CSE 157 - Lab 3

Dallas Meyer

Nov 17, 2023

1 Real Python multi-connection examples

From the Real Python website, are multiconn-client.py and multiconn-server.py files that contain multiple functions that achieve the functionality of client-server communication.

In the multiconn_ed_server.py pieces together the multi-client and multi-server python codes into a class. This Server class template allows accepting connections from multiple clients, whilst also has the feature of being "event-driven" where the server will take action once it receives a message/event from a client.

More detail will go into the different functions/methods.

1.1 accept_wrapper()

Listing 1: monlticonn_ed_server.py: accept_wrapper

```
def accept_wrapper(self, sock):
    """
    Accepts and registers new connections.
    """
    conn, addr = sock.accept()
    slogger.debug(f"Accepted connection from {addr}.")
    # Disable blocking.
    conn.setblocking(False)
    # Create data object to monitor for read and write
        availability.
    data = types.SimpleNamespace(addr=addr, inb=b"",
        outb=b"")
```

```
    events = selectors.EVENT_READ | selectors.EVENT_WRITE
    # Register connection with selector.
    self.sel.register(conn, events, data=data)
```

In the accept_wrapper() server-function, passes the socket into the parameters, and it runs the accept() method on it. This allows the server to accept/complete the connection to the client.

The difference between this and the listen() method – that is called beforehand –, is that listen() sets the server to listen, where the accept() accepts.

There are slogger debug messages, which work like print statements.

The events objects are set for when the server is ready to be read and written to.

A unfamiliar method is select(), which allows high-level and efficient I/O multiplexing. From the documentation, it "waits until some registered file objects become ready, or timeout expires" (docs.python.org). To monitor the file object, or socket in this case, the sel.register() method is used.

1.2 service_connection()

Listing 2: monlticonn_ed_server.py: accept_wrapper

```
def service_connection(key, mask):
    sock = key.fileobj
    data = key.data
    if mask & selectors.EVENT_READ:
        recv_data = sock.recv(1024)  # Should be ready to
            read
        if recv_data:
            data.outb += recv_data
        else:
            print(f"Closing connection to {data.addr}")
            sel.unregister(sock)
            sock.close()
    if mask & selectors.EVENT_WRITE:
        if data.outb:
            print(f"Echoing {data.outb!r} to {data.addr}")
            sent = sock.send(data.outb)  # Should be ready
                to write
            data.outb = data.outb[sent:]
```

As seen in service_connection() server-function, the key returned from select() is passed into it. It also has the socket object, the mask is the events.

As expected, the first if statement corresponds for when the socket is ready to be read from the client, and thus reads the data.

It is then closed – this functionality I had to remove in order to keep the server running on the socket.

Similarly to the reading, the second if state is for writing to the client.

For the client implementation, the client can close its side first after receiving all the necessary info first.

1.3 start_connections()

Listing 3: monlticonn_ed_server.py: accept_wrapper

```
def start_connections(host, port, num_conns):
    server_addr = (host, port)
    for i in range(0, num_conns):
        connid = i + 1
        print(f"Starting connection {connid} to
            {server_addr}")
        sock = socket.socket(socket.AF_INET,
            socket.SOCK_STREAM)
        sock.setblocking(False)
        sock.connect_ex(server_addr)
        events = selectors.EVENT_READ |
            selectors.EVENT_WRITE
        data = types.SimpleNamespace(
            connid=connid,
            msg_total=sum(len(m) for m in messages),
            recv_total=0,
            messages=messages.copy(),
            outb=b"",
        )
        sel.register(sock, events, data=data)
```

In the start_connections() client-function, passed is the number of connections to start.

The socket is created here similarly to the server, however without binding it since it is not a listening socket.

The connect_ex() method doesn't return an exception, ensuring compatibility with the setblocking() method.

With setblocking() method set to False, that means the method won't stop the program counter until its completion. Instead it can allow other tasks to be performs while still waiting. Aka not busy-waiting.

The SimpleNamespace object is used to store the socket's data.

In figure **??**, in the top-left we have the positive and ground in their respective rails at the top of the breadboard. The I2C wires are also seen, white as the data and brown as the clock.

2 Single vs Multi-connection Server

The main difference between the multi-connection server and the single-connection server is in how it handles client connections.

In the single-connection server, only one client is connected to the server at a time. As such its synchronous and cannot handle any other connections until the current client-server connection has ended.

As for the multi-connection server, it can indeed handle multiple connections at a time – concurrently. The use of the non-blocking socket option, and selectors to keep track of events, is what provides the functionality

3 Polling the Raspberry Pis

In this part of the lab, we have to use three Raspberry Pi's.

In a oversimplified summary, one of the Pi's is the primary Pi or data sink that requests data. The data is requested from the two secondary Pi's, of whom collect the data from the sensors and send it back to the primary Pi. The primary Pi then plots the two data from the other Pi's, along with its own data.

3.1 Flow Chart Design choices

In our implementation, we decided that the Primary Pi would alternate between polling each Pi. If one Pi didn't send any data or had some errors reading some sensors, then we would set that value to 0 in the plot.

This is represented in Fig.1, where 1) requests data to the Secondary Pi, 2) Secondary Pi sends back, 3) requests, 4) send.

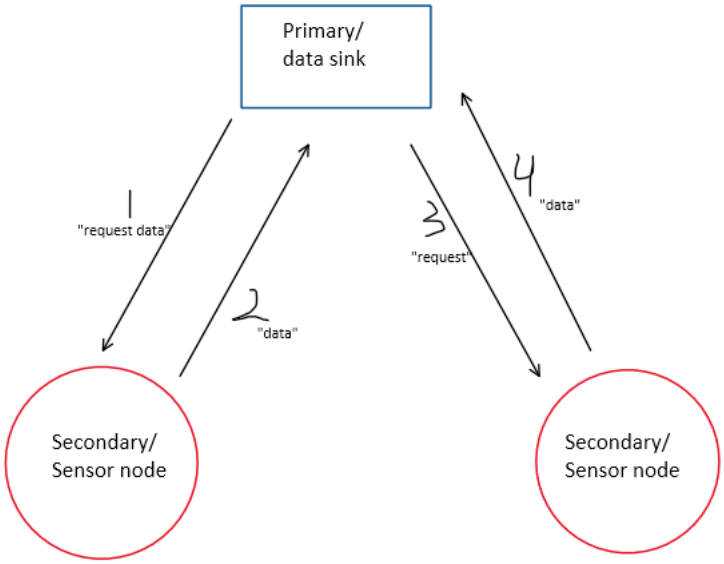

Figure 1: Flow chart of part 2

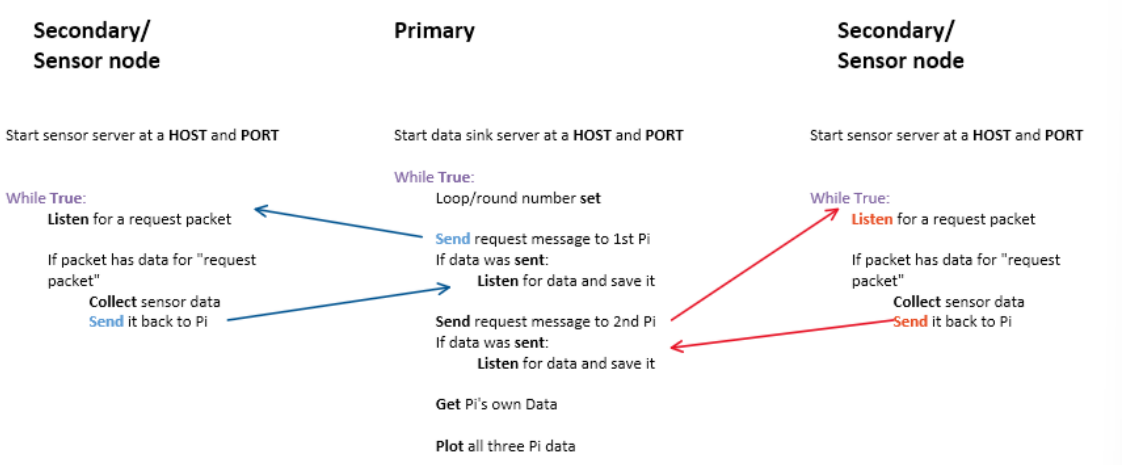

Figure 2: Example flow and pseudo code main loop for part 2

Our implementation heavily focused on modularity. As such all functions were created into a separate class, expanding on that multconn-server.py template. This made it easier to implement the the pseudo code above.

My lab partner Keith was mostly in charge of structuring and modifying the server class files, so that the code can fulfil modularity

On my end, I developed an i2c class that would be used to gather data from the sensors (each having their own simple "getter" function), along with doing some error handling in the case that the sensor cannot be polled. This class also formats the data similarly to a dictionary key:pair, so that it can be parsed more easily.

As for the plotting, my partner Airi was in charge of the plotting, sorting through the data and ensuring the lists didn't have zeros in them.

3.2 Difficulties and debugging

Initially, when setting up the Pi's we had to put it into ad-hoc mode. This was troublesome, but after figuring out that all we had to do was set it into ad-hoc – with different IP addresses – we were able to finally communicate along the network.

Another issue was determining how to send data in one message. We had to come up with different ways, thinking of for loops and such. Eventually we realized we can just take advantage of delimiters.

To make the data work in one message, we simply combined all the data into one string. This string was then sent over in one message. However, a parsing function (Sink_server.packet_parser()) was then required to sort out the data. The good news is that since the data was put into a format with ":" and "," to be used as delimiters, this made it easy for my lab partners to set up the data into a list and plot it.

Some difficulties when testing was the ability to still communicate again (eg. round 2). Determining that the socket or select was being closed, I did a quick fix of removing the close() function. Simply removing this allowed multiple further rounds of communication.

Another difficulty was setting up error handling. It was found that having a keyboard interrupt within one of the class's functions, caused the code to never quit in the terminal. This is because having two nested keyboard exceptions only allows it to escape the inner while True loop.

Continuing on error handling, we had to determine how to send a message for when the sensor data couldn't be found. We used the string "ERR" for

this.

Another error case, was if the Rasperry Pi wasn't existent. This was fixed through a simple try and except if the socket couldn't be reached.

4 Token-ring with the Raspberry Pis

As seen in Fig.3 each Raspberry Pi goes through a one-way communication, where all the Pi's are mostly waiting. Pi #1 (aka the one chosen to have the token) collect and send both its sensor data and the sequence number 2 to Pi #2. Pi #2 will take this data, and since they received a seq#2, combine it into a message with its own collected sensor data. Pi #2 then sends the data and sequence number 3 to Pi #3 notices this sequence number and begins to plot. After that, Pi #3 sends a seq number 1 back to to the first Pi.

The token in this case is sent around within the packet message header. Hence whoever has the token can send a message and the token to the next Raspberry Pi

A design choice by Keith was to have the Pi's all run the same file, where they only have to change their host IP.

It also takes user input on who gets to start with the token, rather then statically setting some specific raspberry pi to start.

The pseudocode or main loop implementation is seen below.

Listing 4: monlticonn_ed_server.py: accept_wrapper

```
def main():
    ask whether user starts with token

    Initialize or create server instance

    If the current Pi is the first token server then start
       with sending a message:
        Create the packet to send the token

        Send the token_packet to the next host using the

        packet_ring_handler function determines what to do
           next

    while true:
```

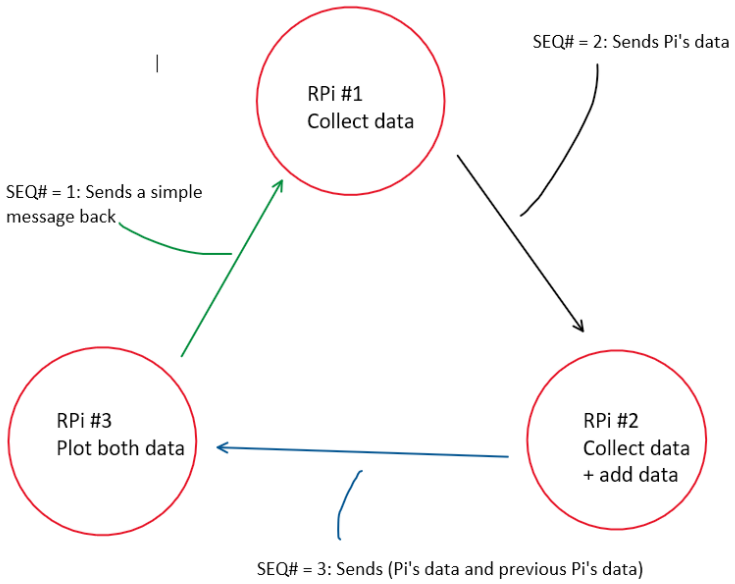

Figure 3: Flow chart of how each Pi's functions.

```
packet = wait and listen for the encoded packet

determine the sequence number from packet

packet_ring_handler function determines what to do
    next
```

4.1 Difficulties and debugging

As with the previous part, this part of the lab also had many communication errors.

One of which was that when sending a message, the data on the receiving end would appear as None or empty. This appeared to be due to decoding the packet message twice. As such we had to use trial and error to determine when and not to decode and encode messages into bytes. Eventually we just implemented a simple if-else statement to determine when to do some type conversions.

The good news is that we've already handled lots of the error handling and cases in the previous part, and as such re-used that implementation here although obviously with a redesign.

The modularlity and class structure for the server communication, again was mostly designed by Keith.

This debugging portion of different connection errors, kept occuring on the recieving and sending Pis. Me and Keith did lots of coordination (viewing each others outputs, running A, running B, etc) and trial and error to get the communication of the first two Pi's (aka data senders) to finally work.

A difficulty when communicating, was trying to get the sequence number to work and send through the message. It turns out the sequence number wasn't being incremented, a silly mistake.

The plotting was still a bit tricky. Again Airi worked on this portion and debugged the errors that occurred when sending packets to the third (aka plotting Pi).

I do enjoy working with the sensors, and as such ensured actual sensor readings (not randomly generated) were still being sent through. There was an issue with one sensor that was fried, which was difficult to point out until after rewiring, trying different breadboards, etc.

5 Implementing the timeout mechanism in the Polling and Token-Ring

5.1 Polling timeouts

When polling, a timeout mechanism would be useful in two cases.

The first case would be when the primary Pi tries sending a message to a secondary Pi. In this case, the send_msg() function must create a socket connection with the secondary Pi and try and send a message. However if this functionality cannot be achieved with that socket (eg. message couldn't be sent), it should timeout and try the other Pi.

The second case is when the Primary Pi accepts a Secondary Pi's incoming connection, but that Secondary Pi doesn't send back any data. If the Secondary Pi doesn't send any data, or if the connection occurs for too long, a time-out can ensure that no resources are wasted and could promptly disconnect. In the code, it would be within the run_listener() function.

5.2 Token-Ring timeouts

As in the Polling case, having a timeout is useful within the run_listener() function. In this case the next Pi in the token-ring would be the "secondary" Pi.

5.3 Corner cases

Some possible corner cases in the polling and token approach, would be for when one Raspberry Pi is not even up or running.

In this case, polling could keep acquiring data from just one Pi. This was handled by filling out any missing data through using zeros, similarly to converting any "ERR" readings to zeros.

However for the token-ring, so far there's not much to do besides having the Pi wait for the offline Pi to join the server, and then send the token and data.

6 Comparing polling and token-ring-based approaches

6.1 Pros and Cons

In regards to latency of calculating the averages, they are most likely similar. This is because in polling, we haven't implemented getting the data in parallel, where the time would equal around just one pi request. As such both approaches are basically sequential since no actual threading was implemented.

The polling approach is good in that it is very expandable in regards how many devices can be polled, and how if one Pi is down, there are still all the other Pi's available to get data from.

Whereas the token-ring network would break if just one Pi malfunctions, unless an corner case was created to choose the next RPi that isn't broken, similar to a linked list.

The token-ring, would still work on devices that have adequate storage, but are limited to connecting to only two raspberry Pi's

6.2 Using the polling approach

The polling approach is useful as the primary acts as a data sink. As such the secondaries don't need to worry about storage or memory management, just collecting data.

To get more storage, the primary device could be replaced or upgraded. This is also the downside with the token-ring approach, where the first Pi will only have to store one set of data, but those near the end of the cycle have to store tons more data – unless that data is aggregated.

6.3 Using the token-ring approach

The token-ring approach is useful for its socket-programming simplicity in large systems, as each machine just needs to two socket connections total. Whereas polling needs to handle lots of incoming connections that may be competing to connect to the data sink, which could cause packet losses.

7 Conclusion

In this lab, lots of topics and concepts were touched on. The ad-hoc system implemented was infrastructure-less, and didn't require WiFi, serving as a peer-to-peer network.The servers implemented were able to have different cases and handling (eg. if a connection fails). Solving the errors one after another, was definitely trial and error.

As for the packets, the data, and parsing functionality, lots of different design ideas had to be taken into consideration.

As a group project, it was also very interesting to combine different perspectives and thoughts of knowledge, to end up creating a server system together.