

## Module 2

### Sockets/Part 2

Prepared by Dr. Manal Mizher

## Lecture 3

جامعة الجميع الذكية

تسعى لتكون الكيان الأول الذي يتبنى مبدأ المجتمع التعليمي  
التعاوني ويقدم العلم مقابل العمل لتنمية وبناء المجتمعات.

# Network programming

Course ID: CYB615



جامعة الجميع الذكية  
EVERYONE'S SMART UNIVERSITY

### TCP/IP Application Protocols

Here is a non-exhaustive list of Internet application protocols:

- SMTP: Simple Mail Transfer Protocol
- SSH: Secure Shell
- FTP: File Transfer Protocol
- SFTP: Secure File Transfer protocol
- SCP: Secure Copy Protocol
- DHCP: Dynamic Host Configuration protocol
- HTTP: Hyper Text Transfer Protocol
- POP: Post Office Protocol (the version currently in use is 3, POP3)
- IMAP: Internet Message Access Protocol

### Quick review:

**TCP/IP:** is actually a collection of protocols. The majority of communications in use today use the TCP protocol.

**Addressing:** In order to make the packet scheme work, there are several details that TCP must take care of:

1. the remote machine (IP address)
2. which application on the remote machine to communicate with (Port)

**DNS protocol:** Domain name system -> mapping between names and numbers of addresses (works with UDP).

### Quick review:

**Port Number:** There's actually an official list of assigned port numbers maintained by the Internet Assigned Numbers Authority (IANA) at [www.iana.org](http://www.iana.org)

- If you're writing a server for a service that isn't on that list, you should pick a port number that's **greater than 1024** and doesn't occur on your list. That will make it least likely to conflict with any other server. Port numbers can be **as high as 65535**.
- Port number of a client is unimportant. Usually, the client will let the operating system pick a random port number(**short-lived**).

### Quick review:

**Reliability:** data should get through to the other end undamaged, complete, unmodified, and in the correct order. This is accomplished with the use of several different algorithms.

**Routing:** routing table, routing protocols, and why do we use netmask?

- Differences between direct (unicast), multicast, broadcast.

**Security:** There are several approaches for security:

- **Secure Sockets Layer (SSL):** is a layer that's normally incorporated in application code above a TCP connection. It provides authentication of the server (so you know you're talking to the machine you think you should be), encryption (so nobody else can read your communications), and data integrity (so no packets can be modified in route without being detected).
- **Transport Layer Security (TLS):** TLS is very similar to SSL, but is included as a part of the protocol stack.

### Quick review:

#### You should use TCP if:

- You need a reliable data transport that ensures that your data arrives complete and undamaged.
- Your protocol requires more than just a single request and a single response from the server.
- You need to send more than a small amount of data across the network.
- A small delay establishing the initial connection is tolerable.



In terminal, run : `netstat -s`

#### You should use UDP if:

- You're unconcerned if some packets don't arrive or arrive out of order, or you can detect and handle this condition yourself.
- Your protocol consists of brief requests and responses.
- You need the conversation setup to be as quick as possible.
- Only a small amount of data is being sent. UDP is limited to 64 KB of data in a single packet, but people often stay below 1 KB when using UDP.

## Module 2

### Sockets/Part 2

Computer Networks Protocols(UDP\_TCP\_IP)

### Lecture 3

**PYTHON  
NETWORK  
PROGRAMMING**



جامعة الجميع الذكية  
EVERYONE'S SMART UNIVERSITY

## (Sockets/part 2\_ Computer Networks Protocols(UDP\_TCP\_IP))

1. Converting an IPv4 address to different formats
2. Converting integers to and from host to network byte order
3. Setting and getting the default socket timeout
4. Handling socket errors gracefully
5. Modifying a socket's send/receive buffer size
6. Changing a socket to the blocking/non-blocking mode
7. Reusing socket addresses
8. Printing the current time from the internet time server
9. Writing an SNTP client
10. Writing a simple UDP echo client/server application
11. Writing a simple TCP echo client/server application

## Converting an IPv4 address to different formats

- When you would like to deal with low-level network functions, sometimes, the usual string notation of IP addresses are not very useful. They need to be converted to the packed 32-bit binary formats → `inet_aton()` and `inet_ntoa()`
- Converts an IP address, which is in 32-bit packed format to the popular human readable dotted-quad string format and vice-versa.

### ▪ Prototypes

`inet_ntoa(ip_address)` where:

- `ip_address` - The address in 32-bit packed format which is to be converted.
- Returns the IP address in dotted quad-string format.



# Module 2

## Converting an IPv4 address to different formats

```
import socket
from binascii import hexlify

def convert_ip4_address():
    for ip_addr in ['127.0.0.1', '192.168.0.1']:
        packed_ip_addr = socket.inet_aton(ip_addr)
        unpacked_ip_addr = socket.inet_ntoa(packed_ip_addr)
        print ("IP Address: %s => Packed: %s, Unpacked: %s"
              %(ip_addr, hexlify(packed_ip_addr), unpacked_ip_addr))
        print ("IP Address: %s => Packed: %s, Unpacked: %s" %
              (ip_addr, packed_ip_addr, unpacked_ip_addr))

if __name__ == '__main__':
    convert_ip4_address()
```

```
import socket
import struct

IPLong = 168496141
IP32Bit = struct.pack('!l', IPLong)
IPQuad = socket.inet_ntoa(IP32Bit)
print ('IP32Bit: %s => IPQuad : %s' % (IP32Bit, IPQuad))
```



## Converting integers to and from host to network byte orders

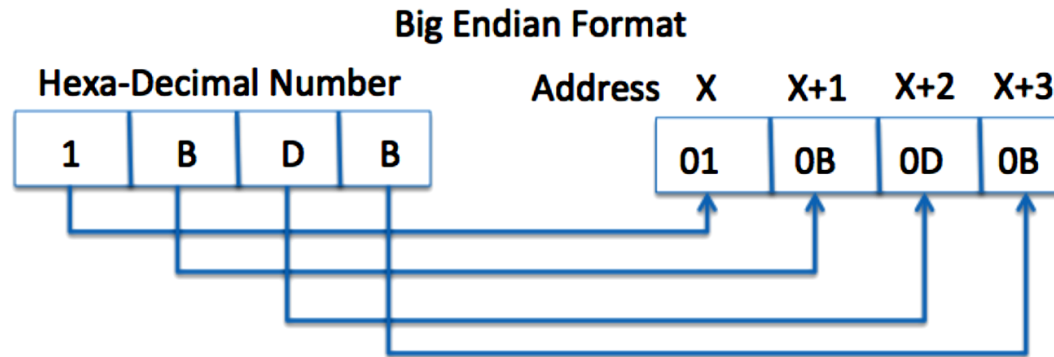
- Handle the low-level data transmission over the wire between two machines. This operation requires some sort of conversion of data from the native host operating system to the network format and vice versa. This is because each one has its own specific representation of data → `ntohl()` and `htonl()`
- Computing world mainly handles data in two formats: **Big Endian Format** and **Little Endian Format**
  1. Most machines handle data in Little Endian Format.
  2. Network transmission and the Internet is using Big Endian Format.
  3. Java runtimes use Big Endian Format.
  4. Python's Endianness is same as the processor in which the Python interpreter is run.
  5. The socket module of Python provides functions to handle translations of integers of different sizes from Little Endian to Big Endian and vice versa.



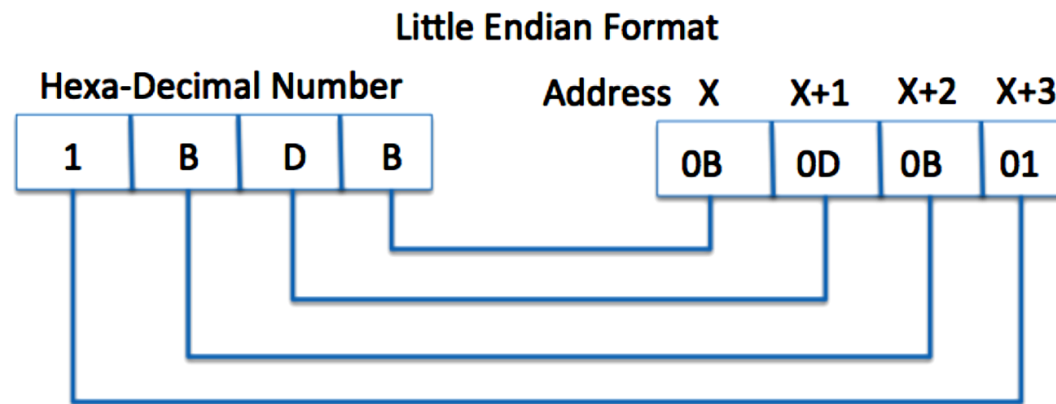
# Module 2

## Converting integers to and from host to network byte orders

### Big Endian and Little Endian Formats



most significant byte(MSB)



least significant byte(LSB)

- The `socket` module of Python provides functions for translating host order to network order and vice versa.

| Python Function Name                         | Description                                                                                                                                                                                                                                                                                                                                                                  | Translation Direction       |
|----------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------|
| <code>socket.ntohl(Int32bit_positive)</code> | <ul style="list-style-type: none"> <li>• <code>ntohl()</code> function converts a 32 bit integer from network order to host order.</li> <li>• If the host order is same as network order, the function just executes a noop instruction.</li> <li>• <code>ntohl()</code> will raise an <code>OverflowError</code> if a negative value is passed.</li> </ul>                  | Big Endian to Little Endian |
| <code>socket.ntohs(Int32bit_positive)</code> | <ul style="list-style-type: none"> <li>• <code>ntohs()</code> function of <code>socket</code> module converts a 16 bit integer from network format to host format.</li> <li>• If the host order and the network order are same, then only a noop is executed.</li> <li>• <code>ntohs()</code> raises an <code>OverflowError</code> if a negative value is passed.</li> </ul> | Big Endian to Little Endian |
| <code>socket.htonl(Int32bit_positive)</code> | <ul style="list-style-type: none"> <li>• <code>htonl()</code> function converts a 32 bit positive integer from host byte order to network byte order.</li> <li>• If the host order and the network order are same, then only a noop is executed.</li> <li>• <code>htonl()</code> will raise an <code>OverflowError</code> if a negative value is passed.</li> </ul>          | Little Endian to Big Endian |
| <code>socket.htons(Int16bit_positive)</code> | <ul style="list-style-type: none"> <li>• <code>htons()</code> function converts a 16 bit positive integer from host byte order to network byte order.</li> <li>• If the host order and the network order are same, then only a noop is executed.</li> <li>• <code>htons()</code> raises an <code>OverflowError</code> if an negative value is passed.</li> </ul>             | Little Endian to Big Endian |

```
import socket
```

```
Int32Bit = 214748300  
Int16Bit = 400
```

```
# Convert a 32 bit integer from network byte order to host byte order  
Int32InHostFormat = socket.ntohl(Int32Bit)
```

```
# Convert a 16 bit integer from network byte order to host byte order  
Int16InHostFormat = socket.ntohs(Int16Bit)
```

```
print("32 bit integer {} converted from Network Byte Order to Host Byte Order: {}".format(Int32Bit, Int32InHostFormat))  
print("16 bit integer {} converted from Network Byte Order to Host Byte Order: {}".format(Int16Bit, Int16InHostFormat))
```

```
# Convert a 32 bit integer from host byte order to network byte order  
Int32InNetworkFormat = socket.htonl(Int32InHostFormat)
```

```
# Convert a 16 bit integer from network byte order to host byte order  
Int16InNetworkFormat = socket.htons(Int16InHostFormat)
```

```
print("32 bit integer {} converted from Host Byte Order to Network Byte Order: {}".format(Int32InHostFormat, Int32InNetworkFormat))  
print("16 bit integer {} converted from Host Byte Order to Network Byte Order: {}".format(Int16InHostFormat, Int16InNetworkFormat))
```

## Setting and getting the default socket timeout

- Socket timeouts can occur when attempting to connect to a remote server, or during communication, especially long-lived ones (TCP).
- Sometimes, you need to manipulate the default values of certain properties of a socket library, for example, the socket timeout → `gettimeout()` and `settimeout()`
- They can be caused by any connectivity problem on the network, such as:
  1. A network partition preventing the two machines from communicating.
  2. The remote machine crashing. This cannot easily be distinguished from a network partitioning.
  3. The remote machine is a VM and it has been destroyed. Again, to the calling application, this looks like a connectivity problem.



4. A change in the firewall settings of one of the machines preventing communication.
5. The settings are wrong and the client is trying to talk to the wrong machine, one that is not on the network. That could be an error in Hadoop configuration files, or an entry in the DNS tables or the /etc/hosts file.
6. If its over a long-haul network (i.e. out of cluster), it may be a transient failure due to the network playing up.
7. If you have some proxy settings, they may be wrong.
8. If using a client of an object store such as the Amazon S3 and OpenStack clients, socket timeouts may be caused by remote-throttling of client requests: your program is making too many PUT/DELETE requests and is being deliberately blocked by the far end. This is most likely to happen when creating many small files, or performing bulk deletes (e.g. deleting a directory with many child entries). It can also arise from a transient failure of the long-haul link.

## Setting and getting the default socket timeout

- Comparing this exception to the *Connection Refused* error, the latter indicates there is a server at the far end, but no program running on it can receive inbound connections on the chosen port. A Socket Timeout usually means that there is something there, but it or the network are not working right.
- Identifying and Fixing Socket Timeouts:
  1. The root cause of a Socket Timeout is a connectivity failure between the machines, so try the usual process.
  2. Check the settings: is this the machine you really wanted to talk to?
  3. From the machine that is raising the exception, can you resolve the hostname.
  4. Is that resolved hostname the correct one?



5. Is the target machine running the relevant Hadoop processes?
6. Can you telnet to the target host and port?
7. Can you telnet to the target host and port from any other machine?
8. On the target machine, can you telnet to the port using localhost as the hostname. If this works but external network connections time out, it's usually a firewall issue.
9. If it is a remote object store: is the address correct? Does it go away when you repeat the operation? Does it only happen on bulk operations? If the latter, it's probably due to throttling at the far end.

**Remember:** These are your network configuration problems. Only you can fix them.

## Setting and getting the default socket timeout

```
import socket

def test_socket_timeout():
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    print ("Default socket timeout: %s" % (s.gettimeout()))
    s.settimeout(100)
    print ("Current socket timeout: %s" % (s.gettimeout()))

if __name__ == '__main__':
    test_socket_timeout()
```



- In any networking application, it is very common that one end is trying to connect, but the other party is not responding due to networking media failure or any other reason → `socket.error`
- Use `argparse` to get a user input or input command.

## Module 2

# Handling socket errors gracefully

```
# First try-except block -- create socket
try:
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
except socket.error as e:
    print ("Error creating socket: %s" % e)
    sys.exit(1)
```

```
# Second try-except block -- connect to given host/port
try:
    s.connect((host, port))
except socket.gaierror as e:
    print ("Address-related error connecting to server: %s" % e)
    sys.exit(1)
except socket.error as e:
    print ("Connection error: %s" % e)
    sys.exit(1)
```



## Module 2

# Handling socket errors gracefully

```
# Third try-except block -- sending data
try:
    msg = "GET %s HTTP/1.0\r\n\r\n" % filename
    s.sendall(msg.encode('utf-8'))
except socket.error as e:
    print ("Error sending data: %s" % e)
    sys.exit(1)
```

```
while 1:
    # Fourth tr-except block -- waiting to receive data from remote host
    try:
        buf = s.recv(2048)
    except socket.error as e:
        print ("Error receiving data: %s" % e)
        sys.exit(1)
    if not len(buf):
        break
    # write the received data
    sys.stdout.write(buf.decode('utf-8'))
```

## Modifying a socket's send/receive buffer sizes

- **Maximum transmission unit (MTU):** in computer networking, MTU is the size of the largest protocol data unit that can be communicated in a single network layer transaction. The MTU relates to, but is not identical to the maximum frame size that can be transported on the data link layer, e.g. Ethernet frame (~1500 bytes). Thus, TCP for example uses the MTU to determine the maximum size of each segment in any transmission.
- **TCP:** for best match with hardware and network realities, the value of buffer size should be a relatively small power of 2, for example,  $4096 = 2^{12} = 4K$
- You should consider the max size of Ethernet packets (~1500 bytes) and the max size of TCP segments (~64K). You really want a buffer larger than the first (so 1024, is probably out of the range) and you probably don't need more than the latter. so go with 2K, 4K, 8K, 16K, 32K, or 64K.



## Modifying a socket's send/receive buffer sizes

- **Example (bottleneck):** Let's say you are using a 1023 byte (< 1k byte) buffer instead of 1024, and since a lot of data is being sent, then the TCP segment is maxed out at 64K. You'll have 64 iterations of 1023 bytes and a wasteful extra iteration of 64 bytes (Notes that there are 64 TCP headers where The minimum size header is 5 words (word e.g. 32 bits) and the maximum is 15 words thus giving the minimum size of 20 bytes and maximum of 60 bytes).
- On the opposite side, what is the problem if the buffer size is the largest one?
- **UDP datagram:** can have any size from 8 to 65,535 bytes (64k). Where it is carried in a single IP packet and is hence limited to a maximum payload of 65,507 bytes for IPv4 and 65,535 bytes for IPv6. The transmission of large IP packets usually requires IP fragmentation.



## Modifying a socket's send/receive buffer sizes

- `setsockopt()` function provides an application program with the means to control socket behavior. An application program can use `setsockopt()` to allocate buffer space, control timeouts, or permit socket data broadcasts → `getsockopt()` and `setsockopt()`
- `Setsockopt()` - set the socket options , while `getsockopt()` - get the socket options
- The default socket buffer size may not be suitable in many circumstances. In such circumstances, you can change the default socket buffer size to a more suitable value

## Modifying a socket's send/receive buffer sizes

- The `setsockopt()` method takes three arguments: `level`, `optname`, and `value`. Here, `optname` takes the option name and `value` is the corresponding value of that option. For the first argument, the needed symbolic constants can be found in the `socket` module (`SO_*etc.`).

```
self: socket, level: int, optname: int, value: Union[int, bytes]
```

```
self: socket, level: int, optname: int, value: None,  
optlen: int
```

```
s.setsockopt()
```

[https://www.masterraghu.com/subjects/np/introduction/unix\\_network\\_programming\\_v1.3/ch07lev1sec2.html](https://www.masterraghu.com/subjects/np/introduction/unix_network_programming_v1.3/ch07lev1sec2.html)

## Modifying a socket's send/receive buffer sizes

```
import socket

SEND_BUF_SIZE = 4096
RECV_BUF_SIZE = 4096

def modify_buff_size():
    sock = socket.socket( socket.AF_INET, socket.SOCK_STREAM )

    # Get the size of the socket's send buffer
    bufsize = sock.getsockopt(socket.SOL_SOCKET, socket.SO_SNDBUF)
    print("Buffer size [Before]:%d" %bufsize)

    sock.setsockopt(socket.SOL_TCP, socket.TCP_NODELAY, 1)
    sock.setsockopt(socket.SOL_SOCKET, socket.SO_SNDBUF, SEND_BUF_SIZE)
    sock.setsockopt(socket.SOL_SOCKET, socket.SO_RCVBUF, RECV_BUF_SIZE)
    bufsize = sock.getsockopt(socket.SOL_SOCKET, socket.SO_SNDBUF)
    print("Buffer size [After]:%d" %bufsize)

if __name__ == '__main__':
    modify_buff_size()
```



## Changing a socket to the blocking/non-blocking mode

- By default, TCP sockets are placed in a blocking mode. This means the control is not returned to your program until some specific operation is complete. If you call the `connect()` API, the connection blocks your program until the operation is complete → `setblocking(0)` and `setblocking(1)` where the value `0` in this method to make it non-blocking and value `1` to make it blocking.
- On many occasions, you don't want to keep your program waiting forever, either for a response from the server or for any error to stop the operation.
- For example, when you write a web browser client that connects to a web server, you should consider a stop functionality that can cancel the connection process in the middle of this operation. This can be achieved by placing the socket in the non-blocking mode.



## Changing a socket to the blocking/non-blocking mode

```
import socket

def test_socket_modes():
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    s.setblocking(1)
    s.settimeout(0.5)
    s.bind(("127.0.0.1", 0))
    socket_address = s.getsockname()
    print ("Trivial Server launched on socket: %s" %str(socket_address))
    while(1):
        s.listen(1)

if __name__ == '__main__':
    test_socket_modes()
```

- You want to run a socket server always on a specific port even after it is closed intentionally or unexpectedly. This is useful in some cases where your client program always connects to that specific server port. So, you don't need to change the server port. → `setsockopt(): SO_REUSEADDR`
- If you run a Python socket server on a specific port and try to rerun it after closing it once, you won't be able to use the same port. It will usually throw an error like the following Command:

```
socket.error: [Errno 98] Address already in use
```

Solution



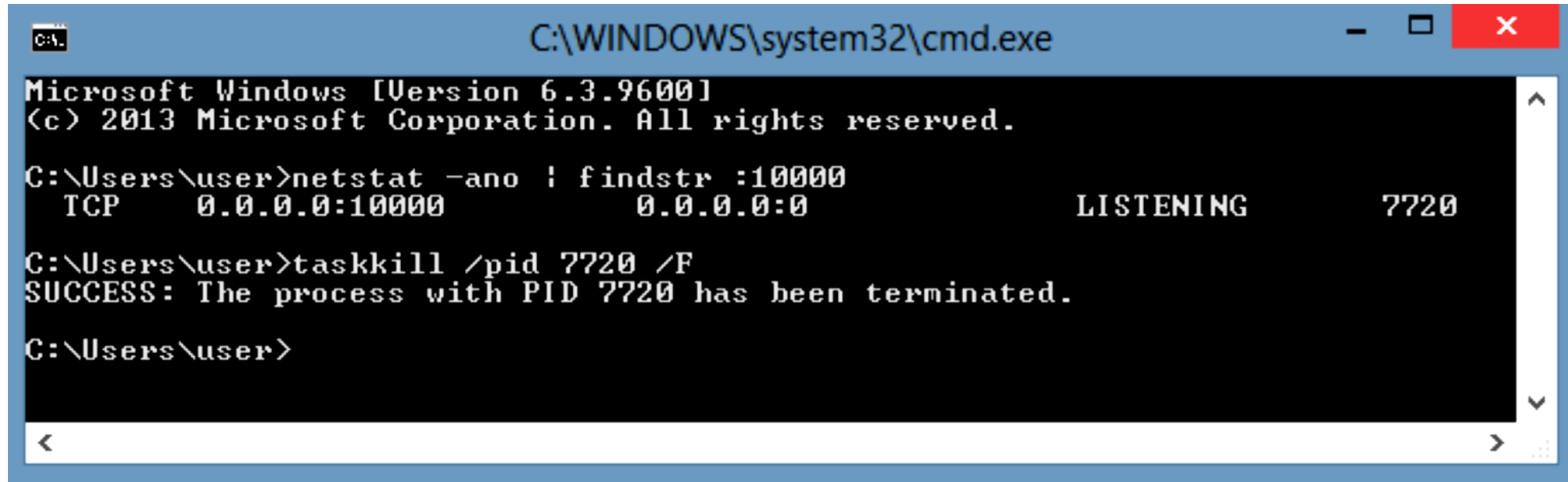
### Solution:

1. Enable the socket reuse option `so_REUSEADDR` after you create a socket object, you can query the status of address reuse, like the old state.
2. Call `setsockopt ()` method to modify the value of the address reuse state.
3. Follow the normal steps to bind the socket to an address.
4. listen for incoming client connections.

```
import socket
import sys

def reuse_socket_addr():
    sock = socket.socket( socket.AF_INET, socket.SOCK_STREAM )
    old_state = sock.getsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR)
    print("Old sock state: {}".format(old_state))
    sock.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
    new_state = sock.getsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR)
    print("New sock state: {}".format(new_state))
    local_port = 8910
    srv = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    srv.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
    srv.bind(("", local_port))
    srv.listen(1)
    print("Listening on port: {}".format(local_port))
    while True:
        try:
            connection, addr = srv.accept()
            print('Connected by {}: {}'.format(addr[0], addr[1]))
        except KeyboardInterrupt:
            break
        except socket.error:
            print('{}'.format(socket.error,))

if __name__ == '__main__':
    reuse_socket_addr()
```



```
C:\WINDOWS\system32\cmd.exe

Microsoft Windows [Version 6.3.9600]
(c) 2013 Microsoft Corporation. All rights reserved.

C:\Users\user>netstat -ano | findstr :10000
    TCP        0.0.0.0:10000          0.0.0.0:0             LISTENING       7720

C:\Users\user>taskkill /pid 7720 /F
SUCCESS: The process with PID 7720 has been terminated.

C:\Users\user>
```

- netstat -ano
- netstat -ano | findstr :**Port number**
- Taskkill /pid (**process id**) /F



- To avoid `TIME_WAIT` state → ensure that the remote end initiates the closure (close event):
  - Server lets the client close first → The application protocol must be designed so that the client knows when to close.
  - Server can safely close in response to an EOF from the client, however it will also need to set a timeout when it is expecting an EOF in case the client has left the network ungracefully.
  - In many cases simply waiting a few seconds before the server closes will be adequate.
- `netstat` (`netstat -l`, `netstat -o`) command used to see which programs (program\_name, pid) tuple) are binded to which ports and what is the socket current state: `TIME_WAIT`, `CLOSING`, `FIN_WAIT` and so on. (`ctrl+c` OR `ctrl+break` to terminate running command-line)

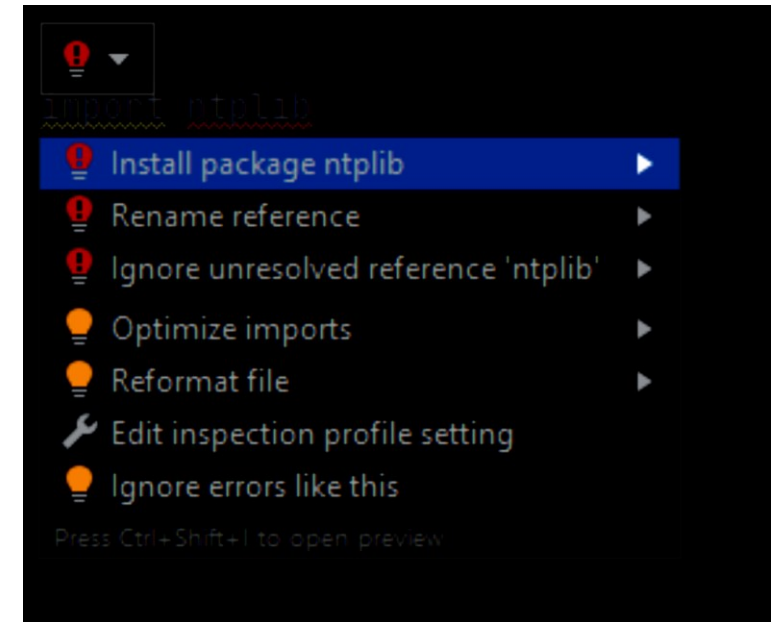
**There are two reasons for the `TIME_WAIT` state:**

1. To implement TCP's full-duplex connection termination reliably
2. To allow old duplicate segments to expire in the network
  - The duration for remaining in this state is twice the maximum segment lifetime (MSL), sometimes called 2MSL.
  - The recommended value in RFC 1122 [Braden 1989] is 2 minutes, although Berkeley-derived implementations have traditionally used a value of 30 seconds instead.

# Module 2

## Printing the current time from the internet time server

- Many programs rely on the accurate machine time, such as the make command in UNIX. Your machine time may be different and need synchronizing with another time server in your network.
- In order to synchronize your machine time with one of the internet time servers, you can write a Python client for that. For this, `ntplib` library will be used. Here, the client/server conversation will be done using **Network Time Protocol (NTP)**.
- If `ntplib` is not installed on your machine.. 



## Printing the current time from the internet time server

```
import ntplib
from time import ctime

def print_time():
    ntp_client = ntplib.NTPClient()
    response = ntp_client.request('pool.ntp.org')
    print (ctime(response.tx_time))

if __name__ == '__main__':
    print_time()
```

- The [pool.ntp.org](https://www.pool.ntp.org/) project is a big virtual cluster of timeservers providing reliable easy to use NTP service for millions of clients.

- Unlike `ntplib`, sometimes, you don't need to get the precise time from the NTP server. You can use a simpler version of NTP called simple network time protocol.
- **Solution:**
  1. Define two constants: `NTP_SERVER` and `TIME1970`. `NTP_SERVER` is the server address to which our client will connect, and `TIME1970` is the reference time on January 1, 1970 (also called Epoch). You may find the value of the Epoch time or convert to the Epoch time at <http://www.epochconverter.com/>.
  2. The actual client creates a UDP socket(`SOCK_DGRAM`) to connect to the server following the UDP protocol.
  3. The client then needs to send the SNTP protocol data (`'\x1b' + 47 * '\0'`) in a packet. Our UDP client sends and receives data using the `sendto()` and `recvfrom()` methods.
  4. When the server returns the time information in a packed array, the client needs a specialized struct module to unpack the data.
  5. The only interesting data is located in the 11th element of the array. Finally, we need to subtract the reference value, `TIME1970`, from the unpacked value to get the actual current time.

```
import socket
import struct
import sys
import time

NTP_SERVER = "0.uk.pool.ntp.org"
TIME1970 = 2208988800

def sntp_client():
    client = socket.socket( socket.AF_INET, socket.SOCK_DGRAM )
    data = '\x1b' + 47 * '\0'
    client.sendto( data.encode('utf-8'), ( NTP_SERVER, 123 ) )
    data, address = client.recvfrom( 1024 )
    if data:
        print ('Response received from:', address)
        t = struct.unpack( '!12I', data )[10]
        t -= TIME1970
        print ('\tTime=%s' % time.ctime(t))

if __name__ == '__main__':
    sntp_client()
```

**pool.ntp.org:**  
*public ntp time server for everyone*



## Writing a simple UDP echo client/server application

```
>>>>
host = 'localhost'
data_payload = 2048

def echo_server(port):
    """ A simple echo server """
    # Create a UDP socket
    sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)

    # Bind the socket to the port
    server_address = (host, port)
    print ("Starting up echo server on %s port %s" % server_address)
    sock.bind(server_address)

    while True:
        print ("Waiting to receive message from client")
        data, address = sock.recvfrom(data_payload)
        print ("received %s bytes from %s" % (len(data), address))
        print ("Data: %s" %(data))
        if data:
            sent = sock.sendto(data, address)
            print ("sent %s bytes back to %s" % (sent, address)) >>>>>
```

## Writing a simple TCP echo client/server application

```
>>>>
host = 'localhost'
data_payload = 2048
backlog = 5

def echo_server(port):
    """ A simple echo server """
    # Create a TCP socket
    sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    # Enable reuse address/port
    sock.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
    # Bind the socket to the port
    server_address = (host, port)
    print ("Starting up echo server on %s port %s" % server_address)
    sock.bind(server_address)
    # Listen to clients, backlog argument specifies the max no. of queued connections
    sock.listen(backlog)
    while True:
        print ("Waiting to receive message from client")
        client, address = sock.accept()
        data = client.recv(data_payload)
        if data: >>>>
```

- The **backlog** parameter to **listen()** sets the maximum number of incoming connections the operating system queues for the application.
- Queued incoming connections are taken of this backlog-queue in the moment the application successfully **accept()**s a connection.
- Backlogged connections are the incoming connections that happen to get into the system when you are in:
  1. the process of duplicating the socket.
  2. forking a new process and **accept(2)**ing them.
- Normally the system allocates a default value of 5 for this queue, almost all time enough for normal purposes.

- Q1:a)** By using windows command-line :`netstat -ano |findstr :1500` where the assumed port number =1500, run the TCP client/server application, then capture the command-line results in these cases?
- b) Run then capture taskkill command-line if possible?
  - c) Explain logically why the results are distinct between these cases?

Case1 (server side) :

```
while True:
    try:
        connection, addr = srv.accept()
        print('Connected by %s:%s' % (addr[0], addr[1]))
        data = connection.recv(data_payload)
        #
        # if data:
        #     print("Data: %s" % data)
        #     connection.send(data)
        #     print("sent %s bytes back to %s" % (data, addr))
        # end connection
        # connection.close()
    except KeyboardInterrupt:
        break
    except socket.error as msg:
        print('%s' % (msg))
```

Case2 (server side) :

```
while True:
    try:
        connection, addr = srv.accept()
        print('Connected by %s:%s' % (addr[0], addr[1]))
        data = connection.recv(data_payload)
        if data:
            print("Data: %s" % data)
            connection.send(data)
            print("sent %s bytes back to %s" % (data, addr))
        # end connection
        # connection.close()
    except KeyboardInterrupt:
        break
    except socket.error as msg:
        print('%s' % (msg))
```

Case3 (server side) :

```
while True:
    try:
        connection, addr = srv.accept()
        print('Connected by %s:%s' % (addr[0], addr[1]))
        data = connection.recv(data_payload)
        if data:
            print("Data: %s" % data)
            connection.send(data)
            print("sent %s bytes back to %s" % (data, addr))
        # end connection
        connection.close()
    except KeyboardInterrupt:
        break
    except socket.error as msg:
        print('%s' % (msg))
```