Python Ready-Responders

Roy Ratcliffe[1,*]

Abstract

This technical note delves into the implementation of a socket-ready multiplexer in synchronous, non-threading environments using the functional paradigm in Python on Windows. It emphasises the advantages of a simpler approach to socket multiplexing and the importance of evaluating use cases before committing to production-grade software development work.

Keywords: Sockets, Python, Threads

Suppose that you want to manage multiple simultaneous socket connections. This is a technical note on how it might be done in a synchronous environment, i.e. without threading. Say that you also want to perform socket operations in a functional paradigm; mainly because of personal preference. Call the required functionality a "socket-based synchronous multiplexor."

"But why?" might be a good question. There are scenarios where a full-blown multi-threaded approach might **not** be apropos. Threads add complexity; not least due to synchronisation requirements but also other considerations including signal handling. It is better not to add complexity before the software project requires it. Normally, developers need to evaluate a use case before committing to production-grade software development work. Call this approach "fake it before you make it!" This is the scenario where a simpler approach to socket multiplexing in software helps.

Let Python be the implementation language and let Windows be the platform.

Socket-Selection Generator

This is a solution in Python. See the listing below. Also available in Gist form.

```
from select import select
from dataclasses import dataclass
import socket

@dataclass
class Read:
    socket: any

@dataclass
class Write:
    socket: any

@dataclass
```

*Corresponding author
Email address: roy@ratcliffe.me (Roy Ratcliffe)
[1]See more at GitHub.

```
class Except:
    socket: any

class Sockets:
    """
    Zero or more sockets on which ready events yield by selection.
    """
    socks = []

    def ready(self, timeout=None):
        """
        Yields zero or more socket-ready events.
        """
        if len(self.socks) != 0:
            rlist, wlist, xlist = select(self.socks, self.socks, self.socks, timeout)
            for r in rlist:
                yield Read(r)
            for w in wlist:
                yield Write(w)
            for x in xlist:
                yield Except(x)

    def add(self, sock):
        self.socks.append(sock)
        return sock

    def bind_dgram(self, address):
        """
        Binds a datagram socket to an address.
        Datagram sockets do not listen with backlog nor accept.
        Sockets default to the Internet address family.
        """
        sock = socket.socket(type=socket.SOCK_DGRAM)
        sock.bind(address)
        return self.add(sock)

    def connect_dgram(self, address):
        """
        Connects a datagram socket.
        """
        sock = socket.socket(type=socket.SOCK_DGRAM)
        sock.connect(address)
        return self.add(sock)

    def remove(self, sock):
        self.socks.remove(sock)
```

Toy Use Case

The exemplar listed below runs a server-client demonstration. Ready events drive the demonstration forward.

The code creates two datagram sockets: one for the server and another for the client. It runs for 10 monotonic seconds while the client sends a rolling counter and the server prints the counter's byte encoding. Just something

very simple helps to illustrate how a pseudo-threaded approach can respond to ready events *without* synchronisation machinations.

```
import socket_select
import time
import sys

sockets = socket_select.Sockets()
server = sockets.bind_dgram(('', 9876))
client = sockets.connect_dgram(('localhost', 9876))

# Shared memory context for demonstration purposes.
# Access is synchronous between ready-responders.
count = 0
start = time.monotonic()

while time.monotonic() < start + 10:
    readies = list(sockets.ready(1.0))
    # In practice, there will always be sockets ready unless they all close.
    if len(readies) == 0:
        break
    for ready in readies:
        match ready:
            case socket_select.Write(sock) if sock == client:
                sock.send(str(count).encode())
                count += 1
                if count == 10:
                    sock.close()
                    sockets.remove(sock)
            case socket_select.Read(sock) if sock.getsockname()[1] == 9876:
                # Demonstrates a guard condition by socket port.
                # Asking for the peer name raises an exception if not connected.
                # This includes bound datagram sockets.
                data = sock.recv(1024)
                print(data)

sys.exit(0)
```

Effectively the ready events act as stream entries, albeit without timestamps. The responders become stream processors applying functions to entries, accumulating state and triggering side effects along the way.

Conclusions

Advantages of this simple approach include:

- File descriptor "select" operations wrap their sockets in the ready context: read, write or except. This makes pattern matching by socket-ready condition along with guard conditions very convenient, and more closely aligns with modern functional decomposition.
- Using a Python generator to yield the ready events gives some flexibility. Iterate them directly or collect them in a container.
- Pulling out the ready events into a common context simplifies prototyping work. The ready responders can access a common context where states may persist in memory during execution, and where precise modular boundaries have not yet become apparent.

The code is not complete. The usage examples do not catch socket exceptions; although they do demonstrate socket closure. It's just a sketch.

Windows machines do not require a socket option to reuse an address, as do Unix-based socket libraries. Tested on Windows only.