Αναφορά υποχρεωτικής εργασίας στο μάθημα «ΔΙΚΤΥΑ ΥΠΟΛΟΓΙΣΤΩΝ 2»

Τσαντίκης Γεώργιος

ΑΕΜ: 10722

ΕΙΣΑΓΩΓΗ

Η εργασία αποτελείται από δύο τμήματα τα οποία αφορούν την υλοποίηση των δύο βασικών λειτουργιών της εφαρμογής, οι οποίες είναι η αποστολή μηνυμάτων (instant messaging - chat) και η φωνητική επικοινωνία σε πραγματικό χρόνο (real-time audio/voice communication) μέσω τοπικού δικτύου με κώδικα σε Java. Στην παρούσα αναφορά περιέχεται μια σύντομη **περιγραφή** του επιπλέον **κώδικα** που υλοποιήθηκε και **εικόνες** από το **Wireshark** οι οποίες δείχνουν:

(i) παραδείγματα μηνυμάτων κειμένου που αποστάλθηκαν μέσω της εφαρμογής μεταξύ των δύο χρηστών (εμφάνιση Network/Internet header και payload τόσο σε εξαδική όσο και σε μορφή κειμένου),

(ii) το stream των πακέτων φωνής που ανταλλάσσονται (η λίστα όπως εμφανίζεται στο main window του Wireshark) και

(iii) παραδείγματα πακέτων φωνής που ανταλλάχθηκαν μέσω της εφαρμογής (εμφάνιση Network/Internet header και payload τουλάχιστον σε δυαδική μορφή).

Βιβλιοθήκες και Μεταβλητές

Το εικονιζόμενο τμήμα κώδικα εισάγει τις κύριες βιβλιοθήκες μέσω των οποίων χρησιμοποιήθηκαν βασικές τεχνολογίες της **Java** για δημιουργία και διαχείριση του **GUI** (*javax.swing.*),* **δικτύωση** *java.net.*),* κρυπτογράφηση μηνυμάτων για **ασφάλεια** (*java.security.Key, javax.crypto.Cipher, javax.crypto.spec.SecretKeySpec, java.util.Base64*), χειρισμό **ήχου** - **φωνητική επικοινωνία** *(javax.sound.sampled.*)* και **πολυνηματική** εκτέλεση για ταυτόχρονη διαχείριση κειμένου και φωνής (*java.lang.Thread*)*:*

```
import java.io.*;
import java.net.*;
import javax.swing.*;
import javax.sound.sampled.*;
import java.awt.*;
import java.awt.event.*;
import java.security.Key; //interface for handling keys
import javax.crypto.Cipher; //functionality for encryption and decryption
import javax.crypto.spec.SecretKeySpec; //construct key from a byte array
import java.util.Base64; //for encoding and decoding data in form Base64(binary representation)
import java.lang.Thread;
```

Figure 1: Αξιοποιούμενες βιβλιοθήκες Java

Παρακάτω αρχικοποιούνται οι βασικές μεταβλητές που θα χρησιμοποιηθούν:

```
// TODO: Please define and initialize your variables here...
// Ports and UDP Sockets
private static final int SEND_PORT_CHAT = 5000, RECEIVE_PORT_CHAT = 50001;
private static final int SEND_PORT_VOICE = 5002, RECEIVE_PORT_VOICE = 50003;
private static DatagramSocket sendSocketChat, receiveSocketChat;
private static DatagramSocket sendSocketVoice, receiveSocketVoice;

// IP Address
private static InetAddress remoteAddress;

// Voice Communication
static volatile boolean isCalling = false;

// Encryption Key
private static final String SECRET_KEY = "2444666668888888"; // 16-byte key
```

Figure 2: Βασικές παράμετροι

Αρχικά, ορίζονται οι θύρες (ports) που θα χρησιμοποιηθούν για την αποστολή και λήψη δεδομένων για κείμενο (chat) και φωνή (voice). Επίσης, χρησιμοποιείται το *DatagramSocket* που παρέχει εύκολη υλοποίηση δικτύωσης μέσω πρωτοκόλλου UDP, για τη δημιουργία UDP sockets τα οποία επικοινωνούν μέσω αυτών των θυρών και προτιμώνται λόγω χαμηλής καθυστέρησης και απλότητας. Η *InetAddress* αποθηκεύει τη διεύθυνση IP του απομακρυσμένου χρήστη που θα χρησιμοποιηθεί για την αποστολή δεδομένων. Επιπλέον, η *isCalling* ελέγχει αν η φωνητική κλήση είναι ενεργή και το *volatile* εξασφαλίζει ότι η τιμή είναι ορατή σε όλα τα *threads* (εξασφαλίζοντας thread safety). Τέλος, υπάρχει το *SECRET_KEY,* το οποίο απαιτείται για την κρυπτογράφηση και αποκρυπτογράφηση των μηνυμάτων μέσω του *AES (*Advanced Encryption Standard*)* που θα αναλυθεί παρακάτω.

Μέρος 1: Αποστολή Μηνυμάτων (Instant Messaging / Chat)

1.1. Περιγραφή Κώδικα

Οι χρήστες θα πρέπει να μπορούν μέσω της εφαρμογής να ανταλλάσσουν μηνύματα κειμένου μεταξύ τους. Η υλοποίηση της εν λόγω λειτουργίας πραγματοποιείται στην μέθοδο ***main(…)*** και χρησιμοποιεί δύο sockets: *sendSocketChat* για αποστολή, και *receiveSocketChat* για λήψη. Η *receiveSocketChat* είναι συνδεδεμένη με συγκεκριμένο port, ονομάτι *RECEIVE_PORT_CHAT*, η τιμή του οποίου καθορίζεται από τους χρήστες. Για την επίτευξη ασύγχρονης λειτουργίας, χρησιμοποιείται **πολυνημάτωση** (multithreading), η οποία παράλληλα επιτρέπει την ταυτόχρονη αποστολή/λήψη μηνυμάτων και δεδομένων φωνής.

Εντός της μεθόδου ***main(…)***, ορίζεται και αρχίζει το thread *chatReceiverThread* . Αξιοποιώντας ένα ατέρμονο βρόγχο *while(true)*, λαμβάνει από το καθορισμένο port κρυπτογραφημένα UDP πακέτα, τα οποία αποκρυπτογραφεί και προβάλλει στο GUI παράθυρο του παραλήπτη (δίπλα από τη λέξη «remote», για να είναι ξεκάθαρο ότι το μήνυμα προέρχεται από τον αποστολέα/απομακρυσμένο χρήστη).

```
        try {
            //create and/or initialize sockets. Specify which IP address you wish to send messages to.
            remoteAddress = InetAddress.getByName("192.168.1.4"); // Replace with the actual remote IP
            sendSocketChat = new DatagramSocket();
            receiveSocketChat = new DatagramSocket(RECEIVE_PORT_CHAT);
            sendSocketVoice = new DatagramSocket();
            receiveSocketVoice = new DatagramSocket(RECEIVE_PORT_VOICE);

            // Thread for receiving text
            Thread chatReceiverThread = new Thread(() -> {
                byte[] buffer = new byte[1024];
                try {
                    while (true) {
                        DatagramPacket packet = new DatagramPacket(buffer, buffer.length);
                        receiveSocketChat.receive(packet);
                        String encryptedMessage = new String(packet.getData(), 0, packet.getLength());
                        String decryptedMessage = decrypt(encryptedMessage);
                        textArea.append("remote: " + decryptedMessage + "\n");
                    }
                } catch (IOException e) {
                    e.printStackTrace();
                } catch (Exception e) {
                    // TODO Auto-generated catch block
                    e.printStackTrace();
                }
            });

            //start the thread
            chatReceiverThread.start();

        } catch (Exception e) {
            e.printStackTrace();
        }
```

Figure 3: Ορισμός και εκκίνηση του νήματος *chatReceiverThread*

Στη μέθοδο ***actionPerformed(…)*** που υλοποιεί τη λειτουργικότητα που εκτελείται όταν ένας χρήστης πατήσει ένα κουμπί στο γραφικό περιβάλλον, περιέχεται και ο παρακάτω κώδικας:

```
if (e.getSource() == sendButton) {
    try {
        String message = inputTextField.getText(); //read the message
        String encryptedMessage = encrypt(message); //encrypt the message for safety
        byte[] buffer = encryptedMessage.getBytes(); //convert to bytes
        /*Object that contains the data, their length, receiver's address, and sending port*/
        DatagramPacket packet = new DatagramPacket(buffer, buffer.length, remoteAddress, SEND_PORT_CHAT);
        sendSocketChat.send(packet); //socket for sending the packet
        textArea.append("local: " + message + "\n");
        inputTextField.setText(""); //empty the text field
    } catch (Exception ex) {
        ex.printStackTrace(); //write down to console any problem about sockets or encryption
    }
```

Figure 4: Κρυπτογράφηση

Έτσι, με το πάτημα του κουμπιού ***Send***, λαμβάνεται το κείμενο που εισήγαγε ο χρήστης στο πεδίο *inputTextField,* αποθηκεύεται σε ένα *String*, κρυπτογραφείται, μετατρέπεται σε byte array, δημιουργείται το UDP πακέτο και αποστέλλεται. Το αρχικό, μη κρυπτογραφημένο String προβάλλεται στην *textArea* του GUI παραθύρου του αποστολέα (δίπλα από τη λέξη «local», για να είναι ξεκάθαρο ότι το μήνυμα προέρχεται από αυτόν και όχι από τον συνομιλητή του).

Η **κρυπτογράφηση** (και αποκρυπτογράφηση) πραγματοποιείται με βάση το Advanced Encryption Standard, εξασφαλίζοντας ότι τα γραπτά μηνύματα είναι ασφαλή κατά τη διάρκεια της μετάδοσης χρησιμοποιώντας τις παρακάτω μεθόδους:

```
/* Use of Advanced Encryption Standard, which returns a string with the encrypted message
   (same key for encryption and decryption).
*/
private static String encrypt(String message) throws Exception {
    Cipher cipher = Cipher.getInstance("AES"); //creates a presence of Cipher for the AES algorithm
    Key key = new SecretKeySpec(SECRET_KEY.getBytes(), "AES"); //converse from a byte array to a key for AES
    cipher.init(Cipher.ENCRYPT_MODE, key); //declares that Cipher is going to be used for encryption
    byte[] encryptedBytes = cipher.doFinal(message.getBytes()); //the encryption algorithms work with bytes
    return Base64.getEncoder().encodeToString(encryptedBytes); //converts the encrypted bytes to a Base64
                                                               //string (instead of using binary data)
}

private static String decrypt(String encryptedMessage) throws Exception {
    Cipher cipher = Cipher.getInstance("AES");
    Key key = new SecretKeySpec(SECRET_KEY.getBytes(), "AES");
    cipher.init(Cipher.DECRYPT_MODE, key);
    byte[] decodedBytes = Base64.getDecoder().decode(encryptedMessage);
    return new String(cipher.doFinal(decodedBytes));
}
```

Figure 5: Μέθοδοι κρυπτογράφησης/αποκρυπτογράφησης

Το String ***SECRET_KEY*** = "2444666668888888" (16 bytes) ορίζεται στην αρχή του constructor της κλάσης ***App***.

Εντός των μεθόδων δημιουργείται ένα αντικείμενο των έτοιμων κλάσεων *Cipher* και *Key (για απλοποίηση της δημιουργίας του κλειδιού)*, και έπειτα χρησιμοποιείται η μέθοδος *init()* της κλάσης *Cipher* για να ορίσει εάν το συγκεκριμένο αντικείμενο θα πραγματοποιήσει :

- κρυπτογράφηση (*cipher.init(Cipher.**ENCRYPT_MODE**, key)*), ή
- αποκρυπτογράφηση (*cipher.init(Cipher.**DECRYPT_MODE**, key)*)

και ποιο κλειδί θα χρησιμοποιήσει για την κάθε ενέργεια. Αφού χρησιμοποιείται Advanced Encryption Standard (AES), το οποίο είναι αλγόριθμος συμμετρικού κλειδιού, το κλειδί είναι το ίδιο και στις δύο περιπτώσεις.

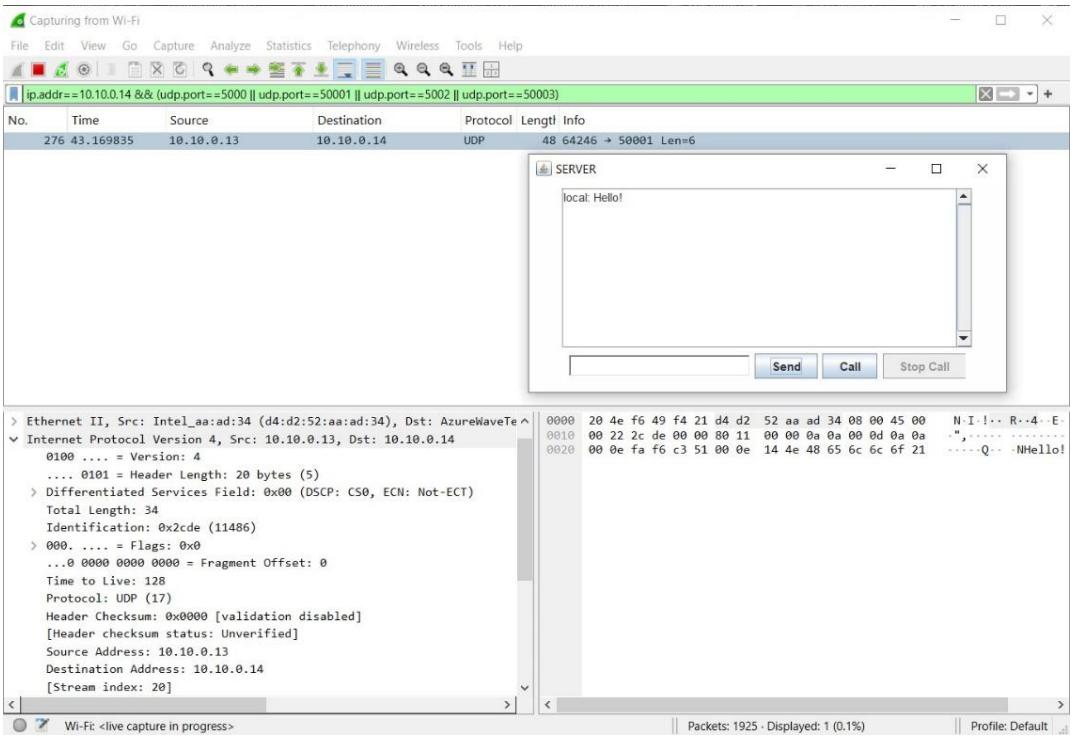
Η μέθοδος *doFinal()* πραγματοποιεί την κρυπτογράφηση (ή αποκρυπτογράφηση) του μηνύματος. Για να διασφαλιστεί η ακεραιότητα των δεδομένων κατά την αποστολή, το μήνυμα κωδικοποιείται επιπλέον σε Base64 (και έπειτα όταν ληφθεί, αποκωδικοποιείται προτού αποκρυπτογραφηθεί).

1.2. Wireshark

Στη συνέχεια παρατίθενται κάποια παραδείγματα **εικόνων** ανταλλαγής γραπτών μηνυμάτων, η οποία παρακολουθήθηκε με χρήση του **Wireshark**. Οι τίτλοι των παραθύρων της εφαρμογής είναι **SERVER** και **CLIENT** για τον πρώτο και τον δεύτερο χρήστη αντίστοιχα.

1. Αρχικά γίνεται δοκιμή με τον ένα χρήστη να χρησιμοποιεί το Advanced Encryption Standard, ενώ ο άλλος όχι.

   Ο SERVER στέλνει «Hello!» :



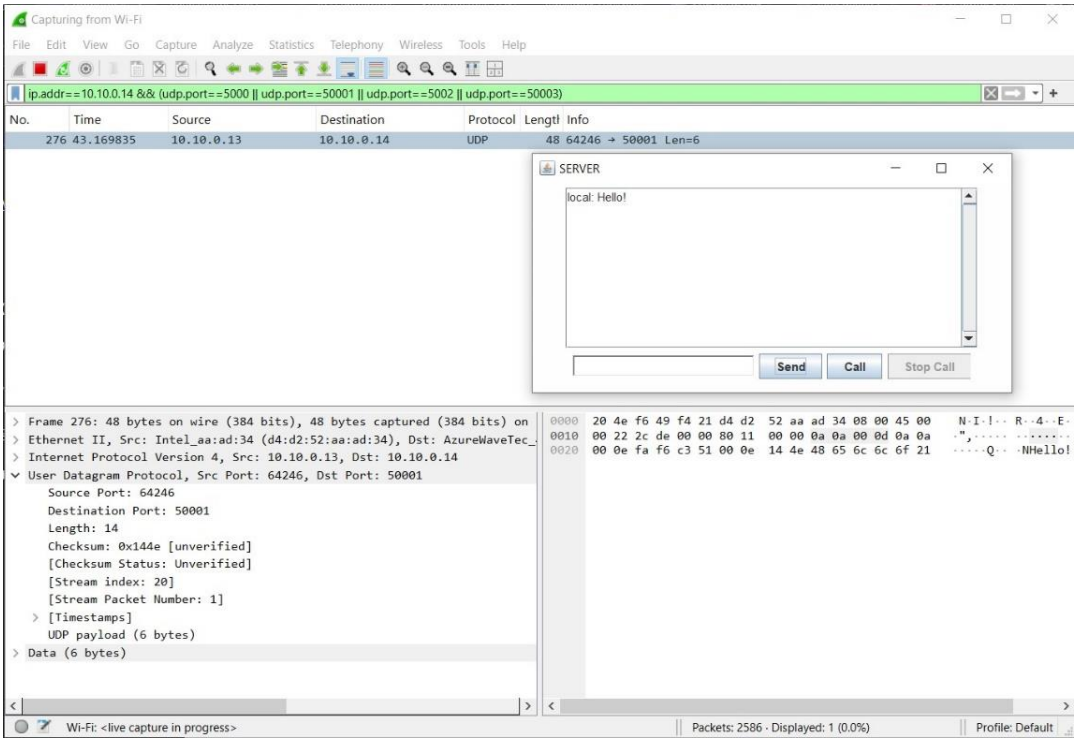
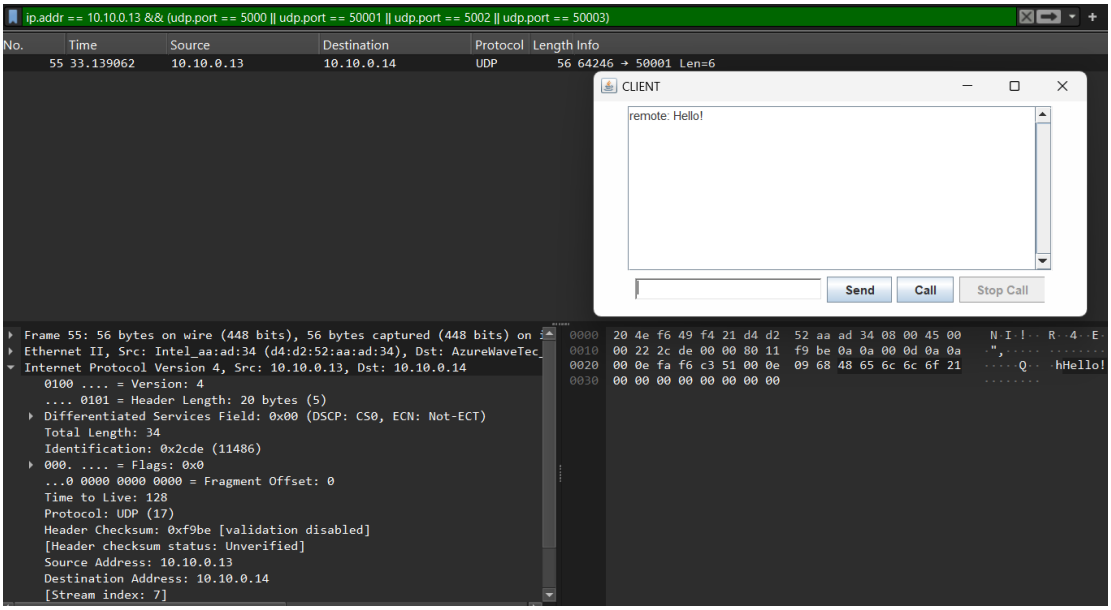


Figure 6: Ο χρήστης SERVER στέλνει « Hello! » χωρίς κρυπτογράφηση. Το περιεχόμενο του μηνύματός του είναι εκτεθειμένο. Επιπλέον δεν είναι προετοιμασμένος για να λάβει κρυπτογραφημένα μηνύματα, πράγμα που μετέπειτα θα του δημιουργήσει πρόβλημα.

Πλευρά CLIENT:



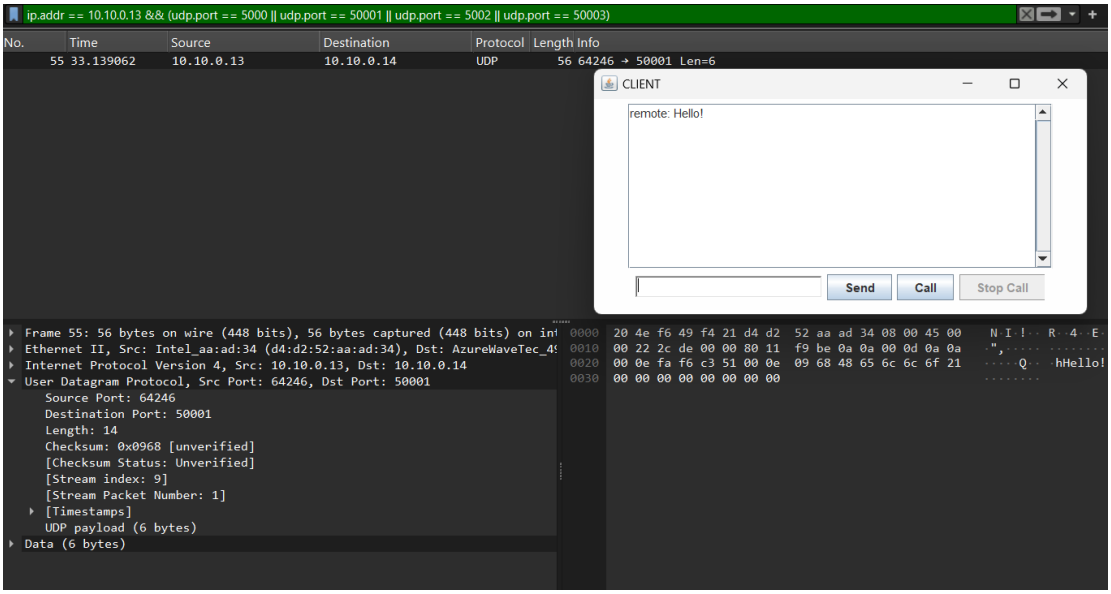
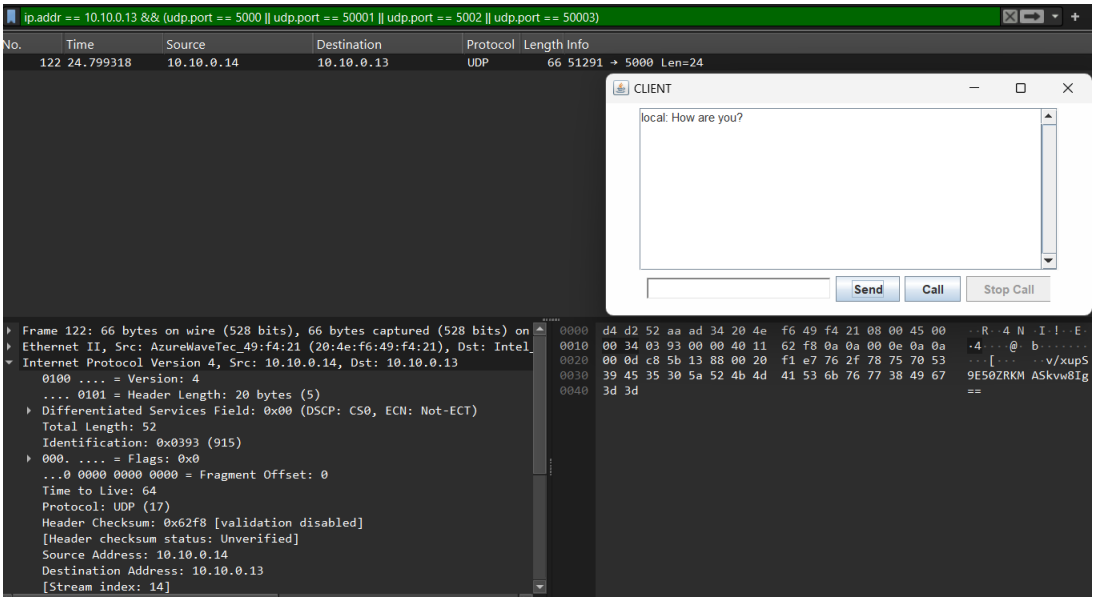


Figure 7: Ο χρήστης CLIENT λαμβάνει το μη κρυπτογραφημένο « Hello! ». Όπως και στην πλευρά του SERVER, το Wireshark ανιχνεύει το περιεχόμενο του μηνύματος.

Όπως θα γίνει εμφανές παρακάτω, ο CLIENT κρυπτογραφεί τα μηνύματά του προτού τα στείλει, και είναι προετοιμασμένος να λάβει κρυπτογραφημένα μηνύματα.

Ο CLIENT απαντά « How are you? » :



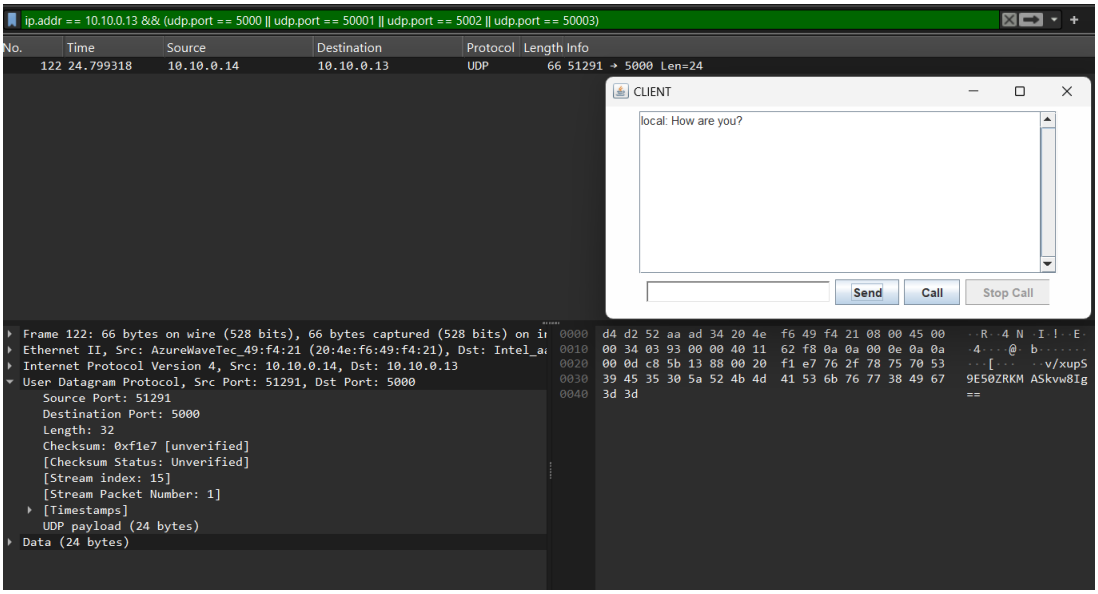
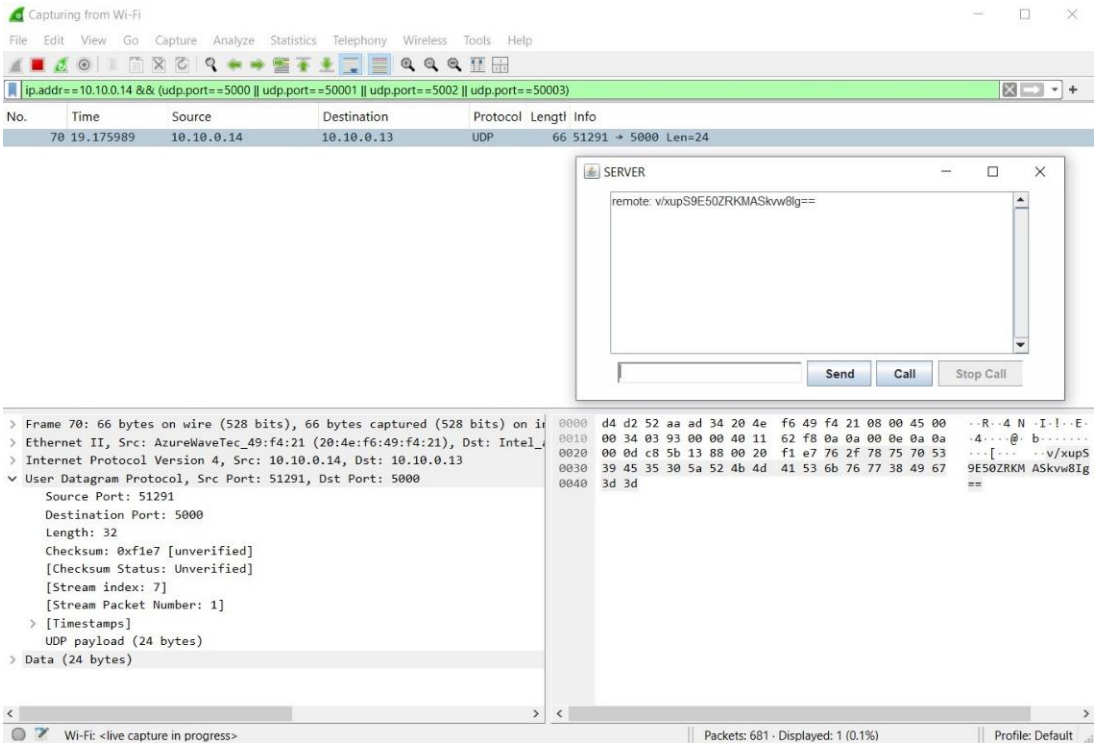


Figure 8: Ο χρήστης CLIENT στέλνει μια κρυπτογραφημένη απάντηση. Το περιεχόμενο του μηνύματος εμφανίζεται στο Wireshark ως μια τυχαία συμβολοσειρά.

Πλευρά SERVER:



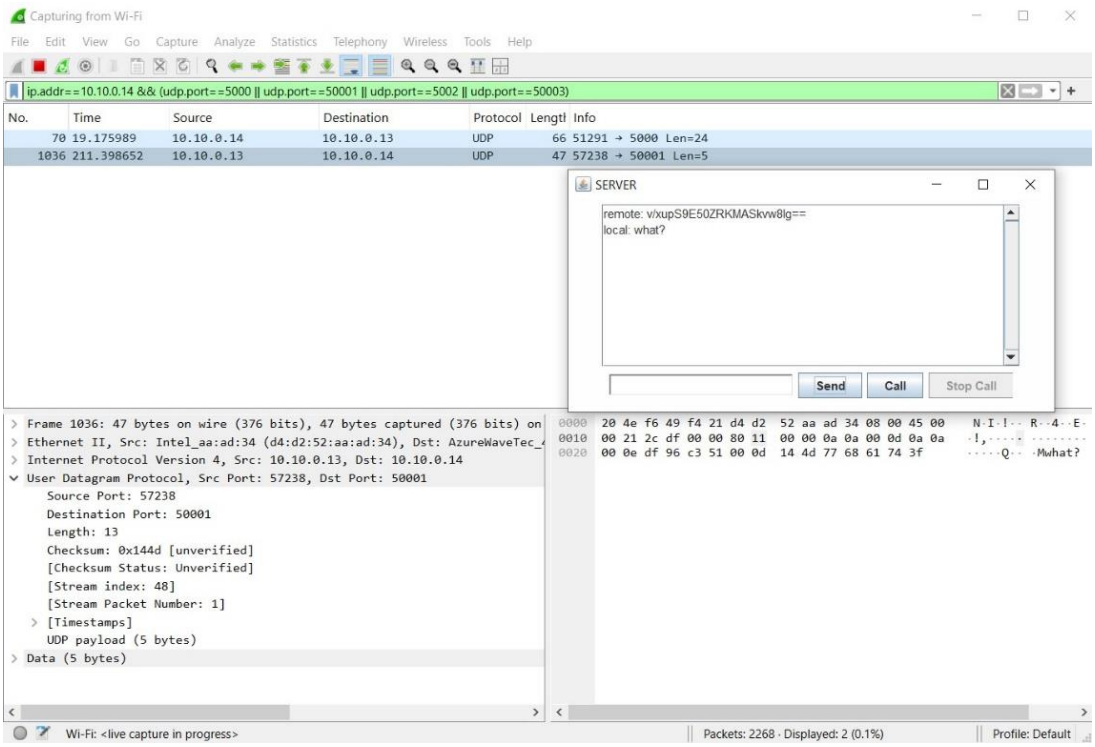
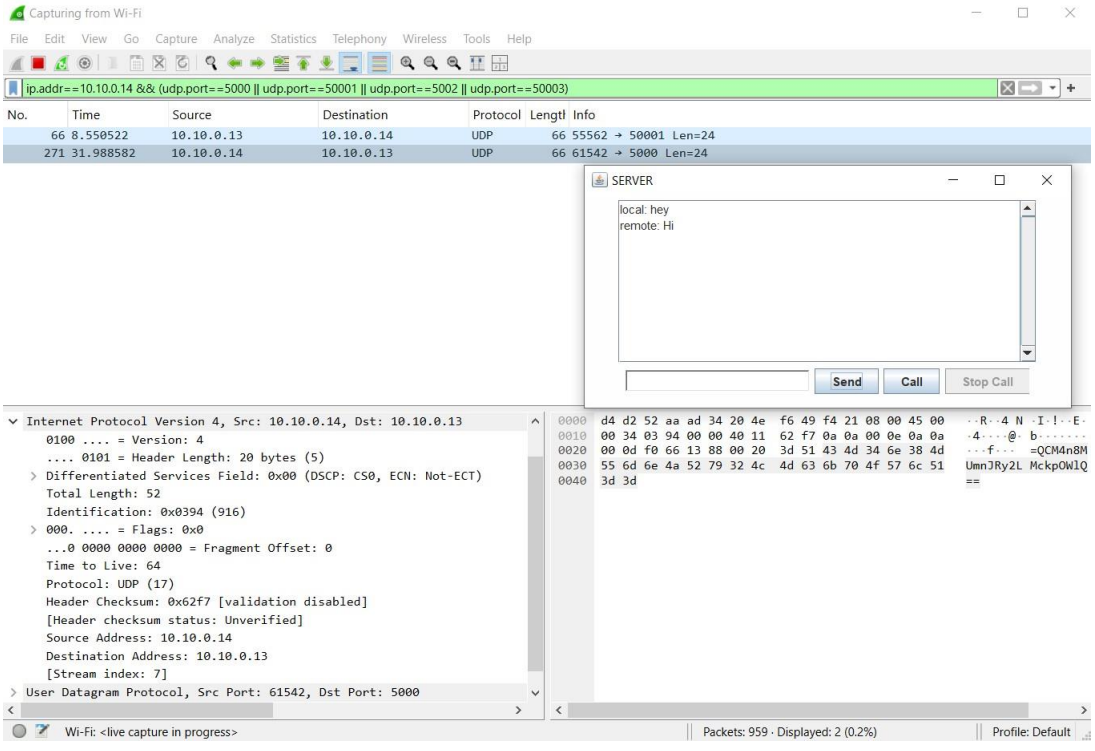


Figure 9: Ο SERVER δεν είναι προετοιμασμένος να λάβει κρυπτογραφημένα μηνύματα. Επομένως βλέπει την ίδια τυχαία συμβολοσειρά που ανίχνευσε προηγουμένως το Wireshark.

2. Εφόσον και οι δύο συνομιλητές κάνουν χρήση του Advanced Encryption Standard, η επικοινωνία τους έχει ως εξής:

Πλευρά SERVER:



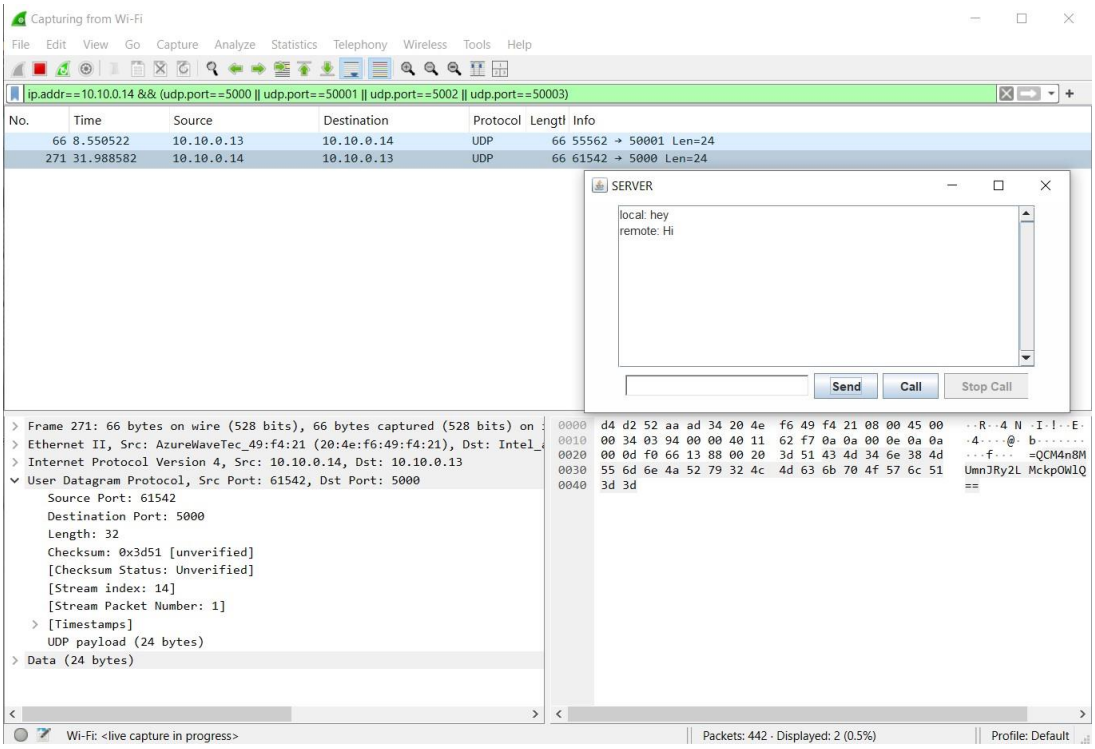
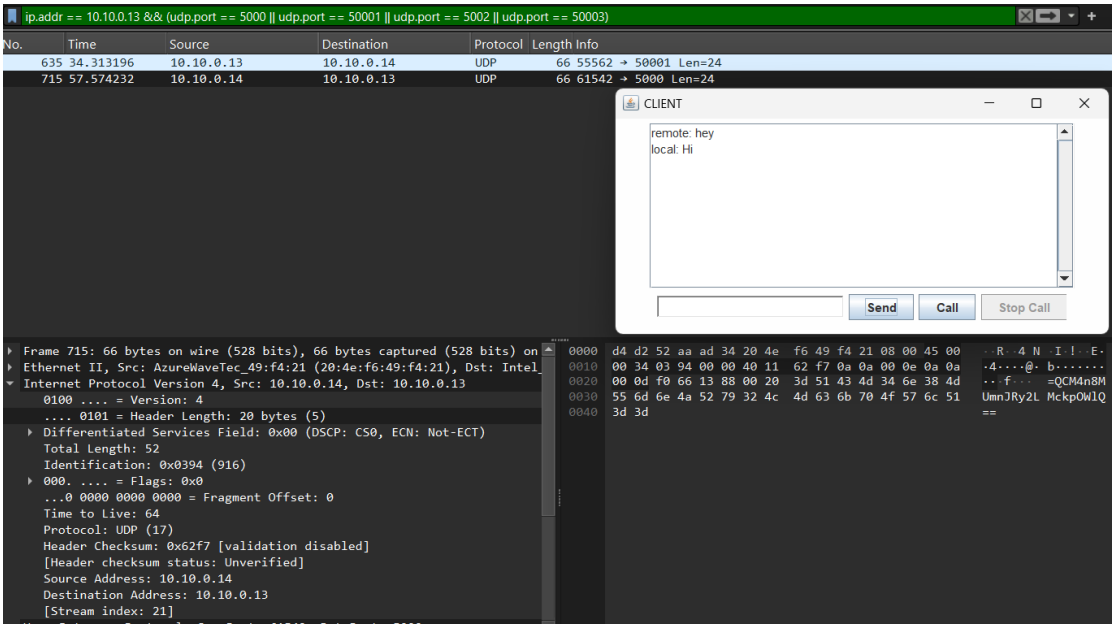


Figure 10: Επιτυχής επικοινωνία με χρήση κρυπτογράφησης, πλευρά SERVER

Πλευρά CLIENT:



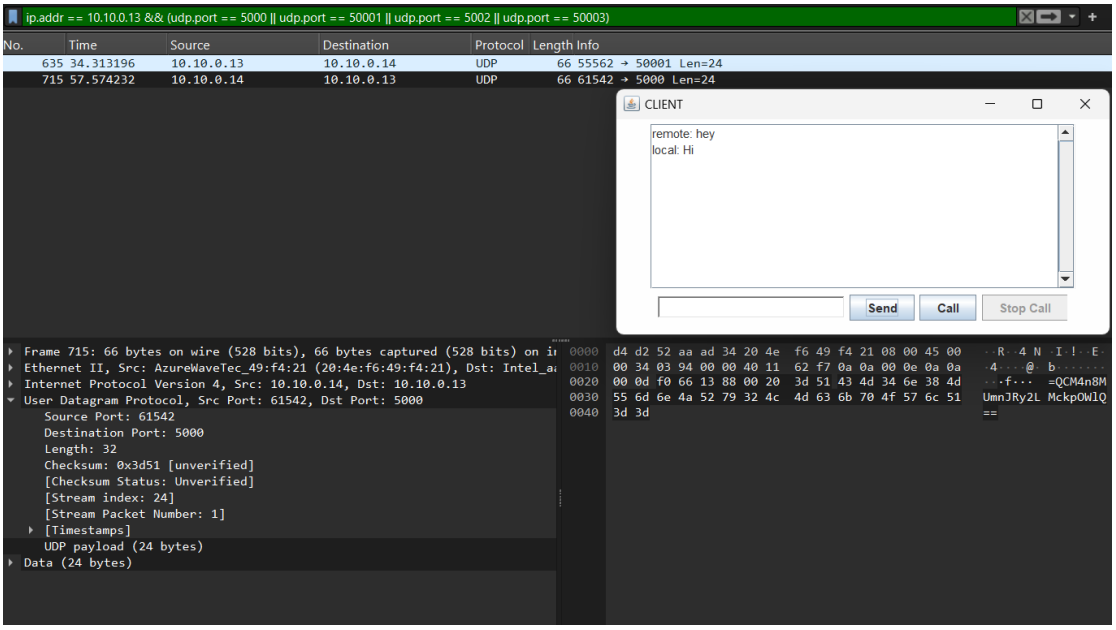


Figure 11: Επιτυχής επικοινωνία με χρήση κρυπτογράφησης, πλευρά CLIENT

Τα περιεχόμενα των μηνυμάτων είναι προστατευμένα κατά τη μεταφορά τους, ενάντια σε οποιονδήποτε αγνοεί το πρότυπο κρυπτογράφησης και το κλειδί που χρησιμοποιήθηκαν. Το Wireshark ανιχνεύει τυχαίες συμβολοσειρές, χωρίς κάποιο προφανές (για ανθρώπινο παρατηρητή) νόημα, ενώ τα μηνύματα εμφανίζονται κανονικά στους χρήστες της εφαρμογής.

Μέρος 2: Φωνητική Επικοινωνία (VoIP)

2.1. Περιγραφή Κώδικα

Οι χρήστες θα πρέπει μέσω της εφαρμογής να επιτύχουν αμφίδρομη φωνητική επικοινωνία σε πραγματικό χρόνο, ανταλλάσσοντας πακέτα φωνής μεταξύ τους. Όπως προαναφέρθηκε, η μέθοδος ***actionPerformed(…)***, η οποία είναι μέρος της διεπαφής *ActionListener*, υλοποιεί τη λειτουργικότητα που εκτελείται όταν ένας χρήστης πατήσει ένα κουμπί στο γραφικό περιβάλλον.

Τα παρακάτω κομμάτια κώδικα περιέχονται στην *actionPerformed(…)* και εκτελούνται όταν ο χρήστης πατήσει το κουμπί ***Call:***

```
} else if (e.getSource() == callButton) {
    isCalling = true;   //this means that the call has started
    callButton.setEnabled(false);   //disable the callButton
    stopCallButton.setEnabled(true);    //and enable the stopCallButton
```

Figure 12: Αρχικοποίηση

Αρχικά, η μεταβλητή *isCalling* τίθεται ως *true,* πράγμα που δηλώνει ότι η κλήση ξεκίνησε, απενεργοποιείται το κουμπί κλήσης (*CallButton*) και ενεργοποιείται το κουμπί διακοπής κλήσης (*stopCallButton*).

Έπειτα, ορίζεται η μορφή του ηχητικού μηνύματος με χρήση της κλάσης *AudioFormat*.

```
AudioFormat format = new AudioFormat(44100, 16, 1, true, false);//(44100, 16, 1, true, false) for CD quality
                                                                //or  (8000, 8, 1, true, true) PCM
```

Figure 13: Audio Format

Το **audio format** που προτείνεται ***(8000Hz, 8-bit samples)*** είναι μονοφωνική παλμοκωδική διαμόρφωση ήχου PCM (pulse code modulation) με τα εξής χαρακτηριστικά:

(i) συχνότητα δειγματοληψίας 8000 samples/sec,
(ii) μέγεθος δείγματος 8 bits (1 byte),
(iii) μονοφωνικό κανάλι (1 κανάλι) και
(iv) signed δείγματα.

Αποτελεί το πρότυπο που χρησιμοποιείται ευρέως στην τηλεφωνία και στις VoIP εφαρμογές, καθώς εξισορροπεί την κατανάλωση bandwidth με την ποιότητα ήχου, επαρκώντας για τη μετάδοση ομιλίας.

Ωστόσο, δοκιμάστηκε και το audio format **(44100Hz και 16-bit samples)** το οποίο είναι επιπέδου CD quality (HQ audio). Θεωρητικά, με αυτήν την επιλογή, προκύπτει το ζήτημα ότι για να επιτύχεις υψηλότερη ποιότητα ήχου κάνεις την εφαρμογή VoIP σου να απαιτεί μεγαλύτερο bandwidth (περίπου 11 φορές περισσότερα bandwidth από το βασικό audio format), δημιουργώντας ενδεχομένως πρόβλημα αν επιχειρήσεις να επικοινωνήσεις μέσω του Διαδικτύου.

Στόχο αποτελεί η εύρεση του μέγιστου δυνατού quality (sample rate + sample size) που οδηγεί στην κατά το δυνατόν πιο seamless και real-time επικοινωνία (low latency). Αυτές οι δύο παράμετροι είναι αντικρουόμενες και πρέπει να βρεθεί η χρυσή τομή.

Οι παρατηρήσεις πάνω σε αυτά θα παρουσιαστούν στη συνέχεια παράλληλα με τα παραδείγματα εικόνων από το Wireshark στην *παράγραφο 2.2*.

Αμέσως μετά τον προσδιορισμό του audio format, αρχικοποιούνται και τίθενται σε λειτουργία οι γραμμές ήχου που θα χρησιμοποιηθούν στην αναπαραγωγή και εγγραφή των μηνυμάτων φωνής. Χρησιμοποιούνται οι διεπαφές *TargetDataLine* και *SourceDataLine,* συνδεδεμένες με το μικρόφωνο και το ηχείο αντίστοιχα.

```
TargetDataLine microphone = AudioSystem.getTargetDataLine(format); //record audio using microphone
microphone.open(format);
microphone.start();

SourceDataLine speakers = AudioSystem.getSourceDataLine(format); //play back audio using speakers
speakers.open(format);
speakers.start();
```

Figure 14: Μικρόφωνο και ηχεία

Τέλος, ορίζονται και εκκινούνται τα νήματα *sendVoiceThread* (για αποστολή ήχου) και *receiveVoiceThread* (για λήψη ήχου).

```
Thread sendVoiceThread = new Thread(() -> { //thread for continuous reading and sending sound
    byte[] buffer = new byte[1024];
    try {
        while (isCalling) {
            int bytesRead = microphone.read(buffer, 0, buffer.length); //convert sound to packets
            DatagramPacket packet = new DatagramPacket(buffer, bytesRead, remoteAddress, SEND_PORT_VOICE);
            sendSocketVoice.send(packet);
        }
        microphone.close();
    } catch (IOException ex) {
        ex.printStackTrace();
    }
});
sendVoiceThread.start();
```

Figure 15: Ορισμός και εκκίνηση του *sendVoiceThread.* Πραγματοποιεί συνεχή ανάγνωση ήχου από το μικρόφωνο και αποστολή του στη θύρα *SEND_PORT_VOICE,* εφόσον μετατραπεί σε πακέτα UDP. Το *while (isCalling)* διασφαλίζει ότι η αποστολή συνεχίζεται όσο η κλήση είναι ενεργή.

```
Thread receiveVoiceThread = new Thread(() -> { //thread for continuous receiving
    byte[] buffer = new byte[1024];
    try {
        while (isCalling) {
            DatagramPacket packet = new DatagramPacket(buffer, buffer.length);
            receiveSocketVoice.receive(packet);
            speakers.write(packet.getData(), 0, packet.getLength());
        }
        speakers.close();
    } catch (IOException ex) {
        ex.printStackTrace();
    }
});
receiveVoiceThread.start();
```

Figure 16: Ορισμός και εκκίνηση της *receiveVoiceThread.* Πραγματοποιεί συνεχή λήψη πακέτων που περιέχουν φωνή, με τα δεδομένα να αναπαράγονται στα ηχεία, για όσο η κλήση είναι ενεργή.

Όλα τα κομμάτια κώδικα από τον ορισμό του audio format έως και αυτό το σημείο περικλείονται σε ένα try catch block, με σκοπό τη διαχείριση τυχόν εξαιρέσεων σε περίπτωση που δεν ανιχνευτεί κάποιο συμβατό διαθέσιμο μικρόφωνο ή ηχείο:

```
} else if (e.getSource() == callButton) {
    isCalling = true; //it means that the call has started
    callButton.setEnabled(false); //disable the callButton
    stopCallButton.setEnabled(true); //and enable the stopCallButton
    try { //(pulse code modulation, 8000 samples/sec, 8 bits and signed samples, single voice channel
        AudioFormat format = new AudioFormat(8000, 8, 1, true, true);//or 44100, 16, 1, true, false
        TargetDataLine microphone = AudioSystem.getTargetDataLine(format); //record audio using microphone
        microphone.open(format);
        microphone.start();

        SourceDataLine speakers = AudioSystem.getSourceDataLine(format); //play back audio using speakers
        speakers.open(format);
        speakers.start();

        Thread sendVoiceThread = new Thread(() -> { //thread for continuous reading of sound and sending through UDP
            byte[] buffer = new byte[100];//or 4096, 1024
            try {
                while (isCalling) { //as far as the call is opened
                    int bytesRead = microphone.read(buffer, 0, buffer.length); //convert sound to packets
                    DatagramPacket packet = new DatagramPacket(buffer, bytesRead, remoteAddress, SEND_PORT_VOICE);
                    sendSocketVoice.send(packet);
                }
                microphone.close();
            } catch (IOException ex) {
                ex.printStackTrace();
            }
        });
        sendVoiceThread.start();

        Thread receiveVoiceThread = new Thread(() -> { //thread for continuous receiving of UDP packets that contain voic
            byte[] buffer = new byte[100];//or 4096, 1024
            try {
                while (isCalling) {
                    DatagramPacket packet = new DatagramPacket(buffer, buffer.length);
                    receiveSocketVoice.receive(packet);
                    speakers.write(packet.getData(), 0, packet.getLength());
                }
                speakers.close();
            } catch (IOException ex) {
                ex.printStackTrace();
            }
        });
        receiveVoiceThread.start();

    } catch (LineUnavailableException ex) {
        ex.printStackTrace();
    }
```

Figure 17: Ολοκληρωμένη εικόνα υλοποίησης της λειτουργικότητας κλήσεων

Στο σημείο αυτό να αναφερθεί ότι προστέθηκε η λειτουργικότητα του κουμπιού **Stop Call** στον *constructor* της κλάσης *App* που υλοποιήθηκε από τον καθηγητή:

```
//Setting up the buttons
sendButton = new JButton("Send");
callButton = new JButton("Call");
stopCallButton = new JButton("Stop Call");
stopCallButton.setEnabled(false); //False so as not to be able to respond to user's input

/*
 * 2. Adding the components to the GUI
 */
add(scrollPane);
add(inputTextField);
add(sendButton);
add(callButton);
add(stopCallButton);

/*
 * 3. Linking the buttons to the ActionListener in order to handle user's activity
 */
sendButton.addActionListener(this);
callButton.addActionListener(this);
stopCallButton.addActionListener(this);
```

Figure 18: Προσθήκη κουμπιού Stop Call για ευκολία χρήσης της εφαρμογής.

Αν πατηθεί το κουμπί διακοπής, η κλήση τερματίζεται:

```
} else if (e.getSource() == stopCallButton) { //stop call and bring buttons to their initial state
    isCalling = false;
    callButton.setEnabled(true);
    stopCallButton.setEnabled(false);
    /*Comment if you want to be able for more than one calls on the opened GUI */
    try { //close UDP sockets for sending and receiving voice
        sendSocketVoice.close();
        receiveSocketVoice.close();
    } catch (Exception ex) {
        ex.printStackTrace();
    }
    /**/
}
```

Figure 19: Υλοποίηση της λειτουργικότητας του κουμπιού Stop Call.

Η *isCalling* γίνεται false, και τα κουμπιά επανέρχονται στην αρχική τους κατάσταση. Έπειτα, κλείνουν τα UDP sockets για αποστολή και λήψη φωνής έχοντας ως αποτέλεσμα να μπορεί ο χρήστης να πατήσει μόνο μία φορά το κάθε κουμπί (Call και Stop Call) και να λάβει λειτουργικότητα από τη στιγμή που θα ανοίξει την εφαρμογή, δηλαδή να έχει δικαίωμα για μία μόνο κλήση. Ωστόσο, βάζοντας σε σχόλια το συγκεκριμένο κομμάτι κώδικα, παρέχεται η δυνατότητα να μπορεί να πραγματοποιεί παραπάνω από μία κλήσεις.

Εξάλλου τα sockets για αποστολή/λήψη μηνυμάτων κειμένου/φωνής κλείνουν πατώντας το ‘Χ’ στο ανοιχτό παράθυρο του GUI:

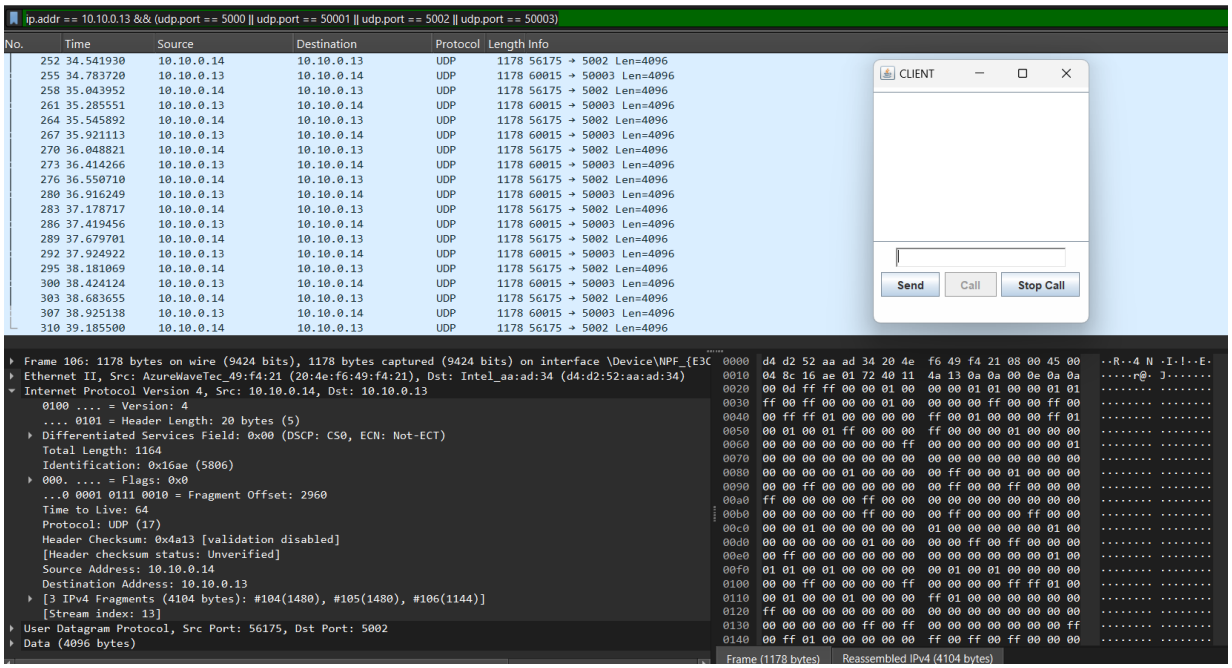
```
public void windowClosing(WindowEvent e) {
    try {
        if (sendSocketChat != null) sendSocketChat.close();
        if (receiveSocketChat != null) receiveSocketChat.close();
        if (sendSocketVoice != null) sendSocketVoice.close();
        if (receiveSocketVoice != null) receiveSocketVoice.close();
    } catch (Exception ex) {
        ex.printStackTrace();
    }
    dispose();
    System.exit(0);
}
```

Figure 20: Κλείσιμο των sockets μαζί με το κλείσιμο του παραθύρου διεπαφής

2.2. Wireshark

Στη συνέχεια παρατίθενται κάποια παραδείγματα **εικόνων** από το **Wireshark** του stream των πακέτων φωνής που ανταλλάσσονται (η λίστα όπως εμφανίζεται στο main window του Wireshark) και πακέτων φωνής που ανταλλάχθηκαν μέσω της εφαρμογής (εμφάνιση Network/Internet header και payload τουλάχιστον σε δυαδική μορφή). Οι τίτλοι των παραθύρων της εφαρμογής είναι **SERVER** και **CLIENT** για τον πρώτο και τον δεύτερο χρήστη αντίστοιχα.

- Τα επακόλουθα παραδείγματα εικόνων είναι με χρήση του **Audio Format [8000Hz και 8-bit samples]** και **buffer size [4096 bytes]**, το οποίο καθορίζει πόσα δεδομένα συλλέγονται πριν αποσταλούν ή αναπαραχθούν. Με 8000 Hz και 8-bit samples, κάθε δευτερόλεπτο παράγει 8000 δείγματα και κάθε δείγμα είναι 1 byte (8 bits), επομένως έχουμε 8000 bytes ανά δευτερόλεπτο, κάτι το οποίο συνεπάγεται ότι ένα buffer μεγέθους 4096 bytes αντιστοιχεί σε ~0.5 δευτερόλεπτο ήχου. Παρατηρήθηκε ότι το χαμηλό βάθος δειγματοληψίας (8-bit) και η χαμηλή συχνότητα (8000 Hz) περιορίζουν την ποιότητα του ήχου, με την ενδεχόμενη απώλεια ή καθυστέρηση πακέτων UDP και το μεγάλο μέγεθος του buffer (4096 bytes) να οδηγούν σε περιττή καθυστέρηση στην αποστολή δεδομένων, εισάγοντας θόρυβο. Συνοψίζοντας, η ποιότητα του ήχο ήταν χαμηλή και το latency μεγάλο.



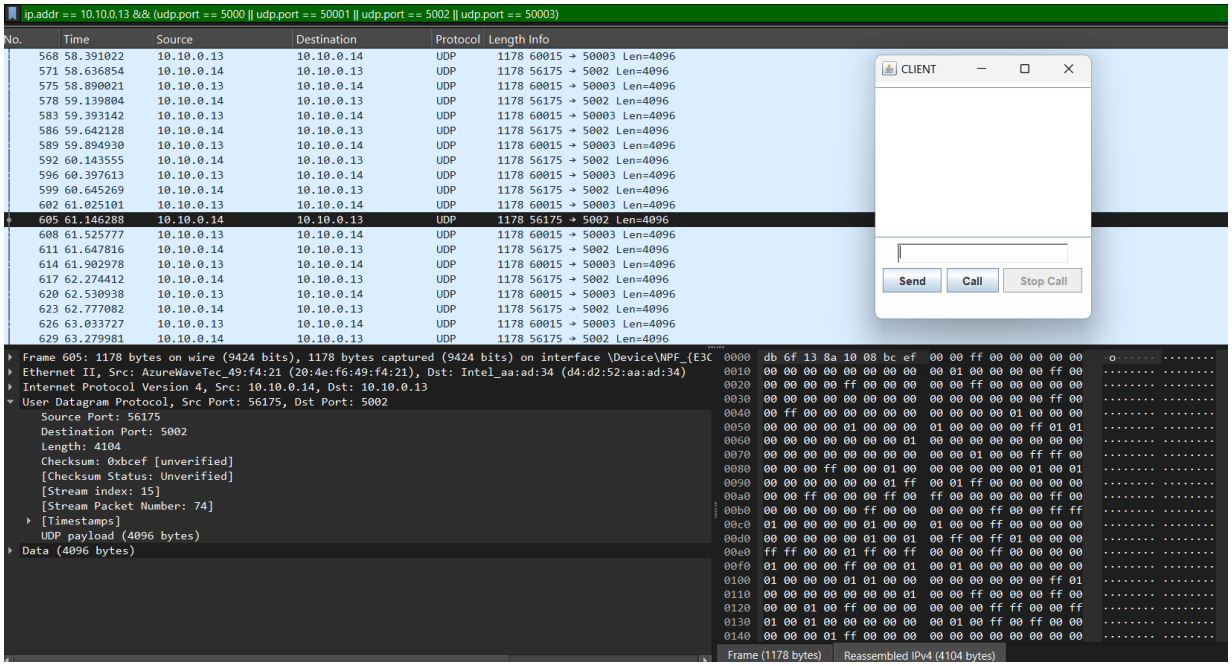


Figure 21: Audio Format (8000Hz, 8-bit samples), buffer size 4096 bytes, CLIENT's side

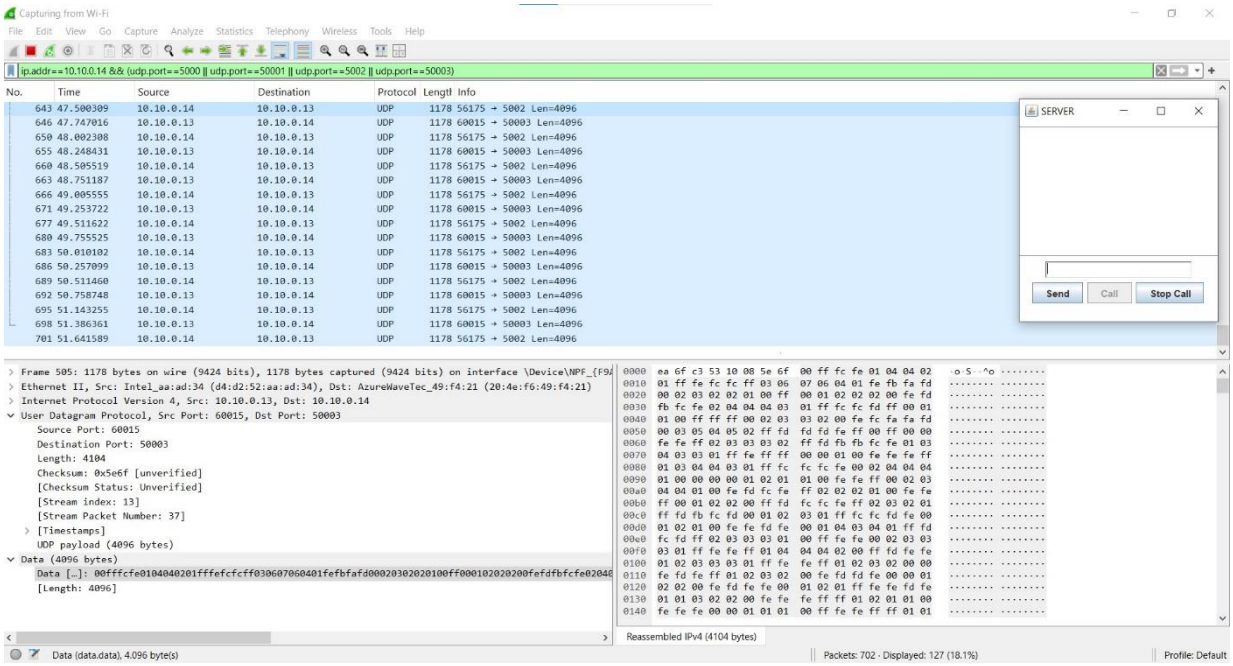


Figure 22: Audio Format (8000Hz, 8-bit samples), buffer size 4096 bytes, SERVER side

- Παρακάτω παρουσιάζονται εικόνες με χρήση του **Audio Format [44100Hz και 16-bit samples]** και **buffer size [4096 bytes]**. Ο ρυθμός δεδομένων είναι $44100 \times \frac{16}{8} = 88200$ bytes ανά δευτερόλεπτο. Ένα buffer 4096 bytes διαχειρίζεται δεδομένα διάρκειας $\frac{4096}{88200} \approx 0.046$ δευτερολέπτων (~46 ms). Αυτή η διάρκεια είναι αρκετά μικρή ώστε να διατηρείται χαμηλό το latency και αρκετά μεγάλη ώστε να απορροφά μικρές διακυμάνσεις στην εισαγωγή/αποστολή δεδομένων. Συνοψίζοντας, συγκριτικά με το προηγούμενο audio format, παρατηρήθηκε μεγάλη βελτίωση στην ποιότητα του ήχου και τεράστια μείωση στο latency.

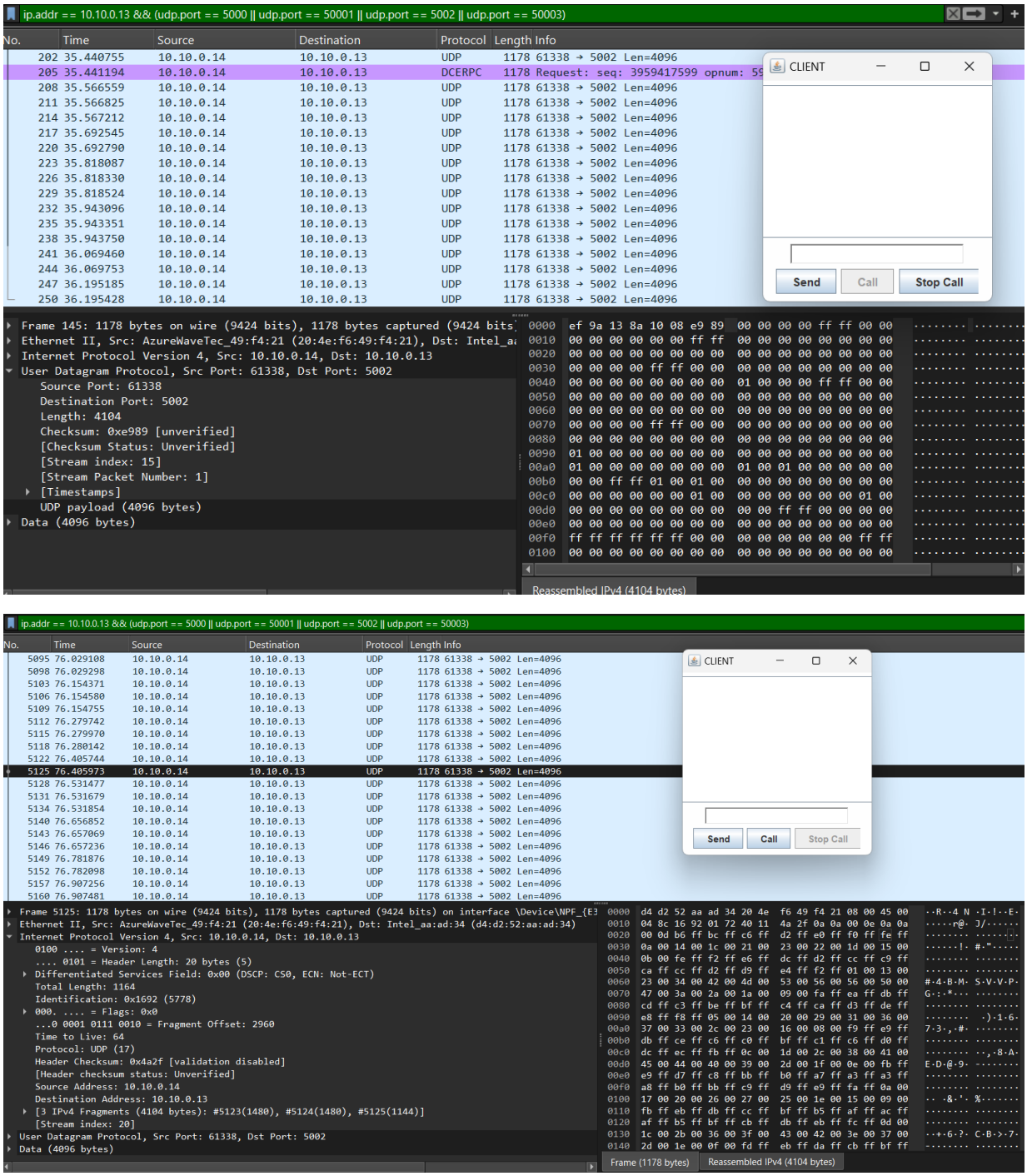


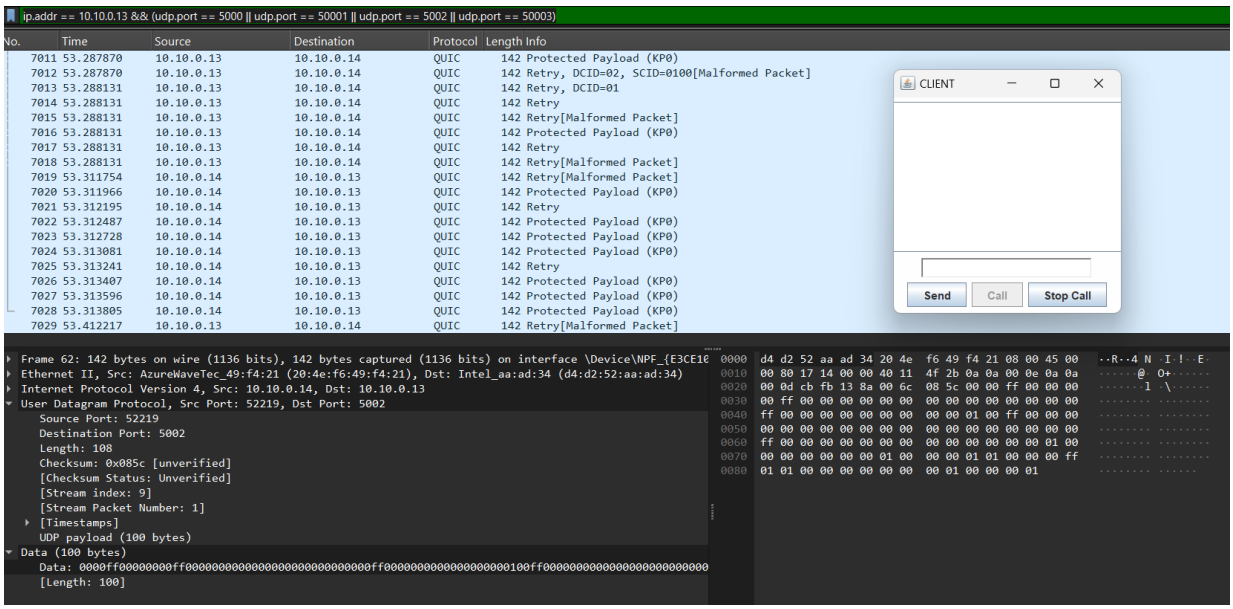
Figure 23: Audio Format (44100Hz και 16-bit samples), buffer size 4096 bytes, CLIENT's side

- Αλλάζοντας το **buffer size** σε **[100]** παρατηρήθηκε ότι και τα δύο audio formats που προαναφέρθηκαν είχαν παρόμοια ποιότητα ήχου και παρόμοιο latency με το audio format **[44100Hz και 16-bit samples]** να υπερτερεί του **[8000Hz και 8-bit samples].** Με 100 bytes, τα δεδομένα αποστέλλονται ή αναπαράγονται πιο συχνά (ανανέωση δεδομένων), μειώνοντας έτσι την καθυστέρηση (latency), καθώς ο χρόνος αναμονής για το γέμισμα του buffer είναι μικρότερος. Παράλληλα, ελαχιστοποιείται και η πιθανότητα θορύβου λόγω αναμονής.

  Με τα συγκεκριμένα χαρακτηριστικά, ένα ενδιαφέρον φαινόμενο που παρατηρήθηκε είναι η εμφάνιση του πρωτοκόλλου **QUIC,** όπως είναι φανερό στην κατηγορία *Protocol* του *main window* του *Wireshark*. Το QUIC είναι πρωτόκολλο βασισμένο στο UDP, το οποίο προσθέτει κάποιες επιπλέον λειτουργίες όπως κρυπτογράφηση, έλεγχο συμφόρησης κτλ. και χρησιμοποιείται κυρίως σε streaming apps (πχ YouTube).

  Ένας πιθανός λόγος που τα πακέτα φωνής καταγράφονται ως QUIC από το Wireshark αντί UDP, είναι το μέγεθός και η συνεχόμενη ροή τους, μιας και με buffer size [100] τα πακέτα UDP είναι πολύ μικρά και συχνά, στοιχεία τα οποία χαρακτηρίζουν τη ροή δεδομένων που χρησιμοποιεί το QUIC. Με αφαίρεση του βρόγχου *while(isCalling)* από τον κώδικα (στον ορισμό των νημάτων *sendVoiceThread* και *receiveVoiceThread*), έτσι ώστε να αποσταλεί και να ληφθεί ένα μόνο πακέτο φωνής (αντί για μία σειρά πακέτων), το Wireshark εμφανίζει τα πακέτα ως UDP. Άρα, φαίνεται πως ο dissection script, που χρησιμοποιείται για να αποφανθεί το είδος του πακέτου, πέρα από τους headers, το payload κτλ. κοιτάει ίσως και τον ρυθμό με τον οποίο έρχονται τα πακέτα (ροή).

  Εναλλακτικά θα μπορούσαμε να ρυθμίσουμε το Wireshark έτσι ώστε να μην αναγνωρίζει σε καμία περίπτωση QUIC πακέτα αποεπιλέγοντας το κουτάκι QUIC πηγαίνοντας στο *Analyze -> Enable Protocols* μέσα από την εφαρμογή.



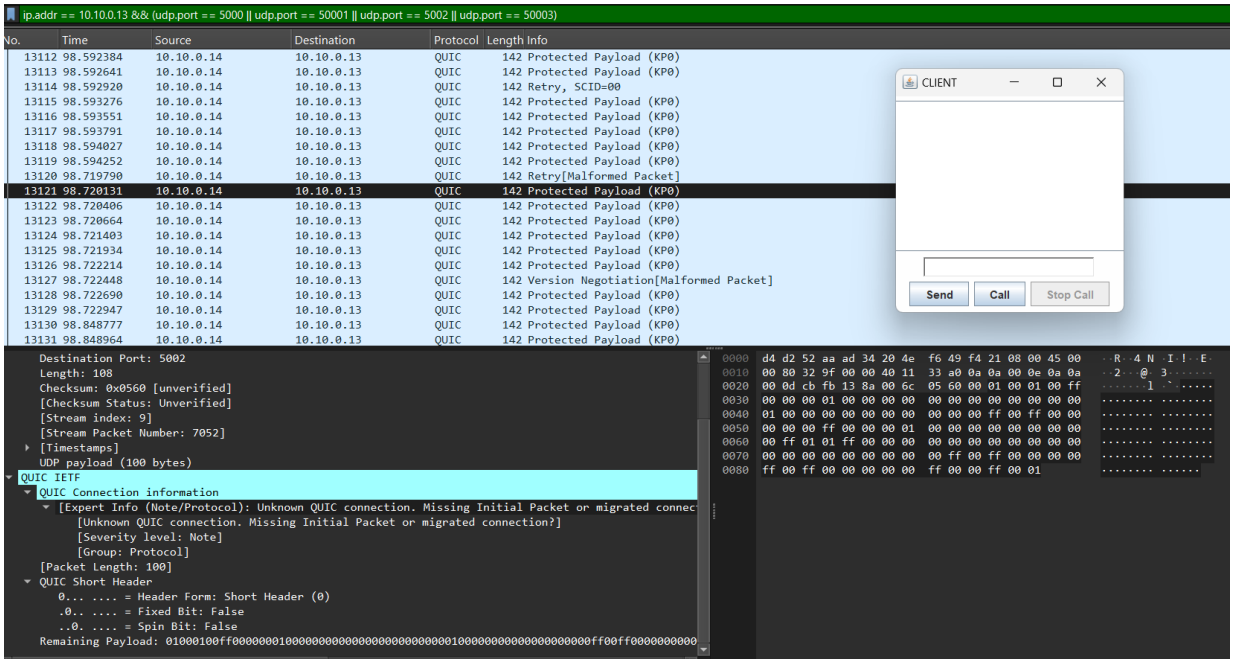


Figure 24: Χαρακτηρισμός των UDP πακέτων ως QUIC από το Wireshark, με buffer size 100 (τα πρώτα πακέτα τα εμφάνιζε ως UDP και μετά από κάποια στιγμή όσο συνέχιζαν να αποστέλλονται πακέτα τα κατέγραφε όλα ως QUIC)

Αντιθέτως, με buffer size [1024] και μεγαλύτερο, τα χαρακτηριστικά των πακέτων γίνονται πιο ξεκάθαρα και καταγράφονται εξαρχής ως UDP από το Wireshark.

- Η εικόνα εφόσον ένας από τους δύο χρήστες τερματίσει την κλήση είναι η εξής:

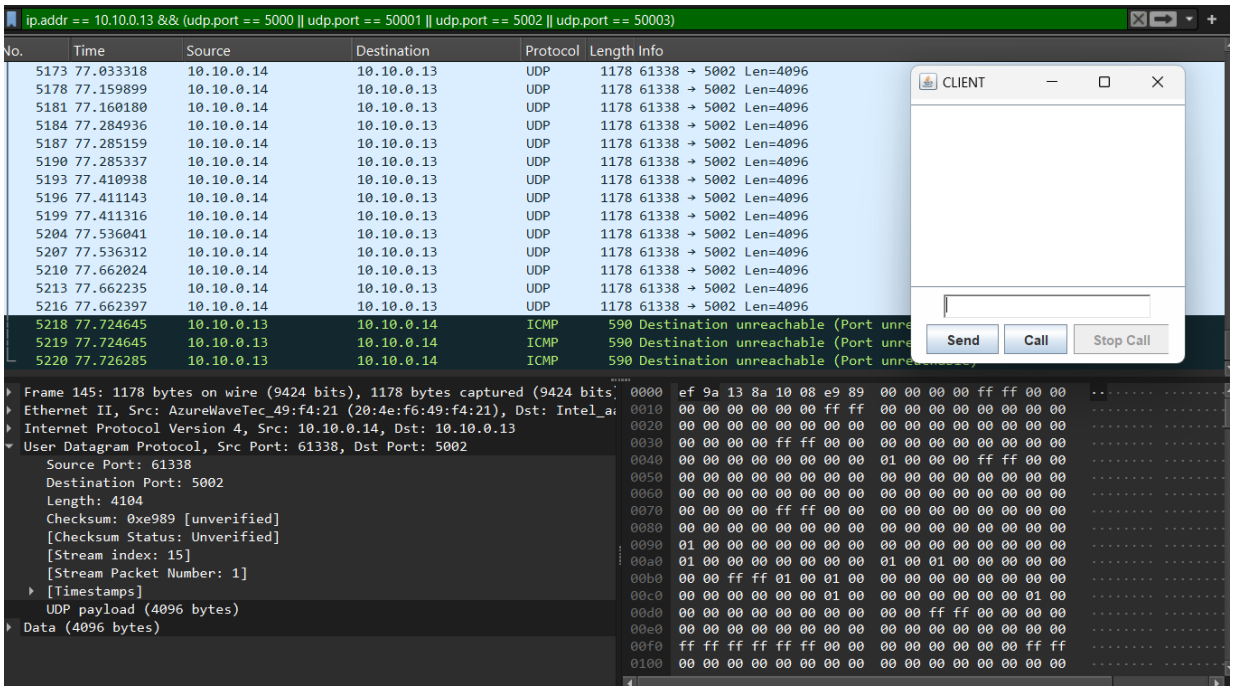


Figure 25: Ο χρήστης SERVER τερμάτισε την κλήση. Μεριά CLIENT: Destination Unreachable.

ΕΠΙΛΟΓΟΣ

Συνοψίζοντας, η εργασία μας εστίασε στην υλοποίηση μιας αποκεντροποιημένης εφαρμογής chat και VoIP (δεν βασίζεται σε έναν κεντρικό server), επιτυγχάνοντας αμφίδρομη επικοινωνία κειμένου και φωνής μέσω του πρωτοκόλλου UDP. Η ανάπτυξη αυτή μας βοήθησε να εμβαθύνουμε στις τεχνολογίες δικτύωσης της Java και να κατανοήσουμε τη σημασία της πολυνημάτωσης για την ασύγχρονη επικοινωνία, καθώς και τη χρήση αλγορίθμων κρυπτογράφησης για την ασφάλεια των δεδομένων.

Παράλληλα, ήρθαμε αντιμέτωποι με προκλήσεις όπως η διαχείριση του latency στις φωνητικές κλήσεις και η εξασφάλιση της συμβατότητας των συσκευών. Μελλοντικές βελτιώσεις θα μπορούσαν να περιλαμβάνουν την υιοθέτηση TCP για μεγαλύτερη αξιοπιστία, την ενσωμάτωση αλγορίθμων βελτιστοποίησης ποιότητας ήχου και τη δυνατότητα επέκτασης της εφαρμογής για χρήση μέσω του διαδικτύου.

Η εργασία αυτή αποτέλεσε ένα πολύτιμο πρακτικό μάθημα, συνδυάζοντας θεωρητικές γνώσεις με πραγματική ανάπτυξη λογισμικού, εμπλουτίζοντας τις δεξιότητές μας και παρέχοντας σημαντική εμπειρία στον τομέα των peer-to-peer εφαρμογών.