Understanding Non-Blocking I/O in Python Socket Programming

- A function is blocking if it has to wait for something to complete.
- So if a function is blocking (for whatever reasons), it is capable of delaying execution of other tasks. And the overall progress of the entire system may get suffered.
- If the function is blocking because it is doing some CPU task, well then we cannot do much. But if it is blocking because of I/O, we know that the CPU is idle and can be used for starting another task that needs CPU.

Server.py

```
import socket
import sys

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

port = 1234 if len(sys.argv) == 1 else
int(sys.argv[1])
sock.bind(('localhost', port))
sock.listen(5)

try:
   while True:
       conn, info = sock.accept()

       data = conn.recv(1024)
       while data:
           print(data)
           data = conn.recv(1024)
except KeyboardInterrupt:
   sock.close
```

Client.py

```
import socket

sock = socket.socket(socket.AF_INET,
socket.SOCK STREAM)
sock.connect(('localhost', 1234))

data = 'foobar\n' * 10 * 1024 * 1024  #
70 MB of data
assert sock.send(data) == len(data)  #
True
```

Client.py

```
import socket

sock = socket.socket(socket.AF_INET,
socket.SOCK_STREAM)
sock.connect(('localhost', 1234))

data = 'foobar\n' * 10 * 1024 * 1024  #
70 MB of data
assert sock.send(data) == len(data)  #
True
```

Let's make this non-blocking:

```
import socket

sock = socket.socket(socket.AF_INET,
socket.SOCK_STREAM)
sock.connect(('localhost', 1234))
sock.setblocking(0)

data = 'foobar\n' * 10 * 1024 * 1024  #
70 MB of data
assert sock.send(data) == len(data)  #
AssertionError
```

- When you run the modified client code, you will notice that it did not block at all.
- There is a problem with the client — it did not send all the data. `socket.send` method returns the number of bytes sent.
- When you make a socket non-blocking by calling `setblocking(0)`, it will never wait for the operation to complete. So when you call the `send()` method, it will put as much data in the buffer as possible and return.

Fixed This:

- In this modified code, we make sure that we keep trying to send the remaining data as long as we have not sent all of it.
- When the write buffer is full and cannot accommodate more data, `EAGAIN` error is raised asking us to try again.
- If you examine the exception object, the exception message is "Resource temporarily unavailable". So we keep trying to send the remaining data until we have sent it all.

```
import errno
import select
import socket

sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
sock.connect(('localhost', 1234))
sock.setblocking(0)

data = 'foobar\n' * 1024 * 1024
data size = len(data)
print 'Bytes to send: ', len(data)

total sent = 0
while len(data):
   try:
       sent = sock.send(data)
       total sent += sent
       data = data[sent:]
       print 'Sending data'
   except socket.error, e:
       if e.errno != errno.EAGAIN:
           raise e
       print 'Blocking with', len(data), 'remaining'
       select.select([], [sock], [])  # This blocks until

assert total_sent == data_size  # True
```

Understanding select()

- The last line of the above example introduces the select module. select module helps us with dealing with multiple file descriptors at once.
- Since we made our socket non-blocking, we don't know when can we actually write to it unless we keep trying to write to it and expect it to not fail.
- This is a major waste of CPU time. In this example, we call the `select()` function to avoid exactly that.
- `select()` expects three arguments - list of file descriptors to watch for reading, list of file descriptors to watch for writing and list of file descriptors to watch for errors.
- Timeout can be passed as an optional 4th argument which can be used to prevent `select()` from blocking indefinitely.
- It returns a subset of all the three lists passed in the same order i.e. all the file descriptors that are ready for reading, writing or have caused some error.

```
import errno
import select
import socket

sock = socket.socket(socket.AF_INET,
socket.SOCK_STREAM)
sock.connect(('localhost', 1234))
sock.setblocking(0)

data = 'foobar\n' * 1024 * 1024
data_size = len(data)
print 'Bytes to send: ', len(data)

total_sent = 0
while len(data):
    try:
        sent = sock.send(data)
        total_sent += sent
        data = data[sent:]
        print 'Sending data'
    except socket.error, e:
        if e.errno != errno.EAGAIN:
            raise e
        print 'Blocking with', len(data), 'remaining'
        select.select([], [sock], [])  # This blocks
until

assert total_sent == data_size  # True
```

But, How does Select work?

- We call the `select()` function and pass it file descriptors asking it to tell us which of these are ready for reading or writing.
- In this example, `select()` blocks if there is no file descriptor that is ready to work with.
- You might say that this is still blocking the execution of our program but this is just the foundation for building better things. As of now, `select()` will just block until our sock object becomes writeable again.
- If we remove that line, our script will continue to work but a lot more useless while loop iterations will be run as most of them will result in exceptions.

```
while len(data):
    try:
        sent = sock.send(data)
        total_sent += sent
        data = data[sent:]
        print 'Sending data'
    except socket.error, e:
        if e.errno != errno.EAGAIN:
            raise e
        print 'Blocking with',
len(data), 'remaining'
        select.select([], [sock], [])
# This blocks until

assert total_sent == data_size  # True
```

Snippet of code from the modified client

For the curious ones, you can read more about in the man page for select and at these links:

http://en.wikipedia.org/wiki/Select_(Unix)

http://www.quora.com/Network-Programming/How-is-select-implemented

Introduction to event loops for network events

- Now that we understand `select` better, lets make use of it to do better than our last example where we actually make use of making a socket non-blocking.
- We are going to make use of generators to make sure that our script does not block execution of other things and let other code proceed as well.
- Consider this example.