Tworzenie narzędzi sieciowych z wykorzystaniem Python, C/C++

- Python posiada wbudowany moduł o nazwie **socket**, który zawiera wszystkie niezbędne definicje i funkcje pozwalające na otwieranie gniazd i komunikację sieciową z ich wykorzystaniem.

Czym są sockety (gniazda)?

- Jest to pewien mechanizm umożliwiający otwarcie kanału komunikacji pomiędzy hostami. Każde gniazdo posiada adres IP oraz numer portu i może przesyłać trzy rodzaje pakietó

datagramy

- datagramy (pakiety UDP) — są to tzw. „datagram sockets” — ten kanał komunikacji wykorzystuje protokół UDP, czyli połączenie jest bezstanowe. Gniazdko wysyła pakiet UDP do docelowego hosta/portu i nie sprawdza, czy komunikacja zakończyła się powodzeniem. Przykładem usługi działającej z użyciem UDP jest DNS (Domain Name Server, port 53);

strumienie

- strumienie (pakiety TCP) — czyli „stream sockets” — te gniazdka wysyłają dane za pomocą protokołu TCP, a zatem otwierają kanał komunikacji na określonym porcie, dane wysyłane są w pakietach TCP, gniazdko (socket) nie zamyka połączenia i oczekuje na odpowiedź od hosta docelowego. Przykładem usługi korzystającej z TCP jest HTTP (HyperText Transfer Protocol, port 80 — gdy przeglądarka internetowa chce pobrać zasób z serwera WWW, robi to właśnie poprzez zestawienie kanału komunikacji TCP na porcie 80);

Raw sockets

- raw sockets (IP sockets) — ostatni rodzaj to gniazdka, które są w stanie wysyłać pakiety IP z pominięciem obu przedstawionych powyżej rodzajów pakietów, które są niejako opakowaniem dla danych (UDP i TCP). Raw IP sockets pozwalają na zdefiniowanie pakietu IP z dowolnym nagłówkiem i przesłanie go dowolnie zdefiniowanym protokołem (np. ICMP).

```
#!/usr/bin/python
#
#  uzycie:
#  ./host.py domena
#

import socket, sys                                                  # [1]

def getHostIP(domain_name):
    ip_addr = socket.gethostbyname(domain_name)                     # [2]
    return ip_addr

if __name__ == '__main__':
    domain = sys.argv[1]
    print "Adres IP dla domeny %s to %s" % (domain, getHostIP(domain))
```

Architektura klient-serwer

```
1  #!/usr/bin/python
2  #
3  # KLIENT
4  #
5  import socket
6
7  def sendPacket():
8      proto = socket.getprotobyname('tcp')                              # [1]
9      s = socket.socket(socket.AF_INET, socket.SOCK_STREAM, proto)      # [2]
10     try:
11         s.connect(("127.0.0.1", 2222))                                # [3]
12         s.send("Hello world")                                         # [4]
13
14         resp = s.recv(1024)                                           # [5]
15         print resp
16     except socket.error:
17         pass
18     finally:
19         s.close()
20
21 if __name__ == '__main__':
22     sendPacket()
```

```
#!/usr/bin/python
#
# SERWER
#

import socket

def server():
    proto = socket.getprotobyname('tcp')                                    # [1]
    serv = socket.socket(socket.AF_INET, socket.SOCK_STREAM, proto)

    serv.bind(("localhost", 2222))                                          # [2]
    serv.listen(1)                                                          # [3]
    return serv

serv = server()

while 1:
    conn, addr = serv.accept()                                              # [4]
    while 1:
        message = conn.recv(64)                                             # [5]
        if message:
            conn.send('Hi, I am a server, I received: ' + message)
        else:
            break
    conn.close()
```

```
#!/usr/bin/python

import socket,sys

def main(dest_name):
    dest_addr = socket.gethostbyname(dest_name)                                 # [1
    port = 33434
    max_hops = 30
    icmp = socket.getprotobyname('icmp')
    udp = socket.getprotobyname('udp')
    ttl = 1
    while True:
        recv_socket = socket.socket(socket.AF_INET, socket.SOCK_RAW, icmp)      # [2
        send_socket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM, udp)
        send_socket.setsockopt(socket.SOL_IP, socket.IP_TTL, ttl)               # [3
        recv_socket.bind(("", port))                                            # [4
        send_socket.sendto("", (dest_name, port))                               # [5
        curr_addr = None
        curr_name = None
        try:
            _, curr_addr = recv_socket.recvfrom(512)
            curr_addr = curr_addr[0]
            try:
                curr_name = socket.gethostbyaddr(curr_addr)[0]
            except socket.error:
                curr_name = curr_addr
        except socket.error:
            pass
        finally:
            send_socket.close()
            recv_socket.close()

        if curr_addr is not None:
            curr_host = "%s (%s)" % (curr_name, curr_addr)
        else:
            curr_host = "*"
        print "%d\t%s" % (ttl, curr_host)

        ttl += 1
        if curr_addr == dest_addr or ttl > max_hops:
            break

if __name__ == "__main__":
    host = sys.argv[1]
    main(host)
```

C/C++

- można wykorzystać moduł **SFML Network** który daje prosty i wygodny dostęp do komunikacji między komputerami.

Połączenie TCP
Tworzenie gniazda klienta

C/C++

```
// klient

sf::TcpSocket socket; // tworzymy gniazdo klienta
sf::IpAddress ip = "adres.ip.serwera.do.ktorego.sie.chcesz.polaczyc";
unsigned int port = 54000 // port na którym nasłuchuje serwer

if( socket.connect( ip, port ) != sf::Socket::Done ) // łączymy się z adresem 'ip' na porcie 'port'
// jeśli funkcja connect zwróci sf::Socket::Done oznacza to, że wszystko poszło dobrze
{
    cerr << "Nie można połączyć się z " << ip.toString() << endl;
    exit( 1 );
}
```

`getLocalAddress()` zwraca lokalne IP.
`getLocalPort()` zwraca lokalny port.
`getRemoteAddress()` zwraca zdalne IP.
`getRemotePort()` zwraca zdalny port.

Tworzenie gniazda serwera TCP

C/C++

```
// serwer

sf::TcpListener listener; // tworzymy gniazdo nasłuchujace
unsigned int port = 54000; // port, na którym będziemy nasłuchiwać

if( listener.listen( port ) != sf::Socket::Done ) // rozpoczynamy nasłuchiwanie na porcie 'port'
{
    cerr << "Nie mogę rozpocząć nasłuchiwania na porcie " << port << endl;
    exit( 1 );
}

//...
```

C/C++

```
// serwer

while( /*...*/ )
{
    sf::TcpSocket client; // tworzymy gniazdo, dzięki któremu będziemy mogli się komunikować z klientem
    listener.accept( client );

    // wysyłanie/odbieranie danych od/do klienta
}
```

Komunikacja między połączonymi gniazdami TCP

C/C++

```
// sposób działa dla klienta i dla serwera

const int datasize = 100; // rozmiar bloku danych
char data[ 100 ] = "..."; // tworzymy blok danych

//...

// wysyłanie danych
if( socket.send( data, datasize ) != sf::Socket::Done ) // i wysyłamy...
{
    // nie można wysłać danych (prawdopodobnie klient/serwer się rozłączył)
    cerr << "Nie można wysłać danych!\n";
    exit( 1 );
}

//...

// odbieranie danych
unsigned int received; // do tej zmiennej zostanie zapisana ilość odebranych danych
if( socket.receive( data, datasize, received ) != sf::Socket::Done ) // i wysyłamy...
{
    // nie można odebrać danych (prawdopodobnie klient/serwer się rozłączył)
    cerr << "Nie można odebrać danych!\n";
    exit( 1 );
}
else
     cout << "Odebrano " << received << " bajtów\n";
```

Komunikacja za pomocą UDP

C/C++

```
sf::UdpSocket sock; // tworzymy gniazdo

const int datasize = 100; // rozmiar danych do wysłania/odebrania
char data[ datasize ] = "..."; // dane

// wysyłanie
sf::IpAddress ip( "adres.komputera.do.którego.chcesz.wysłać.dane" );
unsigned int port = 56000; // port, na który chcesz wysłac dane
if( sock.send( data, datasize, ip, port ) != sf::Socket::Done )
{
    cerr << "Nie można wysłać danych!\n";
    exit( 1 );
}

// odbieranie
sf::IpAddress ip( "adres.komputera.od.którego.chcesz.odebrać.dane" );
unsigned int port = 56000; // port, na którym chcesz odbierać dane
unsigned int senderport = 54000; // port, z którego zostanły wysłane dane
unsigned int received;
sock.bind( port );
if( sock.receive( data, datasize, received, ip, senderport ) != sf::Socket::Done )
{
    cerr << "Nie można odebrać danych!\n";
    exit( 1 );
}
cout << "Odebrano bajtów: " << received << endl;
```

Problem blokujących gniazd

C/C++

```
// dobrze

sf::TcpListener server; // gniazdo nasłuchujące
vector < sf::TcpSocket *> clients; // tutaj przechowujemy klientów

sf::SocketSelector sel; // selektor
sel.add( server ); // dodajemy gniazdo nasłuchujące
//...
while( true )
// pętla główna serwera
{
    if( sel.wait( sf::seconds( 2 ) ) // jeśli metoda wait() zwróci true, to znaczy, że któreś z dodanych gniazd jest gotowe do odbioru
    // jako argument podajemy czas, przez który ma czekać na dane
    {
        if( sel.isReady( server ) ) // metoda isReady() sprawdza, czy dane gniazdo ma dane do odebrania
        // jeśli do motedy isReady() przekażemy gniazdo nasłuchujące, true oznacza, że ktoś chce się do niego podłaczyć
        {
            TcpSocket * tmp = new sf::TcpSocket;
            server.accept( * tmp ); // skoro ktoś chce się do nas połączyć, to go akceptujemy
            clients.push_back( tmp ); // i dodajemy go do listy
            sel.add( * tmp ); // oraz do selektora, żeby można było od niego odbierać dane
            // nie zapomnij, by usunąć(za pomocą delete) gniazdo, kiedy się rozłączy
        }

        // pętla przechodząca po kontenerze gniazd (zależy od typu kontenera)
        for( int i = 0; i < clients.size(); i++ ) // u nas to jest for i indeks i
        {
            if( sel.isReady( * clients[ i ] ) ) // *clients[i] coś nam wysłał
            {
                const int datasize = 100; // rozmiar danych do odebrania
                char data[ datasize ]; // dane
                unsigned int received; // odebrane
                clients[ i ]->receive( data, datasize, received );
                cout << "Odebrano " << received << " bajtów od " << clients[ i ]->getRemoteAddress() << endl;
                // tutaj robimy coś z odebranymi dany
                //...
            }
        }
        //...
        // reszta kodu serwera
    }
```

Pakiety

1) Jeśli wyślesz np. 1kB danych to dotrą one (prawdopodobnie) podzielone na klka części. Serwer odbierze pierwszą serię danych, a później nie będzie wiedział, czy ma oczekiwać na następną, czy to już wszystko.
2) Kiedy wysyłasz dane do jakiejś maszyny, nie wiesz, czy ma ona np. ten sam rozmiar typu int, albo czy używa ona konwencji little-endian (bajty "bardziej znaczące" są na początku), czy big-endian (na końcu).

Jak odbierać dane za pomocą pakietów:

1)Tworzymy zmienną typu sf::Packet

2)Odbieramy dane za pomocą metody receive()

3)Wyciągamy dane z pakietu jak ze strumienia istream z biblioteki standardowej

C/C++

```
sf::Packet pak; // 1
someSocket.receive( pak ); // 2

float x;
int y;
string z;
pak >> x >> y >> z; // 3
```

Jak wysyłać dane z pomocą pakietów:

1) Tworzymy zmienną typu sf::Packet
2) Zapisujemy dane do pakietu jak do strumienia ostream z biblioteki standardowej
3) Wysyłamy dane za pomocą metody send()

C/C++

```
sf::Packet pak; // 1

float x = 3.14f;
int y = 64;
string z = "hello!";
pak << x << y << z; // 2
someSocket.send( pak ); // 3
```