Python socket receive - incoming packets always have a different size

37

I'm using the SocketServer module for a TCP server. I'm experiencing some issue here with the `recv()` function, because the incoming packets always have a different size, so if I specify `recv(1024)` (I tried with a bigger value, and smaller), it gets stuck after 2 or 3 requests because the packet length will be smaller (I think), and then the server gets stuck until a timeout.

```
class Test(SocketServer.BaseRequestHandler):

def handle(self):

  print "From:", self.client_address

  while True:

    data = self.request.recv(1024)
    if not data: break

    if data[4] == "\x20":
      self.request.sendall("hello")
    if data[4] == "\x21":
      self.request.sendall("bye")
    else:
      print "unknow packet"
  self.request.close()
  print "Disconnected", self.client_address

launch = SocketServer.ThreadingTCPServer(('', int(sys.argv[1])),Test)

launch.allow_reuse_address= True;

launch.serve_forever()
```

If the client sends multiples requests over the same source port, but the server gets stuck, any help would be very appreciated, thank !

asked Nov 10 '09 at 15:34

n00bie

51322 gold badges99 silver badges1010 bronze badges

45

The network is *always* unpredictable. TCP makes a lot of this random behavior go away for you. One wonderful thing TCP does: it guarantees that the bytes will arrive in the same order. But! It does *not* guarantee that they will arrive chopped up in the same way. You simply *cannot* assume that every send() from one end of the connection will result in exactly one recv() on the far end with exactly the same number of bytes.

When you say `socket.recv(x)`, you're saying 'don't return until you've read x bytes from the socket'. This is called "blocking I/O": you will block (wait) until your request has been filled. If every message in your protocol was exactly 1024 bytes, calling `socket.recv(1024)` would work great. But it sounds like that's not true. If your messages are a fixed number of bytes, just pass that number in to `socket.recv()` and you're done.

But what if your messages can be of different lengths? The first thing you need to do: stop calling `socket.recv()` with an explicit number. Changing this:

```
data = self.request.recv(1024)
```

to this:

```
data = self.request.recv()
```

means `recv()` will always return whenever it gets new data.

But now you have a new problem: how do you know when the sender has sent you a complete message? The answer is: you don't. You're going to have to make the length of the message an explicit part of your protocol. Here's the best way: prefix every message with a length, either as a fixed-size integer (converted to network byte order using `socket.ntohs()` or `socket.ntohl()` please!) or as a string followed by some delimiter (like '123:'). This second approach often less efficient, but it's easier in Python.

Once you've added that to your protocol, you need to change your code to handle `recv()` returning arbitrary amounts of data at any time. Here's an example of how to do this. I tried writing it as pseudo-code, or with comments to tell you what to do, but it wasn't very clear. So I've written it explicitly using the length prefix as a string of digits terminated by a colon. Here you go:

```
length = None
buffer = ""
while True:
  data += self.request.recv()
  if not data:
    break
  buffer += data
  while True:
    if length is None:
      if ':' not in buffer:
        break
      # remove the length bytes from the front of buffer
      # leave any remaining bytes in the buffer!
      length_str, ignored, buffer = buffer.partition(':')
      length = int(length_str)

    if len(buffer) < length:
      break
    # split off the full message from the remaining bytes
    # leave any remaining bytes in the buffer!
    message = buffer[:length]
    buffer = buffer[length:]
    length = None
    # PROCESS MESSAGE HERE
```

answered Nov 11 '09 at 16:02

Larry Hastings
2,2261616 silver badges1111 bronze badges

130

The answer by Larry Hastings has some great general advice about sockets, but there are a couple of mistakes as it pertains to how the `recv(bufsize)` method works in the Python socket module.

So, to clarify, since this may be confusing to others looking to this for help:

1. The bufsize param for the `recv(bufsize)` method is not optional. You'll get an error if you call `recv()` (without the param).
2. The bufferlen in `recv(bufsize)` is a *maximum* size. The recv will happily return fewer bytes if there are fewer available.

See the documentation for details.

Now, if you're receiving data from a client and want to know when you've received all of the data, you're probably going to have to add it to your protocol -- as Larry suggests. See this recipe for strategies for determining end of message.

As that recipe points out, for some protocols, the client will simply disconnect when it's done sending data. In those cases, your `while True` loop should work fine. If the client does *not* disconnect, you'll need to figure out some way to signal your content length, delimit your messages, or implement a timeout.

I'd be happy to try to help further if you could post your exact client code and a description of your test protocol.

answered Nov 27 '09 at 5:43

Hans L
4,17444 gold badges1818 silver badges2020 bronze badges

17

You can alternatively use `recv(x_bytes, socket.MSG_WAITALL)`, which seems to work only on Unix, and will return exactly `x_bytes`.

answered Dec 2 '09 at 4:40

henrietta
17122 bronze badges

2

That's the nature of TCP: the protocol fills up packets (lower layer being IP packets) and sends them. You can have some degree of control over the MTU (Maximum Transfer Unit).

In other words: you must devise a protocol that rides on top of TCP where your "payload delineation" is defined. By "payload delineation" I mean the way you extract the unit of message your protocol supports. This can be as simple as "every NULL terminated strings".

answered Nov 10 '09 at 15:38

jldupont
74.1k4646 gold badges183183 silver badges298298 bronze badges

1

Note that **exact reason** why your code is frozen is **not** because you set too high request.recv() buffer size. Here is explained What means buffer size in socket.recv(buffer_size)

This code will work until it'll receive an **empty TCP message** (if you'd print this empty message, it'd show `b''`):

```
while True:
  data = self.request.recv(1024)
  if not data: break
```

And note, that there is **no way** to send empty TCP message. `socket.send(b'')` simply won't work.

Why? Because empty message is sent only when you type `socket.close()`, so your script will loop as long as you won't close your connection. As **Hans L** pointed out here are some good methods to end message.

edited Dec 15 '17 at 11:46
answered Dec 15 '17 at 11:36

Qback
1,80511 gold badge99 silver badges2727 bronze badges

0

I know this is old, but I hope this helps someone.

Using regular python sockets I found that you can send and receive information in packets using sendto and recvfrom

```
# tcp_echo_server.py
import socket

ADDRESS = ''
PORT = 54321

connections = []
host = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
host.setblocking(0)
host.bind((ADDRESS, PORT))
host.listen(10)  # 10 is how many clients it accepts

def close_socket(connection):
    try:
        connection.shutdown(socket.SHUT_RDWR)
    except:
        pass
    try:
```

```
        connection.close()
    except:
        pass

def read():
    for i in reversed(range(len(connections))):
        try:
            data, sender = connections[i][0].recvfrom(1500)
            return data
        except (BlockingIOError, socket.timeout, OSError):
            pass
        except (ConnectionResetError, ConnectionAbortedError):
            close_socket(connections[i][0])
            connections.pop(i)
    return b''  # return empty if no data found

def write(data):
    for i in reversed(range(len(connections))):
        try:
            connections[i][0].sendto(data, connections[i][1])
        except (BlockingIOError, socket.timeout, OSError):
            pass
        except (ConnectionResetError, ConnectionAbortedError):
            close_socket(connections[i][0])
            connections.pop(i)

# Run the main loop
while True:
    try:
        con, addr = host.accept()
        connections.append((con, addr))
    except BlockingIOError:
        pass

    data = read()
    if data != b'':
        print(data)
        write(b'ECHO: ' + data)
        if data == b"exit":
            break

# Close the sockets
for i in reversed(range(len(connections))):
    close_socket(connections[i][0])
    connections.pop(i)
close_socket(host)
```

The client is similar

```
# tcp_client.py
import socket

ADDRESS = "localhost"
PORT = 54321

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect((ADDRESS, PORT))
s.setblocking(0)

def close_socket(connection):
    try:
        connection.shutdown(socket.SHUT_RDWR)
    except:
        pass
    try:
        connection.close()
    except:
        pass

def read():
    """Read data and return the read bytes."""
    try:
        data, sender = s.recvfrom(1500)
        return data
    except (BlockingIOError, socket.timeout, AttributeError, OSError):
        return b''
    except (ConnectionResetError, ConnectionAbortedError, AttributeError):
        close_socket(s)
        return b''

def write(data):
    try:
        s.sendto(data, (ADDRESS, PORT))
    except (ConnectionResetError, ConnectionAbortedError):
        close_socket(s)

while True:
    msg = input("Enter a message: ")
    write(msg.encode('utf-8'))

    data = read()
    if data != b"":
        print("Message Received:", data)

    if msg == "exit":
        break

close_socket(s)
```

answered Apr 26 '17 at 18:30

justengel
4,42522 gold badges1616 silver badges3535 bronze badges