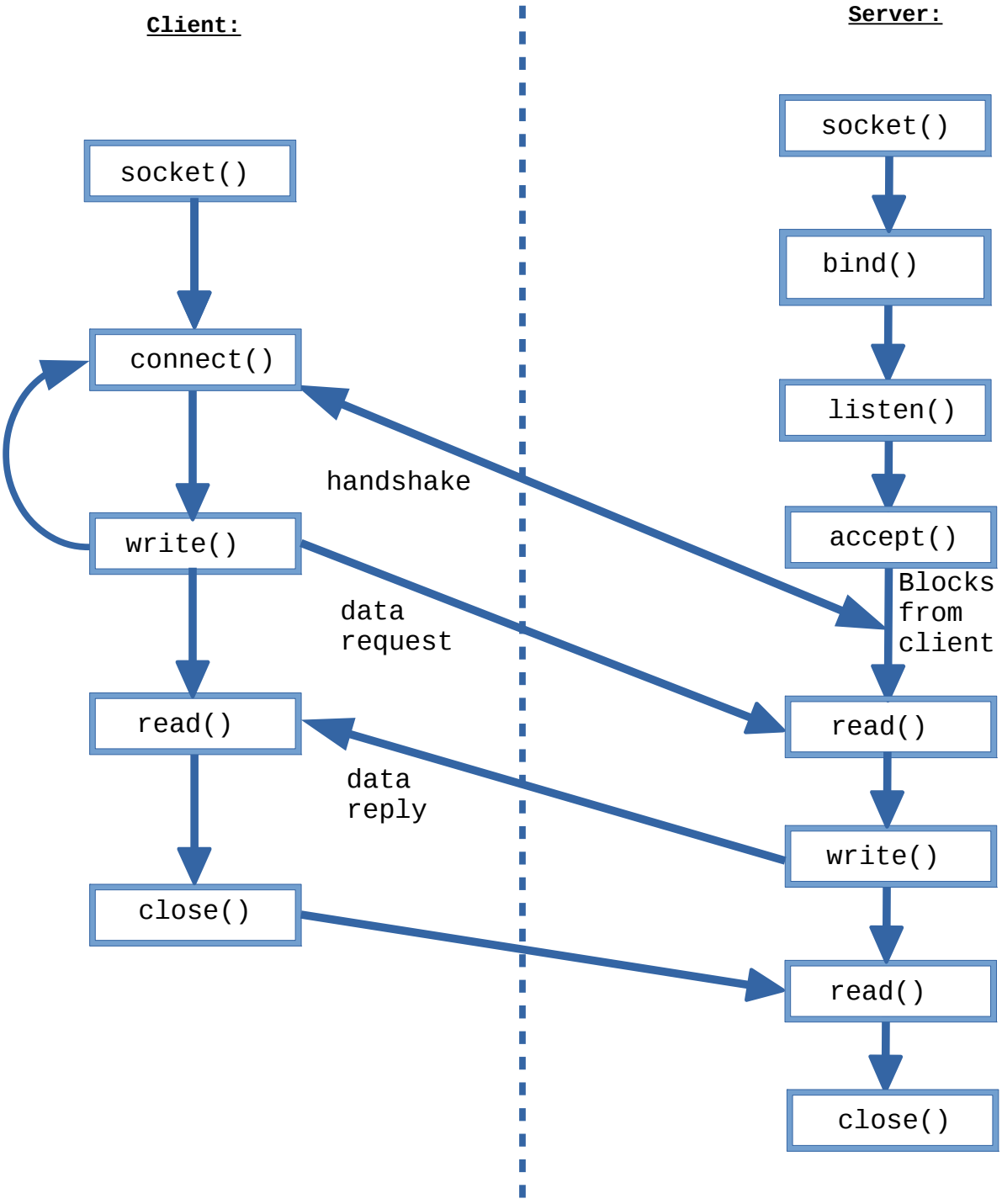
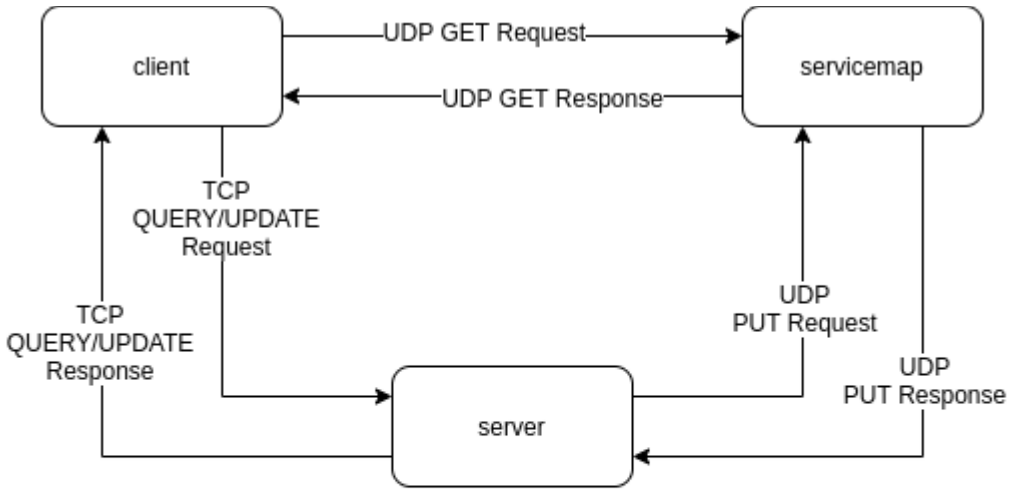
Assignment: Project#03
Name: Christen Ford/Daniel Izadnegahdar
Csu Id#: 2741896/2420596
Turnin Id#: 2741896(chford)
Turnin Date: 04/07/20

1. **Summary:**
   1. The purpose of this project is to design a client-server network system that can query a database. It will exercise functions like TCP/UDP sockets, lseek(), read(), write() and lockf(). The program is broken down into 3 files, the (1) servicemap.c, (2)server.c, and (3)client.c files. Each of them are described in more detail in this report.

2. **Flow:**
   1. **TCP Flow Diagram:**



2. **Client/Server/Servicemap Interaction Diagram**



1. server constructs a UDP PUT request and sends it to the servicemap.
2. servicemap responds with OK or FAIL in a UDP PUT response.
3. client constructs a UDP GET request and sends it to the servicemap.
4. servicemap responds with the service address string or FAIL in a UDP GET Reponse.
5. client creates a new connection to the server and sends a single TCP QUERY/UPDATE request.
6. server responds with the requested resource if QUERY or OK/FAIL if UPDATE.
7. client closes the connection upon receipt of message from the server.
8. client and server repeat steps 5-7 until the client program terminates.

3. **Description:**
   1. This project uses a client-server connection model using both TCP and UDP sockets. Transmission Control Protocol(TCP) is a connection-oriented protocol designed for applications that require high reliability but at the cost of transmission speed. It receives data in the same order it was sent, handles congestion, and provides error checking. On the other hand, the User Datagram Protocol(UDP) is a connection-less protocol that does not provide reliable in-order transmission, does not provide error checking, and does not require connection setup and teardown. UDP is employed in situations where transmission speed is preferred over reliability. When UDP is utilized, it is the responsibility of higher level protocols at the application layer to provide traditional TCP mechanisms for the employing application.
   2. A main component of this client-server model is socket programming. Socket programming is a way of connecting two nodes in a network by having one node listen to a port at a specific Internet protocol (IP) address, while the other socket reaches out to the other to form a connection.
   3. The key elements of the TCP network process is described below, as described during the 02/02/20 lecture.
4. **Socket:**
   1. **Prototype:**
      1. int socket(int domain, int type, int protocol);
   2. **Description:**
      1. Attempts to initialize a new socket or client endpoint. The domain is used to specify which family of protocols the socket will use, AF_INET for the Internet family. The type field specifies the communication semantics to be used at the transport layer and it may also accept a few options using the bitwise or operation on Linux kernel 2.6.27+ to

further customize the behavior of the socket. Lastly, protocol specifies a particular protocol to use with the socket, usually 0 for most socket types. Returns a file descriptor representing the socket on success, -1 on error.

5. **Connect:**
   1. **Prototype:**
      1. int connect(int sockfd, const struct sockaddr *addr, socklen_t addrlen);
   2. **Description:**
      1. Attempts to initiate a connection on socket sockfd with the remote perr address stored in addr. Returns 0 on success, -1 on error.
6. **Bind:**
   1. **Prototype:**
      1. int bind(int sockfd, const struct sockaddr *addr, socklen_t addrlen);
   2. **Description:**
      1. Binds the address addr to the socket specified by sockfd. Returns 0 on success, -1 on error.
7. **Listen:**
   1. **Prototype:**
      1. int listen(int sockfd, int backlog);
   2. **Description:**
      1. Sets the sockfd socket as passive, so it can accept incoming connection requests using the accept() function. Returns 0 on success or -1 on error.
8. **Accept:**
   1. **Prototype:**
      1. int accept(int sockfd, struct sockaddr *addr, socklen_t *addrlen);
   2. **Description:**
      1. It accepts and processes the 1st request from a queue of connection requests. It returns a file descriptor of the newly connected socket if successful or -1 on error.
9. **Read:**
   1. **Prototype:**
      1. ssize_t read(int fd, void *buf, size_t count);
   2. **Description:**
      1. Attempts to read up to count bytes from file descriptor fd into the buffer buf. Returns up to count on success, 0 when fd is at end of file, and -1 on error. It is possible for read() to return less than the number of bytes requested, for instance when the number of requested bytes exceeds the number of bytes available in the file from the file descriptors current position to the end of the file.
10. **Write:**
   1. **Prototype:**
      1. ssize_t write(int fd, const void *buf, size_t count);
   2. **Description:**
      1. Attempts to write count bytes to the file specified by fd from the buffer buf. Writes up to count bytes on success, -1 on error.
11. **Close:**
   1. **Prototype:**
      1. int close(int fd);
   2. **Description:**
      1. Attempts to close the file descriptor fd. Returns 0 on success, -1 on error.

3. **File servicemap.c:**
   1. **Summary:**
      1. This file maps services to sockets, where a socket is specified by an IP

address and port number of the form: [0-255],[0-255],[0-255],[0-255],[0-255],[0-255]. Communication with the servicemap is strictly UDP-based. The servicemap accepts two commands: PUT and GET. The servicemap stores services and addresses from servers using the PUT command and serves clients using the GET command.

2. **Code Description:**
    1. The program begins with a series of networking and socket libraries for connectivity reasons. This includes <sys/socket.h>, <netinet/ip.h>, and <arpa/inet.h>.
    2. Next, the main() function starts by creating a socket and binding. The program will exception handle if either of those functions return less than 0. To convert the port#, the htons() function was used, because the unsigned short integer from the host needs to be converted to network byte order.
    3. Next, the program listens for the incoming string from the UDP socket, and parses it into tokens. The 1st token is the command i.e. 'PUT' or 'GET', the 2nd token is the service name, and the 3rd token is the IP/Port# address string. Once the command is received, a "received" confirmation is printed per the requirements of the assignment.
    4. If the mapper receives a 'PUT' command, it will insert the service name, IP address, and port# into an address table, and confirm success to the server.
    5. If the mapper receives a 'GET' request, it will fetch the IP/Port# address string from the address table(provided a service name argument) and return this to the client.

4. **File server.c:**
    1. **Summary:**
        1. This file is used to run client commands against a database. To achieve this, it needs to make a connection with the client and fork a child process to handle the request. The child process will use lseek() to locate the position to rewrite, and lockf() to protect a critical region from updating simultaneously. The parent process will use a signal-catching routine to clean up child processes that have terminated.
        2. The server will follow a similar sequence of events described on the flow diagram of this report i.e. socket() → bind() → listen() → accept() → read() → write() → read() → close() and is structured similarly to the class Server_TCP.c example.
    2. **Code Description:**
        1. The program begins with a series of networking and socket libraries for network connectivity reasons.
        2. Next, a series of global variables that are re-used throughout the program are defined here. This includes port#s and addresses. The client/port#s are well-known, whereas the service map port# is custom defined as 2 + 4#s of the turnin csuId# as described on the 03/05/20 lecture.
        3. Next, a series of structs are defined for the command messages and socket.
        4. Next, it broadcasts the service to the mapper using a broadcast_service() function. In this function, it first creates a socket using socket() and enables broadcasting by using setsockopt(). The port# is converted between host and network byte order using htons(), for the same reason described in the mapper section. Next, the socket is bound to the address using the bind() function, as described by the sequence in the flow diagram. The program will then run through a series of internet address manipulation routines such as inet_addr() and inet_ntoa(), because different address types have different formats. For example, linux lab broadcast address is 137.148.254.255, local in-home address is 192.168.0/1.255, and generic broadcast address is 255.255.255.255. It

then sends the message to the socket using sendto(), and waits to receive the message using recvfrom(). Finally, the socket is closed using close().

5. Once the broadcast is done, the server is ready to read messages from the client, through a process of binding, listening, and accepting. The program will run through an infinite while loop, to continuously listen for signals from the client. If a message is accepted, it creates a new process using fork(). If the process is a child, it will read the message, process the command with the database, return the results to the client using the send() command, and exit with a success mode of 0. If the process is the parent, the remote_tcp_sk socket will close. If the process is anything else, an error is handled and the program exits with a failure mode of 1.
6. When the program is running the commands from the client, it will either run the query_db() or the update_db() function.
    1. The query_db() function is similar to SELECT in SQL. It begins by calling open(), to bind the db20 binary database to a file descriptor. The descriptor is then read through, one byte at a time, to search for a match with the account# provided in the acctnum argument. If no match is found, query_db() returns -1.
    2. The update_db() function is a data manipulation command similar to UPDATE in SQL. It begins by calling open(), to bind the db20 binary database to a file descriptor. If the account# is found, it will use lseek() to update the location of the pointer in the database data, and update the proper record. The lockf() function is used to protect the critical region from being updated by another concurrent query.
7. The print_db() function is used for printing the database results clearly to the user. It opens and reads the proper records, and uses the printf() command to display properties of the account.
8. Finally, there are a series of other helper functions for string manipulation, token extraction, and byte format conversion. Since the port# is represented in 2 byte format, the quotient(using the / operator) and remainder(using the % operator) can be extracted by dividing by 256.
9. The reason some of these helper functions are not included in their own file is to maintain the syntax of the provided turnin command. In a real application, these functions would live inside their own utility files to avoid code redundancy.

5. File client.c:

1. Summary:

1. This file takes input from the user and communicates with the server to execute database commands(in this case, the ‘query’ and ‘update’ command).
2. The ‘query’ command will return the record for the account# and has the following message structure: 1001(4-byte int)+acct num(4-byte int) in big-endian order.
3. The ‘update’ command will change a person’s record in the database and has the following message structure: 1002(4-byte int)+a ct num(4-byte int) +amount(4-byte float) in big-endian order.
4. This client will follow a similar sequence to the Client_TCP.c example described on 02/02/20, with the following socket() → connect() → write() → read() → close().

2. Code Description:

1. The program begins with a series of networking and socket libraries for network connectivity reasons.
2. Next, a series of global variables are defined here. This includes port#s and addresses similar to server.c.
3. Next, the program begins by creating a socket() to initialize an end

point.
4. Next, a request is made to get the address string by using the get_service_address() function. This function starts by enabling the socket to broadcast, by updating the socket using setsockopt(). A series of conversions are done after that, using htons() and inet_address() similar to what was done and described in the boradcast_service() function of server.c. Once the socket is updated, it assigns the address to the socket using the bind() function and is ready to send and receive messages using the the sendto() and recvfrom() functions. It then uses memcpy() and memset() to store the s_addr_info it received from the database. Finally, it closes the socket with close().
5. Once the address is received, it is parsed for its ip#/port# using the from_addr_string() function. These numbers are then printed to the user.
6. Next, the program is ready to take input commands from the user. To start, it needs to initialize a local TCP socket to communicate commands.
7. The prompt begins with a while loop set to 1, to continuously search for input. It then prints a prompt initiator, in this case the char “>”, and waits for user input. If the string is empty, it will skip the remaining code and run the next iteration to prevent a segmentation error. If the string is not empty, it will parse the string for its tokens using the same parse_string() function used in the server.c file. In this case, the delimiter is the comma instead of the period.
    1. The 1st token is the command type, if it reads “query”, the command will build a query request using init_query() and memcpy() and send it to the database using the connect_and_send() function. The connect_and_send() function is used for both query and update commands, and contains a series of arguments to properly connect from the socket and send a message. The arguments include the local socket to send from, the peer to send a message to, the size of the peer, the buffer containing the message to send, the length of the send buffer, the buffer to write the response back to, and the length of the received buffer. The function will return 0 on success, and -1 on error. The function will first clone the socket because it is not good to connect directly to an original client socket. Next, a connection is made with the clone socket using connect(). It will then send the message using send(), receive using recv(), and finally close().
    2. If the command reads “update”, the command will run a similar routine for the update command.
    3. If the user types “help”, a help menu will show the user the proper syntax for running database commands in this program.
    4. If the user types “quit”, the while loop will break, and the program will exit out naturally.
    5. If the user types neither of those 4 conditions, the user will be asked to try again, because the provided input was not valid.
8. The code finishes by closing the socket using the close() function.

6. File db20 description:

1. Summary:

1. this file is the provided database which contains data in binary format.

2. Code Description:

1. The data contains records of accounts in the form of the following struct record{int acctnum; char name[20]; float value; int age;};

7. File makefile description:

1. Summary:

1. This file defines a set of rules for use by the make program. It is used to execute commands with a simple pseudonym such as ‘make all’ or ‘make

clean’.

2. **Code Description:**
   1. The file starts with a series of header macros and $() variables to maintain one source of truth.
   2. The file then includes a series of commands for cleaning files, linking, compiling, and submitting. It contains 3 main commands: (1)make, (2)clean, (3)submit.
      1. The ‘make’ command will build the object and executable files.
      2. The ‘clean’ command will remove the object and executable files.
      3. The ‘submit’ command will run turnin to compress and send the project to the cis620s assignment account for grading.

8. Results:

1. Run instructions:

1. To run the program, open 3 terminals and ssh connect to 3 Linux computers on the same LAN, i.e. spirit, strauss, and rodin.
2. Browse to the location of the program and type ‘make’.
3. On the 1st terminal, type: ./servicemap.
4. On the 2nd terminal, type: ./server.
5. On the 3rd terminal, type ./client.
6. Still on the 3rd terminal, type the appropriate database command, either `QUERY <acctnum>` or `UPDATE <acctnum> <value>`. The client and server should output the appropriate information if the command succeeded.

2. Run output:

1. run make on any of the 3 terminals:

```
daizadne@spirit:~/Desktop/hw04/04-05-20/submission$ make
gcc -g   -c -o client.o client.c
gcc -o client client.o
gcc -g   -c -o server.o server.c
gcc -o server server.o
gcc -g   -c -o servicemap.o servicemap.c
gcc -o servicemap servicemap.o
daizadne@spirit:~/Desktop/hw04/04-05-20/submission$
```

2. run servicemap on spirit:

```
daizadne@spirit:~/Desktop/hw04/04-05-20/submission$ ./servicemap
Received from 137.148.204.27: PUT CISBANK 137,148,204,27,97,30
Received from 137.148.204.25: GET CISBANK
```

3. run server on rodin:

```
daizadne@rodin:~/Desktop/hw04/04-05-20/submission$ ./server
Registration OK from 137.148.204.14
Service Requested from 137.148.204.25
Service Requested from 137.148.204.25
Service Requested from 137.148.204.25
Service Requested from 137.148.204.25
```

4. run client on strauss:

```
daizadne@strauss:~/Desktop/hw04/04-05-20/submission$ ./client
Service provided by 137.148.204.27 at port 24862
>: query 11111
MULAN HUA 11111 99.9
>: query 34567
BILL SUN 34567 66.0
>: update 34567 8.2
BILL SUN 34567 74.2
>: quit
daizadne@strauss:~/Desktop/hw04/04-05-20/submission$
```

3. **Description:**
   1. The outputs are displayed in sequence of how the program is suppose to be run servicemap→server→client. The servicemap output confirms that it was able to initiate our system and runs 2 simple commands against the database. The server output confirms that it was able to register and accept requests from the provided address, and can now accept requests from the client. The client output shows that the user can now enter queries, and that the output from the queries is consistent with the data in the database.

9. **Experiences in testing/debugging:**
   1. **Error#01:**
      1. **File:**
         1. server.c
      2. **Description:**
         1. Unable to convert the host addresses in the hostent field, h_addr_list to the format needed by the servicemap. Most documentation on gethostbyname() and the hostent struct do not specify what the data type is for the entries stored in h_addr_list field of the hostent struct. Attempts at converting the field to the ###,###,###,### IP portion of the address string stored by the servicemap were failing.
      3. **Solution:**
         1. After much searching, Christen was able to determine that the data type stored in the h_addr_list field is of type `struct inaddr`. This seems like a documentation error for the most part as Christen only resolved this bug by finding the appropriate documentation in an IBM developer documentation database.
   2. **Error#02:**
      1. **File:**
         1. client.c
      2. **Description**
         1. Input strings and their individual tokens were not being interpreted correctly by the command execution sub-system. The client utilize fgets() for input which does not discard the newline character ‘\n’ prior to storing received input in the destination buffer. This led to commands being formed for the server with improper input.
      3. **Solution**
         1. The client removes the newline character from input if the length of the input string is > 0 (see **Error#03**).
   3. **Error#03:**
      1. **File:**
         1. client.c
      2. **Description:**

1. Segmentation error is displayed when there is no input provided from the command prompt.

3. **Solution:**
    1. This bug occurs when the strcmp() attempts to match against the various client commands. Because the user did not input any information into the client, the parsing sub-system returned an array containing all (null) characters. This caused a segmentation fault when strcmp() attempted to match against the first token (the command). To address this, an if statement was added to continue to the next iteration of the command input loop if the length of the input string prior to removing the newline from input is 0.

4. **Error#04**:
    1. **File:**
        1. All files
    2. **Description**
        1. Requesting and sending data over an Internet socket was unreliable with respect to the amount of bytes sent/received. If more data was requested than the sending socket provided, the receiving socket would return garbage data alongside actual data to the receiving buffer. This made interpretation of received data difficult and presented us with an infuriating bug.
    3. **Solution**
        1. To unify the number of bytes sent and received over all socket interfaces in the project, we implemented a packet_t type that wraps all messages sent over the socket. Header level information is stored within the struct per normal while the body of the packet stores information using a union of all the various message types the packet can contain. Senders and receivers are able to correctly identify the type of data stored in the body of the packet by inspecting the ptype field stored within the packet. As long as senders and receivers correctly set the ptype field and write to the proper field in the body union, the program functions as intended. This improvement allowed for sending and receiving a fixed size chunk of information over all socket interfaces eliminating the uncertainty and potential data corruption introduced by sending and requesting messages of varying size. Future extensions of the packet design for UDP could include adding a checksum field and sequence numbers to ensure that received packets over UDP are correct and in-sequence.

5. **Bug#01**
    1. **Description**
        1. During testing on the Linux lab environment, it seemed like the broadcast address 137.148.254.255 handed out by Dr. Sang in class was not correct. When Christen tested the three programs, they refused to communicate using this broadcast address.
    2. **Solution**
        1. Utilizing the Linux command line tool ifconfig. Christen was able to determine that the proper broadcast address for the Linux lab environment is in fact (at the time of this writing) 137.148.205.255 despite the netmask being 255.255.254.0. Christen is unsure why the network is configured this way as usually the broadcast address for an IPV4 network is specified through the LANs netmask. Upon changing the broadcast address to the address returned by ifconfig in the necessary places in the programs for the assignment, UDP communication worked as expected. Please see the attached screenshot for proof of this phenomenon. You are also free to run ifconfig yourself on any machine connected to the Linux lab environment. They should all return the same information.

```
spirit:~% ifconfig
eno1: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 137.148.204.14  netmask 255.255.254.0  broadcast 137.148.205.255
        inet6 fe80::1a60:24ff:fead:2fb  prefixlen 64  scopeid 0x20<link>
        ether 18:60:24:ad:02:fb  txqueuelen 1000  (Ethernet)
        RX packets 541848704  bytes 352928442235 (352.9 GB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 1169190663  bytes 156439907434 (156.4 GB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
        device interrupt 16  memory 0xd3200000-d3220000
```

Here one sees that the IP address of spirits' Ethernet interface, eno1, is 137.148.204.14 while the netmask for the entire LAN is 255.255.254.0. This would lead one to conclude that the broadcast address for the Linux lab environment is 137.148.254.255 however, that is in contradiction to the broadcast address returned by ifconfig, 137.148.205.255. This is in direct conflict with the information provided in class by Dr. Sang but upon testing, the broadcast address returned by ifconfig is in fact the correct broadcast address. Likewise, 137.148.205.255 is the broadcast address our submission is configured to use.