] == 3 : print>>LOG[1], 'RECV EXCEPTION %s\n'%exc,
sys.server_stack; LOG[1].flush()
42         if exc != None : raise exc
43     if LOG[0] == 3 : pprint.pprint( data, LOG[1] ); LOG[1].flush()
44     return data
45 #-----
46 class _mybuf :
47     def __init__( self ) : self.L = []
48     def write( self, s ) : self.L.append( s )

```

```

49 #-----
50 def _excepthook( exctype, value, traceback ) :
51     if hasattr( sys, 'server_stack' ) :
52         old_stderr, sys.stderr = sys.stderr, _mybuf()
53         _sys_excepthook( exctype, value, traceback )
54         print>>old_stderr, ''.join( sys.stderr.L[:-4] ), sys.server_stack
55         del sys.server_stack; sys.stderr = old_stderr
56     else : _sys_excepthook( exctype, value, traceback )
57 #-----
58 def send_exception( connect ) :
59     old_stderr, sys.stderr, exc = sys.stderr, _mybuf(), sys.exc_info()[1]
60     _sys_excepthook( *sys.exc_info() )
61     sys.stderr, server_stack = old_stderr, ''.join( sys.stderr.L[1:-1] )
62     send_string( connect, 'E'+ cPickle.dumps(( exc, server_stack )) )
63     if LOG[0] == 3 : print>>LOG[1], 'SEND EXCEPTION %s\n'%exc,
server_stack; LOG[1].flush()
64 #-----
65 _sys_excepthook, sys.excepthook = sys.excepthook, _excepthook
66 #-----
67 # SERVER
68 #-----
69 _check_name = lambda a : a.startswith('__') or not a.startswith('_')
70 def _is_const( data ) :
71     if data.__class__ in ( int, long, float, complex, str, None.__class__
) : return True
72     if data.__class__ is tuple : return len(data) == len(filter(
_is_const, data ))
73     return False
74 #-----
75 def _thread_func( connect, IP, userclass ) :
76     try:
77         L, D = recv( connect ); C = userclass( *L, **D ); table = { id(C)
: [C,1] }
78         send_wrap( connect, id(C), C.__doc__, C.__class__.__name__,
C.__class__.__module__ )
79         while 1 :
80             Q = recv( connect ); cmd, ID = Q[:2]
81             if cmd in ('DEL','GET','SET') and not _check_name(Q[2]) :
raise AttributeError(Q[2])
82             if ID not in table : raise SocketError('id %S not in
table'%ID)
83             if cmd == 'DIR' : send( connect, dir(table[ID][0]) )
84             elif cmd == 'ALL' : send( connect, table[ID][0] )
85             elif cmd == 'DEL' : delattr( table[ID][0], Q[2] ); send(
connect, None )

```

```

86         elif cmd == 'SET' : setattr( table[ID][0], Q[2], Q[3] );
send( connect, None )
87         elif cmd == 'BYE' :
88             pair = table[ID]; pair[1] -= 1
89             if not pair[1] : del table[ID]
90             del pair; send( connect, None )
91         elif cmd == 'GET' : res = getattr( table[ID][0], Q[2] )
92         elif cmd == 'RUN' : res = table[ID][0]( *Q[2], **Q[3] )
93         else : raise SocketError('Illegal query %s'%cmd)
94         if cmd in ('GET','RUN') :
95             if _is_const( res ) : send( connect, res )
96         else :
97             table.setdefault( id(res), [res,0] )[1] += 1
98             send_wrap( connect, id(res), res.__doc__,
res.__class__.__name__, res.__class__.__module__ )
99     except SocketClosed, e : pass
100     except : send_exception( connect )
101     del C, table; connect.close()
102 #-----
103 def server( servIP, entryIPs, port, userclass ) :
104     s = socket(AF_INET,SOCK_STREAM);
105     s.setsockopt(SOL_SOCKET, SO_REUSEADDR, 1)
106     s.bind((servIP,port))
107     s.listen(5)
108     while 1 :
109         try:
110             connect, address = s.accept(); IP=address[0]
111             if LOG[0] : print>>LOG[1], 'CONNECT FROM',IP; LOG[1].flush()
112             if filter( lambda p : fnmatch.fnmatch(IP,p), entryIPs ) :
thread.start_new_thread( _thread_func, ( connect, IP, userclass ) )
113             else : raise SocketError( 'Connect from %s to %s port %s
locked'%( IP, servIP, port ) )
114             except Exception, e : send_exception( connect ); connect.close()
115 #-----
116 # CLIENT
117 #-----
118 def client( address, port, *L, **D ) :
119     connect = socket(AF_INET,SOCK_STREAM)
120     connect.connect(( address, port ))
121     send( connect, ( L, D ) )
122     return recv( connect )
123 #-----
124 class _wrap :
125     def __init__( self, connect, ID, doc, class_name, module_name ) :
126         self.__dict__.update( [ ('_connect',connect), ('_ID',ID),

```

```

( '__doc__', doc), ( '_class_name', class_name), ( '_module_name', module_name) ] )
127     def _dir( self ) : send( self._connect, ( 'DIR', self._ID ) ); return
recv( self._connect )
128     def _all( self ) : send( self._connect, ( 'ALL', self._ID ) ); return
recv( self._connect )
129     def __getattr__( self, attr ) : send( self._connect, ( 'GET',
self._ID, attr ) ); return recv( self._connect )
130     def __call__( self, *L, **D ) : send( self._connect, ( 'RUN',
self._ID, L, D ) ); return recv( self._connect )
131     def __setattr__( self, attr, val ) : send( self._connect, ( 'SET',
self._ID, attr, val ) ); recv( self._connect )
132     def __delattr__( self, attr ) : send( self._connect, ( 'DEL',
self._ID, attr ) ); recv( self._connect )
133     def __del__( self ) : send( self._connect, ( 'BYE', self._ID ) );
recv( self._connect )
134 #-----
135 def _dir( 0 ) :
136     if isinstance( 0, _wrap ) : return 0._dir()
137     return _dir( 0 )
138 _dir, dir = dir, _dir

```

Поскольку как правило ведущим является клиент, возможен вариант с созданием таблицы пользовательских функций на сервере. Клиент отправляет на сервер запрос, состоящий из имени функции и её аргументов, и получает в ответ результат выполнения функции. Это хороший подход (называется **RPC** — **Remote Procedure Call**), позволяющий изящно решать довольно разнообразные задачи, но мы оставим его реализацию читателю в качестве упражнения, а сами пойдем дальше.

При работе в рамках одного соединения (одной сессии) часто возникает необходимость хранить на сервере какие то данные. Кроме того, задание таблицы функций в виде словаря выглядит довольно громоздко. Лучше всего собрать все данные и функции, относящиеся к одной сессии, в экземпляре пользовательского класса, и обеспечить удалённый доступ к этому классу со стороны клиента. Фактически речь идет о написании так называемого тонкого клиента (оболочки) — он не имеет никаких собственных методов и полей, а оперирует методами и полями пользовательского экземпляра класса на сервере, но пользователь этого не замечает. Вся прелесть идеи состоит в том, что **Python** позволяет делать такие вещи с мизерными трудозатратами.

Все типы данных в **Python** делятся на неизменяемые (значение **None**, строки, числа и кортежи состоящие из неизменяемых данных) и изменяемые (все остальные). Неизменяемые данные можно смело передавать с сервера на клиент — они доступны лишь для чтения. Изменяемые данные требуют особого отношения — все изменения на клиенте должны как то сбрасываться на сервер, иначе целостность данных может быть нарушена. Простейший (но не самый эффективный с точки зрения производительности) способ состоит в том, что бы все изменяемые данные держать на сервере, обеспечивая к ним доступ через оболочку. Отдельно следует упо-

мянуть про методы — это тоже данные, но они могут быть запущены. Для проверки данных «на неизменяемость» служит функция `_is_const` (строки 70–73).

Для создания экземпляра оболочки (класс `_wrap`, строки 124–133), с сервера необходимо передать ID (поскольку изменяемых данных в рамках одной сессии может быть несколько), строку документации и тип. Имеет смысл доработать функцию `recv`, и при передаче оболочки предварять строку с результатами консервирования символом `W`. Для передачи оболочки служит функция `send_wrap` (строки 30–32).

Через оболочку достаточно перегрузить следующие методы и обеспечить осуществление следующих операций (запрос на каждую операцию передается на сервер в виде кортежа):

- `_dir()` → `('DIR', ID)` — вызов функции `dir()` для экземпляра класса, результатом является список атрибутов;
- `_all()` → `('ALL', ID)` — создает и возвращает копию экземпляра класса (на стороне клиента);
- `__delattr__(attr)` → `('DEL', ID, attr)` — удаление атрибута `attr`;
- `__setattr__(attr, val)` → `('SET', ID, attr, val)` — установка значения атрибута `attr`;
- `__getattr__(attr)` → `('GET', ID, attr)` — запрос значения атрибута `attr`, результатом служат неизменяемые данные или оболочка;
- `__call__(*args, **kw_args)` → `('RUN', ID, args, kw_args)` — запуск метода со списком позиционных аргументов `args` и именованных аргументов `kw_args`, результатом служат неизменяемые данные или оболочка;
- `__del__()` → `('BYE', ID)` — уничтожение объекта на сервере.

Для обеспечения безопасности блокируется доступ к атрибутам, с именами начинающимися с единичного знака подчёркивания. Для проверки имени атрибута применяется функция `_check_name` определённая в строке 69. Для доступа к методу `_dir` перегружена стандартная функция `dir` (строки 135–137).

Все объекты от одной сессии на сервере, для которых создаются экземпляры оболочек на клиенте, хранятся в словаре `table` (строка 77). Словарь имеет вид ключ (ID объекта): список из объекта и числа ссылок на объект (поскольку для одного объекта может запрашиваться несколько экземпляров оболочки со стороны клиента). Объект уничтожается, когда число ссылок становится равным нулю (строки 87–90). Сессия завершается, когда сокет закрывается (строка 99).

Для обеспечения стабильности и единообразия в протоколах обмена, на все запросы от экземпляра оболочки сервер присылает какой то ответ. В частности, на запросы `DEL`, `SET`, `BYE`, сервер отвечает значением `None`, что при необходимости позволяет клиенту принять возникшее при запросе исключение.

Для запуска сервера служит функция `server` (строки 103–114), ожидающая в бесконечном цикле новых соединений. Для установления соединения и создания первого экземпляра оболочки служит функция `client` (строки 118–122).

Пример использования (создаем на сервере словарь и управляем им с клиента):

file examples/server3.py:

```
-----  
1 #!/usr/bin/python  
2 from mysocket3 import *  
3 LOG[0], LOG[1] = 3, sys.stdout  
-----
```

file examples/client3.py:

```
-----  
1 #!/usr/bin/python -u  
2 from mysocket3 import *  
3 C = client( '127.0.0.1', 1234 )  
4 C[1]=2  
5 print '*****'  
6 print '***** dir(C) = %s'%dir(C)  
7 print '***** C._all() = %s'%C._all()  
8 print '***** C.items() = %s'%C.items()  
-----
```

Результаты запуска:

```
-----  
1 examples> ./server3.py &  
2 examples> ./client3.py  
3 CONNECT FROM 127.0.0.1  
4 RECV 14 BYTES FROM ('127.0.0.1', 50897)  
5 ((), {})  
6 SEND 428 BYTES TO ('127.0.0.1', 50897)  
7 (24378064,  
8 "dict() -> new empty dictionary\nndict(mapping) -> new dictionary  
initialized from a mapping object's\n    (key, value) pairs\nndict(iterable) ->  
new dictionary initialized as if via:\n    d = {}\n    for k, v in iterable:\n    d[k] = v\nndict(**kwargs) -> new dictionary initialized with the name=value  
pairs\n    in the keyword argument list. For example: dict(one=1, two=2)",  
9 'dict',  
10 '__builtin__')  
11 RECV 45 BYTES FROM ('127.0.0.1', 50897)  
12 ('GET', 24378064, '__setitem__')  
13 SEND 91 BYTES TO ('127.0.0.1', 50897)  
14 (24067600, 'x.__setitem__(i, y) <==> x[i]=y', 'method-wrapper',  
'__builtin__')  
15 RECV 43 BYTES FROM ('127.0.0.1', 50897)  
16 ('RUN', 24067600, (1, 2), {})  
17 SEND 3 BYTES TO ('127.0.0.1', 50897)  
-----
```

```
18 None
19 RECV 27 BYTES FROM ('127.0.0.1', 50897)
20 ('BYE', 24067600)
21 SEND 3 BYTES TO ('127.0.0.1', 50897)
22 None
23 *****
24 RECV 27 BYTES FROM ('127.0.0.1', 50897)
25 ('DIR', 24378064)
26 SEND 735 BYTES TO ('127.0.0.1', 50897)
27 ['__class__',
28  '__cmp__',
29  '__contains__',
30  '__delattr__',
31  '__delitem__',
32  '__doc__',
33  '__eq__',
34  '__format__',
35  '__ge__',
36  '__getattr__',
37  '__getitem__',
38  '__gt__',
39  '__hash__',
40  '__init__',
41  '__iter__',
42  '__le__',
43  '__len__',
44  '__lt__',
45  '__ne__',
46  '__new__',
47  '__reduce__',
48  '__reduce_ex__',
49  '__repr__',
50  '__setattr__',
51  '__setitem__',
52  '__sizeof__',
53  '__str__',
54  '__subclasshook__',
55  'clear',
56  'copy',
57  'fromkeys',
58  'get',
59  'has_key',
60  'items',
61  'iteritems',
62  'iterkeys',
```

```

63  'intervalvalues',
64  'keys',
65  'pop',
66  'popitem',
67  'setdefault',
68  'update',
69  'values']
70 ***** dir(C) = ['__class__', '__cmp__',
 '__contains__', '__delattr__', '__delitem__', '__doc__', '__eq__', '__format__',
 '__ge__', '__getattr__', '__getitem__', '__gt__', '__hash__', '__init__',
 '__iter__', '__le__', '__len__', '__lt__', '__ne__', '__new__', '__reduce__',
 '__reduce_ex__', '__repr__', '__setattr__', '__setitem__', '__sizeof__',
 '__str__', '__subclasshook__', 'clear', 'copy', 'fromkeys', 'get', 'has_key',
 'items', 'iteritems', 'iterkeys', 'intervalvalues', 'keys', 'pop', 'popitem',
 'setdefault', 'update', 'values']
71 RECV 27 BYTES FROM ('127.0.0.1', 50897)
72 ('ALL', 24378064)
73 SEND 14 BYTES TO ('127.0.0.1', 50897)
74 {1: 2}
75 ***** C._all() = {1: 2}
76 RECV 39 BYTES FROM ('127.0.0.1', 50897)
77 ('GET', 24378064, 'items')
78 SEND 128 BYTES TO ('127.0.0.1', 50897)
79 (24083272,
80  "D.items() -> list of D's (key, value) pairs, as 2-tuples",
81  'builtin_function_or_method',
82  '__builtin__')
83 RECV 34 BYTES FROM ('127.0.0.1', 50897)
84 ('RUN', 24083272, (), {})
85 SEND 136 BYTES TO ('127.0.0.1', 50897)
86 (24083776,
87  "list() -> new empty list\nlist(iterable) -> new list initialized from
iterable's items",
88  'list',
89  '__builtin__')
90 RECV 27 BYTES FROM ('127.0.0.1', 50897)
91 ('BYE', 24083272)
92 SEND 3 BYTES TO ('127.0.0.1', 50897)
93 None
94 RECV 41 BYTES FROM ('127.0.0.1', 50897)
95 ('GET', 24083776, '__str__')
96 SEND 83 BYTES TO ('127.0.0.1', 50897)
97 (24067856, 'x.__str__() <==> str(x)', 'method-wrapper', '__builtin__')
98 RECV 34 BYTES FROM ('127.0.0.1', 50897)
99 ('RUN', 24067856, (), {})

```

```

100 SEND 9 BYTES TO ('127.0.0.1', 50897)
101 '[(1, 2)]'
102 RECV 27 BYTES FROM ('127.0.0.1', 50897)
103 ('BYE', 24067856)
104 SEND 3 BYTES TO ('127.0.0.1', 50897)
105 None
106 RECV 27 BYTES FROM ('127.0.0.1', 50897)
107 ('BYE', 24083776)
108 SEND 3 BYTES TO ('127.0.0.1', 50897)
109 None
110 ***** C.items() = [(1, 2)]
111 RECV 45 BYTES FROM ('127.0.0.1', 50897)
112 ('GET', 24378064, '__getitem__')
113 SEND 98 BYTES TO ('127.0.0.1', 50897)
114 (24083272,
115  'x.__getitem__(y) <=> x[y]',
116  'builtin_function_or_method',
117  '__builtin__')
118 RECV 40 BYTES FROM ('127.0.0.1', 50897)
119 ('RUN', 24083272, (1,), {})
120 SEND 5 BYTES TO ('127.0.0.1', 50897)
121 2
122 RECV 27 BYTES FROM ('127.0.0.1', 50897)
123 ('BYE', 24083272)
124 SEND 3 BYTES TO ('127.0.0.1', 50897)
125 None
126 ***** C[1] = 2
127 RECV 27 BYTES FROM ('127.0.0.1', 50897)
128 ('BYE', 24378064)
129 SEND 3 BYTES TO ('127.0.0.1', 50897)
130 None
131 examples> fg
132 ^C

```

Вывод довольно громоздкий, давайте разбираться что тут к чему. В первой строке мы запускаем сервер в фоновом режиме, во второй строке запускаем клиент. Весь остальной вывод — то что печатает клиент вперемешку с логами сервера. Для наглядности вывод клиента предваряется строчкой из звёздочек (см. файл `client3.py`, строки 4–8).

Итак, строка 3 — установлено соединение с клиентом. Строки 4–5 — получены аргументы конструктора. В ответ (строки 6–10) посылается кортеж для создания экземпляра оболочки — ID, строка документации, имя класса и имя модуля в котором класс определён.

В строках 11–12 приходит запрос к атрибуту `__getitem__` (строка 3 файла `client3.py`), в ответ (строки 13–14) посылается кортеж для создания оболочки созданного на сервере связанного метода `__getitem__`, который запускается (строки 15–16) с аргументами (1,2). В ответ

(строки 17–18) приходит `None` — результат выполнения метода.

В строках 19–20 деструктор оболочки метода `__setitem__` информирует сервер об уничтожении экземпляра оболочки на клиенте, связанный метод `__setitem__` должен быть уничтожен на сервере в словаре `table`, сервер отвечает `None` в строках 21–22. Строка 3 файла `client3.py` отработала, выводится строчка из звёздочек (строка 23).

На сервер приходит запрос на список атрибутов экземпляра оболочки для словаря (строки 24–25, запрос иницирован в строке 6 файла `client3.py`), в строках 26–69 приводится ответ сервера, который печатается в строке 70.

В строках 71–72 запрос серверу на копию словаря (строка 7 файла `client3.py`), копия возвращается строках 73–74 и печатается в строке 75.

В строках 76–77 запрос серверу на атрибут словаря `items` (строка 8 файла `client3.py`), информация для создания оболочки для связанного метода `items` возвращается в строках 78–82. В строках 83–84 метод вызывается без аргументов, в результате в строках 85–89 возвращается оболочка для списка — результата работы метода `items` на сервере. В строках 90–91 деструктор оболочки метода `items` информирует сервер об уничтожении оболочки на клиенте, сервер отвечает `None` в строках 82–93.

В строках 94–95 для списка (результата работы метода `items`) запрашивается атрибут `__str__` (поскольку список выводится на печать в той же строке 7 файла `client3.py`). Атрибут возвращается в виде оболочки в строках 96–97, запускается без аргументов в строках 98–99, возвращает результат `'[(1,2)]'` в строках 100–101.

В строках 102–109 деструкторы списка (результата работы метода `items`) и метода списка `__str__` информируют сервер об уничтожении экземпляров оболочек на клиенте, сервер отвечает `None` и `None`.

Результат, возвращенный методом списка `__str__` печатается клиентом (седьмая строка `client3.py`) в строке 110.

В строках 111–112 на сервер отправляется запрос на атрибут `__getitem__` (строка 8 файла `client3.py`, последняя). Атрибут возвращается в виде оболочки для связанного метода (строки 113–117), вызывается с аргументом 1 (строки 118–119) и возвращает результат — число 2, (строки 120–121).

В строках 122–125 деструктор оболочки метода `__getitem__` прощается с сервером, сервер отвечает `None`.

В строке 126 клиент печатает результат метода `__getitem__`.

В строках 127–130 деструктор словаря прощается с сервером, сервер отвечает `None`. Занавес.

6 Заключение

Уже в нынешнем виде модуль `mysocket3.py` позволяет с минимальными усилиями создавать высокоуровневые клиент–серверные приложения, причём обеспечивается работа многопоточного сервера с контролем входящих соединений по IP-адресам, запись логов, передача структурированных данных, передача исключений с клиента на сервер. Концепция оболочек пользовательских классов не только делает разработку сервера простой и наглядной (собственно это и на разработку сервера то перестает быть похоже, получился некий аналог DCOM или

CORBA), но может применяться и в других областях, например для автоматической записи истории действий пользователя.

Тем не менее многие задачи нами ещё не решены. Автоматическая загрузка обновлённых модулей, передача файлов и λ -функций, трансляция портов и перенаправление соединений, восстановление разорванного соединения, масштабирование сервера на несколько машин, обеспечение безопасности — вот неполный список того, что ещё хотелось бы включить в библиотеку. Но вспомнив высказывание «лучшее — враг хорошего», я решил отложить все эти вещи до следующей статьи, иначе **эта** статья никогда не была бы завершена.

Я выражаю глубокую благодарность моему другу Дмитрию Фролову, пробудившему во мне интерес к этой тематике несколько лет назад, и давшему ряд ценных советов.

Список литературы

- [1] Г. Россум, Ф.Л.Дж. Дрейк, Д.С. Откидач. Язык программирования Python. 2001 — 454С.
- [2] Марк Лутц. Программирование на Python. С-Пб.: «Символ». 2002 — 1135С.
- [3] Дэвид Бизли. Python. Подробный справочник (четвертое издание). С-Пб.–М.: «Символ». 2010 — 858С.
- [4] Стивенс У.Р. UNIX. Разработка сетевых приложений. «ПИТЕР». 2003 — 1045С.