Technical Design Report
Server Connection to Robot

Igor Naperkovskiy and Joao Rodrigues
Seneca College – School of ICT
70 The Pond Rd
Toronto, ON, Canada
jmrodriguesgoncalves@myseneca.com
inaperkovskiy@myseneca.com

Abstract**:** *This report aims at documenting the progress and results of the term project for Data Communications. It will also act as a design document, and step by step manual to the server created for this assignment.*

Keywords**:** *TCP/IP, communications link, robotic connection, client, server, encapsulation, packet, parity, reliable connection, application protocol, data link;*

I. Introduction

This report will describe the design and application choices done in order to successfully connect to a client mobile robot with a unique application layer protocol through a Server application using a reliable TCP/IP communications link. The server listens to a robot that is connected to a Wifi access point. The robot begins by sending a status to the server of “Waiting”. The server that is in listening mode, after receiving a client packet, allows for the user to input a number that will correspond to the commands given to the robot. A user can request for a Status, Drive or Sleep, A connection socket is then sent to the robot via a Buffer of char characters with the appropriate commands, and proper data allocation to be validated on the client side that is waiting, listening to the Server. Once the set of commands are completed, the client will again send a “Waiting” status, back to the server, continuing the process in a loop. The user can stop this loop by sending a packet with a Sleep ID, ceasing all communications with the client.

II. Details of Server Design

```
int main() {

    SOCKET ServerSocket, ConnectionSocket;
    char IP[128] = { "127.0.0.1" };
    int Port = 5000;

    while (1){
        PktDef PacketObj;
        int status = 0;
        if (PacketObj.Listen(ServerSocket, IP, Port) != 0) {
            if (PacketObj.Accept(ServerSocket, ConnectionSocket) != 0) {

                PacketObj.SetInfo();
                status = PacketObj.Send(ConnectionSocket);
                if (status == 5){
                    PacketObj.ReceivePkt(ConnectionSocket);
                }
                PacketObj.CloseSocket(ConnectionSocket);
                PacketObj.CloseSocket(ServerSocket);
                PacketObj.WinsockExit();
                if (status == 5){
                    PacketObj.PrintInfo();
                }
            }
        }
        else {
                cout << " Connection Error - Exiting..." << endl;
        }
    }
    return 0;
}
```

Server.cpp - How it looks

The main body of the application. The operation begins with the creating of a packet that listens on the localhost and port 5000, and accepts the connection. If both conditions are met, the object will then be set, and sent to the client. If the Status is called (5), the robot will send a packet back to the server containing information. Lastly, the connections are closed, and if the Status was called for, then printed for the user to see. This is encapsulated in a while loop until the user decides to put the robot to sleep.

Pkt_Def.h - Definition of Server workload

```
struct Header{
    unsigned char id;
    unsigned char size;
};

struct Commands{
    unsigned char direction;
    unsigned char duration;
};

class PktDef {
private:
    Header head;
    Commands *body;
    unsigned char trailer;
```

In order to store the appropriate data in a more organized fashion, the definition for the header and the commands are placed in two different structs, that are then private variables of the main Packet Definition.
The Header struct contains unsigned char data, an ID (for robot action) and a size (how many commands will be allocated).
The Commands struct contains unsigned char data as well, namely the direction and duration of the command given to the client.
These two structs will be sent over the socket.

```
public:
    ~PktDef();
    void SetInfo();
    int Listen(SOCKET &, char *, int);
    int Accept(SOCKET &, SOCKET &);
    int Send(SOCKET &);
    void CloseSocket(SOCKET &);
    void WinsockExit();
    void ReceivePkt(SOCKET &ConnectionSocket);
    void PrintInfo();
};
```

Function definitions for the object. These will be expanded upon from this point onward. Although the names are standard for any TCP/IP connection the design behind these is made to fit with the robot's application protocol.

```
int PktDef::Listen(SOCKET &ServerSocket, char * IP, int Port) {
    //starts Winsock DLLs
    WSADATA wsaData;
    if (WSAStartup(MAKEWORD(2, 2), &wsaData) != 0)
        return 0;

    //create server socket
    ServerSocket = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
    if (ServerSocket == INVALID_SOCKET) {
        WSACleanup();
        return 0;
    }

    //binds socket to address
    struct sockaddr_in SvrAddr;
    SvrAddr.sin_family = AF_INET;
    //Listen on local host
    SvrAddr.sin_addr.s_addr = inet_addr(IP);
    //Listen to port 5000 for robot
    SvrAddr.sin_port = htons(Port);
    if (bind(ServerSocket, (struct sockaddr *)&SvrAddr, sizeof(SvrAddr)) == SOCKET_ERROR)
    {
        closesocket(ServerSocket);
        WSACleanup();
        return 0;
    }

    //listen on a socket
    if (listen(ServerSocket, 1) == SOCKET_ERROR) {
        closesocket(ServerSocket);
        WSACleanup();
        return 0;
    }

    cout << "Waiting for robot connection..." << endl;

    return 1;

}
```

Pkt_Def.cpp - The Design Behind It

*int Listen(SOCKET &, char *, int);*

This function is responsible for listening to the client's transmission. It begins by initiating the winsock DLLs, instantiating a server socket object, and validating whether it is valid, then binds the socket to the IP address and port that were defined in the main. The Server then listens for the IP address and port defined by the bind command, waiting for incoming connection requests from the client which in this case is the robot. The maximum number of connections in this case is 1. Lastly, a simple message is displayed for the user to know that the server is ready to receive data from the client.

```
void PktDef::SetInfo() {
    delete[] body;
    body = nullptr;
    bool tr = true;

    while (tr){ //loop start
        int AuxInt;
        cout << "What is the packet ID(15 DRIVE, 5 STATUS, 0 SLEEP)? ";
        cin >> AuxInt; //enter packet ID
        head.id = AuxInt;

        if (AuxInt == 15){ // if id is drive
            cout << "What is the body size(no more than 10): ";
            cin >> AuxInt; //enter number of commands
            head.size = AuxInt;
            AuxInt = AuxInt * 2;
            int size = AuxInt;

            if (head.size <= 10){ //no more than 10 commands

                body = new Commands[head.size]; //allocates memory for commands

                for (int i = 0; i < head.size; i++){
                    cout << "Enter the command (FORWARD[1],BACKWARD[2],LEFT[3],RIGHT[4]): ";
                    cin >> AuxInt; //enter directio
                    body[i].direction = AuxInt;

                    if (AuxInt <= 4 && AuxInt >= 1){
                        cout << "Enter Duration: ";
                        cin >> AuxInt; //enter duration
                        body[i].duration = AuxInt;
                    }
                    else {
                        i--;
                        cout << "Commands can only go from 1 - 4" << endl;
                    }
                }

                unsigned int parity = 0;
                unsigned char *temp; //temorary variable to calclate parity

                temp = new unsigned char[(head.size * 2) + 2]; //alocate memory for temp
                temp[0] = head.id;
                temp[1] = head.size;

                for (int i = 0; i < head.size; i++)
                    temp[2 + i] = body[i].direction;

                for (int i = 0; i < head.size; i++)
                    temp[2 + head.size + i] = body[i].duration;

                for (int i = 0; i < size + 2; i++){ //loop that calculates parity

                    while (temp[i] > 0){

                        if (((int)temp[i] & 1) == 1) parity++;

                        temp[i] >>= 1;
```

void SetInfo();

This function is responsible for not only setting the correct information for the packet, but also validating user inputted data, making one of the most, if not the most important function in the server. The following information is all encapsulated in a while loop, making it so that the user can continuously input data until a certain condition is met. That condition would be a packet ID of 0, which would be the sleep command, which transmits data to the client that will make it stop running. The other command, is an ID of 2, which is a request for Status. This will send 0 values, and no commands, but will however send a parity of 2 in its trailer. The last, and most important ID command, is 15, which is for Drive. This command will enter the if statement and ask a user to input a body size of the commands. The value is multiplied by 2, since a body contains 2 different bytes, 1 for Direction and one for duration. With this in mind, body is allocated with a doubled value.

```
                    }
                } // end of loop

                trailer = parity;

                cout << "Trailer: " << (int)trailer << endl;
                tr = false;
            }
            else{
                cout << "no more then 10 commands" << endl;
            }
        }
        else if (AuxInt == 5){ // if command is status
            head.size = 0;
            body = new Commands[0];
            trailer = 2; //parity is 2
            tr = false;
        }
        else if (AuxInt == 0){ // if command is sleep
            head.size = 0;
            body = new Commands[0];
            trailer = 0;
            tr = false;
        }
        else {
            cout << "incorrect command" << endl;
        }
    }//end of loop
}
```

The last operation is a calculation for the parity of the object, which is done via a while loop, iterating through the bits.

int Accept(SOCKET &, SOCKET &);

```
int PktDef::Accept(SOCKET &ServerSocket, SOCKET &ConnectionSocket) {
    if ((ConnectionSocket = accept(ServerSocket, NULL, NULL)) == SOCKET_ERROR) {
        closesocket(ServerSocket);
        WSACleanup();
        return 0;
    }
```

Accepts an incoming connection request, initiating the 3-steps handshake. If accepted, closes the server socket.

```
int PktDef::Send(SOCKET &ConnectionSocket){
    unsigned char *TxBuffer;

    TxBuffer = new unsigned char[(head.size*2) + 3];

    TxBuffer[0] = head.id;
    TxBuffer[1] = head.size;
    int b = 0;
    for (int i = 0; i < head.size * 2; i++){
        TxBuffer[2 + i] = body[b].direction;
        TxBuffer[3 + i] = body[b].duration;
        b++; i++;
    }

    TxBuffer[(head.size * 2) + 2] = trailer;

    send(ConnectionSocket, (char *)TxBuffer, (head.size * 2) + 3, 0);

    return (int)head.id;
}
```

int Send(SOCKET &);

With an established connection, this function sends the data that is stored in the Server packet, allocates its bytes acordingly (char = 1 byte). The size must be multiplied by 2, since even though there is only one command, that command comes with 2 separate pieces of data, the direction and duration. As such, the Buffer is properly allocated to accommodate this, and the number of bytes sent is also taking this into consideration.

```
void PktDef::ReceivePkt(SOCKET &ConnectionSocket){
    delete[] body;

    body = nullptr;

    char RxBuffer[128] = {};
    if (recv(ConnectionSocket, RxBuffer, sizeof(RxBuffer), 0) > 0) {
        head.id = RxBuffer[0];
        head.size = RxBuffer[1];
        body = new Commands[RxBuffer[1]];
        for (int i = 0; i < RxBuffer[1]; i++){
            body[i].direction = RxBuffer[2 + i];
        }

        trailer = RxBuffer[2 + RxBuffer[1]];
    }
}

void PktDef::PrintInfo(){

    cout << "Received PacketID: " << (int)head.id << endl;
    cout << "Receive CmdListSize: " << (int)head.size << endl;
    cout << "Received Status: ";
    for (int i = 0; i < head.size; i++)
        cout << (int)body[i].direction << " ";
    cout << endl << "Received Parity: " << (int)trailer << endl;
}
```

void ReceivePkt(SOCKET &ConnectionSocket);

This function receives the data from the client. This only occurs when a Status ID is sent from the server to the client. In order to store the data, the body variable is deallocated, in order to then be reallocated to the correct size of the data being sent back from the robot.

void PrintInfo();

Simply prints out the information delivered by the robot, after being received by the server.

```
void PktDef::CloseSocket(SOCKET &Socket){
    closesocket(Socket);
}
void PktDef::WinsockExit(){
    WSACleanup();
}
PktDef::~PktDef() {
    delete[] body;
    body = nullptr;
}
```

void CloseSocket(SOCKET &); void WinsockExit();~PktDef();

Standard close socket that initiates the 4-step terminating handshake, and freeing of winsock DLLs.
Lastly, deallocates the space in memory that is allocated when the data is created. A standard memory deallocating destructor.

III. The protocol

TCP/IP protocol guarantees a delivery of data to the client and from it. The application layer of our protocol consists of three components a header which contains Packet ID, a body that has a number of commands and list of commands and trailer that has a Parity value in it. Packet Id hold a number that represent what type of command we would like the robot to run (15 is Run, 5 is Status, 0 is Sleep). Number of commands holds the number of how many commands we would like to send to a robot. Packet Id and Command size are both encapsulated within a structure called Header. List of commands hold values that represent direction and duration, both of them are encapsulated within a structure called Commands. Parity is calculated by iterating through all the above numbers in a hexadecimal form and counting how many times the number one appears there. The value that is been calculated stored in a trailer. Transport layer provides channels that application uses in exchange data. The Internet Layer is a transmission layer that transfers all the packet through the network to the destination. And data link layer adds header and tail to the packet to prepare it for transmission.

IV. Summary

When the program was first created, the design and structure of the code would multiply the number of commands imputed by a user twice, and set up an array so that he or she could enter the direction and duration of the robot's movement and the proper memory would be allocated, taking this in consideration. At first, the program wouldn't run, and when it did, it would deliver the user a simple error message. After re-working the server a couple more times we realised that we should be sending the number of commands as we imputed them, and only store the multiplied value in a dynamic array size afterwards, not before. After fixing this, the communications between the robot and the server went smoothly, and worked as intended at all times.

However, another error arrised with the server, and this one would occur whenever the status was received from the client, it would simply crash, before restarting the listening loop. The reason for that was that we had our printing function run right after we received the status from the client. This was resolved after placing the print function after closing the sockets. This error happened because client only waits for the closing byte for a certain period of time and then terminates the connection, so by the time the print function was finished printing the client status the client would terminate the connection and break the loop.

REFERENCES

[1] M.Silva, "Data Communications Programming", Toronto, ON, Canada, https://cs.senecac.on.ca/~marcel.silva/BTN415/Slides/W2_2/

[2] M.Silva, "Data Communications Programming", Toronto, ON, Canada, https://cs.senecac.on.ca/~marcel.silva/BTN415/Slides/W5_2/

[3] M.Rose, "What is TCP/IP(Transmission Control Protocol/Internet Protocol", http://searchnetworking.techtarget.com/definition/TCP-IP