Python Network Programming

by Sebastian V. Tiponut
Technical University Timisoara

Version 0.00, 16. July 2001

Contents

List of Figures

1 Introduction

Network programming is a buzzword now in the soft world. We see the market filled with an avalanche of network oriented applications like database servers, games, Java servlets and applets, CGI scripts, different clients for any imaginable protocol and the examples may continue. Today, more then half of the applications that hit the market are network oriented. Data communication between two machines (on local net or Internet) is not any more a curiosity but is a day to day reality. “The network is the computer” says the Sun Microsystem’s motto and they are right. The computer is no more seen as a separate entity, dialogging only with it’s human operator but as part of a larger system - the network, bound via data links with other thousands of other machines.

This paper is presenting a possible way of designing network-oriented applications using Python. Because the author is a Linux fan, the examples contained in this paper are related to Linux[1] and apologizes all the Windows or Mac OS users (fans ?) for any inconvenience on reading this text. With a little effort, the examples are portable to another non-UNIX operation system. Presenting a quick structure of this paper, first four sections are dealing with *primitive* design – at socket level – of network applications. The remaining sections are treating specific protocols like *http*, *ftp*, *telnet* or *smtp*. The section dealing with *http* will contain a subsection about writing CGI scripts and using the `cgi` module.

Going further on more concrete subjects, we are going to analyze the possibilities of network programming provided in Python. Raw network support is implemented in Python through the `socket` module, this module comprising mostly of the system-calls, functions and constants defined by the 4.3BSD Interprocess Communication facilities (see [1]), implemented in object-oriented style. Python offers a simple interface (much simpler than the corresponding C implementation, though based on this one) to properly create and use a socket. Primarily, is defined the `socket()` function returning a *socket object*[2]. The socket has several methods, corresponding to their pairs from C `sys/socket.h`, like `bind()`, `connect()`, `listen()` or `accept()`. Programmers accustomed with socket usage under C language[3] will find very easy to translate their knowledge in the more-easy-to-use socket implementation under Python. Python eliminates the daunting task of filling structures like `sockaddr_in` or `hostent` and ease the use of previously mentioned methods or functions – parameter passing and functions call are easier to handle. Some network-oriented functions are provided too: `gethostbyname()`, `getprotobyname()` or conversion functions `ntohl()`, `htons()`, useful when converting integers to and from network format. The module provides constants like `SOMAXCONN`, `INADDR_*`, used in `gesockopt()` or `setsockopt()` functions. For a complete list of above mentioned constants check your UNIX documentation on socket implementation.

Python provide beside `socket`, additional modules (in fact there is a whole bundle of them) supporting the most common network protocols at user level. For example we may find useful modules like `httplib`, `ftplib`, `telnetlib`, `smtplib`. There is implemented support for CGI scripting through `cgi` module, a module for URL parsing, classes describing web servers and the examples may continue. This modules are specific implementations of well known protocols, the user being encouraged to use them and not trying to reinvent the wheel. The author hopes that the user will enjoy the richness of Python’s network programming facilities and use them in new and more exciting ways.

Because all the examples below are written in Python, the reader is expected to be fluent with this programming language.

[1]And to other *NIX systems, POSIX compliant.
[2]We call this further just socket.
[3]4.3BSD IPC implementation found on mostly UNIX flavors.

2 Basic socket usage

The *socket* is the basic structure for communication between processes. A socket is defined as *"an endpoint of communication to which a name may be bound"* [1]. The 4.3BSD implementation define three communication domains for a socket: the UNIX domain for on-system communication between processes; the Internet domain for processes communicating over TCP(UDP)/IP protocol; the NS domain used by processes communicating over the old Xerox communication protocol.

Python is using only[4] the first two communication domains: UNIX and Internet domains, the AF_UNIX and AF_INET address families respectively. UNIX domain addresses are represented as strings, naming a local path: for example `/tmp/sock`. This can be a socket created by a local process or, possibly, created by a foreign process. The Internet domain addresses are represented as a *`(host, port)`* tuple, where *`host`* is a string representing a valid Internet hostname, say `matrix.ee.utt.ro` or an IP address in dotted decimal notation and *`port`* is a valid port between 1 and 65535[5]. Is useful to make a remark here: instead of a qualified hostname or a valid IP address, two special forms are provided: an empty string is used instead `INADDR_ANY` and the '<broadcast>' string instead of `INADDR_BROADCAST`.

Python offer all five type of sockets defined in 4.3BSD IPC implementation. Two seem to be generally used in the vastness majority of the new applications. A *stream* socket is a connection-oriented socket, and has the underlaying communication support the TCP protocol, providing bidirectional, reliable, sequenced and unduplicated flow of data. A *datagram* socket is a connectionless communication socket, supported through the UDP protocol. It offers a bidirectional data flow, without being reliable, sequenced or unduplicated. A process receiving a sequence of *datagrams* may find duplicated messages or, possibly, in another order in which the packets were sent. The *raw*, *sequenced* and *reliably delivered message* sockets types are rarely used. *Raw* socket type is needed when one application may require access to the most intimate resources provided by the socket implementation. Our document is focusing on *stream* and *datagram* sockets.

2.1 Creating a socket

A socket is created through the `socket(`*`family, type`*`[, `*`proto`*`])` call; *`family`* is one of the above mentioned address families: `AF_UNIX` and `AF_INET`, *`type`* is represented through the following constants: `SOCK_STREAM, SOCK_DGRAM, SOCK_RAW, SOCK_SEQPACKET and SOCK_RDM`. *`proto`* argument is optional and defaults to 0. We see that `socket()` function returns a socket in the specified domain with the specified type. Because the constants mentioned above are contained in the `socket` module, all of them must be used with the `socket.CONSTANT` notation. Without doing so, the interpreter will generate an error. To create a *stream* socket in the Internet domain we are using the following line:

```
sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
```

Substituting `socket.SOCK_STREAM` with `socket.SOCK_DGRAM` we create a *datagram* socket in the Internet domain. The following call will create a *stream* socket in the UNIX domain:

```
sock = socket.socket(socket.AF_UNIX, socket.SOCK_STREAM)
```

We discussed so far about obtaining a socket of different types in different communication domains.

[4]Xerox Network System is no longer used.
[5]Using a port under 1000 must be done with root privileges.

2.2 Connecting a socket and data transfer

A server from our point of view is a process which listen on a specified port. We may call the association port, process as a service. When another process wants to meet the server or use a specific service it must connect itself to the address and portnumber specified by the server. This is done calling the socket method `connect(address)`, where *`address`* is a pair *`(host, port)`* in the Internet domain and a pathname in the UNIX domain. When using the Internet domain a connection is realized with the following code:

```
sock.connect(('localhost', 8000))
```

while in UNIX domain,

```
sock.connect('/tmp/sock')
```

If the service is unavailable or the server don't want to talk with the client process a socket.error-(111, 'Connection refused') is issued. Elsewhere, after the connection is established with the desired server, data is sent and received with `send(buffer[, flags])` and `recv(buffer[, flags])` methods. These methods accepts as mandatory parameter the size of the buffer in bytes and some optional *`flags`*; for a description about the meaning of the flags consult the UNIX man page for the corresponding function[6].

2.3 Binding a name to socket

The socket, after creation, is nameless, though it have an associated descriptor. Before it can be used it must be bind to a proper *`address`* since this is the only way a foreign process may reference it. The `bind(address)` method is used to "name" a socket. The meaning of the *`address`* is explained above. Next call will bind a socket in the Internet domain with *`address`* composed from hostname *`localhost`* and port number *`8000`*:

```
sock.bind(('localhost', 8000))
```

Please take care when typing: indeed there are *two* pairs of parenthesis. Doing elsewhere the interpreter will issue a TypeError. The purpose of the two pairs of parenthesis is simple: *`address`* is a tuple containing a string and an integer. The hostname must be properly picked, the best method is to use `gethostname()` routine in order to assure host independence and portability[7]. Creating a socket in the UNIX domain use *`address`* as a single string, naming a local path:

```
sock.bind('/tmp/sock')
```

This will create the '/tmp/sock' file (pipe) which will be used for communication between the server and client processes. The user must have read/write permissions in that specific directory where the socket is created and the file itself must be deleted once it's no longer of interest.

2.4 Listening and accepting connections

Once we have a socket with a proper name bound to it, next step is calling the `listen(queue)` method. It instructs the socket to passively listen on port *`port`*. `listen()` take as parameter

[6]Do "man recv(2)" or "man send(2)".

[7]Don't bind a real hostname to a socket unless you do it for testing purposes only; if you do so the program will run solely on that particular system whom hostname was bind to the socket.

an integer representing the maximum queued connection. This argument should be at least 1 and maximum, system-dependent, 5. Until now we have a socket with a proper bounded address. When a connection request arrives, the server decide whether it will be accepted or not. Accepting a connection is made through the `accept()` method. It takes no parameter but it returns a tuple *`(clientsocket, address)`* where *`clientsocket`* is a new socket server uses to communicate with the client and *`address`* is the client's address. `accept()` normally blocks until a connection is realized. This behavior can be overridden running the method in a separate thread, collecting the new created socket descriptors in a list and process them in order. Meantime, the server can do something else. The above mentioned methods are used as follows:

```
sock.listen(5)
clisock, address = sock.accept()
```

The code instructs the socket on listening with a queue of five connections and accept all incoming "calls". As you can see, `accept()` returns a new socket that will be used in further data exchanging. Using the chain *bind-listen-accept* we create TCP servers. Remember, a TCP socket is connection-oriented; when a client wants to speak to a particular server it must connect itself, wait until the server accepts the connection, exchange data then close. This is modeling a phone call: the client dial the number, wait till the other side establish the connection, speak then quit.

2.5 UDP sockets

We chose to deal with connectionless sockets separately because these are less common in day to day client/server design. A *datagram* socket is characterized by a connectionless and symmetric message exchange. Server and client exchange *data packets* not data streams, packets flowing between client and server separately. The UDP connection resemble the postal system: each message is encapsulated in an envelope and received as a separate entity. A large message may be split into multiple parts, each one delivered separately (not in the same order, duplicated and so on). Is the receiver's duty to assemble the message.

The server[8] have a `bind()` method used to append a proper name and port. There are no `listen()` and `accept()` method, because the server is not listening and is not accepts connection. Basically, we create a P.O.Box where is possible to receive messages from client processes. Clients only send packets, data and address being included on each packet.

Data packets are send and received with the `sendto(`*`data, address`*`)` and `recvfrom(`*`buffer`*`[, `*`flags`*`])` methods. First method takes as parameters a string and the server address as explained above in `connect()` and `bind()`. Because is specified the remote end of the socket there is no need to connect it. The second method is similar to `recv()`.

2.6 Closing the socket

After the socket is no longer used, it must be closed with `close()` method. When a user is no more interested on any pending data a *shutdown* may be performed before closing the socket. The method is `shutdown(`*`how`*`)`, where *`how`* is: 0 if no more incoming data will be accepted, 1 will disallow data sending and a value of 2 prevent both send and receive of data. Remember: always close a socket after using it.

[8]Which process is the server and which is the client is hard to predict at socket level, because of the symmetrical connection.

2.7 Using functions provided in socket module

Was presented before that `socket` module contains some useful functions in network design. This functions are related with the *resolver* libraries, */etc/services* or */etc/protocols* mapping files or conversions of quantities.

2.7.1 Functions based on resolver library

There are three functions using BIND8 or whatever resolver you may have. This functions usually converts a hostname to IP address or IP address to hostname. One function is not related with the resolver, `gethostname()`, this function returning a string containing the name of the machine where the script is running. It simply read (through the corresponding C function) the */etc/HOSTNAME*[9] file.We can use (prior to version 2.0) the `getfqdn(`*`hostname`*`)` function, which return the "Fully Qualified Domain Name" for the *`hostname`*. If the parameter is missing it will return the localhost's FQDN. Two functions are used to translate hostnames into valid IP addresses: `gethostbyname(`*`hostname`*`)` and `gethostbyname_ex(`*`hostname`*`)`. Both functions take as mandatory parameter a valid hostname. First function returns a dotted notation IP address and the last function (*ex* from "extended") a tuple *(hostname, aliaslist, ipaddrlist)*. Last function discussed here is `gethostbyaddr(`*`ipaddr`*`)` returning hostname when the IP address is given.

2.7.2 Service-related functions

/etc/services is a file which maps services to portnumbers. For example, http service is mapped on port 80, ftp service on port 21 and ssh service on port 22. `getservbyname(`*`servname`*`)` is a functions that translates a service name into a valid portnumber, based on the file presented above. The method will assure platform independence (or even computer independence) while the same service may not be mapped exactly on the same port.

When we want to translate a protocol into a number suitable for passing as third argument for the `socket()` function use `getprotobyname(`*`protoname`*`)`. It translate, based on */etc/protocols* file, the protocol name, say GGP or ICMP, to the corresponding number.

2.7.3 Miscellaneous functions

To convert short and long integers from host to network order and reversed, four functions were provided. On some systems (i.e. Intel or VAX), host byte ordering is different from network order. Therefore programs are required to perform translations between this two formats[10].The table below synthesize their use:

Function Name	**Synopsis**
`htons(`*`sint`*`)`	Convert short integer from host to network format
`ntohs(`*`sint`*`)`	Convert short integer from network to host format
`htonl(`*`lint`*`)`	Convert long integer from host to network format
`ntohl(`*`lint`*`)`	Convert long integer from network to host format

[9] Or a corresponding file, depending on your system.

[10] In fact there is a simple byte-swap.

3 Basic network structures design

This section is focused in presenting four simple examples to illustrate the previous explained network functions and methods. Examples are providing a simple code designed for clarity not for efficiency. There are presented two pairs of server-client structures: one pair for the TCP protocol and the other one for the UDP protocol. The reader is encouraged to understand this examples before going further.

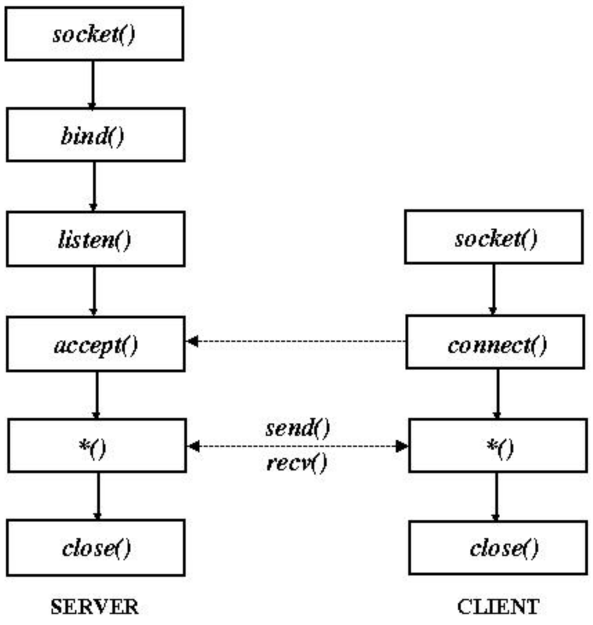

Figure 1: TCP connection

A *stream* connection is presented above, figure 1. Observe how the processes interact: the server is started and must be in *accept* state before the client issue its request. The client's `connect()` method is trying to rendezvous with the server while this one is accepting connections. After the connection have been negotiated follows a data exchange and both sides call `close()` terminating the connection. Remember, this is a one connection diagram. In the real world, after creating a new connection (in a new process or in a new thread) the server return to *accept* state. `*()` functions are user-defined functions to handle a specific protocol. Data transfer is realized through `send()` and `recv()`.

A *datagram* exchange is presented in figure 2. As you can see, server is the process which bind itself a name and a port, being the first that receive a request. The client is the process which first send a request. You may insert a `bind()` method into client code, obtaining identical entities. Which is the server and which is the client it is hard to predict in this case.

Hoping that the presented diagrams were useful, we are going further presenting the code on four simple structures, modeling the TCP and UDP client-server pairs.

3.1 Designing a TCP server

Designing a TCP server must follow the *bind-listen-accept* steps . Producing code for a simple echo server, running on *`localhost`* and listening on port *8000* is fairly simple. It will accept

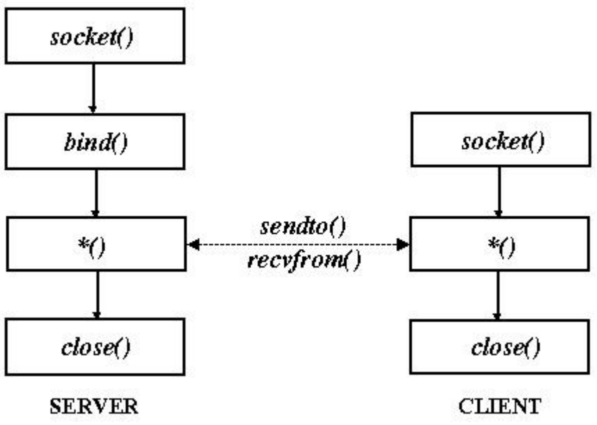

Figure 2: UDP connection

a single connection from the client, will echo back all the data and when it receives nothing[11] close the connection.

```
import socket

serversocket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
serversocket.bind(('localhost', 8000))
serversocket.listen(1)
clientsocket, clientaddress = serversocket.accept()
print 'Connection from ', clientaddress
while 1:
        data = clientsocket.recv(1024)
        if not data: break
        clientsocket.send(data)
clientsocket.close()
```

On first line is declared the import statement of the `socket` module. Lines 3 through 6 is the standard TCP server chain. On line 6 the server accept the connection initiated by the client; the `accept()` method returns `clientsocket` socket, further used to exchange data. The `while 1` loop is a the “echo code”; it sends back the data transmitted by the client. After the loop is broken (when the client send an empty string and the `if not data: break` condition is met) the socket is closed and program exits.

Before getting into client code, a quick way to test the server is a *telnet* client. Because telnet run over TCP protocol there should be no problem in data exchange. Telnet will simply read data from the standard input and send it to server and, when it receives data, display it on the terminal.

```
$ telnet localhost 8000
Trying 127.0.0.1...
Connected to localhost
Escape character is ^] .
```

[11]An empty ” string.

```
$
```

After the telnet send some data it enter in terminal mode: type something and the server will echo it back.

3.2 The TCP client

It is time to focus on writing client's code for our echo server. The client will enter a loop where data is read from the standard input and sent (in 1024 bytes packets - this is the maximum amount of data the server is reading specified in the server code above) to the server. The server is sending back the same data and the client is posting it again. Here is the code:

```
import socket

clientsocket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
clientsocket.connect(('localhost', 8000))
while 1:
        data = raw_input('>')
        clientsocket.send(data)
        if not data: break
        newdata = clientsocket.recv(1024)
        print newdata
clientsocket.close()
```

The client first send data and only after the data was sent, in case data is zero, exits. This prevent server from hanging (remember, the server exit when it receive an empty string). We decide to store the received data in `newdata` to prevent any problems that may occur, if no data or something wrong was transmitted back from the server. Compared with the server the client code is much simpler - only the socket creation and a simple connect, then the protocol follows.

3.3 Modeling datagram applications

Was specified before that is a slightly difference between a datagram client and a datagram server. Let's consider again the example of a server running on *`localhost`* and receiving packets on port *`8000`*.

```
import socket

sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
sock.bind(('localhost', 8000))
while 1:
        data, address = recvfrom(1024)
        if not data: break;
        sock.sendto(data, address)
sock.close()
```

As you can see in the example there are no listen or accept steps. Only binding is necessary, elsewhere the client is not aware where it must send packets to server. In simple terms, the kernel simply 'push' data packets to the server on the specified port, being no need for confirmation of connection and so on. The server decide if the packet will be accepted or not[12]. `recvfrom()` returns beside data the client's address, used further in the `sendto()` method. The client is not much different from the server:

```
import socket

sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
while 1:
        data = raw_input('>')
        sock.sendto(data, ('localhost', 8000))
        if not data: break;
        newdata = sock.recvfrom(1024)
        print newdata
sock.close()
```

We do not have a bind method, while the server is using `address` returned by `recvfrom()` method. Is possible to design an almost symmetric UDP server/client, using the same structure. The client will be the side that will initiate first the connection.

[12]A connection-oriented server may deny all the packets that a client is willing to send to it by simply refusing the connection.

4 Advanced topics on servers

This section cover some advanced techniques extensively used in modern server designs. The four structures previously presented are just some illustrations on how to use socket-related functions. The reader is not encouraged to use this four structures, because the code is very limited and was produced for exemplification purpose only. Going further, is necessary to learn some well-known patterns, enhancing server design.

4.1 Building a pristine environment

When a server is started it must define some parameters needed at run time. This is done through `get*()` functions. Next code is written for a web server and is trying to determine the system's hostname, port number on which *http* protocol is running.

```
hostname = gethostname()
try:
        http_port = getservbyname('http','tcp')
except socket.error:
        print 'Server error: http/tcp: unknown service'
```

We check `getservbyname()` against errors because is possible for a service to be absent on that system. Definitely, on new implemented services/protocols is useless to call this function. `hostname` and `http_port` will be further used as parameters to `bind()`.

After the previous step is over, the server must be disassociated from the controlling terminal of it's invoked, therefore it will run as demon. In C this is done with the `ioctl()` system-call, closing the standard input, output and error and performing some additional operations [1]. In Python the server must be invoked with a "&" after its name. Another way to complete this, the recommended one, is to call a `fork()` which create two processes: the parent, that exits immediately and the child (the child is our application). The child is adopted by *init*, thus running as demon. This is an affordable method and is not implying complicate code:

```
import os

...

pid = os.fork()
if pid: os._exit(0)
else:
        ...   #server code begin here
```

The server will be able no more to send messages through the standard output or standard error but through *syslog* facility. After this two steps are completed, we can select a method for handling multiple connections in the same time.

4.2 Handling multiple connections

Analysing the servers presented above, sections 3.1 and 3.3, it's clear that the design has a huge shortcoming: the server is not able to handle more than *one* client. A real server is designed to serve more then one client and moreover, multiple clients in the same time[13].

[13]Well, not really in the same time but this is a good approximation.

There are three methods to handle this issue: through the `select()` functions, creating a new process for each incoming request via `fork()` or to handle that request on a separate thread. Threading is the most elegant method and we recommend it. Using `select()` save some CPU in case of heavily accessed servers and may be a good option sometime. Creating a new process for each incoming connection is the most used pattern in current server design (for example Apache use it) but it have the disadvantage of wasting CPU time and system resources. Again, we recommend using threaded servers.

4.2.1 Threaded servers

Threaded servers use a separate thread for handling each connection. Threads are defined as a *light processes* and are running in and along with the main process[14] which started them. The diagram is presented below:

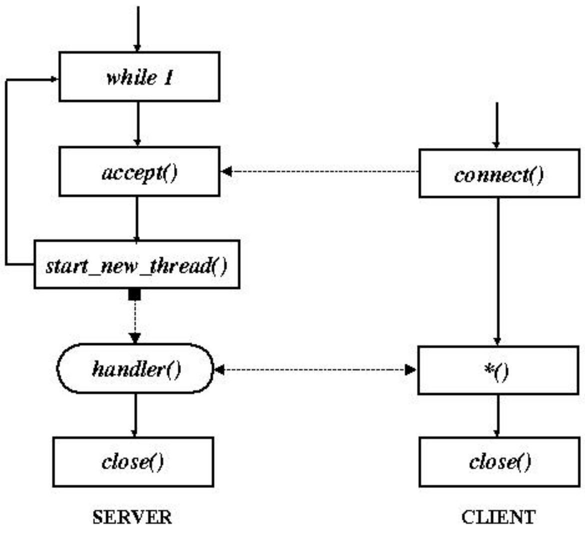

Figure 3: Threaded server diagram

Assuming we have a user-defined function called `handler` to handle a simple connection, each time a client wants to exchange data with the server `handler` is started on a separate thread doing its job. The squared and dashed arrow denote a new thread creation. The new thread will interact further with the client. Bellow is provided an example for a TCP server which main socket is called `sock`.

```
import socket, thread

def handler(socket):

        ...

...

while 1:
```

[14]Well, no more main process but main thread.

```
                clisock, addr = sock.accept()
                syslog.syslog('Incoming connection')
                thread.start_new_thread(handler,  (clisock,))
```

The handling function must be defined *before* being passed as argument for the new thread. In example this function take as parameter the socket corresponding to the client which initiate the connection. The function may perform any kind of operation on that socket. The logging job is no more done through the `print` function but through `syslog`. Alternatively, is possible to write a function to handle this job separately (producing our own logs in user-defined files).

Another choice is to use the `Threading` module which offers a more flexible implementation of threads. In fact, you create a *thread object*, with several methods, more easy to handle. The function's[15] arguments must be passed in a tuple (or if it is a single element use a *singleton* [4]) in order to have an error-free code, elsewhere the interpreter will issue a TypeError. The design may be enhanced putting the `accept()` method in a separate thread; consequently the server will be free (if `accept()` blocks, the normal behavior, it will do this separate from the main thread of the server) and may be used to perform some additional operation or even listen to another port. I suggest you the design of a web server running on standard HTTP port and being remotely controlled *in the same time* on an arbitrary port. Other valuable feature: using threads add a professional "touch" to your code and eliminate the need of write code to handle `fork` system-call and additional functions. In the same time, if you want to design the web server mentioned above, there will be at least three[16] processes running in the same time, causing processor to slow down.

4.2.2 Using select

Python provides a straightforward implementation of the UNIX standard `select()` function. This function, defined in `select` module, is used on multiplexing I/O requests among multiple sockets or file descriptors. There are created three lists of socket descriptors: one list with descriptors for which we want to read from, a list of descriptors to which we want to be able to write and a list which contains sockets with 'special conditions' pending. The only "special condition" implemented is the out-of-band data, specified in the socket implementation as `MSG_OOB` constant and used as special flag for the `send()` and `recv()` methods. This three lists are passed as parameters to function beside a ***`timeout`*** parameter, specifying how long should wait `select()` until returns, in case no descriptor is available. In return, `select()` provide us with three lists: one lists with socket which might be read from, another with writable sockets and the last one corresponding to 'special condition' category.

A special socket feature will help in a more flexible approach to handle `select()` calls. This is the `setblocking()` method, which, if called with a 0 parameter, will set the *nonblocking* option for a socket [7]. That is, requests that cannot complete immediately, causing the process to suspend aren't executed, but an error code is returned. There is provided a code for a better understanding of the subject:

```
import socket, select

preaders = []
pwriters = []

...
```

[15]We talk about the function executed in a separate thread.

[16]The main server process, one process for each handled connection and one for the other connection - remote control.

```
    sock.setblocking(0)
    preaders.append(sock)

    rtoread, rtowrite, pendingerr = select.select(preaders, pwriters, []\
    , 30.0)

    ...
```

As you can see we constructed two lists, one for the potential readers and the other ones for the potential writers. Because there is no interest in special condition, an empty list is passed as parameter to `select()` instead of a list with sockets. The ***timeout*** value is 30 seconds. After the function returns, we have two usable lists, with the available sockets for reading and writing. Is a good idea at last to add the main server socket (here `sock`) to the potential readers list, therefore the server being unable to accept incoming requests. Accepting requests mechanism is fairly simple: when a request is pending on the server socket, it is returned in the ready to read list and one may execute an `accept()` on it. The resulting client socket may be appended to any of the lists (may be on both, depending on protocol). When ***timeout*** parameter is 0 the `select()` takes the form of a poll, returning immediately.

4.2.3 Fork servers

Fork servers are the first choice in network design today[17]. One example will give you a birds-eye view: Apache, the most popular web server today is conceived as a fork server. Fork servers use the flexible process management implemented on most modern UNIX flavors. The basic idea is to get a new process for every incoming request, the parent process only listen, accept connections and "breed" children. Using the fork alternative you must accept its shortcomings: you must deal with processes at professional level, the server is much slower compared to a server designed with threads and is a bit more complicated. This implies that you must design a *fireman* function to collect the *zombies*, adding separate code for parent and for children and so on. In figure 4 is presented a diagram showing the algorithm for a fork server. In the figure is not presented the *fireman* function; usually it is called before calling `fork()`. The code for parent and for children is explicitly marked. The creation of a new process is signaled through a circled and dashed arrow. After the data exchange is over and both parts call `close()` on socket, the child is waited by the parent through a `wait()` or `waitpid()` system-call. This system-calls prevents the birth of so-called *zombies*. Is customary to create a list where to add all the children pids resulted after the calling `fork()`. This list is passed as parameter to the *fireman* function which take care of them. Are presented two fork methods:

- simple fork: this method simply call `fork()` then with an if-then-else, based on the resulting pid execute separate code for parent and for child. If the child is not waited it will became a zombie.

- separate fork: the parent forks and create the child one. This forks again and create child two then exit. Child one is waited in the parent. Because child two was left without a parent it is adopted by *init*, thus, when it[18] call `_exit()`, init will take care of this process not to became a zombie.

[17]At least in the UNIX world - but threaded servers are coming fast.
[18]Child two.

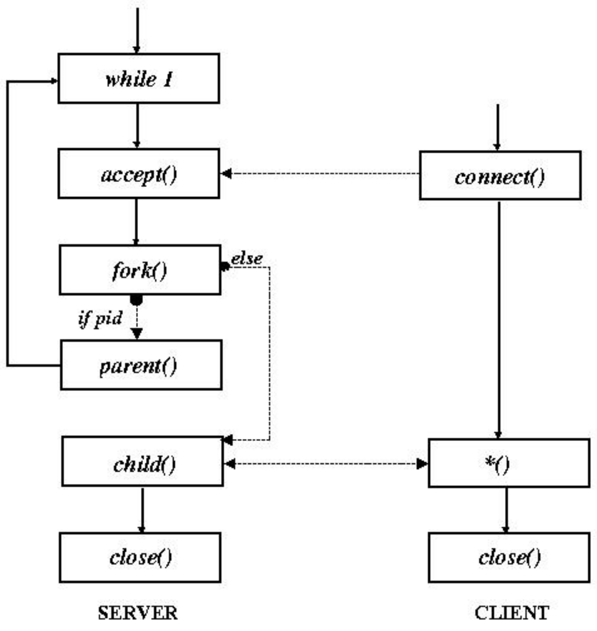

Figure 4: Fork server diagram

We present now code for a fork server with both methods. First method is usually expected. Again, `sock` is the server's socket.

```
import os, socket, syslog

children_list = []

def fireman(pids):
        while children_list:
                pid, status = os.waitpid(0, os.WNOHANG)
                if not pid: break
                pids.remove(pid)

def handler(socket):

        ...

...

while 1:
        clisock, addr = sock.accept()
        syslog.syslog('Incoming connection')
        fireman(children_list)
        pid = os.fork()

        if pid: #parent
                clisock.close()
                children_list.append(pid)

        if not pid: #child
                sock.close()
                handler(clisock)
                os._exit(0)
```

`handler` function is described in 4.2.1. Please see how separate code is executed for parent and for child. It is a good habit to close the server socket `sock` in the child and the client socket `clisock` in the parent. The `fireman` wait all the children from the specified list, passed as parameter. The second method may be implemented as follows:

```
...

 while 1:
        clisock, addr = sock.accept()
        syslog.syslog('Incoming connection')
        fireman(children_list)
        pid = os.fork()

        if pid: #parent
                clisock.close()
                children_list.append(pid)

        if not pid: #child1
                pid = os.fork()
                if pid: os._exit(0)
                if not pid: #child2
                        sock.close()
                        handler(clisock)
                        os._exit(0)
```

First child only forks then exit. It is waited on parent, so there must be no problem on it. The second child is adopted and then it enter in "normal" mode, executing the `handler` function.

4.3 Dealing with classes

This section describes how to do real object-oriented programming. Because it is a common topic, both referring server and clients, we decided to cover both aspects in this section. You must learn ho to construct classes that deals with networking functions, constructing connection objects and much more. The purpose is to design all this objects after a given *pattern* [3]. This will simplify the design and will add more reusability and power to your code. For more information about classes or using classes in Python, see [5] and [6].

4.3.1 Simple connection object

This section's goal is to design a simple connection object. The purpose of designing such an object is to ease the design of a given application. While you have already designed the proper components (objects) you may focus on designing the program at a higher level, at component level. This give you a precious level of abstraction, very useful in almost all the cases.

Let's suppose we want to create a TCP socket server in one object. This will allow connection from TCP clients on the specified port. First, there must be defined several methods to handle the possible situation that appear during runtime. Referring at figure 1, we see that the server must "launch" a socket and listen on it, then, when a connection is coming, accepting the connection then deal with it. The following methods are visible to the user: `OpenServer`, `EstablishConnection`, `CloseConnection`. `OpenServer` is a method that sets the hostname and the port on which the server is running. When a connection is incoming, `EstablishConnection` decide if it will be accepted based on a function passed as parameter. `CloseConnection` is self explanatory.

```
import socket

...

class TCPServer:

        def __init__(self): pass;

        def OpenServer(self, HOST='', PORT=7000):
                try:
                        sock = socket.socket(socket.AF_INET,\
                        socket.SOCK_STREAM);
                except socket.error:
                        print 'Server error: Unable to open socket'
                sock.bind((HOST, PORT));
                sock.listen(5);
                return sock;

       def EstablishConnection(self, sock, connection_handler):
                while 1:
                        client_socket, address  =  sock.accept();
                        print 'Connection from', `address`;
                        thread.start_new_thread(connection_handler,\
                        (client_socket,));

       def CloseConnection(self, sock):
                sock.close();
```

The `__init__()` function does nothing. When the server is started, the *TCPServer object*, through `OpenServer`, takes as parameters the hostname on where it run and the service portnumber. Alternatively, this may be done passing this parameters to the `__init__()` function. The method perform also am error checking on socket[19]. We are going to insist over the `EstablishConnection` method. It implement a threaded server and takes, among other parameters, a function name (here called `connection_handler`) used to handle the connection. A possible enhancement is to design this method to take as parameter a tuple (or a dictionary) containing the `connection_handler` parameters, which is supplied in our example with one parameter, generated inside the method which called it, the `client_socket`. There is no sense to insist over `CloseConnection` method. It was provided to assure a uniform interface to the TCPServer object. Future implementation (through inheritance or through

[19]Errors that may occur: an unsupported or unknown protocol or maybe, but rarely, a lack of memory.

object composition, see [3]) may find useful to specify additional features to this method, but now the ball is in your field.

4.3.2 Applying a design pattern

The *state* design pattern fits our purpose. The design will include five classes: one class, the `TCPServer`, will be the client (the user of the other four classes); `TCPConnection` is a class that realize a TCP connection: through `OpenConnection()`, `HandleConnection()` and `CloseConnection()` it manages the socket; also, are created three state classes, representing the states in which `TCPConnection` may be. This classes are inherited from an *abstract* class, called `TCPState`. Below is presented a diagram illustrating the relationship between the connection object and its states (represented as classes): Here is the code for the `TCPConnection`

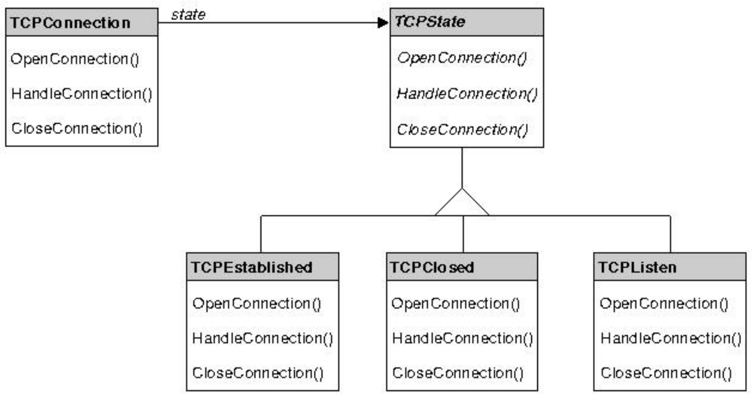

Figure 5: Designing a TCP connection with state pattern

class. The `TCPServer` class (please be careful: when the `TCPServer` example was given in section 4.3.1, it was a server; here we design only a connection class, which will further be used in designing a server or a server class) simply instantiate a connection object and perform some additional operations.

```
class TCPConnection:

        self.state = TCPClosed(self.sock)

        def __init__(self, host, port):
                self.host = host
                self.port = port
                self.sock = socket.socket(socket.AF_INET,\
                 socket.SOCK_STREAM)
                self.sock.bind((host, port))
                self.sock.listen(5)
                self.state = TCPListen(self.sock)

        def OpenConnection(self):
                if state is TCPClosed: self.state = TCPListen
                if state is TCPListen:
                         self.clisock = self.state.OpenConnection()
```

```
                                  self.state = TCPEstablished(self.clisock)

                def HandleConnection(self):
                        if state is TCPClosed: raise CloseStateError
                        if state is TCPListen: raise ListenStateError
                        if state is TCPEstablished:
                                  self.state.HandleConnection(some_handler)
                                  self.state = TCPListen(self.sock)

                def CloseConnection():
                        self.state = TCPClosed(self.sock)
                        self.state.CloseConnection()
```

As you can see in the example presented above, the behavior of the connection object depend on its state. The methods cannot be used in any order. When the object is instantiated, the `__init__()` performs the classic three steps initialization on server socket and brings the object to *listen* state. When in *listen* state, the client[20] may call just `OpenConnection()` and `CloseConnection()` methods. After the connection is accepted (via `OpenConnection` - also see the code below, describing the *states* classes) the object is in the *established* state, when it deals with the other end of the connection. Now, the client may use `handleConnection()` to run the protocol. By default, the object returns itself to *listen* state and then `OpenConection` must be called again to realize a new connection. When `CloseConnection()` is called, the server socket is closed and further operation on it will be discarded, thus the object is no longer of interest. This is a very basic approach, providing just a skeleton for a real `TCPConnection` object. `raise` statements are used in signaling the client over an improper use of the object's methods. This errors are constructed through inheritance from the `Exception` class. Bellow are described the four state classes:

```
        class TCPState:

               def  __init__(self, sock):
                        self.sock = sock

               def OpenConnection(self): pass

               def HandleConnection(self): pass

               def CloseConnection(self): pass


        class TCPClosed(TCPState):

               def CloseConnection(self, sock):
                        sock.close()

        class TCPListen(TCPState):

               def OpenConnection(self, sock):
```

[20]The code that instantiate and use a connection object.

```
                        clisock , addr = sock.accept()
                        return clisock

        class TCPEstablished(TCPState):

                def HandleConnection(self, handler):
                        thread.start_new_thread(handler,\
                         (self.sock, ))
```

All the concrete classes are inherited from the `TCPState` abstract class. This class provides a uniform interface for the other three. Please see that only the necessary methods were overridden. When another state is wanted, the state class is inherited from `TCPState`, methods are redefined and proper code is added to the `TCPConnection` class to include this state as well.

4.4 Advanced aspects concerning clients

In majority of this section 4 we focused on server design. This is true, because a server design is far much complicated related to client design. A server is running as *demon* in most of the cases, dealing with many clients, so it must be fast, reliable and must not be prone to errors. A client is maintaining a single connection with the server (this for the huge majority of clients), therefore the pure networking code[21] is simpler and is not involving threading, selecting or forking. However, is recommended to apply object-oriented programming in this case or even a design pattern (the presented *state* design pattern is easy applicable to a client), but no fork, thread or select. Life is not so hard when one have to design the networking part of a client.

[21]We not discuss about he protocol here.

5 HTTP protocol

HTTP is, beside electronic mail, the most popular protocol on Internet today. Please refer [8] and [9] for detailed informations about this protocol. HTTP is implemented in web servers like Apache or in browsers like Netscape or Mosaic. Python provides a few modules to access this protocol. We are mentioning a few of them: `webbrowser`, `httplib`, `cgi`, `urllib`, `urllib2`, `urlparse` and three modules for already build HTTP servers: `BaseHTTPServer`, `SimpleHTTPServer` and `CGIHTTPServer`. Is provided the `Cookie` module for cookies support.

The `webbrowser` module offers a way in displaying and controlling *HTML* documents via a web browser (this web browser is system dependent). Using the `cgi` module the user may be able to write complex *CGI*[22] scripts, replacing "standard" CGI scripting languages as Perl. `httplib` is a module implementing the client side of the HTTP protocol. The `urllib` is module that provides a high-level interface for retrieving web-based content across World Wide Web. This module (and `urllib2` too) uses the previous mentioned module, `httplib`. As it's name says, the `urlparse` module is used for parsing URLs into components.

5.1 CGI module

CGI (Common Gateway Interface) is a mean through one may send information from a HTML page (i.e. using the `<form>...</form>` structures) to a program or script via a web server. The program receive this information in a suitable form – depending on web server – and perform a task: add user to a database, check a password, generate a HTML page in response or any other action. Traditionally, this scripts live in the server's special directory 'cgi-bin', whom path must be set in the server configuration file.

5.1.1 Build a simple CGI script

When the user is calling a CGI script, a special string is passed to the server specifying the script, parameters (fed from a form or an URL with query string) and other things. The server is passing all this stuff to the mentioned script which is supposed to understand what's happening, process the query and execute an action. If the script is replying something to client (all the `print` statements, the script output respectively), it will send the data chunk through the web server. Now is clear why this interface is called *Common Gateway* – every transaction between a client and a CGI script is made using the web server, more specific, using the interface between the server and the script.

A simple script which simply displays a message on the screen is presented bellow:

```
print 'Content-Type: text/html'
print

print 'Hello World!'
```

First two lines are mandatory: these lines tells the browser that the following content must be interpreted as HTML language. The `print` statement is used to create a blank line. Now, when the header is prepared, we may send to the client any output. If the output will contain HTML tags, they will be interpreted as a normal web page. Save this script in the 'cgi-bin' directory and set the permissions for it as world readable and executable. Make the appropriate modifications to the server's configuration file[23]. Calling 'http://your.server.com/cgi-

[22] Common Gateway Interface

[23] If you are running Apache on Linux please see the last section on this topic.

bin/your-script.py' you should see a small 'Hello World!' placed in the top left corner of your browser.

Let's modify the script to display the same message centered and with a reasonable size. We are going to use the `<title>`,`<h1>` and `<center>` HTML tags:

```
print 'Content-Type: text/html'
print

print '<title>CGI Test Page</title>'
print '<center><h1>Hello World!</h1></center>'
```

Now the road is open. Try to generate your own HTML pages with CGI scripts. You can insert pictures, Java Script or VBScript in you pages. If the browser supports them, no problem.

5.1.2 Using CGI module

This module is useful when one might want to read information submitted by a user through a form. The module name is (well!) `cgi` and must be used with an `import cgi` statement (see [4]). The module defines a few classes (some for backward compatibility) that can handle the web server's output in order to make it accessible to programmer. The recommended class is `FieldStorage` class though in some cases `MiniFieldStorage` class is used. The class instance is a dictionary-like object and we can apply the dictionary methods. Bellow is the HTML page and the Python code to extract the informations from the form fields:

```
<html>
<form name='myform' method='POST' action='/cgi-bin/mycgi.py'>
<input type='text' name='yourname' size='30'>
<input type='textarea' name='comment' width='30' heigth='20'>
<input type='submit'>
</form>
</html>
```

```
import cgi

FormOK = 0;
MyForm = cgi.FieldStorage();
if MyForm.has_key('yourname') and MyForm.has_key('comment' ):
        FormOK = 1

print 'Content-Type: text/html'
print

if FormOK:
        print '<p>Your name: ', Myform['yourname'].value, '<br>'
        print '<p>You comment: ', MyForm['comment'].value
else:
        print 'Error: fields have not been filled.'
```

As you can see in the example, `MyForm` is an instance of the `FieldStorage` class and will contain the name-value pairs described in the `<form>` structure in the HTML page: the `text field` with name=*yourname* and `textarea` with name=*comment*. Applying `has_key` method to `MyForm` object one may be able to check if a particular field have been filled and then, with `MyForm[''name''].value` can check the value of that field[24]. Once the values have been assigned to some internal variables, the programmer may use them as any trivial variable. We present now a small example that read a form submitted by a user (containing the user name, address and phone number) and write all the values in a file.

```
<html>
<form name='personal_data' action='/cgi-bin/my-cgi.py' method='POST'>
<p>Insert your name :&nbsp;<input type='text' name='name' size='30'>
<p>Insert your address:&nbsp;<input type='text' name='address' size='30'>
<p>Phone:&nbsp;<input type='text' name='tel' size='15'>
<p><input type='submit'>
</form>
</html>
```

```
import cgi

MyForm = cgi.FieldStorage()

# we suppose that all the fields have been filled
# and there is no need to check them

# extracting the values from the form
Name = MyForm['name'].value
Address = MyForm['address'].value
Phone = myForm['tel'].value

# opening the file for writing
File = open('/my/path/to/file', 'r+')

File.write('Name: ' + Name)
File.write('Address: ' + Address)
File.write('Phone: ' + Phone)

File.close

print 'Content-Type: text/html'
print
print 'Form submited...ok.'
```

5.1.3 Configuring Apache on Linux for using with CGI scripts

This section is a brief troubleshooter for those which have some difficulties on setting up a script. If you are using Apache on Linux or whatever *NIX system you should see an error

[24]The value of a text field or textarea is a string, the value of a pop-up box is the selected value and the values of radio buttons, checkboxes are boolean values.

with the code 500 – Internal Server Error. Another error is File Not Found but I think that it's too trivial to discuss this here. If you receive a 500 error code here is a list of 'to do' in order to make you script working:

- Check if the interpreter is specified on the first line of you script:

  ```
  #!/usr/bin/env python
  ```

- Please check if the interpreter is world readable and executable.
- Check if the files accessed by the script are readable and executable by 'others' because, due to security reasons, Apache is running you script as user 'nobody'.
- Make sure that in *httpd.conf* exist the following lines:

  ```
  ScriptAlias /cgi-bin/ "/var/www/cgi-bin/"
  ```

  and

  ```
  <Directory "/var/www/cgi-bin">
    AllowOverride None
    Options ExecCGI
    Order allow,deny
    Allow from all
  </Directory>
  ```

  The path to your 'cgi-bin' directory may vary, depending on how you have set the *DocumentRoot* option in the same file.

Also it's a good practice to run the script into a console, the interpreter printing a traceback in case any error occurs, this being the simplest way to correct syntax errors. For any other solutions and tips please refer [4].

6 Common protocols

6.1 Designing Telnet applications

Telnet is a simple protocol that emulate a terminal over a networked link. With telnet you can access a foreign machine on a specified port and then send and receive data over the established link without any restrictions. Is possible to access via telnet clients a web server, a POP3 server, a mail server and so on. The purpose of the `telnetlib` is to offer a simple way to design a telnet client. This client may be used in further applications as the support for communication between two machines, using some protocol. To understand better the previous affirmations, please consider the next two examples:

```
$ telnet localhost

 Trying 127.0.0.1...
 Connected to localhost.
 Escape character is '^]'.

 login:
```

This is a trivial telnet session. The remote machine will request you a user name and password (in case that the telnet service is up and running on that host and you are allowed to access it) and then will drop you in a shell. The main disadvantage: the communication is not confidential: telnet is transmitting everything in clear text. Let's consider the second example:

```
$ telnet localhost 80

 Trying 127.0.0.1...
 Connected to localhost.
 Escape character is '^]'.


 GET /
 ...
```

In this case we try to access a web server (port 80) on some location. To receive a web page GET method is used followed by the name of the page (in this case / is the root page, commonly named index.html). We did not specified the complete GET request, mentioning the protocol and other things (if someone is interested on this topic check [9]).

Conclusion: the same client but different protocols. When we used the GET method in the second example, the telnet client was our *network support* and the protocol was emulated typing it. The core idea is to use a telnet client as support for the networking part while the user is writing (or using) it's own protocol over this link.

Python is implementing the Telnet protocol with the `telnetlib` module which offers a class called `Telnet(`*`host, port`*`)`. The instance of this class is a *telnet object* with methods for opening and closing a connection, reading and writing data, checking the status of the link and debugging the script. The previous examples are translated now in Python code using the `telnetlib`:

```
import telnetlib

telnet = telnetlib.Telnet('localhost')
telnet.read_until('login: ')
user = raw_input('Login: ')
telnet.write(user + '\n')
telnet.read_until('Password: ')
passwd = raw_input('Password: ')
telnet.write(passwd + '\n')

# now the user may send commands
# and read the proper outputs
```

A telnet object is created (in this case `telnet`) and the output of the server is read until the string 'login: ' is reached. Now the server is waiting for the login name, which is supplied reading it from the keyboard and then passing it through a variable to the `write()` method. The same algorithm is used for submitting the password. One the user is authenticated is possible to send commands and read the output send by the server. The second example is a simple retrieval of a web page from a certain address, save it to a file and display it using the `webbrowser` module.

```
import telnetlib
import webbrowser
```

```
telnet = telnetlib.Telnet('localhost', 80)
telnet.read_until('^].\n')

telnet.write('GET /')
html_page = telnet.read_all()

html_file = open('/some/where/file', 'w')
html_file.write(html_page)
html_file.close()

webbrowser.open('/some/where/file')
```

The code above will read the index page from the specified location, will save it to a file and then, through the `webbrowser` module will display it into a browser[25]. For a complete list of commands described in the `telnetlib` module please refer to [4].

6.2 File Transfer Protocol

Python currently implement this protocol using the `ftplib` module. This module is very useful when one may want to write a script for automatic retrieval of files or mirroring ftp sites on a daily (hourly, weekly) basis. The module is a simple translation in Python code, as methods of the `FTP` class, of the usual commands during a ftp session: get, put, cd, ls etc.

As was stated before, this module implement the `FTP` class. To test a ftp session, open a terminal, enter the Python interactive shell and type commands as bellow:

```
>>> import ftplib
>>> ftp_session = ftplib.FTP('ftp.some.host.com')
>>> ftp_session.login()
'230 Guest login ok, access restriction apply.'
>>> ftp_session.dir()

....

>>>ftp_session.close()
```

The `ftp_session` is the instance of the `FTP` class. There are currently two methods implemented to open a connection to a given server: specifying the host name and port when instantiating the *ftp object* or applying the `open(`*`host, port`*`)` method for an unconnected ftp object.

```
>>> import ftplib
>>> ftp_session = ftplib.FTP()
>>> ftp_session.open('ftp.some.host.com')

....
```

After the connection is realized the user must submit it's login name and password (in this case was used the 'anonymous' user name and user@host password). The server will send back a string stating if it accepts or reject the user's login and password. After the user has successfully entered the site, the following methods may be useful:

[25]What browser is dependent on the configuration of you system.

- `pwd()` for listing the pathname of the current directory, `cwd(`*`pathname`*`)` for changing the directory and `dir(`*`argument[,...]`*`)` to list the content of the current (or *`argument`*) directory, methods for navigation and browsing through the directories.
- `rename(`*`fromname, toname`*`)`, `delete(`*`filename`*`)`, `mkd(`*`pathname`*`)`, `rmd(`*`pathname`*`)` and `size(`*`filename`*`)` methods (all of them are self-explanatory) to manage files and directories inside an ftp site.
- `retrbinary(`*`command, callback[, maxblocksize[, rest]]`*`)` method used to retrieve a binary file (or to retrieve a text file in binary mode). *`command`* is a 'RETR filename' (see [4]), *`callback`* is a function called for each data block received. To receive a file or the output of a command in text mode `retrlines(`*`command[, callback]`*`)` method is provided.

Many others commands are provided and you may find a complete description in *Python Library Reference* [4].

6.3 SMTP protocol

The Simple Mail Transfer Protocol described in [11], is a protocol that may be used to send email to any machine on the Internet with a SMTP daemon listening on port 25 (default port for SMTP service). Python implements the client side of this protocol (one may only send but not receive mail) in the `smtplib` module. As for `telnetlib` and `ftplib` modules, is defined a `SMTP(`*`[host[, port]]`*`)` class which instance, an *SMTP object*, has several methods that emulate SMTP. The module also offers a rich set of exception, useful in different situations. An exhaustive list of all the exceptions defined in this module is presented bellow for the sake of completeness:

- `SMTPException`
- `SMTPServerDisconnected`
- `SMTPResponseException`
- `SMTPSenderRefused`
- `SMTPRecipientsRefused`
- `SMTPDataError`
- `SMTPConnectionError`
- `SMTPHeloError`

Let's see in a short example, how do we use the methods available in this module to send a simple test mail to a remote machine named 'foreign.host.com' for user 'xxx':

```
import smtplib, sys

s = smtplib.SMTP()
s.set_debuglevel(1)
s.connect('foreign.host.com') #default port 25
s.helo('foreign.host.com')
```

```
        ret_code = s.verify('xxx@foreign.host.com')
        if ret_code[0] == 550:
                print 'Error, no such user.'
                sys.exit(1)

        s.sendmail('user@localhost', 'xxx@forein.host.com', 'Test message!')
        s.quit()
```

The script begin by instantiation the `SMTP()` class, here without any parameters. *s.connect ('foreign.host.com')* is used to connect the client to the remote SMTP server. After the connection is realized our client will send an `HELO` message. Here is useful to check if the server accepts this message (with a `try...` clause using the `SMTPHeloError` exception). There is no reason for the server to reject the `HELO` greeting but is useful to make such a error checking. With `s.verify('xxx@foreig.host.com')` the script checks if the receiver address exists on that server. `ret_code` is a variable (list) which will contain the server reply after the instruction is executed, containing two elements: an integer, meaning the reply code and a string for human consumption. In case the user exist on that machine, the code is 250 and in case of error we receive a 550. The most important method is `sendmail(`*`fromaddr, toaddr, message[, mail_options, recipient_options]`*`)`, which is the real sender of the message. Because we are such polite people, we are doing a `quit()` before effectively exiting the script. This will assure that everything it's alright when leaving the SMTP session.

7 TO DOs

This is the version 0.00 (as stated in the title) of *Network Programming with Python*, so it is very young and very incomplete. The purpose for the first version was fulfilled but next versions will be richer in information. I plan to append a subsection (included in section 'HTTP protocol') that will teach how to design a HTTP server using the predefined classes and one programmed from ground up. I want to add more information about miscellaneous module related to HTTP protocol (all mentioned in the beginning of the 'HTTP protocol' section). Future versions will contain something about designing some clients for POP3 and IMAP4 protocols: if I will have enough time there will be provided a complete example of such a client, including a graphical interface. The author is interested in any feedback with suggestions, error reports, things that may be added. Please contact me at: *seba@trinity.ee.utt.ro*.

References

[1] S. Leffler et al: *An Advanced 4.3BSD Interprocess Communication Tutorial*

[2] A. Tanenbaum: *Computer Networks* 3^{r}*ded.*, Prentice-Hall, 1996

[3] E. Gramma et al: *Design patterns - Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1994

[4] G. van Rossum: *Python Library Reference*

[5] G. van Rossum: *Pyhton Tutorial*

[6] G. van Rossum: *Pyhton Language Reference*

[7] G. McMillan: *Socket Programming HOWTO*

[8] ***: *HTTP 1.0 - RFC1945*

[9] ***: *HTTP 1.1 - RFC2068*

[10] ***: *Telnet Protocol Specifications - RFC854*

[11] ***: *Simple Mail Treanfer Protocol - RFC821*

[12] ***: *SMTP Service Extensions - RFC1869*