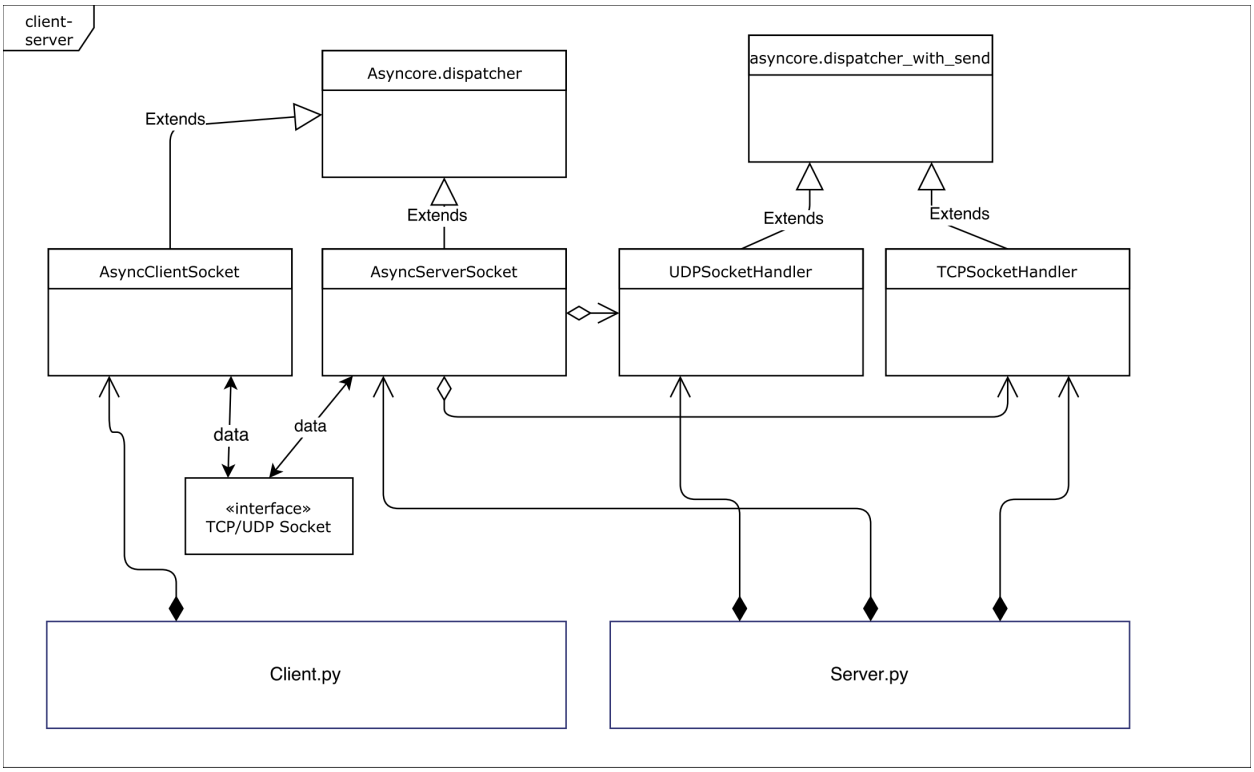
Tony Gao
CS453
HW5
Programming Assignment 1

Abstract

client.py, server.py, my_socket_stuff.py encapsulate the functionality of a client socket, server socket, and wrapper to asynchronous sockets.

The purpose was to create a robust, extensible application that could be configured to use any protocol within reason, and have specific client and server functionalities abstracted from the core of the socket handling smarts.

Design



- Client.py composes an AsyncClientSocket.
  - AsyncClientSocket extends Asyncore.dispatcher, which is a python standard API wrapper for dispatching callback functions bound to asynchronous sockets.
  - AsyncClientSocket handles the reading, writing, and thread synchronization (write with timeout on read response) expected of an asynchronous client socket
- Server.py composes an AsyncServerSocket.
  - AsyncServerSocket extends asyncore.dispatcher.
  - AsyncServerSocket handles the reading of data from the server socket.
  - AsyncServerSocket aggregates either a TCPSocketHandler or UDPSocketHandler instance of super class asyncore.dispatcher_with_send. This is so that the behavior of the server can be swapped for any derived class of asyncore.dispatcher_with_send, so the server API can be swapped at will and more protocols can be added.

    - AsyncServerSocket communicates with AsyncClientSocket with (currently) a TCP or UDP socket. There is no limitation on what socket protocol is used, and conceivably any socket protocol can be used with the necessary extensions.
- Because asyncore.dispatcher does not support socket.sendto() and socket.recvfrom() functionality, this functionality had to be hacked into the python api via class function binding. This code was copied from the python 3.5.2 socket.sendto() and socket.recvfrom() from the python api source code.

TCP Reliable Send Behavior:

Preconditions: Server is running TCP mode.

1. Client connects to server. TCP handshake takes place. If connection is successful, client obtains a socket to the underlying connection.
2. Client sends '+,1,1'.
3. '+,1,1' is written to the buffer, which AsyncClientSocket picks up and writes into the socket.
4. The information on the socket goes through the transport layer, the network layer, the link layer, and written as a bitstream on the physical layer.
5. Client waits for response given some timeout
6. The server physical layer reads the written stream, forwarding it up the network stack to the transport layer.
7. TCP decodes the segment and extracts the payload, sending it to the socket.
8. TCPSocketHandler detects bytes on the socket, reads the bytes, and processes the request.
    1. If a request is valid, code 200 and the result of the operation is written back to the client
    2. If a request is invalid, code 300 and '-1' is written back to the client
9. If client does not receive response within timeout, show error to user.

UDP Send Behavior (success modifier 100%):

Preconditions: Server is running UDP mode with success = 1

1. Client sends '+,1,1'.
2. '+,1,1' is written to the buffer, which AsyncClientSocket picks up and writes into the socket.
3. The information on the socket goes through the transport layer, the network layer, the link layer, and written as a bitstream on the physical layer.
4. Client waits for response given some timeout.
5. The server physical layer reads the written stream, forwarding it up the network stack to the transport layer.
6. UDP decodes the datagram and extracts the payload, sending it to the socket.
7. UDPSocketHandler detects bytes on the socket, reads the bytes (storing the address of the client, since UDP is connectionless), and processes the request.
    1. If a request is valid, code 200 and the result of the operation is written back to the client
    2. If a request is invalid, code 300 and '-1' is written back to the client
8. If client does not receive response within timeout, show error to user.

UDP Send Behavior (success modifier <100%):

Preconditions: Server is running UDP mode with success < 1

1. Client sends '+,1,1'.

2. '+,1,1' is written to the buffer, which AsyncClientSocket picks up and writes into the socket.
3. The information on the socket goes through the transport layer, the network layer, the link layer, and written as a bitstream on the physical layer.
4. Client waits for response given some timeout.
5. The server physical layer reads the written stream, forwarding it up the network stack to the transport layer.
6. UDP decodes the datagram and extracts the payload, sending it to the socket.
7. UDPSocketHandler detects bytes on the socket, reads the bytes (storing the address of the client, since UDP is connectionless), and processes the request.
    1. UDPSocketHandler rolls a dice that causes some percent of packets = success modifier to be dropped (no response is returned)
        1. If a request is valid, code 200 and the result of the operation is written back to the client
        2. If a request is invalid, code 300 and '-1' is written back to the client
8. If client does not receive response within timeout, show error to user.

UDP Send Behavior (success modifier <100%, client exponential backoff):

Preconditions: Server is running UDP mode with success = 1. Client is launched with —expb modifier.

1. Client sends '+,1,1'.
2. '+,1,1' is written to the buffer, which AsyncClientSocket picks up and writes into the socket.
3. The information on the socket goes through the transport layer, the network layer, the link layer, and written as a bitstream on the physical layer.
4. IF timeout <= 2:
    1. Client waits for response given some timeout = initial timeout value.
5. Else:
    1. Time out, show error to user.
6. The server physical layer reads the written stream, forwarding it up the network stack to the transport layer.
7. UDP decodes the datagram and extracts the payload, sending it to the socket.
8. UDPSocketHandler detects bytes on the socket, reads the bytes (storing the address of the client, since UDP is connectionless), and processes the request.
    1. UDPSocketHandler rolls a dice that causes some percent of packets = success modifier to be dropped (no response is returned)
        1. If a request is valid, code 200 and the result of the operation is written back to the client
        2. If a request is invalid, code 300 and '-1' is written back to the client
9. If client does not receive response, timeout = timeout * 2. Go to 4

Considerations for improvement

- Multiple TCP connections could be supported in parallel in AsyncClientSocket with few modifications.
    - A hashmap of unique identifier keys to TCP socket instances can be created. This will have to be managed in real time to detect shut down, timed out, and so on sockets.
    - Each sender will send by keying into the hashmap with this unique identifier key (call it "session id")

    - Each session id must have a unique read and write buffer. As of the current implementation, the client uses a global read and write buffer, so multiple connections on the same AsyncClientSocket are not possible.
- Reconnection functionality could be added to AsyncClientSocket
    - With some exponential backoff function, retry connection to a host if a socket fails to be opened
    - If a socket cannot be opened after the time limit is exceeded, give up and allow user to decide how to proceed
- Different protocols can be implemented by adding more supported modes to client.py, and implementing the connection logic for that mode in AsyncClientSocket.__init__()
- The server could be extended to utilize a second socket listening on the port for some UDP discovery protocol, such as 1900 for SSDP. The client could be extended to utilize this discovery protocol to discover the IP:Port of all servers that implement my server library. The results of this discovery could be presented to the user, for the user to choose what server to connect to.
- Unit tests can be added to test the my_socket_stuff module for proper expected behavior on both positive test cases, and negative test cases.
- Server.py and Client.py could be merged into one ClientServer.py, which has an additional argparse required parameter to toggle between server and client mode. The server mode would require a subparser with the various server arguments, and same for the client.
- A parameter for an upper limit for the exponential backup can be added to the client, which will change the maximum amount of time that the timeout value will approach following the exponential backoff algorithm.

Testing done

- Manual testing of correct behavior for requests in UDP/TCP that follow the API (‘opcode,number,number’) was done.
- Manual testing of correct error handling for requests in UDP/TCP that do not adhere to the API was done. The edge case of an empty write buffer (due to user pressing enter without entering any text) was handled because of the results of this test. Arithmetic errors and any other expression that cannot be evaluated such as division by 0 are captured in the server socket handlers in server.py.
- Wireshark was run on the transmission for UDP. The capture file has been included named ‘udp_test.pcapng’. This was done to ensure that no matter how small the timeout (within reason), the client socket attempted to send to the UDP end point. This was only done for UDP because of the unreliable nature of UDP. A bug in the calls on both the server and client to asyncore.loop() was found as a result. This bug was due to the timeout parameter, which should be set to 0 so that incoming socket requests do not block the asynchronous callback loop. This was seen with short timeout values, where the server seemed to not be receiving packets and the client was not sending packets. This was because the asyncore.loop() was calling the select() syscall with a certain timeout, and send requests that came faster than that timeout were lost.
- Manual testing of correct client and server behavior for exponential backoff for UDP was tested. There was no requirement for exponential backoff for TCP, so this was not tested.
- Through manual performance testing of the behavior of asyncore.loop(), it was discovered that asyncore.loop(timeout = 0) causes 100% CPU loading due to the infinite 0 timeout callback loop. This is internal to asyncore. As a fix, the client callback polling interval was set to the timeout / 2, and the server callback polling interval was set to timeout / 4. This ensures that any request pending on the client socket will always be written and have enough time left to timeout regardless of timeout backoff configuration, and any data

pending on the server socket will always be read without incurring the performance cost of an infinite nonblocking callback loop.