Final Report

Project: Auction System

A Report
Presented to
The Department of Electrical & Computer Engineering
Concordia University

In Partial Fulfillment
of the Requirements
of COEN 445

By

Adam Ibrahim-Disuky	ID: 27003722
Ryan Nichols	ID: 29739983
Anas Shakra	ID: 40004648

Presented To
Dr. F. Khendek

Concordia University
December 2018

Introduction

The purpose of this project is to further the students knowledge of protocol design, the client server model, use of sockets both UDP and TCP as well as use of multithreading. This project is the implementation of an auction system over a LAN. Functionalities implemented over UDP are client registration as well as de-registration both requests and responses from the server. Also the two way communication from client to server for making items available for bid. Functionalities implemented over TCP include the two way communication of bidding on items and all the associated messages such as messages indication who won the bid, a new highest bid, that the bid has ended etc.

Overall View of Protocol

The interaction between a client and a server over the communication protocol in question is facilitated by two transport protocols: UDP and TCP. The former is used to receive and verify registration requests while the latter is utilized to facilitate the bidding process on one item by multiple clients. Initially, the interaction between the client and the server occurs solely through UDP. The client inputs the required registration information (i.e. Name, IP, Port) and sends this data to the server. Upon receipt of this request, the server will verify that the request is valid by ensuring that no user with the same information is already registered in its log, in addition to ensuring some formatting standards are met (e.g. valid IP). If the request is successful, the server responds to the user with a positive acknowledgment. If the request is not successful, the server responds to the user with a negative acknowledgment.

At this point, the user is allowed to bid on currently offered items, or to put an item for bid themselves. Assuming the latter, the user would provide their name, the description of the item they'd like to offer, and the minimum bid acceptable for this item. Upon receipt of this offer request, the server will first verify that the user making the offer is, in fact, a valid user. If not, an unsuccessful offer acknowledgment is sent. However, if the offer is successful, the server will begin the process of establishing a dedicated TCP connection for each item on offer that will allow up to six clients to bid on this item. The UDP connection passes the relevant server information to this TCP connection, including the port of the item that it will be associated to.

Now that a TCP connection has been established for the item, the item is available to be bid on by all valid clients. To due so, the client submits his/her registration information, the number of the item to be bid upon, in addition to the bid amount. Each client can potentially interact with a maximum of six items, each with a dedicated TCP socket. When the bid request is received, the server will first acknowledge that the user is valid. Second, it will use the item number given in the request to verify if the item is valid. If that’s the case, the server will return the port number of the item to the client which will then be used to connect one of the client’s dedicated sockets to the port associated with the relevant item. Finally, the bid amount will be compared to two values; The minimum bid possible on the item and the current maximum bid. If it is larger than both, the server will respond with a positive acknowledgment, represented by the “HIGHEST” message. If not, the server will ignore the bid request. If the same user, or another user, places a larger bid, the “HIGHEST” message, is resent to all users currently bidding on the item. This is done through TCP using the connection established between each client’s socket and the TCP socket associated with the item. The bidding process for an item lasts for a total of five minutes, after which a number of messages are sent; “BID_OVER” is sent to announce the end of the bidding process, “BID_SOLDTO” is sent if there is at least one valid bid on the item, and “BID_NOTSOLD” is sent if no valid bid was made on an item. All these messages are sent to each client through the TCP connection between each client and the item on bid. The “WIN” message is sent to the winning client through the server’s UDP connection to the client.

The above is a brief overview on how the protocol establishes an auction system capable of handling multiple items, each with its own set of clients.

Assumptions

Firstly our main assumption is that regardless of which computer a client registers on they can bid from any computer that establishes a UDP connection to the server. This meant that we could run the client software on a seperate computer from the one running the server, or even run the client on the same computer. Our GUI asks for a name and IP address when registering. We had planned on refactoring the codebase to have a password instead of the IP but in the end we didn’t have time, since some of our tests that were successful on our own machines were causing problems on the computers in the lab. Essentially the IP address here is functioning as a password though. Furthermore, the IP needed to be valid. The only difference being the name and the fact that if the user inputs an invalid IP they won't be allowed to register. One of our assumptions was that the client would have the servers IP address when it started. This made it a lot easier for testing and simplified the passing of IPs.

After registration all interactions with the server require this IP address along with the name of the user. If either are incorrect the action won't be allowed by the server.

We also assumed that a client can bid on their own items. There are ways to cheat and bid up the price of your own items that way but there is also risk that instead of getting a buyer for the item the item is sold to the seller so we didn't see a need to step in and limit this behaviour.

It was difficult to decide what to do with items once the server would crash. What we decided to assume was that once a server would go down, all items would have their open states set to 0 and to be given back to the person who put it up for bid. This decision was based on the fact that it would be unfair to the seller if an item would be sold before the 5 minutes was over as it could sell for a lot less than it was worth. When the server would crash, users would get a message telling them all items were no longer up for bid and to put them back up if they wish to sell them.

Overall Design

As a language we decided on Python3 since it's fast for prototyping, has a concise syntax and is very popular so it's easy to find examples and discussions online on the topics of multithreading, using sockets and the client server model. Python is also a language that is very high level and takes care of a lot of underlying mechanisms. It simplifies a lot of implementation for threading and sockets specifically compared to a language like C++. This really helped in our decision to choose Python over using C, C++. Java is similar to Python as it takes care of a lot of nuisances for threads and sockets, but when it came down to it, the three of us had more experience with Python than Java and wanted to further our Python knowledge.

The more important libraries we required were the *threading* and *socket* libraries for obvious reasons but also we required the *ast* library for restoring the state of the system from our backup file as well as the *time* library for getting starting times for bids and checking to see if the bidding time is up for open items. We also used some basic libraries such as *JSON* to encode and decode JSON objects and *random* to generate random numbers**.**

To store the state of the system we have a python dictionary with the name *state* in main.py. This is a shared resource so must be protected by a mutex lock which we called *state_lock.* The structure of the state variable can be seen in figure 1. We have a list of dictionaries that hold information on clients who are registered, items up for bid, connections made over UDP and a variable that indicates if a change has been made recently to the state. If so this would imply that the latest information on items for offer should be broadcast to all clients over UDP. This is because we have a GUI and we wanted our users to have this information before they register in case they see some item that makes them want to register. Both *state* and *state_lock* are passed to the constructor for our class *UDPServer* which then passes them on to each *TCPServer* instance created.

For handling the UDP connections we have a class *UDPServer* in *udp_server.py* in the *SERVER* directory. This class overrides the Thread class in a similar manner to what is discussed in this stackoverflow question[1] and this tutorial[2]. That is to say we pass the host ip address and port number as parameters to the constructor and then create the socket and bind to it in the __*init*__ function which can be seen in figure 2 of the appendix.

The *run* function, seen in figure 3, is of course overridden and this is where we start other threads for and checking to see if any threads have modified the shared data structure *self.state*. If so we update all clients with UDP connections as mentioned previously. The heart of the *UDPServer* thread is the while loop. It listens for messages on the socket that it has been bound to. When it receives a message it checks to see if this is a new ip and port that it hasn't seen yet. If so it adds the address for sending return messages to *self.connected_clients.* Regardless of whether or not this is a new unseen address the message is decoded using *ast.literal_eval()* since the message being received is decoded into string format but then needs to be decoded further since this string should be of the form of a python dictionary.

The decoded python dictionary is then passed to *self.handle_response()* which should determine which type of message its dealing with as well which response message should be returned. As you can see in figure 4 it does this by checking a parameter in the incoming dictionary 'type' and calling the appropriate helper function to prepare a return message which then gets passed back to the run function where the message is

[1] https://stackoverflow.com/questions/23828264/how-to-make-a-simple-multithreaded-socket-server-in-python-that-remembers-client

[2] https://www.techbeamers.com/python-tutorial-write-multithreaded-python-server/

turned into bytes format to be sent over UDP to the return address associated with the message just received.

In the function *self.ack_offer(),* seen in figure 6, in the *UDPServer* class we can see how TCP connections are established for items up for bid. Should the latest incoming UDP message be of type *OFFER* and should the offer be accepted then a new instance of TCPServer class is spawned and *self.offer_success()* is called to update the *state* variable as well as the backup txt file to reflect that a new item is up for bid. *self.ack_offer()* also sends out the NEW_ITEM message to all clients who've made UDP connections to the *UDPServer* which is a superset containing all registered clients as well. The TCPServer class also overloads the Thread module so after adding the new item server to *self.item_servers* the thread is started and the item can now be bid on.

The other UDP messages received are handled in a similar fashion to what we have seen for a successful offer. The 'type' of the incoming message is checked in *self.handle_response(),* the corresponding helper function for that type is called which will check the incoming request, compare some data from that request to the shared state variable, determine if the request is valid and produces a response which gets passed up the chain of return statements to the run function where the request was received. For example in figure 7 we can see part of one such helper function *self.ack_register()* the name from the incoming request is obtained and checked against the shared state variable to make sure that no one with that name is already registered. If someone is then a registration denied message is returned.

On the client side for UDP connections we have multiple ways of getting information from the user. Our legacy code was to get the input from the terminal and package a message to be sent to the server and our GUI did essentially the same thing so we have messages that needed to be sent over UDP connection but coming from different places during development. To handle this we used the producer consumer model where it doesn't matter where the messages come from, they only need to get into the correct queue and we have a thread that only checks if that queue is non empty. If so it pops the next message from the queue and sends it. The function provided to this thread is pictured in figure 9.

Once an item was put up for bid, a TCP socket would be opened for that item. This meant that every item had its own socket, and every client had its own socket. Furthermore, there were threads built to handle waiting to receive messages and threads used for the items themselves. If we needed to communicate with a client, we could send a TCP message through their socket, and if a client wanted to bid on an item, they could give us the item number with a bid and this would start a connection to the item to place the bid. We had two functions for TCP messages, *tcp_incoming()* and *tcp_outgoing()*. These functions were built for incoming and outgoing messages respectively. Both these functions run in a while loop so they never end, and are running on a thread. When messages come in, they get taken from the socket and parsed for handling. We used an array of connections and would go through that list to check if any messages had come through. We set the blocking on these TCP connections to 0 so that we wouldn't get held up waiting for a socket to get a message when another one had one waiting. By throwing an exception *BlockingIOError,* if a socket had no message, it would just continue to the next one. Depending on the type of TCP message received, we could take the message and pass it to the GUI to render on the screen. On the other hand, if a message was outgoing, we could simply use another lock and send the message using *sendTCPMessage()*.

sendTCPMessage() works by taking the message (a dictionary) and passing it through the socket as bytes. It loops through the connections and finds the proper port number for the item you wish to bid on and passes it through that socket.

After an item went up for bid, a TCP connection would need to be made for said item. Our function *establishTcpConnection(Host, portNumber)* would set this up for us. Firstly, it would check if a connection was already made for an item by going through our connection list and checking previous port numbers. If a socket wasn't created, we would set up the socket by using *tcp_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)* and *tcp_socket.connection((HOST, portNumber)).* Lastly, we would add this new connection to our connection list so that we could send and receive messages from it.

On the server side, once an item went up for bid, a few things had to happen. Firstly, a TCP connection had to be made on the server side for the item. This was done immediately after an item would go up for bid in our *udp_server* code. After the item was handled, a TCP connection was set up for it by using the following code *server_for_item = TCPServer(self.host, item['port #'], self.state, self.state_lock, self.txt_file, response['item #'])*. By then calling the start function on this, we could create a thread for the item. Secondly, a timer had to be started on each item to ensure that it wasn't up for bid for longer than 5 minutes. We did this by having a thread that continuously checked if the 5 minutes had elapsed on an item and to then close it if it did. Once this was handled, the server would need to wait for bids to come in. When a bid was received, the function *ack_bid(self, msg_recieved)* would handle it. This function worked by getting the current highest bid on an item and first checking if the new bid was higher. If it was, the state file would updated with the new bid, and then a highest message would be sent out. The response would be sent with the function *respond_bid(self, msg_recieved, amount, success)*. This function would take the new information from the state and put it into a dictionary. Once the message was built, the highest_message would be sent out to the clients.

Once a bid was over, a few things had to happen to have the proper functionality. Our function *handle_end_of_bid(self)* would handle all things that needed to be done after the 5 minutes elapsed. Firstly, the item open status would be set to 0 in the state. Secondly, if there was a highest bid, a sold_to message would be sent to the highest bidder. If there were no bids, a not_sold message would be sent off. The socket would then be closed and the thread would be terminated as to not keep it opened and use resource for nothing.

Since we didn't have any experience coding GUI's most of the code for the client communicating with the server is kept separate from the GUI script which is meant to be run as its own process. We kept the GUI as minimal as possible. It needs to receive and display messages from the client process which is in communication with the server. Other than that is just needs to package information given by the user, pass them to the client process which forwards the messages to the server. To accomplish this we came up with a message passing system that uses a text file for each direction of communication. On the client side a function is run in a separate thread that constantly checks which the contents of *toClient.txt.* Messages in this file are numbered so this function checks the latest message number and if it's new the packages the message to be put in one of the queues, TCP or UDP, to be sent to the server. This numbering system can be seen in figure 10 of the *toClient.txt* file and communication from the server to the client process and ultimately on to the GUI works in the same way but messages are passed from the client process to the GUI in a text file named *toGUI.txt*.

Testing

If we test registration through the GUI the message displayed in figure 11 is the message received by the server. The message displayed in figure 12 is what is received back at the GUI. If we try to register again with the same info we get the message displayed by the server seen in figure 13 where the request #2 has been sent twice since the registration was unsuccessful and the resulting return message is displayed in the GUI in figure 14. Pressing the deregister button works as expected and the results are seen in figure 15 and 16. We should not be able to offer and item for bid because we have just unregistered so trying to do so correctly produces the responses seen in figure 17 and 18. If I register again and try to make the same offer it is successful and the messages produced by the server and received at the GUI are shown in figure 19 and 20. You can also see the item up for bid in figure 20. After performing these actions you can see that our shared variable *state* has been updated to show that we now have an UDP connection, a registered client and an item up for bid.

If I offer another item up for bid and try to deregister I should not be able to because I have items up for bid and you can see the correct responses generated in figures 21 and 22. If I make two more offers the last one shouldn't go through due to having more than 3 items up for bid and the correct response is returned to the GUI in figure 23. The open status of item 2 had changed to zero but that's just because I didn't snap the screenshot quick enough.

Since the demo of this project had to be done on linux machines at school, it was imperative that our code be tested on these computers to ensure that it would work smoothly. Most of our initial testing was done on windows machines and a bit on linux but on our own personal computers. Before the demo, we ran the client and server on the school machines to ensure when it came time to demo that it would work properly. It was a good thing we did because we did find a couple issues that needed to be fixed to make it work properly.

Conclusion

When the server was connected to three clients, the system worked well; that is, the UDP communication, in addition to its TCP variant, processed requests efficiently and outputted the expected behaviour. Moreover, error handling techniques, such as preventing identical users and making bids smaller than the minimum or current maximum bid, worked correctly.

However, complications arose when the number of clients exceeded three. While no definitive answer has been determined, it could be that the use of the same account on multiple computers in the lab could have resulted in some errors. From past experiences in other group projects, the use of one student account on multiple computers has been known to produce unexpected behavior.

In future iterations, we would finish refactoring the code to change the IP address entered by the user when registering, bidding etc. to a password for authentication. We would also like to improve on the appearance of the gui. It functioned well enough but doesn't look appealing yet. Also it would be preferable if the gui and client were one process. Passing messages back and forth between text files works but adds latency and requires additional threads for reading the files. It would be better if the client side logic of making socket connections and packaging messages to be sent were all implemented in the same process that handles the gui.

Also, an important aspect that would needed to be fixed in the future would be to take better care of how threads are created and joined. In some of our testing we found that at some points in running the code, we would have around 35 to 40 threads running simultaneously. In the future, we would like to make sure that when threads are created, we manage them properly. This means that we only create as many threads as we need, and when threads are done running they are properly joined back to the main process. We wanted to make most process multithreaded so that the program would run as quickly as possible. Moreover, we were very happy with how our threads and locks were functioning since we had very good atomicity for important variables. For example if 5 people bidded at the same time, the functions would execute properly one at a time and ensured that the variables were accessed atomicly. We were very happy with this result, but we would really like to take better care of all the threads that are created.

Contributions

Ryan was responsible for coming up with the architecture of having one *UDPServer* instance running in its own thread while having a list of items for bid or TCP connections as a list of attributes in that instance. He was responsible for coding the GUI as well as the protocol for transferring messages to and from the GUI from the client process. He was responsible for implementing the UDP communication on both the server and client side. Furthermore, Ryan came up with the flow of communication between the GUI the client and server by implementing files that would hold the data that needed to be sent either way. Ryan took on a team leader role for the project and was very good at ensuring that the team knew what tasks needed to be completed to keep us well organized.

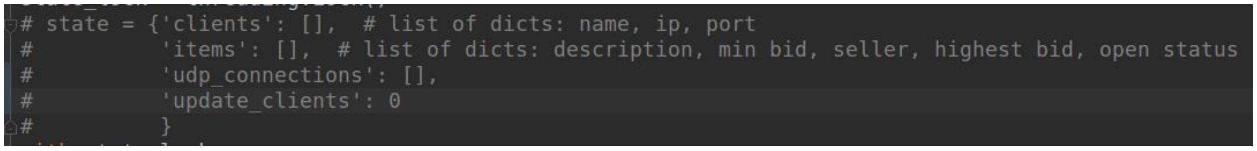
Anas and Adam were responsible for implementing the TCP connection between the client and the server. Anas focused on implementing the structure of the TCP connection between the client and the item on sale. Moreover, he implemented each TCP message and ensured that the server was able to support multiple items, each of which was connected to multiple clients. Adam focused on the client side by finding a way to reliably communicate server status to the clients in a way that ensured efficiency and atomicity. Furthermore, he was responsible for ensuring that the correct TCP messages were sent to the GUI.

The interaction between the UDP and TCP server required major adjustments to the preexisting UDPServer code, which Anas and Adam completed over the duration of the project.

All three group members were extensively involved in the debugging process, which took place over three sessions during the last two weeks before the project demo deadline. There were a lot of moving parts that needed to work in unison to ensure proper functionality. This lead to a lot of testing and debugging on school computers which the whole team was apart of.

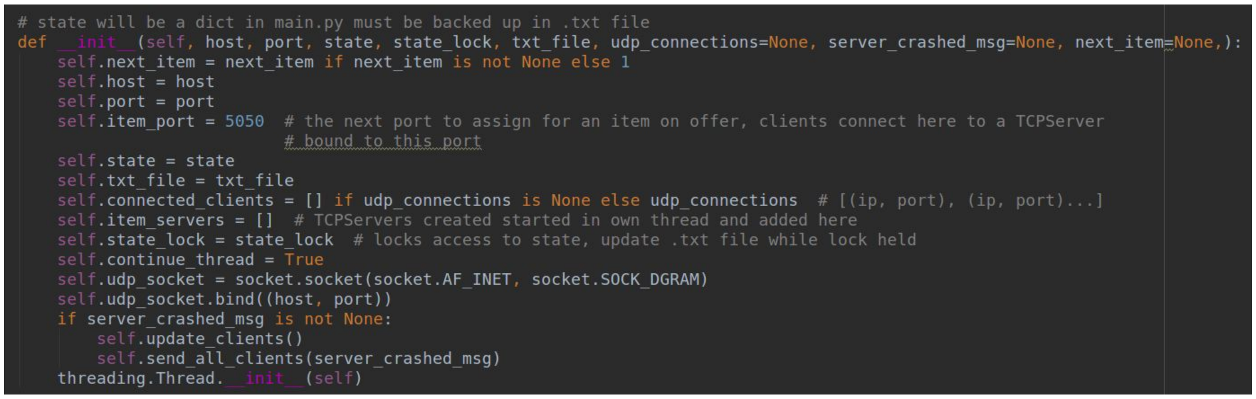
Appendix:

Figure 1



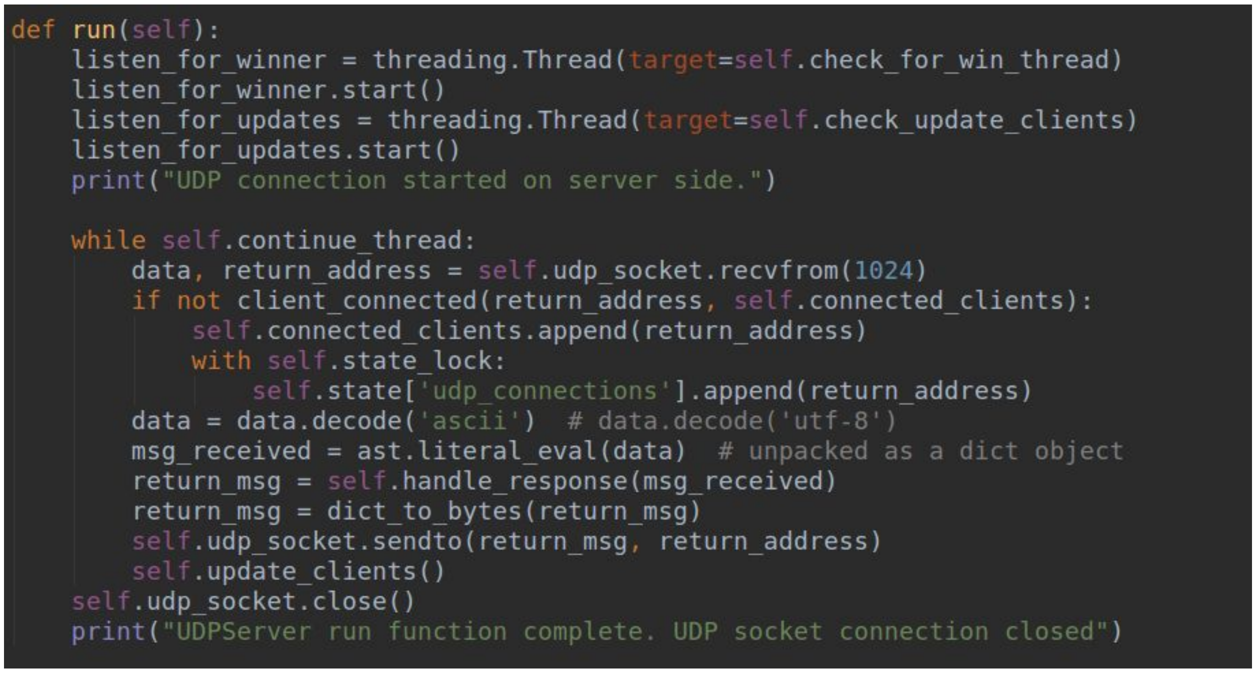
```
# state = {'clients': [],  # list of dicts: name, ip, port
#          'items': [],  # list of dicts: description, min bid, seller, highest bid, open status
#          'udp_connections': [],
#          'update_clients': 0
#          }
```

Figure 2



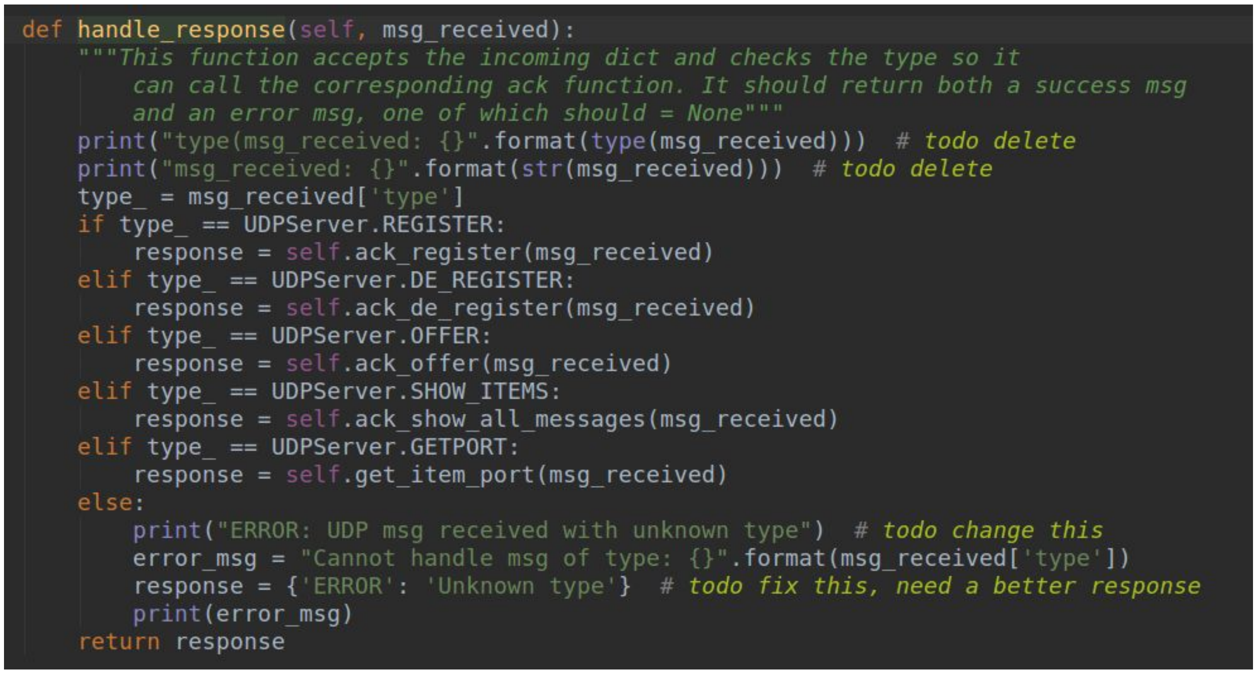
```
# state will be a dict in main.py must be backed up in .txt file
def __init__(self, host, port, state, state_lock, txt_file, udp_connections=None, server_crashed_msg=None, next_item=None,):
    self.next_item = next_item if next_item is not None else 1
    self.host = host
    self.port = port
    self.item_port = 5050  # the next port to assign for an item on offer, clients connect here to a TCPServer
                           # bound to this port
    self.state = state
    self.txt_file = txt_file
    self.connected_clients = [] if udp_connections is None else udp_connections  # [(ip, port), (ip, port)...]
    self.item_servers = []  # TCPServers created started in own thread and added here
    self.state_lock = state_lock  # locks access to state, update .txt file while lock held
    self.continue_thread = True
    self.udp_socket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    self.udp_socket.bind((host, port))
    if server_crashed_msg is not None:
        self.update_clients()
        self.send_all_clients(server_crashed_msg)
    threading.Thread.__init__(self)
```

Figure 3



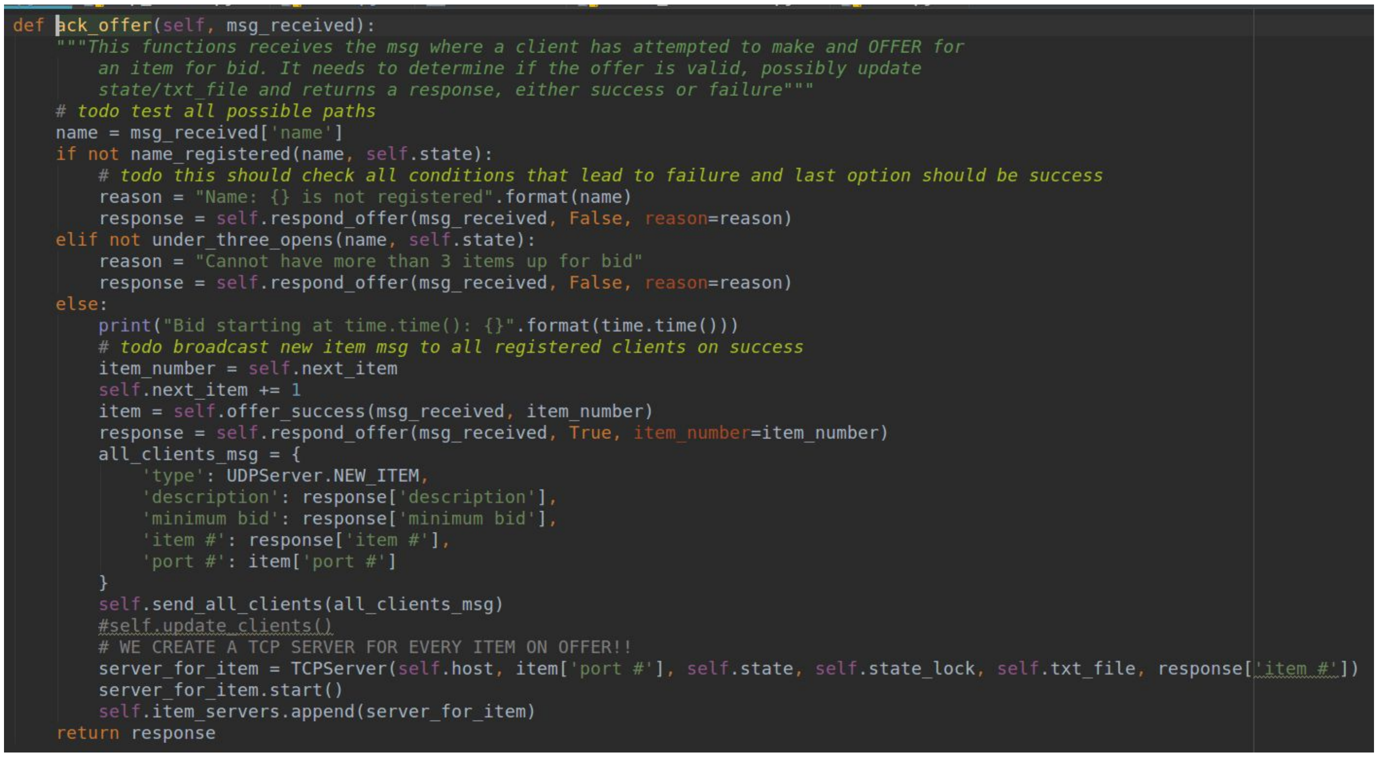
```python
def run(self):
    listen_for_winner = threading.Thread(target=self.check_for_win_thread)
    listen_for_winner.start()
    listen_for_updates = threading.Thread(target=self.check_update_clients)
    listen_for_updates.start()
    print("UDP connection started on server side.")

    while self.continue_thread:
        data, return_address = self.udp_socket.recvfrom(1024)
        if not client_connected(return_address, self.connected_clients):
            self.connected_clients.append(return_address)
            with self.state_lock:
                self.state['udp_connections'].append(return_address)
        data = data.decode('ascii')  # data.decode('utf-8')
        msg_received = ast.literal_eval(data)  # unpacked as a dict object
        return_msg = self.handle_response(msg_received)
        return_msg = dict_to_bytes(return_msg)
        self.udp_socket.sendto(return_msg, return_address)
        self.update_clients()
    self.udp_socket.close()
    print("UDPServer run function complete. UDP socket connection closed")
```

Figure 5



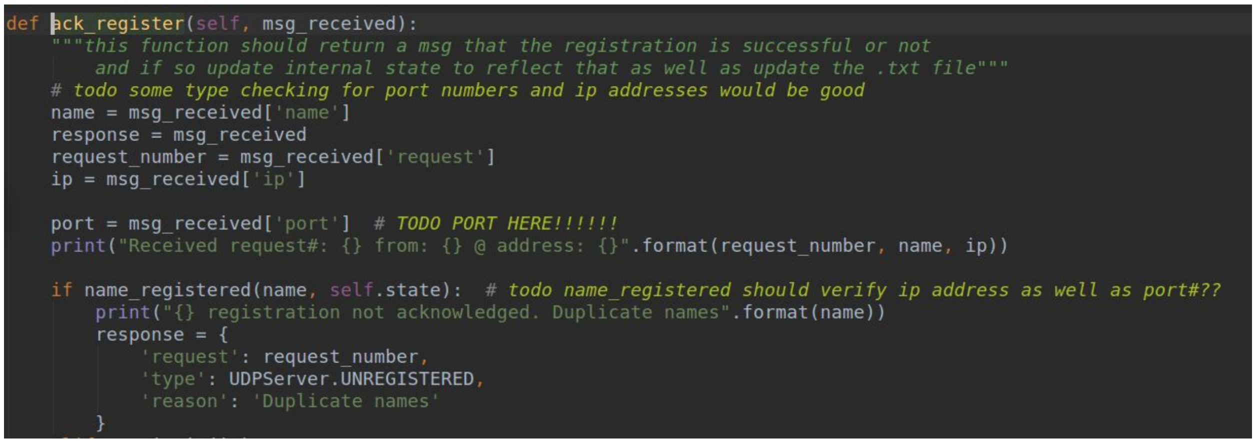
```python
def handle_response(self, msg_received):
    """This function accepts the incoming dict and checks the type so it
        can call the corresponding ack function. It should return both a success msg
        and an error msg, one of which should = None"""
    print("type(msg_received: {}".format(type(msg_received)))  # todo delete
    print("msg_received: {}".format(str(msg_received)))  # todo delete
    type_ = msg_received['type']
    if type_ == UDPServer.REGISTER:
        response = self.ack_register(msg_received)
    elif type_ == UDPServer.DE_REGISTER:
        response = self.ack_de_register(msg_received)
    elif type_ == UDPServer.OFFER:
        response = self.ack_offer(msg_received)
    elif type_ == UDPServer.SHOW_ITEMS:
        response = self.ack_show_all_messages(msg_received)
    elif type_ == UDPServer.GETPORT:
        response = self.get_item_port(msg_received)
    else:
        print("ERROR: UDP msg received with unknown type")  # todo change this
        error_msg = "Cannot handle msg of type: {}".format(msg_received['type'])
        response = {'ERROR': 'Unknown type'}  # todo fix this, need a better response
        print(error_msg)
    return response
```

Figure 6



```python
def ack_offer(self, msg_received):
    """This functions receives the msg where a client has attempted to make and OFFER for
       an item for bid. It needs to determine if the offer is valid, possibly update
       state/txt_file and returns a response, either success or failure"""
    # todo test all possible paths
    name = msg_received['name']
    if not name_registered(name, self.state):
        # todo this should check all conditions that lead to failure and last option should be success
        reason = "Name: {} is not registered".format(name)
        response = self.respond_offer(msg_received, False, reason=reason)
    elif not under_three_opens(name, self.state):
        reason = "Cannot have more than 3 items up for bid"
        response = self.respond_offer(msg_received, False, reason=reason)
    else:
        print("Bid starting at time.time(): {}".format(time.time()))
        # todo broadcast new item msg to all registered clients on success
        item_number = self.next_item
        self.next_item += 1
        item = self.offer_success(msg_received, item_number)
        response = self.respond_offer(msg_received, True, item_number=item_number)
        all_clients_msg = {
            'type': UDPServer.NEW_ITEM,
            'description': response['description'],
            'minimum bid': response['minimum bid'],
            'item #': response['item #'],
            'port #': item['port #']
        }
        self.send_all_clients(all_clients_msg)
        #self.update_clients()
        # WE CREATE A TCP SERVER FOR EVERY ITEM ON OFFER!!
        server_for_item = TCPServer(self.host, item['port #'], self.state, self.state_lock, self.txt_file, response['item #'])
        server_for_item.start()
        self.item_servers.append(server_for_item)
    return response
```

Figure 7



```python
def ack_register(self, msg_received):
    """this function should return a msg that the registration is successful or not
       and if so update internal state to reflect that as well as update the .txt file"""
    # todo some type checking for port numbers and ip addresses would be good
    name = msg_received['name']
    response = msg_received
    request_number = msg_received['request']
    ip = msg_received['ip']

    port = msg_received['port']  # TODO PORT HERE!!!!!!
    print("Received request#: {} from: {} @ address: {}".format(request_number, name, ip))

    if name_registered(name, self.state):  # todo name_registered should verify ip address as well as port#??
        print("{} registration not acknowledged. Duplicate names".format(name))
        response = {
            'request': request_number,
            'type': UDPServer.UNREGISTERED,
            'reason': 'Duplicate names'
        }
```

Figure 8

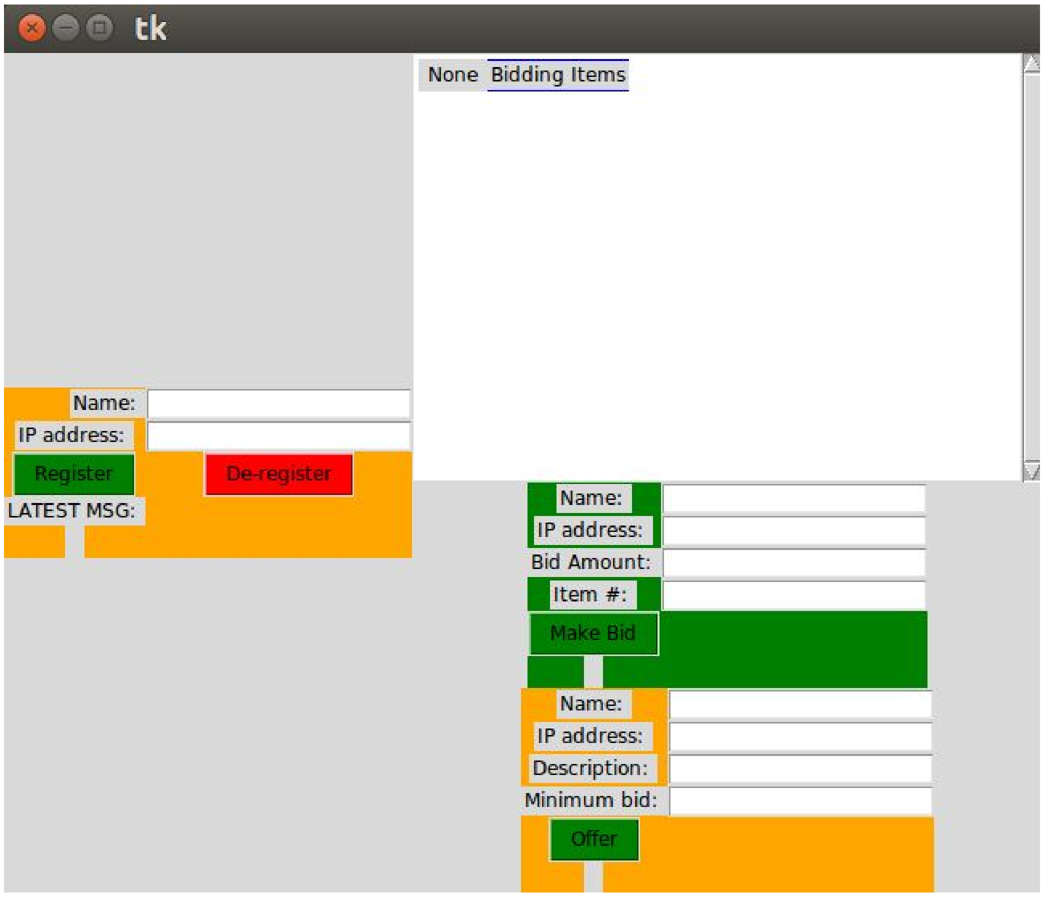
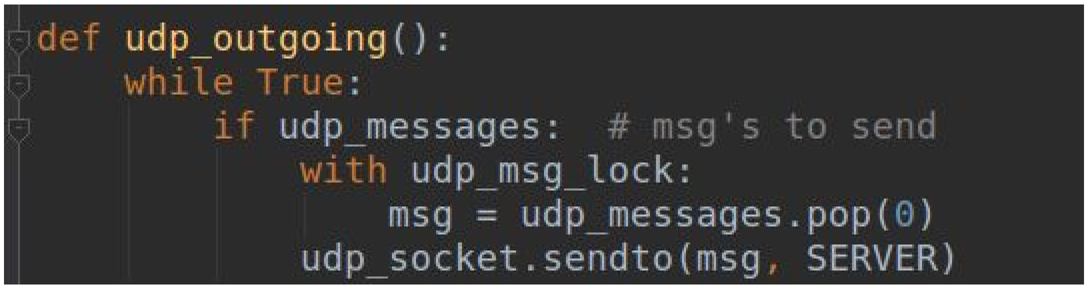
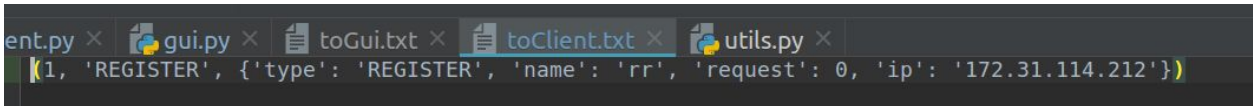


Figure 9



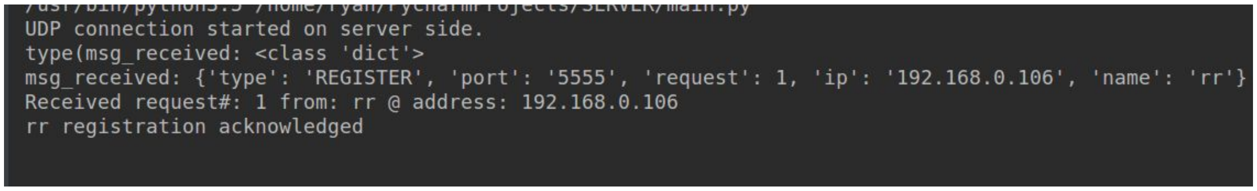
```
def udp_outgoing():
    while True:
        if udp_messages:  # msg's to send
            with udp_msg_lock:
                msg = udp_messages.pop(0)
            udp_socket.sendto(msg, SERVER)
```

Figure 10



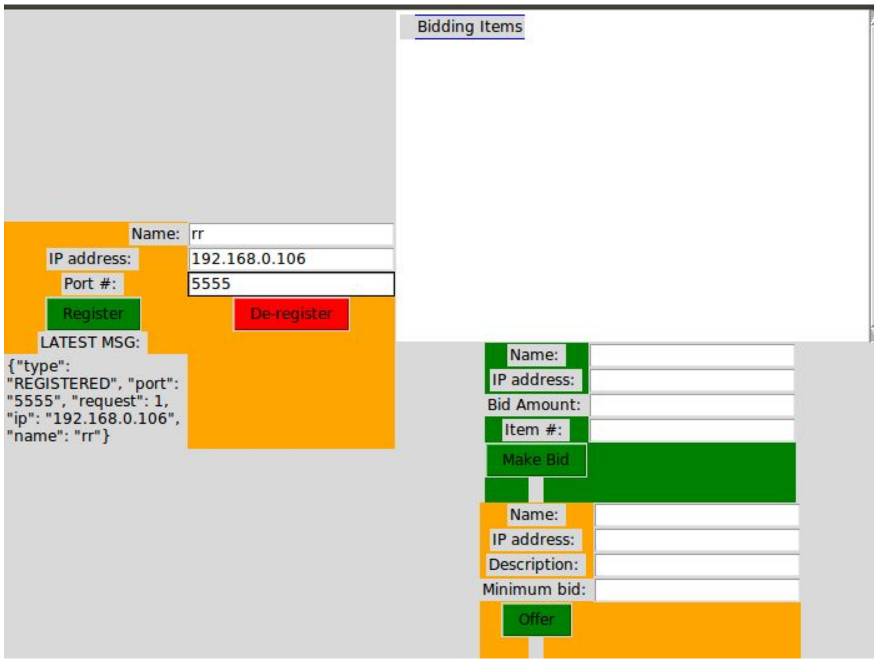
```
ent.py × gui.py × toGui.txt × toClient.txt × utils.py ×
(1, 'REGISTER', {'type': 'REGISTER', 'name': 'rr', 'request': 0, 'ip': '172.31.114.212'})
```

Figure 11



```
/usr/bin/python3.5 /home/ryan/PycharmProjects/SERVER/main.py
UDP connection started on server side.
type(msg_received: <class 'dict'>
msg_received: {'type': 'REGISTER', 'port': '5555', 'request': 1, 'ip': '192.168.0.106', 'name': 'rr'}
Received request#: 1 from: rr @ address: 192.168.0.106
rr registration acknowledged
```

Figure 12



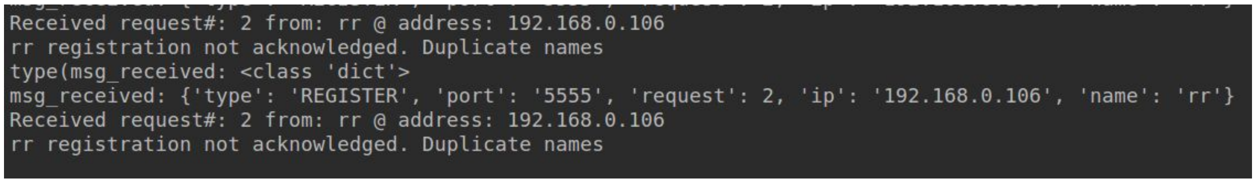
Name: rr
IP address: 192.168.0.106
Port #: 5555

LATEST MSG:
{"type": "REGISTERED", "port": "5555", "request": 1, "ip": "192.168.0.106", "name": "rr"}

Name:
IP address:
Bid Amount:
Item #:

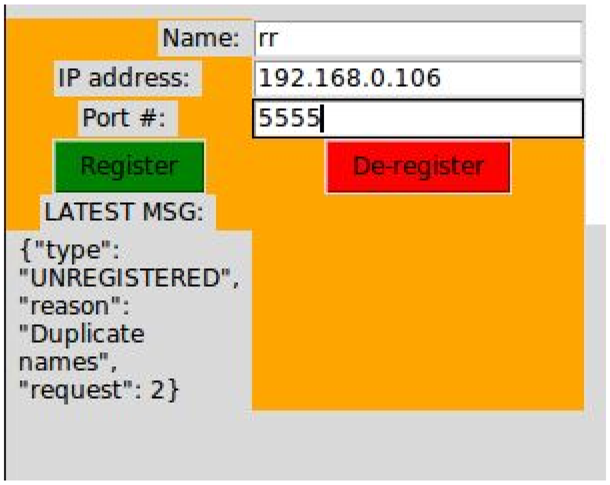
Name:
IP address:
Description:
Minimum bid:

Figure 13



```
Received request#: 2 from: rr @ address: 192.168.0.106
rr registration not acknowledged. Duplicate names
type(msg_received: <class 'dict'>
msg_received: {'type': 'REGISTER', 'port': '5555', 'request': 2, 'ip': '192.168.0.106', 'name': 'rr'}
Received request#: 2 from: rr @ address: 192.168.0.106
rr registration not acknowledged. Duplicate names
```

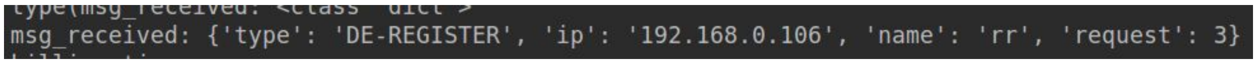
Figure 14



Name: rr
IP address: 192.168.0.106
Port #: 5555

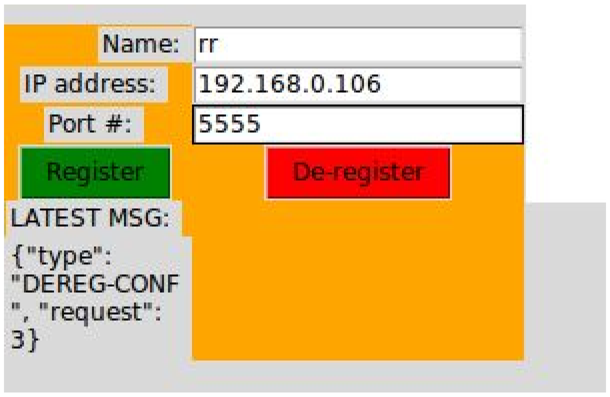
LATEST MSG:
{"type": "UNREGISTERED", "reason": "Duplicate names", "request": 2}

Figure 15



```
type(msg_received: <class 'dict'>
msg_received: {'type': 'DE-REGISTER', 'ip': '192.168.0.106', 'name': 'rr', 'request': 3}
```

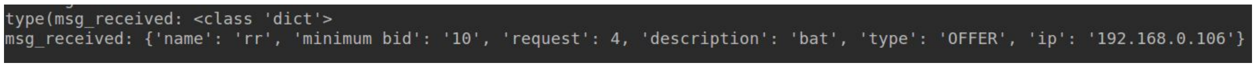
Figure 16



Name: rr
IP address: 192.168.0.106
Port #: 5555

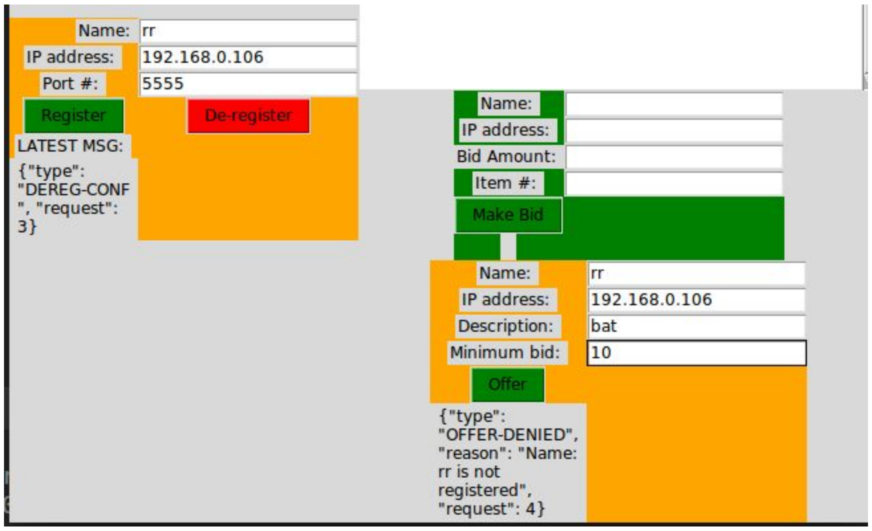
LATEST MSG:
{"type": "DEREG-CONF", "request": 3}

Figure 17



```
type(msg_received: <class 'dict'>
msg_received: {'name': 'rr', 'minimum bid': '10', 'request': 4, 'description': 'bat', 'type': 'OFFER', 'ip': '192.168.0.106'}
```

Figure 18



Name: rr
IP address: 192.168.0.106
Port #: 5555

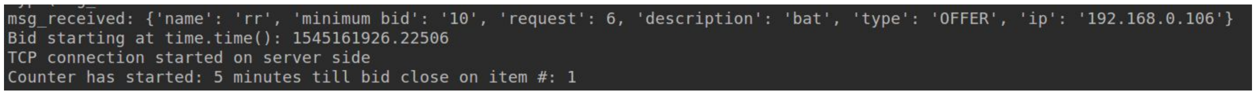
LATEST MSG:
{"type": "DEREG-CONF", "request": 3}

Name:
IP address:
Bid Amount:
Item #:

Name: rr
IP address: 192.168.0.106
Description: bat
Minimum bid: 10

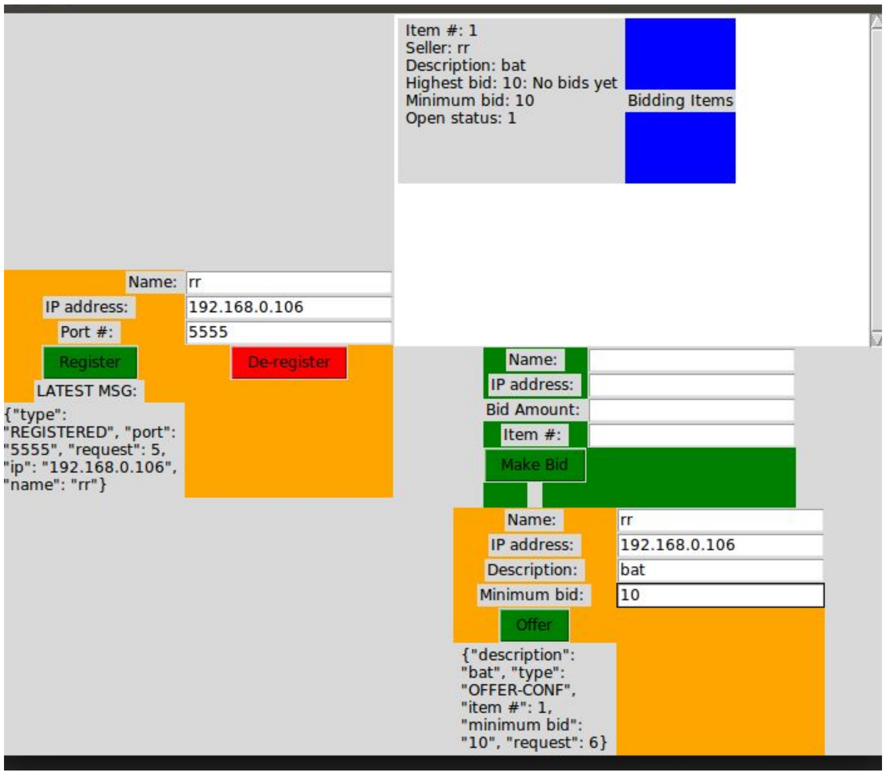
{"type": "OFFER-DENIED", "reason": "Name: rr is not registered", "request": 4}

Figure 19



```
msg_received: {'name': 'rr', 'minimum bid': '10', 'request': 6, 'description': 'bat', 'type': 'OFFER', 'ip': '192.168.0.106'}
Bid starting at time.time(): 1545161926.22506
TCP connection started on server side
Counter has started: 5 minutes till bid close on item #: 1
```

Figure 20



Item #: 1
Seller: rr
Description: bat
Highest bid: 10: No bids yet
Minimum bid: 10
Open status: 1

Name: rr
IP address: 192.168.0.106
Port #: 5555

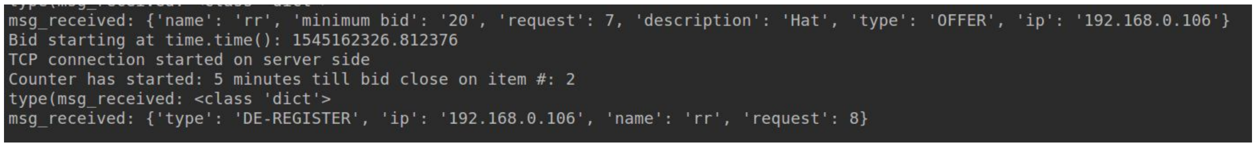
LATEST MSG:
{"type": "REGISTERED", "port": "5555", "request": 5, "ip": "192.168.0.106", "name": "rr"}

Name:
IP address:
Bid Amount:
Item #:

Name: rr
IP address: 192.168.0.106
Description: bat
Minimum bid: 10

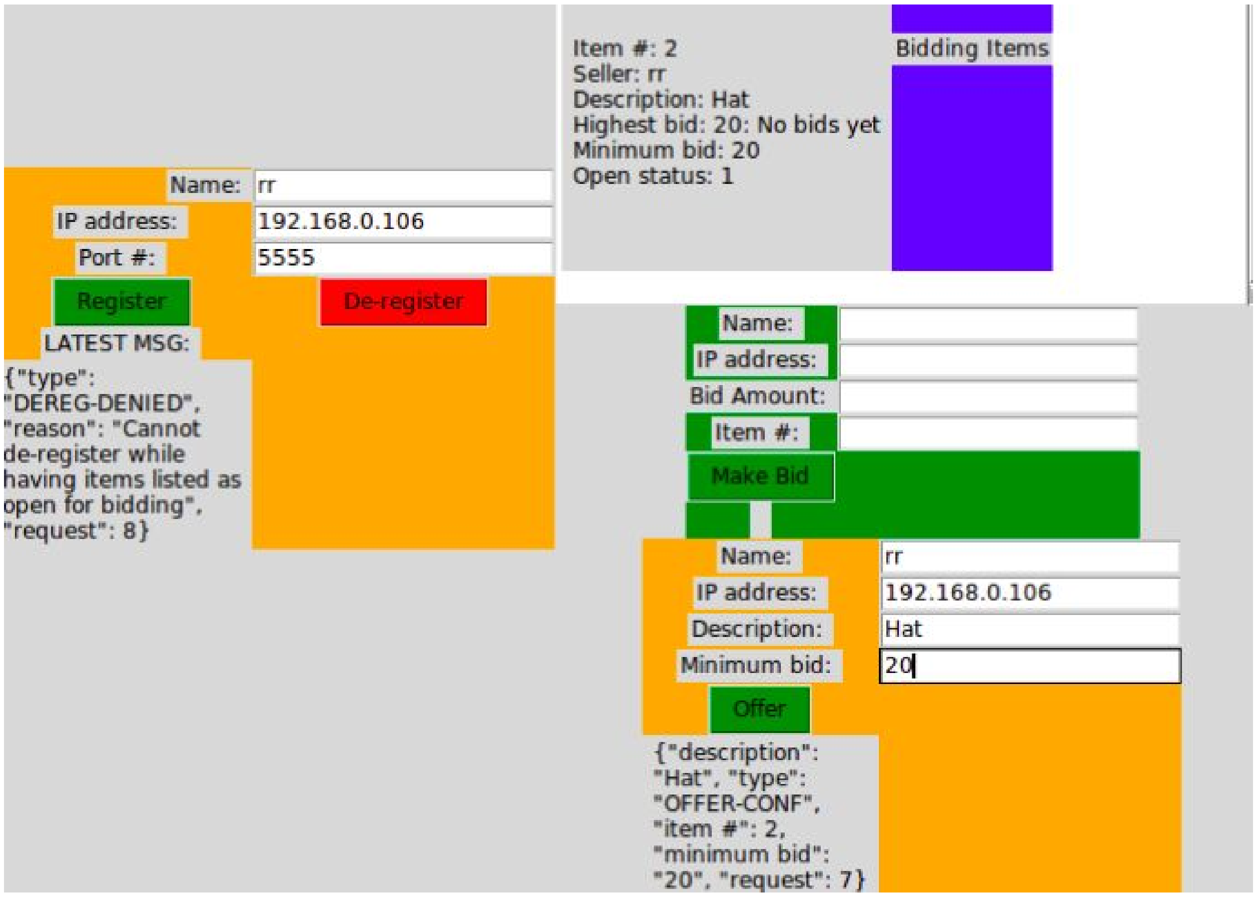
{"description": "bat", "type": "OFFER-CONF", "item #": 1, "minimum bid": "10", "request": 6}

Figure 21



```
msg_received: {'name': 'rr', 'minimum bid': '20', 'request': 7, 'description': 'Hat', 'type': 'OFFER', 'ip': '192.168.0.106'}
Bid starting at time.time(): 1545162326.812376
TCP connection started on server side
Counter has started: 5 minutes till bid close on item #: 2
type(msg_received: <class 'dict'>
msg_received: {'type': 'DE-REGISTER', 'ip': '192.168.0.106', 'name': 'rr', 'request': 8}
```

Figure 22



Item #: 2
Seller: rr
Description: Hat
Highest bid: 20: No bids yet
Minimum bid: 20
Open status: 1

Name: rr
IP address: 192.168.0.106
Port #: 5555

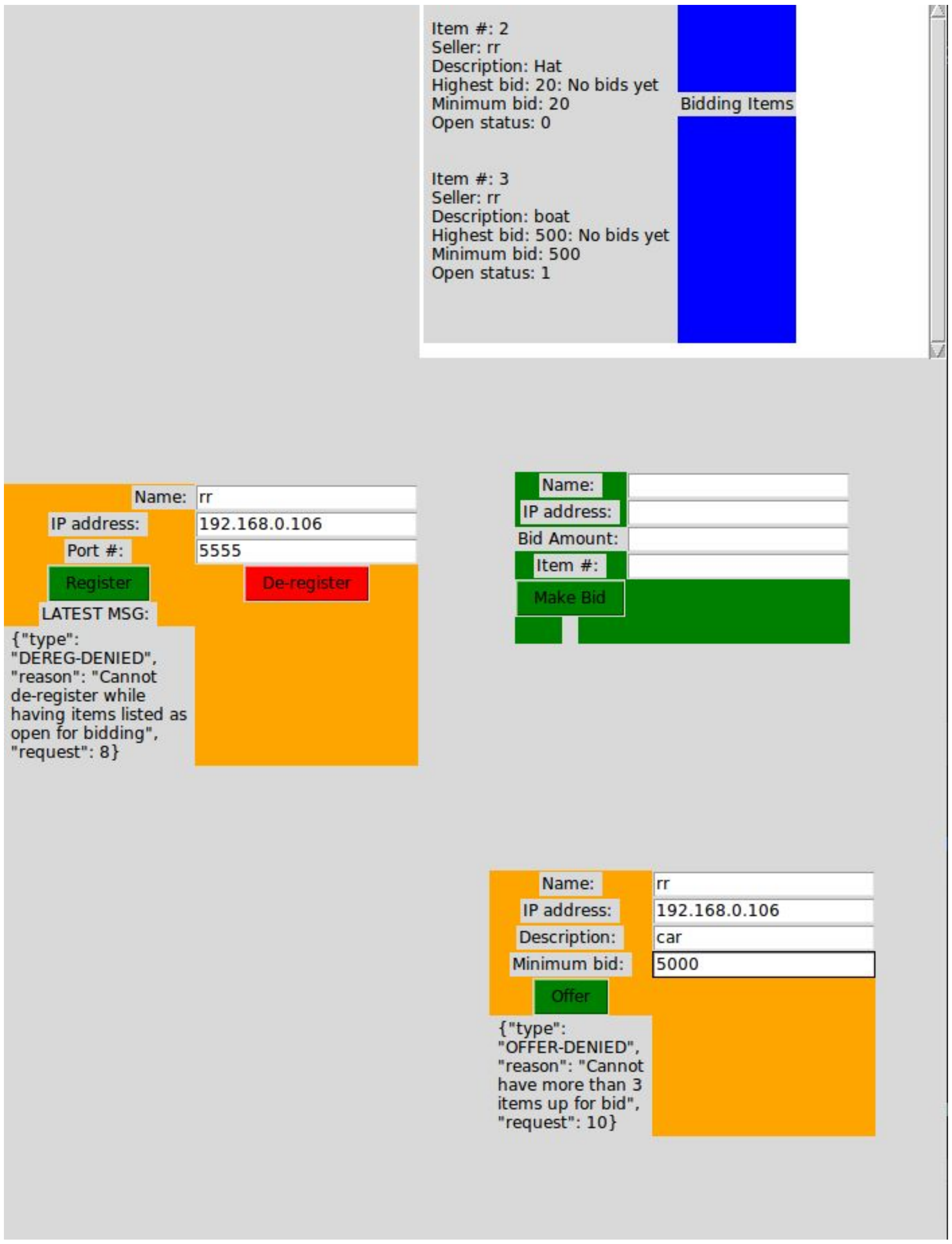
LATEST MSG:
{"type": "DEREG-DENIED", "reason": "Cannot de-register while having items listed as open for bidding", "request": 8}

Name:
IP address:
Bid Amount:
Item #:

Name: rr
IP address: 192.168.0.106
Description: Hat
Minimum bid: 20

{"description": "Hat", "type": "OFFER-CONF", "item #": 2, "minimum bid": "20", "request": 7}

Figure 23



Item #: 2
Seller: rr
Description: Hat
Highest bid: 20: No bids yet
Minimum bid: 20
Open status: 0

Item #: 3
Seller: rr
Description: boat
Highest bid: 500: No bids yet
Minimum bid: 500
Open status: 1

Name:	rr
IP address:	192.168.0.106
Port #:	5555

{"type": "DEREG-DENIED", "reason": "Cannot de-register while having items listed as open for bidding", "request": 8}

Name:	
IP address:	
Bid Amount:	
Item #:	

Name:	rr
IP address:	192.168.0.106
Description:	car
Minimum bid:	5000

{"type": "OFFER-DENIED", "reason": "Cannot have more than 3 items up for bid", "request": 10}