Delft University of Technology

ET4399 - Extra Project

Adhoc networking interface for ESP8266 Wifi module

July 23, 2017

Contents

1 Introduction

An ad hoc network is a collection of wireless mobile nodes dynamically forming a temporary network without the use of any existing network infrastructure or centralized administration. Due to the limited transmission range of wireless network interfaces, multiple network hops maybe needed for one node to exchange data with another across the network.

In areas in which there is little or no communication infrastructure or the existing infrastructure is expensive or inconvenient to use, wireless mobile users may still be able to communicate through the formation of an adhoc network. In such a network, each mobile node operates not only as a host but also as a router, forwarding packets for other mobile nodes in the network that may not be within direct wireless transmission range of each other. Each node participates in an ad hoc routing protocol that allows it to discover multi-hop paths through the network to any other node. Some examples of the possible uses of adhoc networking include students using laptop computers to participate in an interactive lecture, business associates sharing information during a meeting, soldiers relaying information for situational awareness on the battlefield, and emergency disaster relief personnel coordinating efforts after a hurricane or earthquake.

This document describes hardware and software guidelines required to implement an adhoc networking interface using ESP8266 wifi modules. The document describes in detail the protocol used to implement an adhoc network of nodes and design decisions taken. The document provides a description of all APIs that can be used by other applications who wish to use this framework.

2 Hardware overview

The project is aimed at specifically developing protocols to facilitate adhoc communication in ESP8266 wifi modules. The pin diagram of hardware used is shown in Figure 1.

Figure 1: ESP8266 WiFi chip

The ESP8266 is a low-cost wifi chip with full TCP/IP stack and an on board MCU (Micro Controller Unit). Its low cost has attracted lot of attention for potential IoT applications. Some of the key technical features of this chip are as follows.

- Supports 2.4GHz WiFi 802.11 b/g/n protocol
- Integrated TCP/IP protocol stack.
- An onboard temperature sensor
- A 10 bit ADC
- A low power 32-bit CPU running at 80MHz which can be used like an application processor
- GPIO pins.
- SPI and UART capabilities

2.1 ESP8266 Adhoc functionality

The default firmware in the wifi module does not support an adhoc networking interface . This highly restricts the capabilities of the chip to be used in a system of multiple nodes communicating with each other. This project involves design and development of protocol to enable a robust multihop adhoc communication interface for the Wifi module. Such an interface can be useful for a number of applications where data is exchanged among multiple wifi modules.

3 Getting started with ESP8266 wifi module

This section gives an overview of hardware connections required to hook up ESP8266 board. Although hardware connections are simple there are special needs required for ESP modules related to power. Some of important things to consider are listed below.

- The ESP8266 requires 3.3V power, do not power it with 5 volts.
- The ESP8266 needs to communicate via serial at 3.3V and does not have 5V tolerant inputs, so you need level conversion to communicate with a 5V microcontroller like most Arduinos use.

3.1 Hardware connections

In this section we describe hardware connection with a typical microcontroller such as an arduino. In this example we make use of an arduino UNO however same example can be extended to any standard microcontroller platform. As shown in figure 10 connect TX of arduino to RX of the ESP, RX of arduino to TX of ESP, connect both grounds together, connect VCC and CH_PD to 3.3V supply of the arduino.

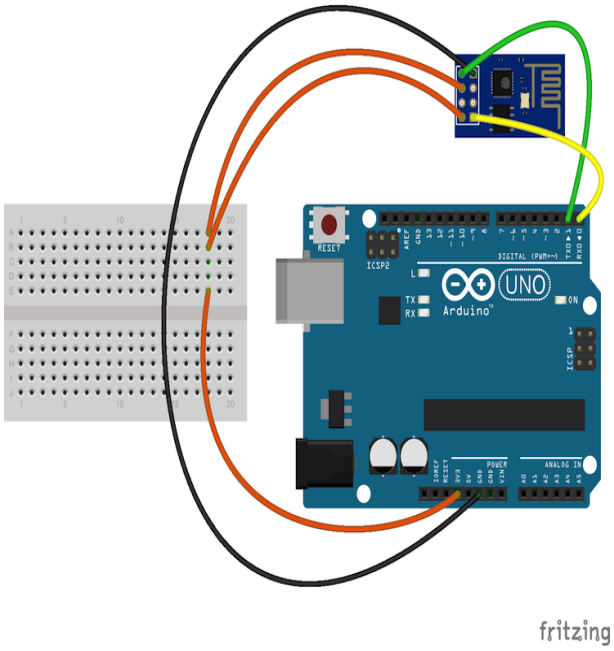

Figure 2: ESP - Arduino connection settings

3.2 ESP8266 Software

All ESP modules come with a default firmware that supports AT commands. These commands can be used to set up network, configure mode of operation and data transfer. However for developers who wish to implement their own custom protocols and functions there are many open source software development kits (SDKs) available for ESP modules. Some of the popular ones are listed below :

- NodeMCU - It is an eLua based firmware for the ESP8266 WiFi SOC from Espressif.
- Arduino - Its a C++ based library based on the popular Arduino. It lets writing skethes in Arduino style and also flashing via the same IDE.
- MicroPython - A port of the MicroPython to the ESP8266 platform.

In this project we make use of arduino libraries based on their ease of usage and immense support from the developer community. The library is open source and be found in Github here. The repository contains working sketches and detailed API documentation.

3.3 ESP8266 Flashing procedure

In this section we provide steps required to flash custom firmware to ESP8266 modules.

Hardware required :

- USB FTDI cable
- ESP8266
- Connecting Cables
- Breadboard

Connect the hardware as shown in figure 4 for flashing the board. It is import to connect GPIO0 to ground which forces the ESP into flashing mode. The chip has to be reset just before flashing the board this can be done either manually (which can be tricky) or by connecting RTS (Request To Send) pin on the FTDI to the Reset pin of the ESP module.

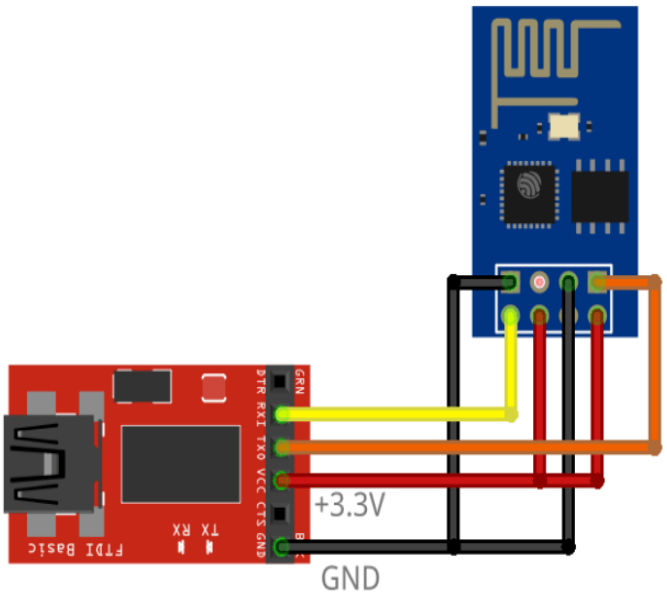

Figure 3: ESP connections for flashing

Complete step by step procedure for flashing custom firmware using Arduino IDE can be found at this webpage.

4 Packet format

In this section we describe the packet format used by the routing protocol. The protocol itself is described in the later section.

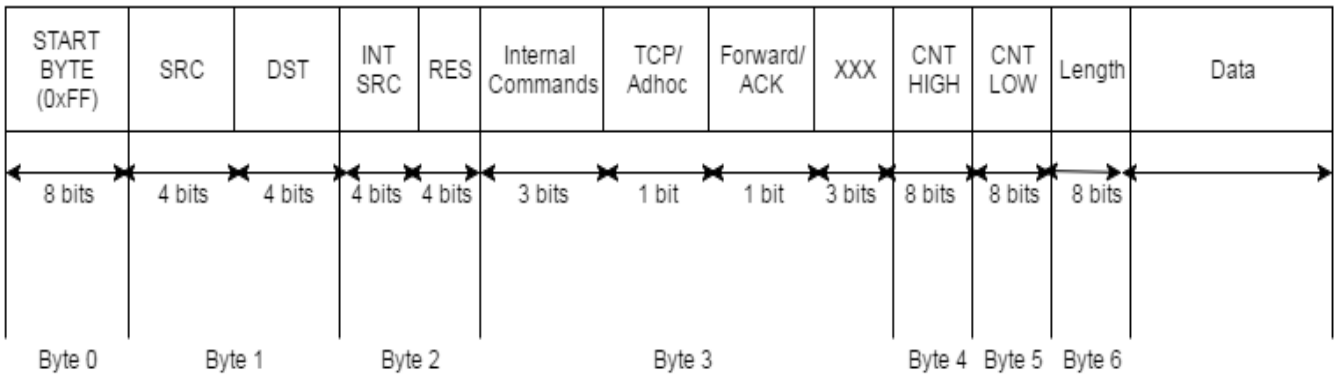

Figure 4: Packet format

Figure 4 represents the packet format used in the protocol for communication.
Contents of each field is explained below

- **START BYTE** : Its the first byte. Represents the start sequence of the packet. Its is denoted by 0xFF.
- **SRC** : A 4 bit field of the second byte represents the unique identifier of the source Ad-hoc node.
- **DST** : A 4 bit field of the second byte represents the unique identifier of the destination Ad-hoc node.
- **INT SRC** : A 4 bit field represents the ID of intermediate hop node in the network from which the packet reached the current node.
- **Internal Commands** : This 3 bit field is used for communication between a master MCU and ESP. It represents certain commands that are exchanged between MCU and ESP during initialization. Internal command should be set to '0' when communicating with TCP or Ad-hoc node. It should be set only when the robot communicates with its ESP8266 module.
- **TCP/Adhoc** : 1 bit information used to identify whether the packet is intended for TCP server or Ad-hoc network.
  Note: When communicating with TCP, TCP bit should be made high and data should be packed as per the length. Rest of the bytes such as SRC, DST, Internal commands, Forward/ACK are don't cares.
- **Forward/ACK** : 1 bit information used to identify if the packet is ACK packet from a destination node or a forwarding packet.
- **Length** : Represents the length of data in bytes present in the packet.
- **Data** : Represents data to be sent. This field can be up to 254 bytes in size.
- **CNT HIGH and LOW** : These fields represents the high and low values of the 16-bit packet sequence counter.
- **Checksum** : Represents checksum to make sure that the complete packet is received and to avoid receiving corrupt packets.
- **RES**: Empty 4 bit field reserved for future use.

4.1 Internal Commands

The internal commands are represented by bit 6 to bit 8 (3 bits) in 5th byte of the packet. These commands are used by Arduino to communicate with ESP8266. The commands are always sent from Arduino to ESP8266. The commands used are:

- INT_ID (0x00) : Represents that data contains ID
- INT_SSID_PWD (0x03) : Represents that data contains SSID and Password
- INT_MATRIX (0x02) : Represents that data contains its connection matrix

- INT_RSSI (0x05) : Represents that Arduino is requesting RSSI value from ESP8266. The data length can be zero since this is request command
- INT_IP (0x06) : Represents that data contains IP address of the server
- INT_DEMO (0x07) : Enable demo mode in ESP8266

Note: When internal commands are sent, the first byte in data does not contain COMMAND unlike external commands.

5 Protocol description

5.1 Network structure

The proposed framework supports two modes of communication. In TCP mode it can communicate with the TCP server to receive commands from it or send data to the server. In Adhoc mode, it can communicate with other nodes without any intervention from the server. This can be done by setting **TCP/Adhoc** bit in the packet. Setting this bit sends the data to the TCP server whereas clearing this bit sends the data to the Adhoc network. Similarly, this bit can serve as an indication if the packet originated from the server or from the Adhoc network.

The nodes can communicate with each other based on a network graph which is represented as a matrix. Consider an example of a simple network of four robots as show in figure 5.

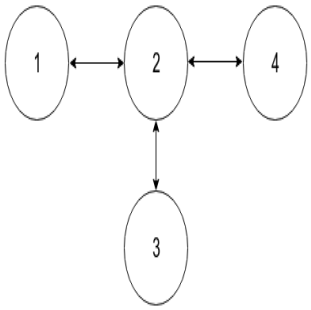

Figure 5: Sample Network graph for a set of three bots

In this topology, it can be observed that node **1** can communicate only with node **2**. Whereas node **2** can communicate with nodes **1**,**3** and **4**. This relation is represented in form a matrix as follows.

$$\begin{bmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix}$$

5.2 Initialization

During the initialization routine, the master MCU sends initialization commands to the Wifi module. These commands are needed by the Wifi module to connect to the specified access point and the IP address of the TCP server.

The data sent to the wifi module during initialization routine are:

- ID of the node.
- connection matrix containing information of 1 hop neighbours of the node.
- IP address of TCP server.
- SSID and password of the access point.

This can be seen as 4 blue blinks with a delay of one second on the Wifi module. After this, the Wifi module tries to connect to the specified access point which can be seen as pulsating blink on the Wifi module.

5.3 Algorithm

In this section we provide a detailed description of the algorithm used to create a multi hop mesh network of ESP8266 wifi modules. Consider a scenario where data has to be transmitted through multiple nodes as shown in figure 6

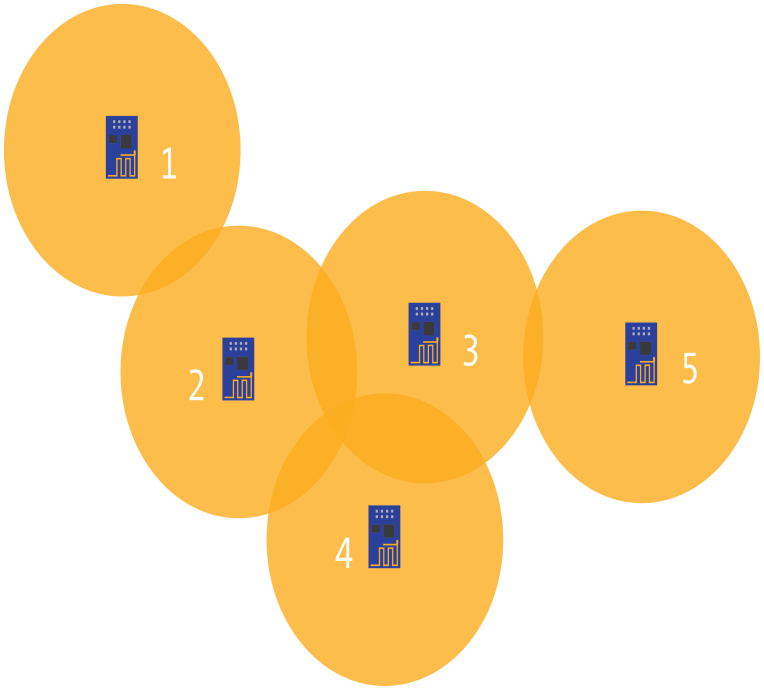

Figure 6: A mesh network of wifi modules

In figure 6 if node 1 needs to transmit data to node 5 it has do it through other nodes (2, 3 or 4) as node 5 is not in the transmission range of node 1. In order to facilitate communication we go for a multicast approach where all the nodes in the network subscribe to a specific multicast address. Each node in the network maintains a table henceforth called as packet transaction table containing the history of transmissions with a time stamp, packet sequence counter value, source, destination and whether acknowledgement has been received for a particular packet. Whenever a node has to transmit data it records these metrics in its local table and sends packet on a multicast address. All the nodes which are in the reception range will receive this packet. The receiving nodes first check their own table to prevent reception of duplicate packets. Similarly the destination node also stores the packet in its table to avoid reception of multiple duplicate packets. The packet is removed from the list when nodes receive an ACK for a packet that was stored in the table.

5.4 Packet transaction table

In order to combat issues due to arrival of duplicate packets and indefinite looping of packets within the network a data structure is maintained in all nodes to keep track packet transactions. Each entry in the packet transaction table contains following elements.

- Source node ID in the packet
- Destination node ID in the packet
- Time stamp information in milliseconds.
- A Boolean indicating if an acknowledgment was received for the packet.
- Sequence counter value from the packet.

Table 1: An illustration of entries in packet transaction table

Entry Number	Counter value	SRC	DST	Forward/ACK	Time
1	254	1	5	False	24367
2	354	2	4	True	25432
3					
.					
N					

Whenever a node has to transmit data to a multicast address it makes an entry in the packet transaction table. The transmitting node sets the acknowledgement field in the packet transaction table to false indicating that ACK has not been received for that specific packet. The field is set to true once the receiving node sends an acknowledgment to the transmitting node by setting the ACK bit in the packet. The entry in packet transaction table is not immediately removed. The entry is kept alive for a small amount of time (around 100ms) to avoid reception of duplicate packets. Whenever a packet is received the packet transaction table is searched based on the sequence counter and source-destination pair. If the ACK was already received for the packet then the duplicate packet is ignored. Similarly intermediate hop nodes also make an entry in the packet transaction table to keep track of traffic going through them. Each entry in the table is flushed after a timeout of 5000 milliseconds after which a retransmission is initiated.

An illustration of packet transaction table is as shown as in table 1. The first entry in the table shows details of a packet originating from node 1 to node 5 for which the acknowledgement has not yet been received. The entry is kept alive for a small amount of time waiting for the ACK to be received, if no ACK is received from the destination this entry is deleted and retransmission occurs. Similarly second entry in the table shows details of a packet for which ACK has been received this entry is also kept alive for a small instance of time to avoid reception of duplicate packets from the neighbouring nodes.

5.5 Packet reception logic

The flowchart shown in figure 7 represents the steps each node goes through when it receives a packet on the wifi interface.

The reception logic is explained briefly as follows :

- A node upon reception of a packet checks the validity of the packet by checking the if the packet begins with a valid start sequence, 0xFF in our case. The packet is ignored if it does not contain a valid start byte.
- The node then checks the connection matrix to infer if it is allowed to receive from the source node from where the packet originated. If it is not allowed then the packet is simply ignored.
- The node then checks the destination field in the packet to check if the packet was directed towards itself.
- If the destination ID field of the packet matches with nodes ID then the node consults its local packet transaction table to see if it already received this packet. This is done to avoid reception of duplicate packets from different paths. The packet is ignored if an entry is found in the packet transaction table.
- If the node is the destination node and packet is received for the forst time, and entry is made in packet transaction table and acknowledgment is sent to source by reversing source and destination nodes in the packet and transmitting on multicast address.
- In case the node is not the destination node then the node changes the intermediate source field to match its node ID , makes an entry in the packet transaction table and transmits again on multicast address.

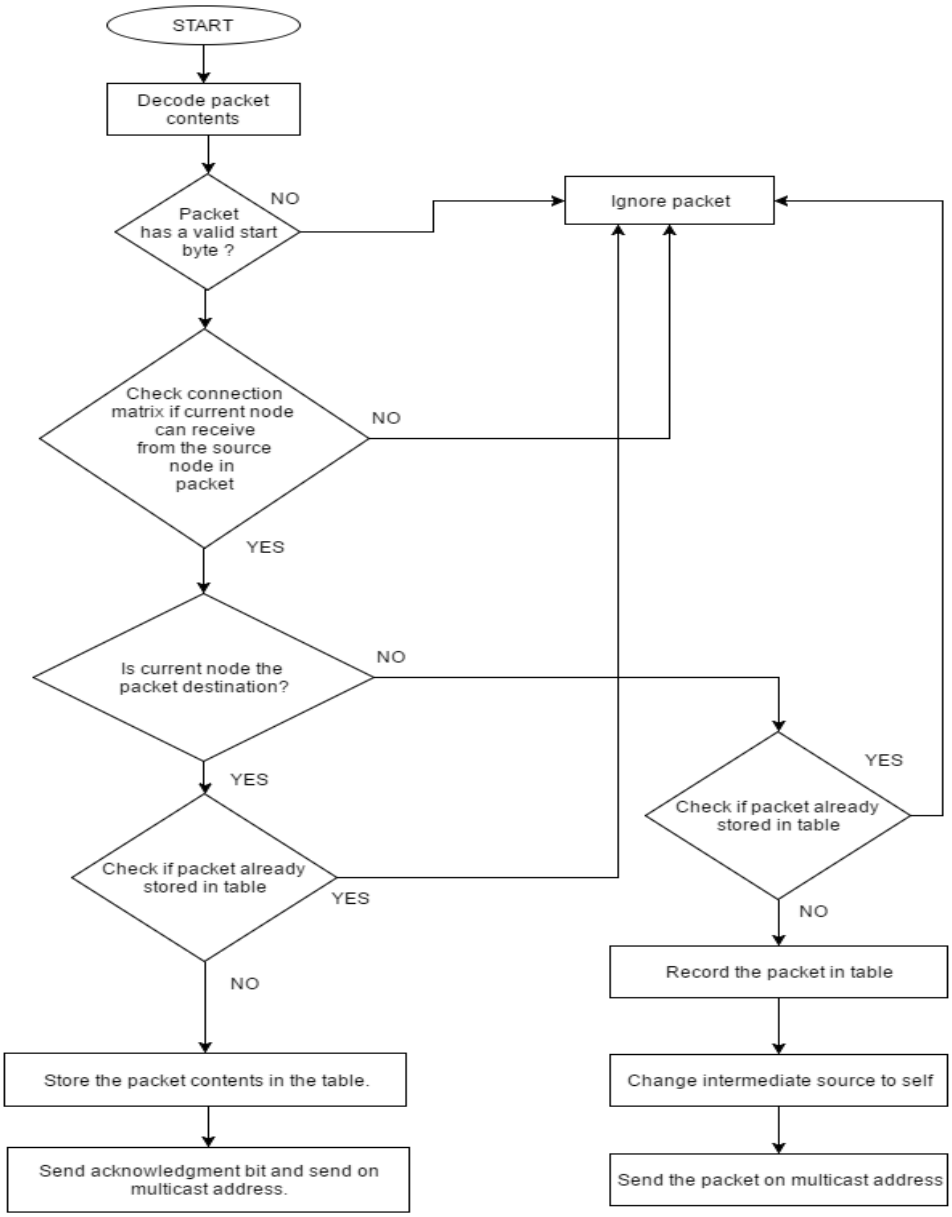

Figure 7: Flow chart describing the reception logic at each node

5.6 Packet transmission logic

Transmission logic is relatively simple, the flowchart describing the procedure is as shown in figure 8

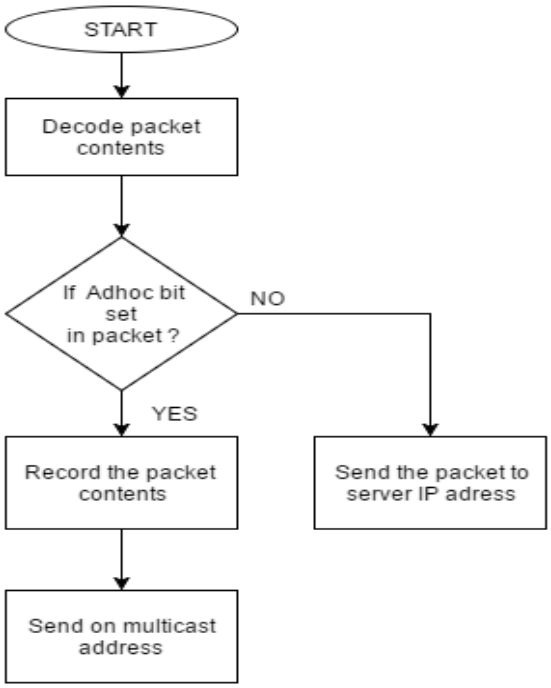

Figure 8: Flow chart describing the transmission logic at each node

The transmission logic is explained briefly as follows :

- The node first decodes the packet to identify if it was intended towards another node or the TCP server. This is done by checking the TCP/Adhoc bit in the packet.
- If the packet is intended for the TCP server then the packet is sent to the server IP address.
- If the packet is intended towards the adhoc network, an entry is created in the Packet transaction table and then its sent to the multicast address.

6 Swarm robotics framework

The protocol developed was used to create a network of robots that could communicate among themselves to perform various tasks. In such a framework the wifi module running the protocol was connected to an arduino micrcontroller which acted as brains of the robot. A number of sensors to measure direction and obstacles were placed on the robot. The robot could controlled individually by means of a central TCP server. Robots could also exchange sensor data and movement commands between each other by means of the protocol described earlier. The finished bot is as shown in figure 10.
Some of the experiments conducted using the framework are as follows :

- Using robots to analyze the variation of RSSI values in an environment.
- Implementing a simple platooning approach based on RSSI values where a slave bot follows a leader bot.
- Distance synchronization experiments in which slave bots maintain same distance from the AP as the master bot.

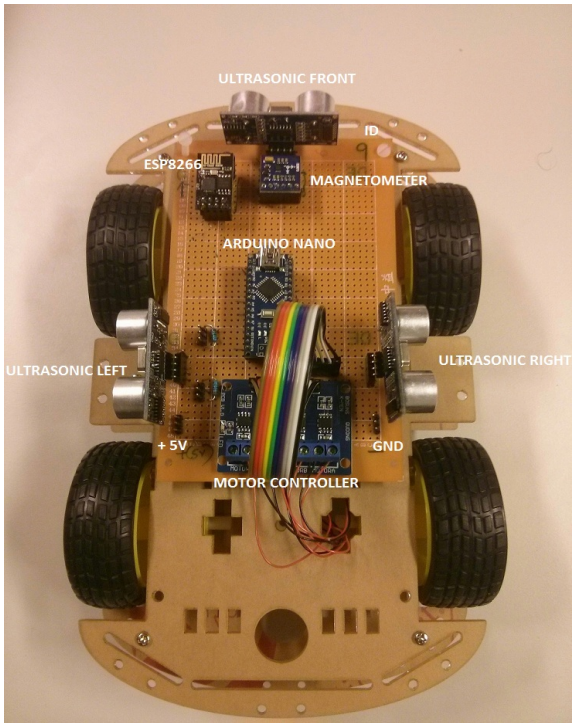

Figure 9: Robot hardware overview

7 TCP Server

The Server program is built using linux socket APIs. The server starts listening for a fixed number of nodes on a specific IP address, it keeps listening until all the clients connect to the server. This procedure can be illustrated as shown in figure The server maintains a different sockets for each connected client and maintains a table which maps client is to the socket id. This allows the server to choose appropriate socket id for each of the available clients.

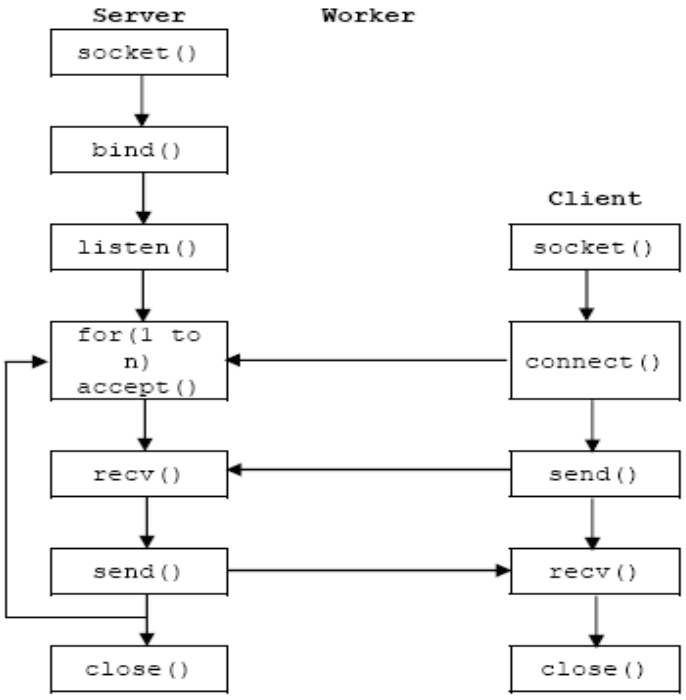

Figure 10: TCP client server communication

The TCP server provides a way to send commands to each node individually by means of TCP sockets. The server is able to establish one-to-one communication with any node in order to send commands or retrieve data from it. The server IP address should be configured in the Arduino code. This information is sent to the Wifi module during the initialization routine.

In order to execute the Server type the following command.

```
./tcpserver <Number-of-bots>
```

The server should be turned ON and should be connected to the AP before any of the clients (nodes) are turned ON. The server waits for the robots to get connected. It displays the ID of the connected bots as soon as it gets connected.

```
*************** Server control program (Adhoc networking course)  ************
* Binding done
* Waiting for bots to connect
  - Accepted connection
  - Bot with ID : <2> Connected
```

In order to send command to a specific connected robot type the ID of the bot you wish to communicate with and select the appropriate command from the menu.

```
Enter Bot ID to send the packet
Enter the command(1-12) to the bot-2 :
   1. Move forward
   2. Move forward for time in seconds
   3. Move reverse
   4. Move reverse for time in seconds
   5. Move left time
   6. Move right time
   7. Stop the bot
   8. Get obstacle distance left
   9. Get obstacle distance right
  10. Get obstacle distance front
  11. Get RSSI value
  12. Execute commands from file (cmd_file.txt)
```

NOTE : Guide to turning Android phone into wifi hotspot can be found here. Likewise guide to turning laptop into a hotspot can be found here, the server source code and working examples can be found at the following Github link

8 API listing

This section gives an overview of the APIs that can be used on the TCP server and ESP library. The APIs that can be used on the TCP server are summarized in table 3 and the APIs used for implementation of the protocol are listed in table 2.

Table 2: ESP8266 library APIs

Name	USAGE	DESCRIPTION
Transmission and reception APIs		
adhoc_recv_filter	void adhoc_recv_filter(char packet[]);	This API is used to filter the incoming packet based on network topology and packet transaction table entries. It takes raw packets which are received from the wireless interface as input.
adhoc_send_filter	void adhoc_send_filter(char packet[], int numBytes);	This API is used for transmitting a packet over the adhoc network or to the TCP server. It takes two arguments, packet as char array and the size of the packet as integer.
Packet tranaction table APIs		
store_record	void store_record(char packet[]);	This API is used to store a packet in the packet transaction table before it is sent over the wireless interface.
is_record_present	int is_record_present(unsigned count);	This API is used to check if a packet having counter value specified by count is present in the packet transaction table.
set_ack_table	void set_ack_table(unsigned count);	This API is used to set the ACK bit in the transaction table for a packet with counter value specified by count indicating that an ACK was received
ack_in_table	int ack_in_table(unsigned count);	This API is used to check if ACK bit is set in the transaction table for a packet with counter value specified by count value to check reception of duplicate packets
fwd_in_table	int fwd_in_table(unsigned count);	This API is used to check if forward bit is set in the transaction table for a packet with counter value specified by count value to check reception of duplicate packets
get_table_index	int get_table_index(int count);	This API is used to get the index of an entry in packet transaction table given counter value of a packet.
Packet data extraction —API		
get_src	int get_src(char packet[]);	This API is used to extract the source node ID from a packet
get_dst	int get_dst(char packet[]);	This API is used to extract the destination node ID from a packet
get_counter	unsigned int get_counter(char packet[]);	This API is used to extract the counter value from a packet
get_inter	int get_inter(char packet[]);	This API is used to extract intermediate counter value from the packet.
is_ACK	int is_ACK(char packet[]);	This API is used to check if ACk is set in a packet
is_TCP	int is_TCP(char packet[]);	This API is used to determine if packet is intended for ahoc network or the TCP server
can_rcv	int can_rcv(char src);	This API is used to determine if packet can be received from a source specified by src

Table 3: Server APIs

NAME	USAGE	DESCRIPTION
send_forward_time	void send_forward_time(int src, int dst, int time)	This API is used to move the bot with ID **dst** forward for **time** seconds. Time can be any positive number between (0-254). Specifying zero in time field moves the bot in forward direction indefinitely.
send_reverse_time	void send_reverse_time(int src, int dst, int time)	This API is used to move the bot with ID **dst** reverse for, **time** seconds. Time can be any positive number between (0-254). Specifying zero in time field moves the bot in reverse direction indefinitely.
send_rotate_left	void send_rotate_left(int src, int dst, int time)	This API is used to rotate the bot with ID **dst** left for **time** seconds.
send_rotate_right	void send_rotate_right(int src, int dst, int time)	This API is used to rotate the bot with ID **dst**right for **time** seconds.
stop_bot	void stop_bot(int src, int dst)	This API is used to stop the bot with ID **dst**.
get_obstacle_data	int get_obstacle_data(int src, int dst, int sensor_num)	This API is used to obtain the reading from obstacle sensor of the bot with ID **dst**. **sensor_num** is used to select the sensor on the bot. **sensor_num** can be ULTRASONIC_LEFT for left obstacle sensor. ULTRASONIC_RIGHT for right obstacle sensor and ULTRASONIC_FRONT for front obstacle sensor.
get_RSSI	long get_RSSI(int src, int dst)	This API is used to get received signal strength of the bot with ID **dst**.

The complete source code can be found in the appendix section.

9 Future work

In current implementation the total number of nodes in the topology are already known to each node within the topology. Thus dynamically adding a new node to the system may not work. Thus in future implementation a dynamic protocol where nodes can be added to the topology at any instant of time would be very useful. This would require a continuous neighbour discovery process where each node would constantly update its 1 hop neighbours. There has been significant research in this area and many protocols already exist, such protocols can be adapted for ESP8266 for a dynamic operation.

10 Conclusion

ESP8266 is an inexpensive yet versatile piece of hardware which can be used for various wifi based applications. The default firmware does not allow multihop adhoc networking interface which restricts its capabilities in a number of applications. A multihop adhoc networking protocol was implemented in this project using which a network of multiple nodes can communicate with each other. The protocol was used in a swarm robotics framework in which a system of multiple robots had to exchange data among themselves to accomplish a set of tasks. Although the protocol performed well it was noticed that it was significantly slower than anticipated. This limitation might be because of serial link between the microcontroller unit and the ESP module. This acts like a bottleneck and limits the rate at which data can be exchanged between the ESP module and the microcontroller. The protocol is implemented using the arduino SDK which processes the commands sent to the wifi module sequentially this also limits speed to an extent as sending commands rapidly overwrites the previously sent commands. In future a different SDK that supports multithreading can be used to speed up execution significantly.

References

[1] https://www.arduino.cc/en/Main/ArduinoBoardNano

[2] https://espressif.com/

[3] http://www.pridopia.co.uk/pi-doc/ESP8266ATCommandsSet.pdf

[4] http://iot-playground.com/blog/2-uncategorised/17-esp8266-wifi-module-and-5v-arduino-connection

[5] https://github.com/esp8266/Arduino

[6] https://docs.micropython.org/en/latest/esp8266/esp8266/tutorial/intro.html

A Source code

The source code for protocol implementation with ESP8266 arduino library is as follows.

```
/*
  ESP8266Adhoc.h - Library for creating an Adhoc network
  of ESP8266 nodes.
*/
#ifndef ESP8266Adhoc_h
#define ESP8266Adhoc_h
#include <ESP8266WiFi.h>
#include <WiFiUdp.h>
#include "Arduino.h"
#include "matrix.h"


#define ADHOC_UDP_PORT 8080
#define TCP_PORT 8089



/* Packet format details */


#define START_BYTE 0xFF

#define PACKET_START_BYTE_LOC 0
#define PACKET_SRC_DST_LOC 1
#define PACKET_INTERMEDIATE_SRC_LOC 2
#define PACKET_INTERNAL_CMD_LOC 3
#define PACKET_COUNTER_HIGH_LOC 4
#define PACKET_COUNTER_LOW_LOC 5
#define PACKET_DATA_LENGTH_LOC 6
#define PACKET_DATA_LOC 7

/* Process the internal commands between
 * Arduino and ESP.
 * 3-bits
 *
 * |CMD|Desription|
 * |000|No internal command
 * |001|Set the ID of the bot
 * |010|Set ssid and password of wifi
 * |011|get init_matrix for bot ID
 * |100|Get RSSI
 * |101|Server IP set
 * |110|Reserved
 * |111|Reserved
 *
 */

/* List of internal commands */

#define INTERNAL_SET_ID 1
#define INTERNAL_SET_SSID_PWD 2
#define INTERNAL_SET_MATRIX 3
#define INTERNAL_GET_RSSI 4
#define INTERNAL_SET_SERVER_IP 5
```

```
#define INTERNAL_NON_DEMO_MODE 6


/* Number of records to maintain */
#define NUM_RECORDS 512

class ESP8266Adhoc
{
    public:

        /* ID of the bot */
        char ID_SELF;
        /* Wifi and pssword */
        char* ssid ;
        char* password;
        char *con_matrix;
        /* Connection matrix row for the ID */
        int wifi_status = 0;


        /* To Enable or disable demo mode
         * (Disabled by default)
         */

        int DEMO_MODE = 0;


        /* TCP server ip address */
        char server_ip[16];
        /* Global variable used to maintain current record count */
        unsigned int record_count = 0;

        /* To keep track of received bytes from UDP */

        int recvBytes;

        /* Table used to record the movement of packets in network */

        struct table {
            unsigned int counter;
            char src;
            char dst;
            unsigned long arrival_time;
            char forward:1;
            char ack:1;
            char tcp:1;
            char valid:1;
        };

        struct table record[NUM_RECORDS];

        /* Constructor */

//        ESP8266Adhoc(int port);


        /* Start Adhoc mode on the specified port */
```

```
    void start();

    int parseBytes();


    /* For debugging purposes */
    void print_packet(char packet[],int numBytes);
    void Debug(String str);
    void print_record(unsigned int count);
    int is_valid(char packet[]);

    /* Database storage */
    void store_record(char packet[]);
    int is_record_present(unsigned count);
    void set_ack_table(unsigned count);
    int ack_in_table(unsigned count);
    int ack_to_TCP(unsigned count);
    int fwd_in_table(unsigned count);
    int get_table_index(int count);
    int get_src(char packet[]);
    int get_dst(char packet[]);
    unsigned int get_counter(char packet[]);
    int get_inter(char packet[]);
    int is_ACK(char packet[]);
    int is_TCP(char packet[]);
    int can_rcv(char src);

    /* For Checksum calculation */

    int calc_checksum(char packet[]);

    /* To handle internal commands */


    int is_internal_cmd(char packet[]);
    void process_internal_cmd(char packet[]);
    char *get_pwd(char packet[]);
    char *get_data(char packet[]);
    char *get_ssid(char packet[]);
    void set_con_matrix(char packet[]);
    void set_server_ip(char packet[]);
    char get_self_ID(char packet[]);
    int connect_wifi();
    long get_RSSI();
    void set_id(char data);


    /* Adhoc reception logic */
    void adhoc_recv_filter(char packet[]);

    /* Adhoc send logic */
    void adhoc_send_filter(char packet[],int numBytes);
    void extract_ssid_pwd(char packet[],int recvBytes);



private:
```

```
        /* port used by the adhoc network nodes */
        int adhoc_port;
        /* Function to extract ssid and password from packet */
        int get_index(char data[],int size,char item);


};

#endif
```

The CPP source code of all functions.

```
#include "Arduino.h"
#include "ESP8266Adhoc.h"
//WiFiUDP Udp;
//IPAddress ipMulti (239,0,0,57);


/* Comment to disable debugging here */
#define __DEBUG__ 1




void ESP8266Adhoc::Debug(String str) {
#ifdef __DEBUG__
  Serial.println(str);
#endif

}

/*
 * Function used to check if the packet
 * is valid or not
 * returns 1 if packet begins with start byte
 * else returns zero
 */


int ESP8266Adhoc::is_valid(char packet[]) {

#ifdef __DEBUG__
  Serial.print("Startbyte : ");
  Serial.println(packet[PACKET_START_BYTE_LOC],HEX);
#endif
  if(packet[PACKET_START_BYTE_LOC] == START_BYTE)
    return 1;
  else
    return 0;

}


#if 0
ESP8266Adhoc::ESP8266Adhoc(int port) {

  //Udp.beginMulticast(WiFi.localIP(),  ipMulti, port);
```

```
  adhoc_port = port;

}

void ESP8266Adhoc::start() {


  Udp.beginMulticast(WiFi.localIP(),  ipMulti, adhoc_port);

#ifdef __DEBUG__
  Serial.print(ID_SELF,DEC);
  Serial.print(" Started listen on port ");
  Serial.println(adhoc_port);
#endif

}


int ESP8266Adhoc::parseBytes() {
  return Udp.parsePacket();

}
#endif

/*
 * Following function is used for debugging purpose
 * it prints the contents of the packet
 */


void ESP8266Adhoc::print_packet(char packet[],int numBytes) {

  Serial.println("");
  Serial.println("*****************************************");

  if(packet[PACKET_START_BYTE_LOC] != START_BYTE) {
    Serial.println("Unknown packet format found");
    return;
  }

  Serial.print("Data length in packet : ");
  Serial.print(packet[PACKET_DATA_LENGTH_LOC],DEC);
  Serial.print(" ");
  Serial.println("bytes");

  Serial.println("|Packet contents|");
  for(int i = 0; i < numBytes ; i++) {
    Serial.print(packet[i],HEX);
    Serial.print(" ");
  }
  Serial.println("");
  Serial.print("Source : ");
  Serial.println(ESP8266Adhoc::get_src(packet));
  Serial.print("Destination : ");
  Serial.println(ESP8266Adhoc::get_dst(packet));
  Serial.print("Intermediate : ");
  Serial.println(ESP8266Adhoc::get_inter(packet));
```

```
  Serial.print("Counter : ");
  Serial.println(ESP8266Adhoc::get_counter(packet));
  Serial.print("Current record count : ");
  Serial.println(ESP8266Adhoc::record_count);
  Serial.print("Packet type : ");
  if(ESP8266Adhoc::is_ACK(packet))
    Serial.println("Acknowledgement packet");
  else
    Serial.println("Forwarding packet");

  Serial.print("Length : ");
  Serial.println(packet[PACKET_DATA_LENGTH_LOC],DEC);
  Serial.print("Data : ");
  for(int i = 0; i < packet[PACKET_DATA_LENGTH_LOC]; i++) {
    Serial.print(packet[PACKET_DATA_LOC + i],HEX);
  }
  Serial.println(" ");

#if 0
  Serial.print("Checksum : ");
  Serial.println(packet[7 + packet[6]],HEX);
  Serial.println("");
#endif
  Serial.println("*****************************************");
}




void ESP8266Adhoc::print_record(unsigned int count) {

  Serial.println("Contents of record " );
  Serial.print("Counter ");
  Serial.println(record[count].counter);
  Serial.print("src ");
  Serial.println(record[count].src,DEC);
  Serial.print("dest ");
  Serial.println(record[count].dst,DEC);
  Serial.print("Arrival time ");
  Serial.println(record[count].arrival_time,DEC);
  Serial.print("Forward ");
  Serial.println(record[count].forward,DEC);
  Serial.print("Ack ");
  Serial.println(record[count].ack,DEC);

}

int ESP8266Adhoc::calc_checksum(char packet[]) {

  char checksum = packet[1];
  int length = (int) packet[6];
  for(int i=2; i< (6 + length) ; i++ )
  {
    checksum=checksum ^ packet[i];
  }

#ifdef __DEBUG__
```

```
  Serial.print("Calculated checksum : ");
  Serial.println(checksum,HEX);
#endif
}


void ESP8266Adhoc::store_record(char packet[]) {

  record[record_count].counter = ESP8266Adhoc::get_counter(packet);
  record[record_count].src = ESP8266Adhoc::get_src(packet);
  record[record_count].dst = ESP8266Adhoc::get_dst(packet);
  record[record_count].arrival_time = millis();
  if (ESP8266Adhoc::is_ACK(packet)) {
    record[record_count].ack = 1;
  }
  else {
    record[record_count].forward = 1;
  }
  if (ESP8266Adhoc::is_TCP(packet)) {
    record[record_count].tcp = 1;
  }
  else {
    record[record_count].tcp = 0;
  }
  ESP8266Adhoc::record_count++;

}



int  ESP8266Adhoc::get_table_index(int count) {

  for(int i = 0 ; i < NUM_RECORDS; i++) {
    if(record[i].counter == count) {
      return i;
    }
  }
  return -1;
}


int ESP8266Adhoc::can_rcv(char source) {

#if 1
  if(con_matrix[source - 1] == 1) {
    return 1;
  }
  else {
    return 0;
  }
#endif

#if 0
  if(connection_matrix[ID_SELF - 1][source - 1] == 1) {
    return 1;
  }
  else {
```

```
    return 0;
  }

#endif

}
/* Checks if ACK is set in the table for
 * the count value
 */

int ESP8266Adhoc::fwd_in_table(unsigned count) {

  if(record[count].forward == 1) {
#ifdef __DEBUG__
    Serial.println("");
    Serial.println("In is_fwd_in_table()");
    Serial.print("forward set in table for counter : ");
    Serial.println(count);
#endif
    return 1;
  }
  else {
#ifdef __DEBUG__
    Serial.println("");
    Serial.println("In is_fwd_in_table()");
    Serial.print("forward not set in table for counter : ");
    Serial.println(count);
#   endif
    return 0;
  }

}

int ESP8266Adhoc::ack_in_table(unsigned count) {

  if(record[count].ack == 1) {
#ifdef __DEBUG__
    Serial.println("");
    Serial.println("In is_ack_in_table()");
    Serial.print("ACK set in table for counter : ");
    Serial.println(count);
#endif
    return 1;
  }
  else {
#ifdef __DEBUG__
    Serial.println("");
    Serial.println("In is_ack_in_table()");
    Serial.print("ACK not set in table for counter : ");
    Serial.println(count);
#endif
    return 0;
  }

}
int ESP8266Adhoc::ack_to_TCP(unsigned count) {
```

```
  if(record[count].tcp == 1) {
#ifdef __DEBUG__
    Serial.println("");
    Serial.println("In ack_to_TCP()");
    Serial.print("TCP set in table for counter:");
    Serial.println(count);
#endif
    return 1;
  }
  else {
#ifdef __DEBUG__
    Serial.println("");
    Serial.println("In ack_to_TCP()");
    Serial.print("TCP not set in table for counter:");
    Serial.println(count);
#endif
    return 0;
  }

}

/* Set the ACK in the table */

void ESP8266Adhoc::set_ack_table(unsigned count) {

  record[count].ack == 1;

}

int ESP8266Adhoc::is_record_present(unsigned count) {

  int i = 0;

  Serial.print("is_record_present called with Arg : ");
  Serial.println(count);
  for(i = 0 ; i < NUM_RECORDS; i++) {

#if 0
    Serial.print("Record [");
    Serial.print(i);
    Serial.print("].counter = ");
    Serial.println(record[i].counter);
#endif
    if(record[i].counter == count) {
#if 1
      Serial.println("HIT");
      Serial.print("Record [");
      Serial.print(i);
      Serial.print("].counter = ");
      Serial.println(record[i].counter);
      Serial.println("Count = ");
      Serial.println(count);
#endif
      return 1;
    }

  }
```

```
  return 0;
}



int ESP8266Adhoc::is_TCP(char packet[]) {

  /* Changed during debugging */
  //return (packet[3] & (1 << 4));
  return (packet[PACKET_INTERNAL_CMD_LOC] & (1 << 4));

}



int ESP8266Adhoc::get_src(char packet[]) {

  return (packet[PACKET_SRC_DST_LOC]& 0xf0) >> 4;
}
unsigned int ESP8266Adhoc::get_counter(char packet[]) {

  return (packet[PACKET_COUNTER_HIGH_LOC] << 8)| (packet[PACKET_COUNTER_LOW_LOC]);
}

int ESP8266Adhoc::get_dst(char packet[]) {

  return (packet[PACKET_SRC_DST_LOC]& 0x0f);
}


int ESP8266Adhoc::get_inter(char packet[]) {

  return ((packet[PACKET_INTERMEDIATE_SRC_LOC]& 0xf0) >> 4);

}

int ESP8266Adhoc::is_ACK(char packet[]) {

  return (packet[PACKET_INTERNAL_CMD_LOC] & 0x08 ) >> 3;

}


void ESP8266Adhoc::set_id(char data){

#ifdef __DEBUG__
  Serial.print("Current SELF ID : ");
  Serial.println(ID_SELF,DEC);
#endif

  ID_SELF = data;
#ifdef __DEBUG__
  Serial.print("SELF ID changed to : ");
  Serial.println(ID_SELF,DEC);
#endif
```

```
}

long ESP8266Adhoc::get_RSSI() {
  return WiFi.RSSI();
}


int ESP8266Adhoc::connect_wifi() {

  WiFi.begin(ESP8266Adhoc::ssid, ESP8266Adhoc::password);
  Debug("Connecting to Wifi");
  while (WiFi.status() != WL_CONNECTED) {
    delay(500);
    Serial.print(".");
  }
  ESP8266Adhoc::wifi_status = 1;
#ifdef __DEBUG__
  Debug("WiFi connected");
  Debug("IP address: ");
  Serial.println(WiFi.localIP());
#endif

}

char ESP8266Adhoc::get_self_ID(char packet[]) {

  int i;
  char len = packet[PACKET_DATA_LENGTH_LOC];
  char id;
  char *data = (char *)malloc(len * sizeof(char));
  //memcpy(data,packet+7,len);
  for(i = 0 ; i < len ; i++) {
    data[i] = packet[PACKET_DATA_LOC + i];
  }
  id = data[0];
  free(data);
  return id;

}

char *ESP8266Adhoc::get_ssid(char packet[]) {

  int i;
  char len = packet[6];

  char *data = (char *)malloc(len * sizeof(char));

  //memcpy(data,packet+7,len);
  for(i = 0 ; i < len ; i++) {
    data[i] = packet[7 + i];
  }
#ifdef __DEBUG__
  Serial.println("Wifi SSID changed ");
#endif
  return data;

}
```

```
char *ESP8266Adhoc::get_data(char packet[]) {

  int i;
  char len = packet[PACKET_DATA_LENGTH_LOC];
  char *data = (char *)malloc(len * sizeof(char));
  for(i = 0 ; i < len ; i++) {
    data[i] = packet[PACKET_DATA_LOC + i];
  }
  return data;
}
char *ESP8266Adhoc::get_pwd(char packet[]) {

  int i;
  char len = packet[6];
  char *data = (char *)malloc(len * sizeof(char));
  for(i = 0 ; i < len ; i++) {
    data[i] = packet[7 + i];
  }
#ifdef __DEBUG__
  Serial.println("Wifi password changed ");
#endif
  return data;
}

void ESP8266Adhoc::set_con_matrix(char packet[]) {

  int i;
  char len = packet[PACKET_DATA_LENGTH_LOC];
  char *data = (char *)malloc(len * sizeof(char));
  for(i = 0 ; i < len ; i++) {
    data[i] = packet[PACKET_DATA_LOC + i];
  }
  ESP8266Adhoc::con_matrix = data;
#ifdef __DEBUG__
  Serial.println("Connection matrix set");
  for(i = 0 ; i < len ; i++) {
    Serial.print(con_matrix[i],DEC);

  }
  Serial.println("");
#endif
}

void ESP8266Adhoc::set_server_ip(char packet[]) {

  int i;
  char len = packet[PACKET_DATA_LENGTH_LOC];
  unsigned char *data = (unsigned char *)malloc(len * sizeof(char));

  for(i = 0 ; i < len ; i++) {
    data[i] = packet[PACKET_DATA_LOC + i];
  }

  sprintf(ESP8266Adhoc::server_ip,"%d.%d.%d.%d",data[0],data[1],data[2],data[3]);
#ifdef __DEBUG__
  Serial.println("Server IP address set");
  Serial.println(ESP8266Adhoc::server_ip);
```

```
  Serial.println("");
#endif
}



int ESP8266Adhoc::is_internal_cmd(char packet[]) {

  char value = packet[PACKET_INTERNAL_CMD_LOC];
  char cmd = (value & 0xE0 ) >> 5;
#ifdef __DEBUG__
  Serial.print("Received internal Command : ");
  Serial.println(cmd,DEC);
#endif
  return cmd;

}



int ESP8266Adhoc::get_index(char data[],int size,char item) {

  for(int i = 0; i < size ; i++) {
#ifdef __DEBUG__
    Serial.println("Data : ");
    Serial.println(data[i],HEX);
#endif
    if(data[i] == item) {
#ifdef __DEBUG__
      Serial.println("Index : ");
      Serial.println(i,DEC);
#endif
      return i;
    }
  }

  return -1;

}


void ESP8266Adhoc::extract_ssid_pwd(char packet[],int size) {

  int delim_index = -1;
  int data_length = (int)packet[PACKET_DATA_LENGTH_LOC];

  char *data = get_data(packet);

#ifdef __DEBUG__
  Serial.println("Data obtained :");
  for(int j =0; j < data_length;j++) {
    Serial.print(data[j],HEX);
    Serial.print(" ");
  }
#endif
```

```
  delim_index = get_index(data,data_length,0xA9);
#ifdef __DEBUG__
  Serial.print("Delimiter index : " + delim_index);
  Serial.println(delim_index,DEC);
#endif

  char *wifi_ssid = (char *)malloc((delim_index + 1) * sizeof(char));
  //char *wifi_ssid = (char *)malloc((delim_index ) * sizeof(char));
  memset(wifi_ssid,'\0',(delim_index + 1) * sizeof(char));
  //memset(wifi_ssid,'\0',(delim_index) * sizeof(char));
  /*
     for(int i =0 ; i < delim_index;i++) {
     wifi_ssid[i] = data[i];
     }
   */
  memcpy(wifi_ssid,data,delim_index);
#ifdef __DEBUG__
  Serial.print("New SSID : ");
  Serial.println(wifi_ssid);
#endif

  char *wifi_pwd = (char *)malloc(((data_length - (delim_index))) * sizeof(char));
  memset(wifi_pwd,'\0',((data_length - (delim_index ))) * sizeof(char));
  //char *wifi_pwd = (char *)malloc(((data_length - (delim_index))) * sizeof(char));
  //memset(wifi_pwd,'\0',((data_length - (delim_index ))) * sizeof(char));
  for(int i = 0 ; i < ((data_length - delim_index) - 1 );i++) {
    wifi_pwd[i] = data[delim_index + 1 + i];
  }
#ifdef __DEBUG__
  Serial.print("New PWD : ");
  Serial.println(wifi_pwd);
#endif
  //memcpy(wifi_pwd,data + delim_index+1,(data_length -(delim_index )));

#ifdef __DEBUG__
  Serial.println("Setting SSID and password");
#endif
  ESP8266Adhoc::ssid = wifi_ssid;
  ESP8266Adhoc::password = wifi_pwd;
#ifdef __DEBUG__
  Serial.print("SSID set to : ");
  Serial.println(ESP8266Adhoc::ssid);
  Serial.print("Password set to : ");
  Serial.println(ESP8266Adhoc::password);
#endif
}

/* Process the internal commands between
 * Arduino and ESP.
 * 3-bits
 *
 * |CMD|Desription|
 * |000|No internal command
 * |001|Set the ID of the bot
 * |010|Set ssid and password of wifi
 * |011|get init_matrix for bot ID
 * |100|Get RSSI
```

```
 * |101|Server IP set
 * |110|Reserved
 * |111|Reserved
 *
 */

void ESP8266Adhoc::process_internal_cmd(char packet[]) {

  int cmd = is_internal_cmd(packet);
  char len = packet[PACKET_DATA_LENGTH_LOC];
  char data;
  long RSSI;

  switch(cmd) {
    case INTERNAL_SET_ID:
      data = get_self_ID(packet);
      set_id(data);
      break;

    case INTERNAL_SET_SSID_PWD:

      /* Will be handled in arduino loop itself */
      //ESP8266Adhoc::ssid = get_ssid(packet);
      // extract_ssid_pwd(packet,ESP8266Adhoc::recvBytes);
#ifdef __DEBUG__
      Serial.println("Initiating connection to WiFi");
#endif
      //connect_wifi();
#ifdef __DEBUG__
      Serial.println("Done");
#endif
      //delay(1000);
      break;
    case INTERNAL_SET_MATRIX :
      set_con_matrix(packet);
      break;
    case INTERNAL_GET_RSSI:
      // serial.send(packet,sizeof(packet));
      /* Handle RSSI will be taken care in loop itself */
      break;
    case INTERNAL_SET_SERVER_IP:
      set_server_ip(packet);
      break;
    case INTERNAL_NON_DEMO_MODE:
      DEMO_MODE = 1;
      break;
    default :
      Serial.println("Unknown command");

  }


}
#if 0

void ESP8266Adhoc::adhoc_send_filter(char packet[],int numBytes) {
```

```
  store_record(packet);
  Udp.beginPacketMulticast(ipMulti,adhoc_port, WiFi.localIP());
#ifdef __DEBUG__
  Serial.print(ID_SELF,DEC);
  Serial.print(" Sent broadcast on port ");
  Serial.println(adhoc_port);
#endif
  Udp.write(packet,numBytes);
  Udp.endPacket();
}




void ESP8266Adhoc::adhoc_recv_filter(char packet[]) {

  /* if self == dst -> forward the packet to arduino
   * Enter in the table if no entry present
   * Send ACK
   *
   */
  char src = get_src(packet);
  char dst = get_dst(packet);
  char inter = get_inter(packet);
  unsigned int counter = get_counter(packet);
  char ACK = is_ACK(packet);


  if(src == ID_SELF ) {

    /* should not receive from myself */
#ifdef __DEBUG__
    Serial.println("Receving from self ignored");
#endif
    return;
  }

  /* Drop the packet if you cannot
   * receive from this source
   */

  if(!can_rcv(inter)) {
    /* drop the packet */
#ifdef __DEBUG__
    Serial.print("Cannot receive from ");
    Serial.print(inter);
    Serial.println("Ignored ");
#endif
    return;
  }

  /* Chck if self is the destination
   *
   */
```

```
  if(ID_SELF == dst) {

#ifdef __DEBUG__
    Serial.println("I am the destination ");

#endif

    /* Check if entry for the packet already present
     *  Ignore if already present
     */

    if(is_record_present(counter)) {
#ifdef __DEBUG__
      Serial.print("Record already present for counter : ");
      Serial.print(counter);
      Serial.println("Ignored ");
#endif
      /* Drop the packet */
      return;
    }
    else {
      /* Make an entry in table
       *  Send to arduino over serial
       *  SEND ACK by reversing SRC and DST in
       *  same packet
       */
      store_record(packet);
      Serial.write(packet,sizeof(packet));

      Udp.beginPacketMulticast(ipMulti,8080, WiFi.localIP());

      /* Reverse source and destination */
      packet[1] = (dst << 4 | src );

      /* Set ACK bit */
      packet[3] & ( 1 << 4);
      Udp.write(packet,9);
      Udp.endPacket();

#ifdef __DEBUG__
      Serial.print("Stored the record for counter : ");
      Serial.print(counter);
      Serial.println("Sending ACK ");
#endif


      return;

    }

  }

  /* If SELF is not the destination
   *  then make intermediate source == SELF and broadcast
   *  again.
   *
```

```
   */

#ifdef __DEBUG__
  Serial.print("Checking if record for counter : ");
  Serial.print(counter);
  Serial.print(" present ");
  Serial.println(" && ACK ");
#endif

  if(!is_record_present(counter) && !ACK) {

#ifdef __DEBUG__
    Serial.print("Record not present for counter : ");
    Serial.print(counter);
    Serial.print(" Storing it ");
    Serial.println("");
    Serial.print("Changing inter to SELF ");
    Serial.print(ID_SELF);
    Serial.println("");
    Serial.println("Broadcasting again");
#endif


    store_record(packet);
    packet[2] = ID_SELF << 4;
    Udp.beginPacketMulticast(ipMulti,8080, WiFi.localIP());
    Udp.write(packet,9);
    Udp.endPacket();
    return;

  }

  /* Record present in counter and ACK in packet */

#ifdef __DEBUG__
  Serial.print("Checking if record for counter : ");
  Serial.print(counter);
  Serial.print(" && ACK in packet ");
  Serial.println("");
#endif


  if(is_record_present(counter) && ACK) {

    /* Check in table if ACK is set for this
     *  counter
     */

    if(ack_in_table(counter)) {
      //ignore it
#ifdef __DEBUG__
      Serial.print("Already received ACK for counter : ");
      Serial.print(counter);
      Serial.print(" Ignoring packet ");
      Serial.println("");
#endif
      return;
```

```
    }
    else {
      /* set ACK bit in table
       * and broadcast the ACK
       */
#ifdef __DEBUG__
      Serial.print("Received ACK for counter : ");
      Serial.print(counter);
      Serial.print(" Setting ACK bit in table ");
      Serial.println("");
      Serial.println("Broadcasting again");
#endif
      set_ack_table(counter);
      packet[2] = ID_SELF << 4;
      Udp.beginPacketMulticast(ipMulti,8080, WiFi.localIP());
      Udp.write(packet,9);
      Udp.endPacket();
      return;

    }

  }


}

#endif
```