Python
Programowanie Sieciowe

---

by Sebastian V. Tiponut
Technical University Timisoara

Wersja 0.00, 16. Lipiec 2001

Tłumaczenie K.remik
OliverQ Translators
Listopad/Grudzień 2006

Zawartość

1 Wprowadzenie

Programowanie sieciowe to słowo – wytrych w świecie oprogramowania. Widzimy ,że rynek zapełni się lawinowo zorientowanymi sieciowo aplikacjami.takimi jak serwery baz danych, gry, servlety i aplety Javy, skrypty CGI, różne klienty dla różnych protokołów , a przykłady można mnożyć. Dzisiaj, więcej niż połowa aplikacji trafiających na rynek jest zorientowana sieciowo.Przesyłanie danych między dwoma maszynami (w sieci lokalnej lub Internecie) nie jest żadną ciekawostką, ale codzienną rzeczywistością. "Sieć to komputer" jak mówi motto firmy Sun Microsystem i to jest prawda. Komputer nie jest niczym więcej jak oddzielną jednostką, komunikującą się tylko z udziałem człowieka, ale jako część dużego systemu - sieci, powiązaną poprzez łącze danych z tysiącami innych maszyn.

Ten tekst przedstawia możliwe sposoby projektowania sieciowo zorientowanych aplikacji przy użyciu Pythona. Ponieważ autor jest miłośnikiem Linuksa, przykłady tu zawarte są związane z Linuksem więc przepraszam wszystkich użytkowników (fanów?) Windows lub MacOS za niedogodności przy czytaniu tego tekstu. Przy odrobinie wysiłku, przykłady można przenieść na inne nie – UNIX'owe systemy operacyjne. Przedstawię szybko strukturę tego tekstu, pierwsze cztery sekcje dotyczą projektu *elementarnego* – na poziomie gniazda – aplikacji sieciowej. Pozostałe sekcje traktują o określonych protokołach takich jak *http, ftp, telnet* lub *smtp.* Sekcja dotycząca *http* będzie zawierała podsekcję o pisaniu skryptów CGI I stosowaniu modułu CGI.

Posuwając się dalej, w bardziej konkretne tematy, przejdziemy do analizy możliwości programowania sieciowego jakimi dysponuje Python. Surowa obsługa sieci jest implementowana w Pythonie poprzez moduł gniazda, moduł ten stanowi najczęstszy system wywołań, funkcji i stałych definiowanych przez urządzenia 4.3BSD Interprocess Communication, zimplementowane w stylu zorientowanym obiektowo. Python oferuje prosty interfejs (dużo prostszy niż odpowiednia implementacja C, chociaż oparty na niej) dla właściwego stworzenia i stosowania gniazda. Przede wszystkim jest zdefiniowana funkcja socket() zwracająca *obiekt gniazda.* Gniazdo ma kilka metod, odpowiadających parze z C sys/socket.h, takie jak bind(), connect(), listen() lub accept (). Programiści obyci z gniazdami w języku C bardzo łatwo odnajdą się w dużo łatwiejszej do zastosowania implementacji gniazd w Pythonie. Python eliminuje zniechęcające zadanie wypełniania struktur takich jak sockaddrin lub hostent i łatwo stosujej się w nim poprzednio wspomniane metody lub funkcje – przekazywanie parametrów i wywołanie funkcji są łatwe do obsłużenia. Wprowadzono również sieciowo zorientowane funkcje: gethostbyname(), getprotobyname () lub funkcje konwersji ntohl (), htons (), użyteczne kiedy konwertujemy liczby całkowite do i z formatu sieciowego.Moduł dostarcza stałych takich jak SOMAXCONN, INADDR_*, używanych w funkcjach gesockopt () lub setsockopt (). Aby znaleźć kompletną listę powyżej wspomnianych metod sprawdź dokumentację swojego UNIX'a w implementacji gniazda.

Python dostarcza poza gniazdem , dodatkowych modułów (w zasadzie cały jest z nich zbudowany) obsługując najpopularniejsze protokoły sieciowe z poziomu użytkownika. Na przykład możemy znaleźć użyteczne moduły takie jak httplib, ftplib, telnetlib, smtplib. Ma również zaimplementowaną obsługę skryptowania CGI poprzez moduł cgi, moduł dla parsowania URL'i, klas opisujących serwery sieciowe i wiele innych. Moduły te określają implementację dobrze znanych protokołów, wiec użytkownik nie będzie miał z nimi żadnych problemów.Autor ma nadzieję,że czytelnik będzie się dobrze bawił odkrywając możliwości sieciowe Pythona, i będzie ich używał w nowy i bardziej ekscytujący sposób.

Ponieważ wszystkie przykłady są napisane w Pythonie, zakładam ,że czytelnik ma jakieś pojęcie o tym języku programowania.

2 Podstawy używania gniazd

Gniazdo jest podstawową strukturą dla komunikacji miedzy procesami. Gniazdo jest definiowane jako *"punkt końcowy komunikacji do którego nazwy może być ograniczona"*. Implementacja 4.3BSD definiuje trzy domeny komunikacji dla gniazda: domenę UNIX dla systemowej komunikacji pomiędzy procesami; domenę Internetu dla procesów komunikacji przez protokół TCP(UDP)/IP; domenę NS używaną przez proces komunikacyjny starym protokołem komunikacyjnym Xerox'a.

Python stosuje tylko pierwsze dwie domeny komunikacji: domeny UNIX'a i Internetowa, odpowiednio AF_UNIX i AF_INET. Adresy domenowe UNIX'a są przedstawiane jako ciągi ,nazywając lokalne ścieżki dostępu,na przykład: /tmp/sock. To może być gniazdem stworzonym przez lokalny proces, lub możliwe, stworzony przez obcy proces. Adresy domenowe Internetu są przedstawiane jako *(host, port)* gdzie *host* to ciąg przedstawiający poprawną nazwę hosta, powiedzmy matrix.ee.utt.ro lub adres IP w dziesiętnej notacji a *port* jest poprawnym portem pomiędzy 1 i 65535. Można poczynić pewne spostrzeżenie: zamiast kwalifikowanej nazwy hosta lub poprawnego adresu IP są dwie specjalne formy: pusty ciąg jest używany zamiast INADDR_ANY i ciąg '<broadcast>' zamiast INADDR_BROADCAST.

Python oferuje wszystkie pięć typów gniazd zdefiniowanych w implementacji 4.3BSD IPC. Dwa wydają się być generalnie używane w dużej mierze w nowych aplikacjach. Gniazdo *stream* jest gniazdem zorientowanym na połączenia, i wykorzystuje obsługę komunikacji w protokole TCP zakładając dwukierunkowy, niezawodny, sekwencyjny i niezduplikowany przepływ danych. Gniazdo *datagram* jest bezpołączeniowym gniazdem komunikacyjnym, obsługiwanym przez protokół UDP. Oferuje ono dwukierunkowy przepływ danych, bez tej niezawodności, sekwencyjności lub niezduplikowania. Proces odbioru sekwencji *datagramów* może wykryć zduplikowane wiadomości, lub ,prawdopodobnie, w innej kolejności niż pakiet został wysłany. Typy gniazd *pierwotnych, sekwencyjnych* i *niezawodnych dostarczania wiadomości* są rzadko używane. Typ gniazda *pierwotnego* jest potrzebny kiedy jedna aplikacja może żądać dostępu do najbardziej skrytych zasobów dostarczonych przez implementację gniazda. Nasz dokument skupia się na gniazdach *stream* i *datagram*.

2.1 Tworzenie gniazda

Gniazdo jest tworzone przez wywołanie *socket (family, type* [, *proto*]); *family* to jeden z powyżej wspomnianych adresów: AF_UNIX i AF_INET, *type* jest przedstawiany przez następujące stałe: SOCK_STREAM, SOCK_DGRAM, SOCK_RAW, SOCK_SEQPACKET i SOCK_RDM. Argument *proto* jest opcjonalny i domyślnie wynosi 0. Zobaczymy,że funkcja socket () zwraca gniazdo w określonej domenie z określonym typem. Ponieważ stałe wspomniane powyżej są zawarte w module gniazda , każda z nich musi być użyta z notacją socket .STAŁA. Bez zrobienia tego, interpreter wygeneruje błąd. Aby stworzyć gniazdo *stream* w domenie Internet, użyjemy następującej linii:

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

Zastępując socket .SOCK_STREAM na socket .SOCK_DGRAM tworzymy gniazdo *datagram* w domenie Internet. Poniższe wywołanie stworzy gniazdo *stream* w domenie UNIX:

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

Jak narazie omówiliśmy uzyskiwanie gniazd różnych typów w różnych domenach komunikacyjnych.

.

2.2 Połączenie gniazda a transfer danych

Serwer z naszego punktu widzenia jest procesem, który nasłuchuje na określonym porcie.Możemy wywołać skojarzony port procesu jako usługę. Kiedy inny proces chce użyć serwera lub użyć określonej usługi, musi sam się połączyć z adresem i numerem portu określonym przez serwer. Jest to wykonywane przez wywołanie metody gniazda **connect (address**), gdzie ***address*** to para ***(host, port)*** w domenie Internet a ściezka dostępu w domenie UNIX. Kiedy używamy domeny Internet połączenie jest realizowane następującym kodem:

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

podczas gdy w domenie UNIX ,

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

Jeśli usługa jest niedostępna lub serwer nie chce rozmawiać z procesem klienta zostanie wygenerowany socket.error-(111, 'Connection refused').Jednakowoż, po nawiązaniu połączeniu z żądanym serwerem, dana jest wysyłana i odbierana przez metody **send *(buffer* [*, flags*])** i **recv *(buffer* [*, flags*]**). Metody te przyjmują obowiązkowo jako parametr rozmiar bufora w bajtach i opcjonalne ***flags***; po opis znaczenia flag ,zajrzyj do podręcznika UNIXA, na stronę z odpowiednią funkcją.

2.3 Wiązanie nazwy z gniazdem

Gniazdo, po stworzeniu, jest bezimienne, chociaż ma przypisany deskryptor Zanim jednak zostanie użyty, musi być powiązany z właściwym ***adresem*** ponieważ jest to jedyny sposób aby mógł się do niego odnieść obcy proces. Metoda **bind *(address)*** jest używana do "nazwania" gniazda. Znaczenie ***address*** wyjaśniono powyżej.Poniższe wywołanie powiąże gniazdo w domenie Intrenet z ***adresem*** złożonym z nazwy hosta ***localhost*** i portu o numerzer *8000:*

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

Proszę uważać przy wpisywaniu: w rzeczywistości są dwie pary nawiasów.Inaczej interpreter wyświetli TypeError. Cel tych dwóch par nawiasów jest prosty: ***address*** jest krotką zawierającą ciąg i liczbę całkowitą. Nazwa hosta musi być właściwie dobrana, najlepszą metoda jest zastosowanie podprogramu **gethostname()** aby zapewnić niezależność i przenośność hosta. Stworzenie gniazda w domenie UNIX użyj ***adresu*** jako pojedynczego ciągu, nazywający lokalną ścieżkę dostępu:

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

To stworzy plik '/tmp/sock' , który będzie używany do komunikacji pomiędzy procesami serwera a klientem. Użytkownik musi mieć uprawnienia do odczytu/zapisu aby określić katalog, w którym stworzone będzie gniazdo, a sam plik musi być usunięty kiedy tylko nie będzie dłużej potrzebny.

2.4 Nasłuchiwanie i akceptacja połączeń

Kiedy już mamy gniazdo o poprawnej nazwie, kolejnym krokiem jest wywołanie metody **listen *(queue*)**.Instruuje on gniazdo o pasywnym nasłuchiwaniu na porcie ***port.*** **listen()** pobierając jako parametr

.

liczbę całkowitą przedstawiającą maksymalną kolejkę połączenia. Ten argument powinien mieć wartość przynajmniej 1 a maksimum, w zależności od systemu, 5. Od teraz mamy gniazdo z właściwym adresem granicznym. Kiedy przyszło żądanie połączenia , serwer decyduje czy będzie ono zaakceptowane czy nie. Akceptacja połączenia jest wykonywana poprzez metodę **accept()**. Nie pobiera żadnych parametrów ale wraca krotkę ***(clientsocket, address)*** gdzie ***clientsocket*** jest nowym gniazdem serwera używanym do komunikowania się z klientem a **address** jest adresem klienta. Zazwyczaj **accept** () jest zablokowany dopóki połączenie jest realizowane.. To zachowanie może być pominięte przez uruchomienie metody w oddzielnym wątku, zbierającym deskryptory nowo stworzonych gniazd na liście a następnie je przetworzyć. Czasami serwer może zrobić coś takiego. Powyżej wspomniane metody są używane jak poniżej:

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

Kod instruuje gniazdo aby nasłuchiwało z kolejką pięciu połączeń i aby akceptowało wszystkie przychodzące "wywołania". Jak widzisz, **accept**() zwraca nowe gniazdo, które będzie używana dalej do wymiany danych. Używając łańcucha *bind-listen-accept* tworzymy serwery TCP. Pamiętaj , gniazdo TCP jest połączeniowe; kiedy klient chce pogadać z określonym serwerem musi połączyć się, poczekać dopóki serwer nie zaakceptuje połączenia, wymienić dane i zamknąć. To jest modelowa rozmowa telefoniczna: klient wybiera numer, czeka aż druga strona nawiąże połączenie, rozmawia potem kończy.

2.5 Gniazda UDP

Wybraliśmy oddzielne gniazda bezpołączeniowe ponieważ są one mniej powszechne w dniach projektu klient//serwer. Gniazdo *datagram* charakteryzuje bezpołączeniowość i symetryczna wymiana wiadomości. Serwer i klient wymieniają *pakiety danych* a nie strumień danych, pakiety przepływają pomiędzy klientem a serwerem oddzielnie. Połączenie UDP przypomina system pocztowy: każda wiadomość jest zamknięta w kopercie i odbierana jako oddzielna część. Duże wiadomości mogą być dzielone na wiele części, każda dostarczana oddzielnie (nie w takim samym porządku,duplikowane itd). Odbiorca musi połączyć wszystko w jedną wiadomość.

Serwer ma metodę **bind()** używaną do dopisania właściwej nazwy i portu. Nie ma metod **listen()** i **accept**() ponieważ serwer nie nasłuchuje i nie akceptuje połączenia. Przede wszystkim tworzymy Skrzynkę Pocztową, gdzie jest możliwe odebranie wiadomości od procesów klienta. Klienci tylko wysyłają pakiety, w których zawarty jest już dana i adres.

Pakiety danych są wysyłane i odbierane przez metody **sendto(data, address)** i **recvfrom(buf*fer* [, flags])**. Pierwsza metoda pobiera parametry jako ciąg i adres serwera ,jak wspomniano powyżej w **connect()** i **bind().** Ponieważ jest określony zdalny koniec gniazda ,nie ma potrzeby go podłączać.Druga metoda jest podobna do **recv().**

2.6 Zamknięcie gniazda

Kiedy gniazdo nie jest już dłużej potrzebne musi być zamknięte metodą **close** (). Kiedy użytkownik nie jest zainteresowany oczekiwaniem na dane ,przed zamknięciem gniazda może wykonać *shutdown*. Metoda to **shutdown *(how),***gdzie ***how*** to: 0 jeśli brak przychodzących danych do akceptacji, 1 odrzucające wysylanie danych i wartość 2 zapobiegająca zarówno wysyłaniu jak i odbieraniu danych. Pamiętaj: zawsze zamykaj gniazdo po jego wykorzystaniu.

.

2.7 Używanie funkcji dostarczonych w module socket

Moduł **socket** zawiera pewne użyteczne funkcje w projektowaniu sieciowym. Te funkcje są powiązane z bibliotekami *resolver* libraries, plikami odwzorowanymi */etc/services* lub */etc/protocols* lub konwersją liczb

2.7.1 Funkcje oparte o bibliotekę resolver

Są trzy funkcje używające BIND8 lub możesz mieć niezależny resolver. Te funkcje zazwyczaj konwertują nazwę hosta na adres IP lub adres IP na nazwę hosta. Jedna funkcja nie jest powiązana z resolverem, **gethostname** (), ta funkcja zwraca ciąg zawierający nazwę maszyny na której jest uruchomiony skrypt. Po prostu odczytuje (przez odpowiednią funkcję w C) plik */etc/HOSTNAME*. Możemy użyć (przed wersją 2.0) funkcji **getfqdn**(***hos tname***), która zwraca "W Pełni Kwalifikowaną Nazwę Domeny" dla ***hostname*** . Jeśli parametr jest niewiadomy, zwróci ona WPKND hosta lokalnego. Dwie funkcje są używane do tłumaczenia nazwy hostów na poprawne adresy IP: **gethostbyname***(hostname* **)** i **gethostbyname_ex(hostname**). Obie funkcje pobierają jako obowiązkowy parametr poprawną nazwę hosta. Pierwsza funkcja zwraca adres IP w notacji dziesiętnej a druga *(ex* od "extended (rozszerzona)") krotkę *(hostname, aliaslist, ipaddrlist).* Ostatnia funkcja tu omawiana **gethostbyaddr(ipaddr)** zwraca nazwę hosta kiedy podany jest adres IP.

2.7.2 Funkcje związane z usługami

*/etc/services*jest plikiem,który odwzorowuje usługi do numerów portów. Na przykład, usługa http jest odwzorowana do portu 80, usługa ftp do portu 21 a ssh do portu 22. **getservbyname** ***(servname***) jest funkcją, która tłumaczy nazwę usługi na poprawny numer portu, w oparciu o plik prezentowany powyżej.Metoda zakłada niezależność platformy (lub nawet niezleżność komputera) chociaż ta sama usług może nie być odwzorowana dokładnie do tego samego portu.

Kiedy chcemy przetłumaczyć protokół na numer odpowiedni dla przekazania trzeciego argumentu do funkcji **socket** () użyjemy **getprotobyname** ***(protoname***). Tłumaczy ona, w oparciu o plik */etc/protocols*, nazwę protokołu, powiedzmy GGP lub ICMP, na odpowiedni numer.

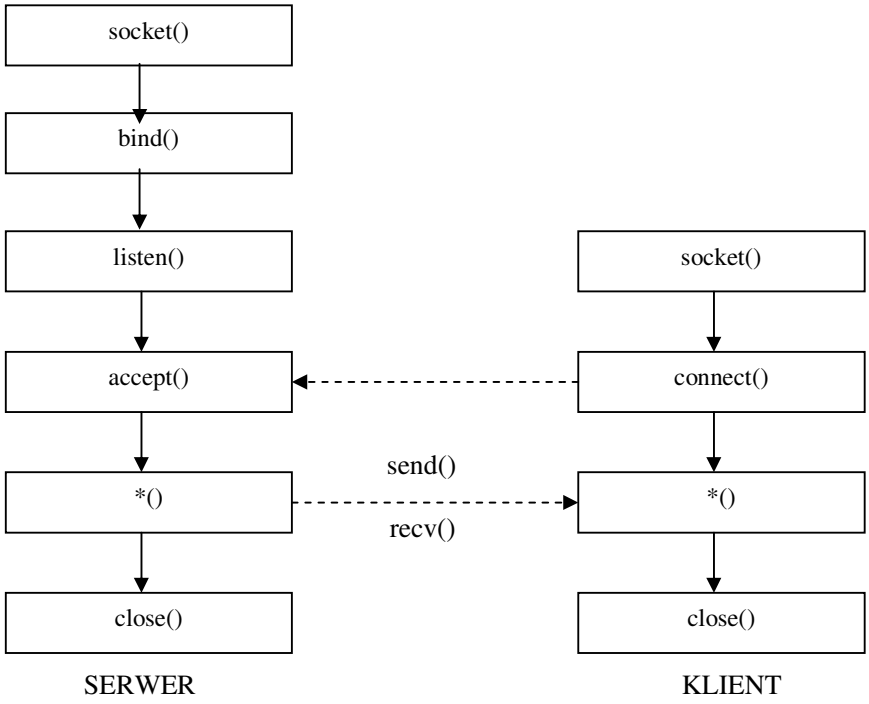
2.7.3 Różne funkcje

Do konwersji krótkich i długich liczb całkowitych z hosta na porządek sieciowy i odwrotnie mamy cztery funkcje.W tym samym systemie (tj Intel lub VAX), porządek bajtów hosta jest inny niż porządek sieciowy. Dlatego też programy wymagają dokonania tłumaczenia między tymi dwoma formatami.Znajdują się te funkcje w poniższej tabeli:

Nazwa Funkcji	Opis
htons*(sint)*	Konwersja krótkiej liczby całkowitej z formatu hosta na sieciowy
ntohs*(sint)*	Konwersja krótkiej liczby całkowitej z formatu sieciowego na host
htonl*(lint)*	Konwersja długiej liczby całkowitej z formatu hosta na sieciowy
ntohl*(lint)*	Konwersja długiej liczby całkowitej z formatu sieciowego na host

3 Podstawy projektowania struktury sieciowej

Ta sekcja skupia się na zaprezentowaniu czterech najprostszych przykładów ilustrujących poprzednio omówione funkcje i metody sieciowe. Przykłady są dostarczone w postaci prostego kodu, stworzonego dla jasności nie zaś efektywności. Przedstawiam dwie pary struktur serwer - klient: jedna para dla protokołu i druga dla protokołu UDP. Czytelnik powinien zapoznać się z tymi przykładami przed dalszym czytaniem.



Rysunek 1: Połączenie TCP

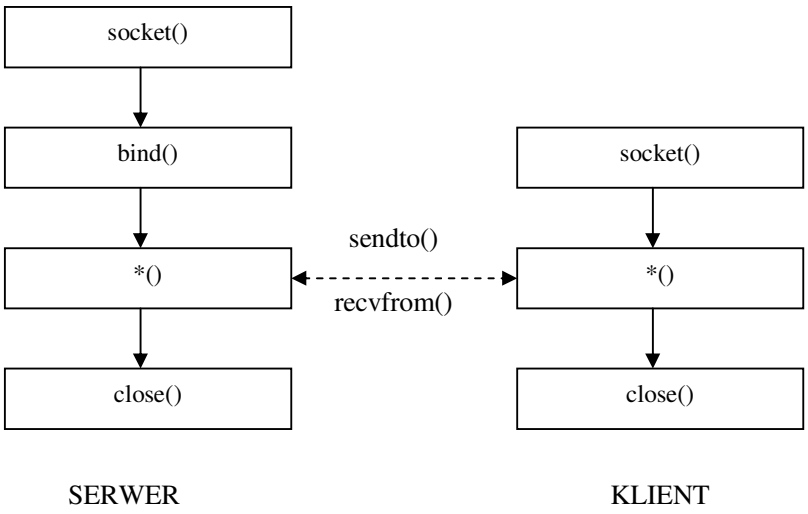
Powyżej prezentuję połączenie *stream*. Zauważmy jak procesy współdziałają: serwer staruje i musi być w stanie *akceptacji* nim klient wysunie swoje żądanie. Metoda **connect** () klienta próbuje spotkać się z serwerem gdy ten jest w stanie akceptacji połączenia. Po połączeniu następuje faza negocjowania wymiany danych po czym obie strony wywołują **close**() kończącą połączenie. Pamiętaj, to jest diagram jednego połączenia. W realnym świecie, po stworzeniu nowego połączenia (w nowym procesie lub nowym wątku) serwer wraca do stanu *akceptacji*. Funkcja *() jest funkcją zdefiniowaną przez użytkownika do obsługi określonego protokołu . Transfer danych jest realizowany poprzez **send()** i **recv().**

Wymiana *datagram* jest prezentowana na Rysunku 2. Jak widać, serwer jest procesem ,który wiąże swoją nazwę i port, będąc pierwszym który odpowiada na żądanie. Klient jest procesem, który pierwszy wysyła żądanie. Możesz wstawić metodę **bind()** do kodu klienta , uzyskując identyczne części. Kto jest serwerem a kto klientem, to trudno w tym przypadku ustalić

Mając nadzieję, że zaprezentowane diagramy okazały się użyteczne, przejdziemy dalej do kodów naszych czterech prostych struktur, modelujących pary klient-serwer TCP i UDP.

3.1 Projektowanie serwera TCP

Projektując serwer TCP musisz podjąć następujące kroki *wiązanie-nasłuchiwanie-akceptacja* . Tworząc kod dla prostego serwera echo, uruchom ***host lokalny*** i nasłuchuj na porcie *8000*.Zaakceptuje



Rysunek 2: Połączenie UDP

pojedyncze połączenie od klienta, zwróci echo wszystkich danych a kiedy nic nie odbierze zamknie połączenie.

```
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()
```

W pierwszej lini deklarujemy ważną instrukcję modułu **socket**. Linie 3 do 6 to standardowy łańcuch serwera TCP. W lini 6 serwer akceptuje połączenie zainicjowane przez klienta; metoda **accept**() zwróci gniazdo **clientsocket**, dalej używane do wymiany danych. Pętla **while 1** jest "kodem echa"; odsyła dane transmitowane przez klienta z powrotem. Po pętli która została przerwana (kiedy klient wysłał pusty ciąg i napotkany jest warunek **if not data: break**) gniazdo jest zamykane i program się kończy.

Przed przejściem do kodu klienta, szybkim sposobem na przetestowanie serwera jest klient *telnet*. Ponieważ telent uruchamia się pod protokołem TCP nie powinno być problemu z wymianą danych. Telnet po prostu będzie odczytywał dane ze standardowego wejścia I wysyłał je do serwera, a kiedy odbiera dane wyświetla je na terminalu.

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

```
$
```

Po wysłaniu przez telnet jakichś danych wejdź w tryb terminala:wpisz coś a serwer zwróci echo.

3.2 Klient TCP

Teraz skupimy się na napisaniu kodu klienta dla naszego serwera echa. Klient będzie miał pętlę w której dana jest odczytywana ze standardowego wejścia i wysyłana (w 1024 bajtowych pakietach – to jest maksymalna ilość danych jakie serwer odczytuje z określonego kodu klienta) do serwera. Serwer odsyła te same dane a klient ogłasza je ponownie.Tu mamy kod:

```
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()
```

Klient najpierw wysyła dane i tylko po wysłaniu danych, w przypadku danych zerowych, kończy się. Zabezpiecza to serwer przed zawieszeniem (pamiętaj ,że serwer kończy kiedy odbierze pusty ciąg). Zdecydowaliśmy się,że przechowuje odebrane dane w newdata aby zabezpieczyć się przed problemami jakie mogą wystąpić, jeśli żadna dana lub coś niewłaściwego zostanie przekazane do serwera. W porównaniu do serwera kod klienta jest dużo prostszy – tworzy tylko gniado i proste połączenie, potem nadchodzi protokół.

3.3 Aplikacje modelowania datagramu

Wcześniej określiliśmy ,że jest trochę różnic miedzy klientem datagram a serwerem datagram. Rozważmy ponownie przykład serwera uruchamianego na *hoście lokalnym* i odbierającym pakiety na porcie *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()
```

Jak widać w tym przykładzie ,nie ma kroków nasłuchiwania ani akceptacji. Konieczne jest tylko powiązanie, gdziekolwiek by nie był klient musi wysłać pakiety do serwera. W prostej terminologii, jądro po prostu 'wypycha' pakiety danych do serwera na określonym porcie, bez konieczności potwierdzania połączenia. Serwer decyduje czy zaakceptować pakiet czy

nie. recvfrom() zwraca poza tym dane adresu klienta, używanego później w metodzie sendto(). Klient niewile się różni od serwera:

```
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()
```

Nie musimy wiązać metod, kiedy serwer używa adresu zwracanego przez metodę recvfrom(). Możliwe jest zaprojektowanie prawie symetrycznego serwer/klient UDP, używającego tej samej struktury. Klient będzie stroną,która pierwszy zainicjuje połączenie.

4 Zaawansowane tematy dotyczące serwerów

Ta sekcja dotyczy zaawansowanych technik poszerzających zastosowanie w projektach nowoczesnych serwerów. Cztery struktury omówione poprzednio są ilustracją jak stosować funkcje związane z gniazdami. Czytelnik nie musi stosować tych czterech struktur,ponieważ kod jest bardzo ograniczony i stworzony dla zilustrowania tego celu.Idąc dalej, konieczne jest nauczyć się pewnych dobrze znanych wzorców ,ulepszających projektowanie serwerów.

4.1 Budowanie nieskazitelnego środowiska

Kiedy serwer zaczyna musi zdefiniować parametry konieczne w czasie uruchamiania. Jest to wykonywane przez funkcję get*(). Kolejny kod jest napisany dla serwera sieciowego I próbą określenia nazwy hosta systemu, numeru portu, na którym uruchamiany jest protokół *http*.

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

Sprawdzamy getservbyname() na istnienie błędów ponieważ jest możliwe dla usługi nieobecnej w tym systemie. Bez wątpienia, w nowo zaimplementowanych usługach/protokołach jest bezużyteczne wywoływanie tej funkcji. **hostname** i **http_port** będą używane jako parametry dla **bind().**

Po przejściu poprzedniego kroku, serwer musi być odłączony od terminala sterującego ,dlatego też będzie uruchomiony jako demon. W C jest to robione z systemem wywołania **ioctl()**, obejmującym standardowe wejście, wyjście, obsługę błędów i wykonujący dodatkowe operacje. W Pythonie serwer musi być uruchomiony z "&" po swojej nazwie. Innym sposobem jest. zalecanym, jest wywołanie fork(),która tworzy dwa procesy: rodzica, który istnieje bezpośrednio i potomny (potomkiem jest nasza aplikacja). Potomek jest adoptowany przez *init,* zatem uruchomi się jako demon. To jest przystępna metoda i nie komplikuje kodu:

```
import os



pid = os.fork()
if pid: os._exit(0)
else:
                .... kod #serwera będzie tu
```

Serwer nie będzie mógł wysyłać nigdy więcej wiadomości przez standardowe wyjście lub błędy standardowe ale przez funkcję *syslog*. Po tych dwóch krokach mamy komplet, możemy wybrać metodę do obsługi wielu połączeń w tym samym czasie.

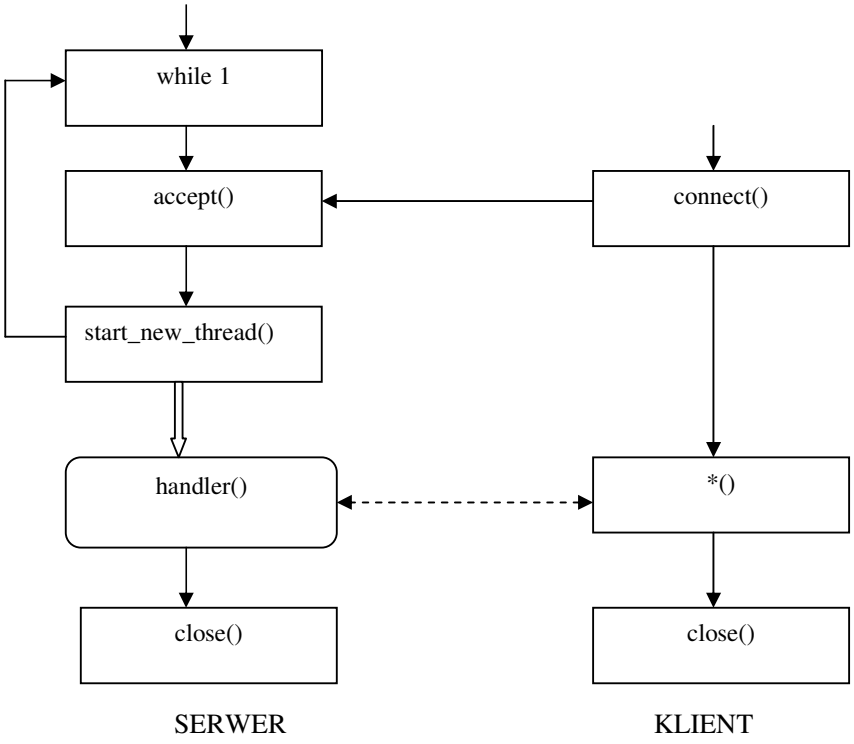
4.2 Obsługa wielu połączeń

Po przeanalizowaniu serwerów w sekcjach 3.1 i 3.3, jasne jest ,że projekt ma wiele mankamentów: serwer nie może obsłużyć więcej niż *jednego* klienta.Rzeczywisty serwer jest zaprojektowany do obsługi więcej niż jednego klienta a ponadto, wielu klientów w tym samym czasie.

Są trzy metody obsługi takiego zdarzenia: poprzez funkcję select(), tworząca nowy proces dla każdego przychodzącego żądania poprzez fork() lub przez obsługę wymagająca oddzielnego wątku. Wątkowanie jest najbardziej elegancką metodą I zalecamy ją. Użycie select () oszczędza CPU w przypadku dostępu wielu serwerów I może być czasami dobrą opcją. Tworzenie nowego procesu dla każdego przychodzącego połączenia jest najbardziej użytecznym wzorcem w aktualnym projektowaniu serwerów (na przykład używa tego Apache) ale ma wady w postaci marnowania czasu CPU i zasobów systemowych. Zalecamy stosowanie serwerów wątków.

4.2.1 Serwery wątków

Serwery wątków używają oddzielnych wątków do obsługi każdego połączenia. Wątki są definiowane jako *lekkie procesy* i są uruchamiane wraz z procesem głównym, startującym wraz z nimi. Diagram jest przedstawiony poniżej:



Rysunek 3: Diagram serwera wątków

Załóżmy,że mamy zdefiniowaną przez użytkownika funkcję nazwaną handler , do obsługi prostego połączenia, za każdym razem kiedy klient chce wymienić dane z serwerem handler jest uruchamiany jako oddzielny wątek. Strzałka blokowa I strzałka przerywana oznaczają tworzenie nowego wątku. Nowy wątek współdziałał będzie dalej z klientem. Poniżej zamieszczam przykład dla serwera TCP z głównym gniazdem nazwanym sock.

```
import socket, thread
def handler(socket):

    …….

……………

while 1:
```

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

Funkcja obsługująca musi być zdefiniowana *przed* przekazaniem jako argument dla nowego wątku. W przykładzie ta funkcja pobiera parametr gniazda odpowiedni dla klienta, który inicjuje to połączenie. Funkcja ,może wykonać jakiś rodzaj działań na tym gnieździe. Wprowadzanie danych nie jest wykonywana przez funkcję **print** ale przez **syslog.** Alternatywnie, możliwe jest napisanie funkcji obsługującej to zadanie oddzielnie (tworzącej własne logi w pliku zdefiniowanym przez użytkownika). Innym wyborem jest użycie modułu **Threading** który oferuje bardziej elastyczną implementację wątków. Faktycznie możesz stworzyć *obiekt wątku,* kilkoma metodami, łatwiejszymi do obsługi. Argumenty funkcji muszą być przekazane w krotce (lub jeśli jest to pojedynczy element, użyjemy *singleton*) aby mieć kod wolny od błędów, a interpreter nie wstawi TypeError. Projekt może być ulepszony przez wstawienie metody **accept** () w oddzielnym wątku; w wyniku czego serwer będzie wolny (jeśli blok **accept** (), przy normalnym zachowaniu, będzie oddzielony od wątku głównego serwera) i może być użyty do wykonania dodatkowych operacji lub nawet nasłuchu na innym porcie. Proponuję stworzenie serwera sieciowego uruchamianego na standardowym porcie HTTP i zdalnie sterowanego *w tym samym czasie* z dowolnego portu. Inna wartościowa cecha: używanie wątków dodaje profesjonalizmu twojemu kodowi I eliminuje potrzebę pisania kodu obsługującego system wywołań **fork** i dodatkowych funkcji. Kiedyś, jeśli zechcesz zaprojektować wspomniany powyżej serwer sieciowy, będą potrzebne przynajmniej trzy procesy uruchomione w tym samym czasie powodując zwolnienie procesora.

4.2.2 Używanie funkcji select

Python prostej implementacji standardowej funkcji **select** () UNIX'a. Ta funkcja, zdefiniowana w module **select**, jest używana w multipleksowych żądaniach I/O pośród wielu gniazd i deskryptorów pliku. Są tworzone trzy listy deskryptorów gniazd: jedna lista z deskryptorami które chcemy odczytywać, lista deskryptorów do który chcielibyśmy zapisywać i lista która zawiera gniazda ze 'specjalnymi warunkami' oczekującymi. Jedynie "specjalny warunek" jest zaimplementowany poza pasmem danych, określony w implementacji gniazda jako stała MSG_OOB i używana jako specjalna flaga dla metod **send()** i **recv()**. Te trzy listy są przekazywane jako parametry do funkcji poza parametrem ***timeout***, określającym jak długo powinna oczekiwać funkcja **select**() aby zwrócić wartość,jeśli nie jest dostępny deskryptor. W zamian, **select**() dostarcza nam trzech list: jedna lista z gniazdem, które może być odczytane, inna z gniazdem zapisywalnym i ostatnia odpowiadająca kategorii 'specjalnego warunku'.

Specjalna funkcja gniazda pomoże ci w bardziej elastycznym podejściu do obsługi wywołania **select**(). Jest to metoda **setblocking()**, która jeśli zostanie wywołana z parametrem 0, ustawi dla gniazda opcję *nonblocking*].To znaczy,że żądanie nie wykonało się bezpośrednio, spowoduje zawieszenie procesu i jego nie wykonanie, ale zwrócony zostanie kod błędu. Poniżej podałem kod dla lepszego zrozumienia tego tematu:

```
import socket,  select

preaders =  []
pwriters =  []
```

.

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

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

……
```

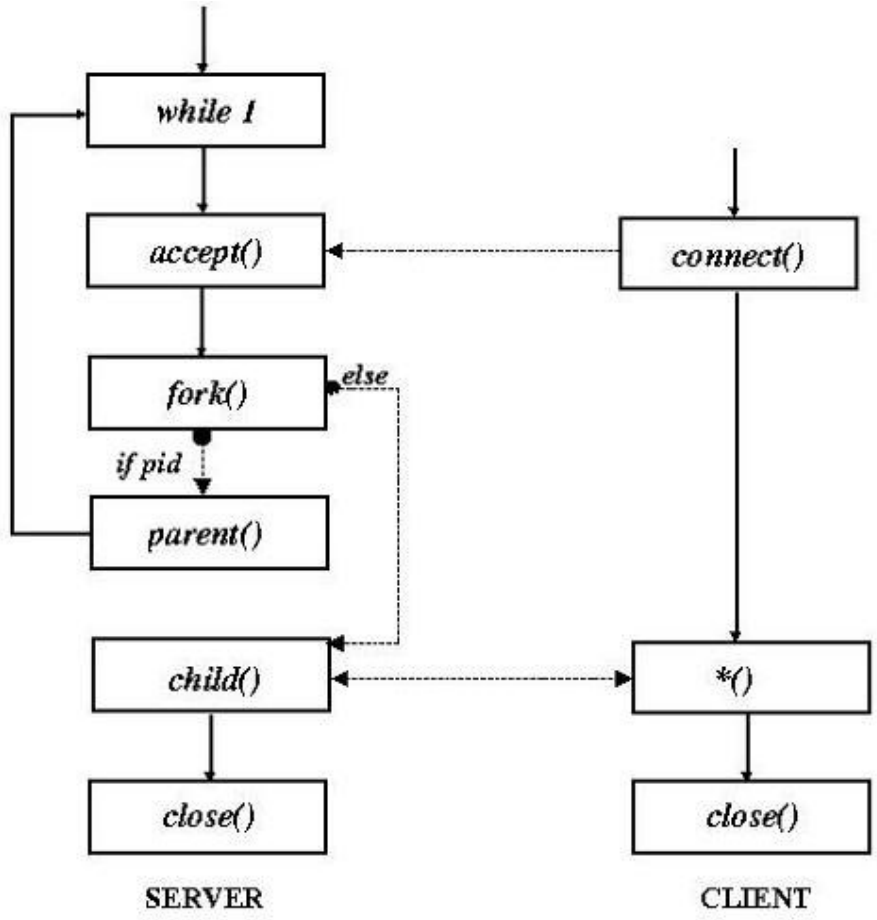
Jak widać stworzyliśmy dwie listy, jedną dla potencjalnych czytelników a drugą dla potencjalnych piszących. Ponieważ nie ma zainteresowania specjalnym warunkiem, przekazywana jest pusta lista jako parametr dla **select()** zamiast listy z gniazdami. Wartość *timeout* to 30 sekund. Po zwróceniu funkcji, mamy dwie używalne listy, z dostępnymi gniazdami dla czytających i piszących. Dobrym pomysłem jest dodanie przynajmniej gniazda głównego serwera (tutaj **sock)** dla listy potencjalnych czytelników, dlatego też nie będzie zdolny do akceptacji przychodzących żądań. Mechanizm akceptacji żądań jest stosunkowo prosty: kiedy żądanie jest odłożone na gnieździe serwera, zwrócone jest w gotowości do odczytu listy i może się na nim wykonać **accept** (). Gniazdo wynikowe klienta może być dołączone do dowolnej listy (może być do obu, w zależności od protokołu). Kiedy parametr *timeout* wynosi **0 select** () przybierze formę odpytywania, zwróconą bezpośrednio.

4.2.3 Serwery rozgałęzione

Serwery rozgałęzione są pierwszym wyborem w dzisiejszym projektowaniu sieciowym. Przykład pierwszy z brzegu: Apache, najpopularniejszy serwer webowy ma strukturę serwera rozgałęzionego. Serwery rozgałęzione używają elastycznego procesu zarządzania zaimplementowanego w większości nowoczesnych UNIX'ów. Podstawową ideą jest uzyskanie nowego procesu dla każdego przychodzącego żądania, proces rodzicielski tylko nasłuchuje, akceptuje połączenie i "tworzy" potomka. Używając alternatywnie rozgałęzienia musisz zaakceptować jego niedostatki: musisz działać z procesami na poziomie profesjonalnym, serwer jest dużo wolniejszy w porównaniu z serwerem zaprojektowanym dla wątków, i jest trochę bardziej skomplikowany. Implikuje to ,że musisz zaprojektować funkcję *fireman* zbierającą *zombies,* dodać oddzielne kody dla rodzica i dla potomków. Na rysunku 4 przedstawiam diagram pokazujący algorytm dla serwera rozgałęzionego. Na rysunku nie przedstawiłem funkcji *fireman*; zazwyczaj jest wywoływana przed wywołaniem **fork().** Kod dla rodzica i dla potomków jest wyraźnie oznaczony. Tworzenie nowego procesu jest sygnalizowane przez przerywane strzałki. Po wymianie danych wywoływane są funkcje **close** () w gnieździe, potomek oczekuje na wywołanie przez rodzica systemowej funkcji **wait()** lub **waitpid()**. Ten system wywołań zabezpiecza przed narodzinami czegoś co się zwie *zombies.* Zwyczajowo tworzymy listę gdzie dodajemy wszystkich wynikowych potomków po wywołaniu **fork().** Ta lista jest przekazywana jako parametr do funkcji *fireman* która zważa na nie. Tu mamy dwie metody rozgałęzienia:

- Proste rozgałęzienie: ta metoda po prostu wywołuje **fork()** z if-then-else, opierając się na wynikach wykonania oddzielnego kodu dla rodzica i potomka.Jeśli potomek nie jest oczekiwany stanie się zombie.

- Oddzielne rozgałęzienie: rodzic rozgałęzia się i tworzy potomka jeden.Ten rozdziela się I tworzy potomka dwa a potem następuje wyjście.Potomek jeden jest oczekiwany przez rodzica.Ponieważ potomek dwa stworzono bez rodzica jest adoptowany przez *init,* a zatem kiedy wywołuje **exit(),** init pilnuje aby ten proces nie stał się zombie.



Rysunek 4: Diagram serwera rozgałęzionego

Przedstawię teraz kod dla serwera rozgałęzionego z oboma metodami.Pierwsza metoda jest zazwyczaj oczekiwana.Ponownie, **sock** jest gniazdem serwera.

```
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)
```

Funkcja **handler** jest opisana w paragrafie 4.2.1. Proszę zobaczyć jak oddzielny kod jest wykonywany dla rodzica i dla potomka. Dobrym zwyczajem jest zamykać gniazdo serwera sock w potomku a gniazdo klienta **clisock** w rodzicu. **fireman** oczekuje na wszystkich potomków z określonej listy, przekazując ją jako parametr.Druga metoda może być zaimplementowana jak następuje:

```
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)
```

Pierwszy potomek tylko rozgałęzia się i kończy. Jest oczekiwany w rodzicu, więc musi być bezproblemowy.Drugi potomek jest adoptowany a potem wprowadzony w "normalnym" trybie, wykonując funkcję **handler**.

4.3 Praca z klasami

Ta sekcja opisuje jak pracować w rzeczywistym programowaniu zorientowanym obiektowo. Ponieważ jest to ważny temat, zarówno odnoszący się do serwera jak i klienta, zdecydowałem się opisać oba aspekty w tej sekcji.Musisz się nauczyć jak konstruować klasy, które pracują z funkcjami sieciowymi, budować obiekty połączeń i dużo więcej. Celem jest zaprojektowanie wszystkich tych obiektów według danego wzorca. Uprości to projekt i doda możliwość wielokrotnego użytkowania i mocy do twojego kodu..

4.3.1 Obiekt prostego połączenia

Celem tej sekcji jest zaprojektowanie obiektu prostego połączenia. Celem zaprojektowania takiego obiektu jest łatwe zaprojektowanie danej aplikacji. Kiedy już zaprojektowałeś właściwe komponenty (obiekty) możesz skupić się na zaprojektowaniu programu wyższego poziom, poziomu komponentu. Da ci to poziom abstarkcji, bardzo użyteczny w prawie wszystkich przypadkach.

Przypuśćmy,że chcemy stworzyć serwer gniazda TCP w jednym obiekcie. Pozwoli ci to łączyć się z klientami TCP na określonym porcie. Najpierw musimy zdefiniować kilka metod do obsługi możliwych sytuacji jakie pojawią się podczas uruchamiania. Odnosząc się do Rysunku 1, widzimy ,że serwer musi "uruchomić" gniazdo i nasłuchiwać na nim, wtedy, kiedy połączenie przychodzi,zaakeptować połączenie a potem działać z nim. Poniższe metody są widzialne dla użytkowników: OpenServer, EstablishConnection, CloseConnection. OpenServer jest metoda, która ustawia nazwę hosta i port na którym serwer jest uruchomiony.Kiedy nadchodzi połączenie, EstablishConnection decyduje czy ma być zaakceptowane w oparciu o funkcje przekazana jako parametr. CloseConnection nie wymaga tłumaczenia.

```
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() ;
```

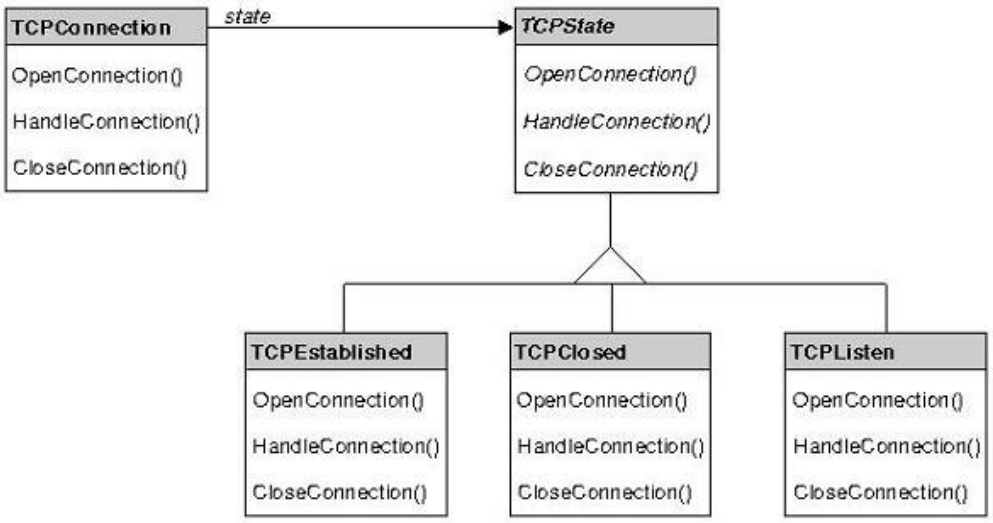
Funkcja__init__() nie robi nic. Kiedy serwer startuje, *obiekt TCPServer* ,przez OpenServer, pobiera jako parametry nazwę hosta na którym jest uruchomiony I numer portu usługi.Alternatywnie, może być zrobione przekazanie parametrów do funkcji __ init__(). Ta metoda wykonuje również kontrolę błędów w gnieździe. Będziemy się upierać przy metodzie EstablishConnection. Implementuje ona serwer wątków i pobiera , wraz z innymi parametrami, nazwę funkcji (tu nazwaną connection_handler) używaną do obsługi połączenia. Możliwe polepszenie to zaprojektowanie tej metody tak aby jako parametr krotkę (lub słownik) zawierającą parametry connection_handler, którą mamy w naszym przykładzie z jednym parametrem, wygenerowaną wewnątrz metody, która ją wywołała, client_socket. Nie ma sensu upierać się przy metodzie CloseConnection Została stworzona do zapewnienia jednolitego interfejsu dla obiektu TCPServer. Przyszłe implementacje (przez dziedziczenie lub przez

.

skład obiektu) mogą odkryć użyteczną, określoną dodatkową cechę dla tej metody.

4.3.2 Nałożenie wzorca projektowego

Stan wzorca projektowego pasuje do naszych celów. Projekt dostarcza pięć klas: jedna klasa, **TCPServer**, będzie klientem (użytkownikiem pozostałych czterech klas); **TCPConnection** jest klasą ,która realizuje połączenie TCP: poprzez **OpenConnection()**, **HandleConnection()** i **CloseConnection()** zarządza gniazdem; również są tworzone trzy trójwartościowe klasy, reprezentujące stany w jakich może się znajdować **TCPConnection** .Te klasy są dziedziczone z klasy *abstrakcyjnej* , nazywanej **TCPState.** Poniżej przedstawiam diagram ilustrujący związki pomiędzy obiektami połączenia a ich stanami (reprezentowanymi jako klasy): Tu mamy kod dla klasy **TCPConnection**



Rysunek 5: Zaprojektowanie połączenia TCP ze stanem wzorca

Klasa **TCPServer** (proszę uważać: kiedy podawałem przykład **TCPServer** w sekcji 4.3.1, był to serwer; tu projektujemy tylko klasę połączenia, która dalej będzie używana w projektowaniu serwera lub klasy serwera) po prostu tworzy egzemplarze obiektu połączenia i wykonuje pewne dodatkowe działania.

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()
```

Jak widać w powyższym przykładzie,zachowanie obiektu połączenia zależy od jego stanu.Metody nie mogą być użyte w dowolny porządku. Kiedy obiekt został stworzony **__init__**() wykonuje trzy klasyczne kroki inicjalizacji gniazda serwera i ustawia obiekt w stan *nasłuchu*. W stanie *nasłuchu,*klient może tylko wywołać metody OpenConnection() i CloseConnection(). Po zaakceptowaniu połączenia (przez OpenConnection – zobacz powyższy kod, opisujący klasę *states*) obiekt jest w stanie *established*, kiedy działa z końcem połączenia. Teraz, klient może użyć handleConnection() do uruchomienia protokołu. Domyślnie, obiekt wraca sam do stanu *nasłuchu* a potem OpenConection musi być wywołana ponownie dla realizacji owego połączenia. Kiedy jest wywoływana CloseConnection(), gniazdo serwera jest zamykane a dalsze operacje na nim stają się niemożliwe. Jest to bardzo proste podejście dajace raczej szkielet rzeczywistego obiektu TCPConnection. Instrukcja **raise** jest używana do sygnalizowania klientowi niewłaściwego używania metod obiektu. Te błędy są tworzone poprzez dziedziczenie z klasy Exception.Poniżej mamy cztery stany klasy:

```
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):
```

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

class TCPEstablished(TCPState) :

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

Wszystkie konkretne klasy są dziedziczone z klasy abstrakcyjnej TCPState. Ta klasa dostarcza jednolitego interfejsu dla pozostałych trzech. Proszę zobaczyć ,że tylko konieczne metody są unieważniane. Kiedy jest oczekiwany inny stan, klasa stanu jest dziedziczona z TCPState, metody są redefiniowane a właściwy kod jest dodawany do klasy TCPConnection obejmującej również ten stan.

4.4 Zaawansowane aspekty dotyczące klientów

Głownie w tej sekcji 4 skupiliśmy się na projektowaniu serwera. To prawda, ponieważ projektowanie serwera jest dużo bardziej skomplikowanym zadaniem niż projektowanie klienta. Serwer jest uruchamiany jako *demon* w większości przypadków, pracując z wieloma klientami, więc musi być szybki, niezaawodny i nie może być nieodporny na błędy. Klient jest traktowany jako pojedyncze połączenie z serwerem (dla większości dużych klientów), dlatego też czysty kod sieciowy jest prostszy i nie obejmuje wątków, selekcji czy rozgałęzienia. Jednak, zalecane jest stosowanie programowania zorientowanego obiektowo w tym przypadku lub nawet wzorca projektowego (prezentowany wzorzec projektowy *state* jest łatwo dający się zastosować w kliencie), ale nie rozgałęzienie, wątek lub selekcja. Życie nie jest ciężkie kiedy mamy projekt części sieciowej klienta.

5 Protokół HTTP

HTTP jest, poza pocztą elektroniczną, najpopularniejszym protokołem w Internecie. HTTP jest implementowany w serwerach sieciowych takich jak Apache lub w przeglądarkach takich jak Netscape lub Mosaic. Python dostarcza kilka modułów dostępu do tego protokołu. Zajmiemy się kilkoma z nich: webbrowser, httplib, cgi, urllib, urllib2, urlparse i trzy modułu dla już wbudowanych serwerów HTTP: BaseHTTPServer, SimpleHTTPServer i CGIHTTPServer. Jest także moduł Cookie dla obsługi cookies. Moduł webbrowser oferuje sposób wyświetlania i sterowania dokumentami *HTML* poprzez przeglądarkę (zależną od systemu). Używając modułu cgi użytkownik może pisać złożone skrypty *CGI*, zastępując "standardowy" język skryptowy CGI jakim jest Perl. httplib jest modułem implementującym po stronie klienta protokół HTTP. urllib jest modułem, który dostarcza wysokopoziomowego interfejsu dla odzyskiwania zawartości stron w World Wide Web. Ten moduł (również i urllib2) używa poprzednio wspomnianego modułu, httplib. Jak wskazuje nazwa, moduł urlparse jest używany do parsowania URL'i.

5.1 Moduł CGI

CGI (Wspólny Interfejs Bramkowy) i jest środkiem przez który można wysyłać informacje ze strony HTML (to znaczy używamy struktury <f orm>...</f orm>) do programu lub skryptu poprzez serwer webowy.Program odbiera tą informację w dogodnej formie – w zależności od serwera webowego – i wykonuje zadanie: dodanie użytkownika do bazy danych, sprawdzenie hasła, generowanie strony HTML w odpowiedzi lub jakaś inna akcja. Zazwyczaj, te skrypty mieszczą się specjalnym katalogu na serwerze „cgi-bin", do którego ścieżka dostępu musi być ustawiona w pliku konfiguracyjnym serwera.

5.1.1 Budowa prostego skryptu CGI

Kiedy użytkownik wywołuje skrypt CGI, przekazywany jest specjalny ciąg do serwera określający skrypt, parametry (podawane z formy lub URL'a z ciągiem zapytania) i inne rzeczy. Serwer przekazuje wszystkie te rzeczy do wspomnianego skryptu, który rozumie o co chodzi, przetwarza zapytanie i wykonuje akcję. Jeśli skrypt odpowiada coś klientowi (jakieś instrukcje drukowania, dane wyjściowe skryptu), wyśle kawałki danych poprzez serwer sieciowy. Teraz jest jasne dlaczego ten interfejs nosi nazwę *Wspólną Bramką* – każda transakcja pomiędzy klientem a skryptem CGI jest wykonywana przy użyciu serwera webowego, bardziej szczegółowo, używając interfejsu między serwerem a skryptem.

Poniżej umieszczony jest skrypt upraszczający wyświetlanie wiadomości na ekranie:

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

print 'Hello World!'
```

Pierwsze dwie linie są obowiązkowe: linie te mówią przeglądarce ,że zawartość musi być interpretowana przez język HTML. Instrukcja print jest używana do tworzenia pustej linii.Teraz, kiedy nagłówek jest spreparowany, możemy wysłać dane do klienta. Jeśli te dane zawierają znaczniki HTML, będą interpretowane jako normalna strona WWW. Zapisz ten skrypt w katalogu 'cgi-bin' i ustaw zezwolenie dla niego jako odczytywalny I wykonywalny. Dokonaj właściwych modyfikacji w pliku konfiguracyjnym serwera. Wywołaj 'http://your.server.com/cgi-

bin/your-script.py' i powinieneś zobaczyć 'Hello World!' umieszczone w lewym górnym rogu przeglądarki.

Zmodyfikujmy skrypt wyświetlający tą samą wiadomość wycentrowaną i w rozsądnym rozmiarze. Użyjemy znaczników HTML <title>,<h1> and :

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

print

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

Teraz droga jest otwarta.Spróbuj wygenerować swoją własną stronę HTML ze skryptami CGI. Możesz wstawić obrazki, JavęScrip lub VBScript na stronie. Jeśli przeglądarka je obsługuje to dobrze.

5.1.2 Użycie modułu CGI

Ten moduł jest użyteczny kiedy chcemy odczytać informację przekazane przez użytkownika poprzez formę. Nazwa modułu to (cóż!) cgi i musi być używana z ważną instrukcją cgi Moduł definiuje kilka klas (niektóre dla wstecznej kompatybilności) które mogą obsługiwać dane wyjściowe serwera sieciowego aby uczynić go dostępnym dla programisty. Zalecam klasę **FieldStorage** chociaż w niektórych przypadkach jest stosowana **MiniFieldStorage**. Instancja klasy jest obiektem słownikowym i możemy zastosować metody słownikowe. Poniżej mamy stronę HTML i kod Pythona do wyodrębnienia informacji z pól formy:

```
<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.'
```

Jak widać w tym przykładzie, MyForm jest instancją klasy **FieldStorage** i zawiera parę nazwa – wartość opisaną w strukturze <form> na stronie HTML: **text field** z *name=yourname* i **textarea** z name= *comment.* Stosując metodę **has_key** w obiekcie MyForm można sprawdzić czy określone pole zostało wypełnione a potem, z **MyForm [**" **name**" **]** . **value** można sprawdzić wartość tego pola. Ponieważ wartość przypisano do jakiejś wewnętrznej zmiennej, programista może użyć ich jako prostej zmiennej. Przedstawię teraz mały przykład ,który odczytuje formę wypełnioną przez użytkownika (zawierającą nazwę użytkownika, adres I numer telefonu) i zapisuje wszystkie zmienne w pliku.

```
<html>
<form name='personal_data' action='/cgi-bin/my-cgi.py'
method='POST'>
<p>Insert your name : <input type='text' name='name'
size='30'>
<p>Insert your address: <input type='text'
name='address' size='30'>
<p>Phone: <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
Addre s s = MyForm['addre s
s'].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 Konfigurowanie Apache'a w Linuksie do użycia ze skryptami CGI

Ta sekcja jest krótkim wstępem dla tych ,którzy mają trudności w ustawieniu skryptów. Jeśli używasz Apache'a w Linuksie lub systemem *NIX powinieneś zobaczyć błąd

.

z kodem 500 - Internal Server Error. Inny błąd to File Not Found ale myślę,że jest ot zbyt trywialne aby to omawiać tutaj. Jeśli uzyskasz kod błędu 500 , to tu jest list 'co zrobić' aby zapewnić działanie skryptu:

- Sprawdź czy interpreter jest określony w pierwszej lini:

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

- Sprawdź czy interpreter jest odczytywalny i wykonywalny.
- Sprawdź czy pliki dostępne przez skrypt są odczytywalne i wykonywalne przez 'inne' ponieważ, ze względu na bezpieczeństwo, Apache uruchamia skrypt jako użytkownik 'nobody'.
- Upewnij się,że w *httpd.conf* istnieją linie:

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

i

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

  Ścieżka do twojego katalogu 'cgi-bin' może się różnić, w zależności od tego jak ustawisz opcję *DocumentRoot* w tym samym pliku.

Również dobrą praktyką jest uruchamianie skryptu z konsoli, interpreter pokaże wystąpienie błędu, to będzie najprostszy sposób na korekcję składni błędu.

6 Protokoły wspólne

6.1 Zaprojektowanie aplikacji Telnet

Telnet jest prostym protokołem, który emuluje terminal w połączeniach sieciowych. Wraz z telnetem możesz uzyskać dostęp do obcych maszyn na określonym porcie a potem wysłać I odebrać dane przez przyjęte łącze bez żadnych ograniczeń.Możliwy jest dostęp przez klienta telnet do serwera sieciowego, serwera POP3, serwera maili itd. Zadaniem telnetlib jest zaoferowanie prostego sposobu zaprojektowania klienta telnet. Ten klient może być używany w innych aplikacjach obsługujących komunikację między dwoma maszynami, używającymi tego protokołu. Aby to lepiej zrozumieć, przypatrzmy się kolejnym dwóm przykładom:

```
$ telnet localhost

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

login:
```

To jest trywialna sesja telentu. Zdalna maszyna zażąda nazwy użytkownika i hasła (w przypadku kiedy usługa telnet jest uruchomiona ma tym hoście i zezwalasz na dostęp do niego) a potem przechodzisz do powłoki. Główna wadą takiej komunikacji jest to,że nie jest tajna, telent przesyła wszystko otwartym tekstem. Spójrzmy na drugi przykład:

```
$ telnet localhost 80

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


GET /
```

W tym przypadku próbujemy uzyskać dostęp do serwera sieciowego (port 80) w tej samej lokacji. Do odebrania strony wykorzystujemy metodą GET po której następuje nazwa strony (w tym przypadku / to znaczy strony głównej ,powszechniej zwanej index.html). Nie określamy całkowicie żądania GET, wspominając protokół i pozostałe rzeczy.

Wniosek: ten sam klient ale różne protokoły.Kiedy używamy metody GET w drugim przykładzie, klient telnet był naszym *sieciowym wsparciem* a ten protokół został zaemulowany. Głównym pomysłem jest zastosowanie klienta telent jako wsparcia dla części sieciowej, podczas gdy użytkownik pisze (lub używa) swojego własnego protokołu podczas tego łączenia.

Python implementuje protokół Telnet z modułem telnetlib, który oferuje klasę nazywaną Telnet *(host, port).* Instancją tej klasy jest *obiekt telnet* z metodami otwierającymi i zamykającymi połączenie, odczytującymi i zapisującymi dane, sprawdzającymi status łącza i debuggowania skryptu. Poprzednie przykłady przetłumaczone na kod Pythona przy zastosowaniu 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
```

Obiekt telnet jest tworzony (w tym przypadku telnet) a dane wyjściowe serwera są odczytywane dopóki nie pojawi się ciąg 'login: '. Teraz serwer oczekuje na login, który wpisywany jest z klawiatury, a potem przekazywany jest przez zmienną do metody write () .Ten sam algorytm jest użyty przy wprowadzaniu hasła.Jeden użytkownik jest autoryzowany czy jest możliwe wysłanie polecenia i odczyt danych wyjściowych wysłanych przez serwer. Drugi przykład jest prostym odzyskiwaniem trony WWW z pewnego adresu, zapisanie do pliku i wyświetlenie go używając modułu webbrowser.

```
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_f ile.
close()

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

Powyższy kod odczytuje indeks strony z określonej lokacji, zapisuje go o pliku a potem przez moduł webbrowser wyświetla w przeglądarce.

6.2 File Transfer Protocol

Python obecnie implementuje ten protokół używając modułu ftplib. Ten moduł jest bardzo użyteczny kiedy ktoś może chcieć napisać skrypt do automatycznego odzyskiwania plików lub mirrorów stron ftp dziennego (godzinowo, tygodniowo). Moduł jest prostym tłumaczeniem w kodzie Pythona, podobnie metody klasy FTP, z typowymi poleceniami podczas sesji ftp: get, put, cd, ls etc.

Ten moduł implementuje klasę FTP. Aby przetestować sesję ftp, otwórz terminal, wprowadź interaktywną powłokę Pythona i wpisz polecenia:

```
>>> 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()
```

ftp_session jest instancją klasy FTP. Są aktualnie dwie metody implementowane do otwarcia połączenia na dany serwer: określająca nazwę hosta i port kiedy tworzony jest *obiekt ftp* lub stosując metodę *open (host, port)* dla niepołączonego obiektu ftp.

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

Po zrealizowaniu połączenia, użytkownik musi wysłać swój login i hasło (w przypadku używania nazwy 'anonymous' , hasło to user@host). Serwer wysyła z powrotem ciąg określający czy dane zostały zaakceptowane czy też odrzucone. Po pozytywnej weryfikacji, można zastosować następujące metody:

.

- **pwd()** dla wylistowania nazwy ścieżki dostępu bieżącego katalogu, **cwd *(pathname***) dla zmiany katalogu i **dir(*argument[, . . .]***) do listowania zawartości bieżącego (lub *argumentu)*katalogu, metody dla nawigacji i przeglądania katalogów.
- Metody **rename(*fromname, toname*), delete(filename), m*kd(pathname*),*rmd(pathname*)** i **size(filename)**(żadna nie wymaga objaśnień) do zarządzania plikami i katalogami wewnątrz strony ftp.
- Metody **retrbinary(*command, callback[, maxblocksize[, rest]]***) używane do odzyskiwania plików binarnych (lub odzyskiwania pliku tekstowego w trybie binarnym. *command* jest 'nazwą pliku RETR', *callback* jest funkcją wywoływaną dla każdego odebranego bloku danych. Aby odebrać plik lub dane wyjściowe polecenia w trybie tekstowym dostarczana jest metoda retrlines(*command[, callback]*).

6.3 Protokół SMTP

Protokół Przesyłania Poczty , jest protokołem, który może być używany do wysyłania poczty z dowolnej maszyny w Internecie z demonem SMTP nasłuchującym na porcie 25 (domyślny port usługi SMTP). Python implementuje stronę klienta tego protokołu (może tylko wysyłać ale nie może odbierać maili) w module smtplib. Podobnie jak dla modułów telnetlib i ftplib, jest zdefiniowana klasa SMTP (*[host[, port]])* ,której instancja, *obiekt SMTP,* ma kilka metod, które emulują SMTP. Ten moduł oferuje również bogaty zbiór wyjątków użytecznych w różnych sytuacjach. Dokładna lista wszystkich wyjątków zdefiniowanych w tym module jest przedstawiona poniżej:

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

Spójrzmy na krótki przykład, jak używać metod dostępnych w tym module aby wysłać prosty mail testowy do zdalnej maszyny nazwanej 'foreign.host.com' do użytkownika '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()
```

Ten skrypt jest instancją klasy SMTP(), tu bez żadnych parametrów. *s.connect ('foreign.host.com')* jest używane do połączenia klienta do zdalnego serwera SMTP. Po zrealizowanym połączeniu nasz klient wyśle wiadomość HELLO. Użyteczne jest sprawdzenie czy serwer akceptuje tą wiadomość (z klauzulą try. . . używającą wyjątku SMTPHeloError). Nie ma powodów aby serwer odrzucił powitanie HELO ale dobrze jest sprawdzić błędy. s.verif y(' xxx@f oreig. host. com') sprawdza skrypt czy adres odbiorcy istnieje na tym serwerze. ret_code jest zmienna (listą) która zawierać będzie odpowiedź serwera po wykonaniu instrukcji, zawierając dwa elementy: liczbę całkowitą oznaczającą kod odpowiedzią i ciąg zrozumiały dla człowieka.W przypadku.istnienia użytkownika na tamtej maszynie kod to 250 a w przypadku błędu odbioru 550. Najważniejszą metodą jest sendmail(fromaddr, *toaddr, message[, mailoptions, recipientoptions]*), która jest rzeczywistym nadawcą wiadomości. Ponieważ jesteśmy zwykłymi ludźmi, robimy quit() przed efektownym wyjściem ze skryptu. Zapewni to ,że wszystko będzie w porządku kiedy opuścimy sesję SMTP.