

# **Redes de Computadoras:**

## **Proyecto Final**

*Creación e implementación de un protocolo de la capa de Aplicación*

M. CONCHA VÁZQUEZ  
A. FLORES MARTÍNEZ  
A.A. GLADÍN GARCÍA

Facultad de Ciencias, UNAM

**Título del proyecto:**

Proyecto Final

**Tema del proyecto:**

Modelo *OSI*; Modelo *TCP/IP*; Modelo de cinco capas; capa de Aplicación; datos; transmisión de mensajes; orientado a conexión; concurrencia; persistencia de datos; Python; *sockets*; cliente; servidor; *pokémon*; *Wireshark*; captura de tráfico.

**Periodo:**

Diciembre, 2018.

**Número de Grupo:**

7003

**Participantes:**

Concha Vázquez Miguel.  
Flores Martínez Andrés.  
Gladín García Ángel Iván.

**Supervisores:**

Profesor Paulo Santiago de Jesús Contreras Flores.  
Virgilio Castro Rendón  
José Daniel Campuzano Barajas.

**Fecha en que se completó el proyecto:**

16 de diciembre del 2018.

**Facultad de Ciencias**

Universidad Nacional Autónoma de México  
<http://www.ciencias.unam.mx/>

**Resumen:**

Se presenta la implementación de un protocolo de la capa de Aplicación para la captura de *Pokémones*. El cliente interactúa con el servidor luego de iniciar sesión y tiene un número de intentos para capturar aleatoriamente el *pokémon* ofrecido por el servidor, en cuyo caso de éxito recibirá una imagen del *pokémon* en cuestión que es guardada en su equipo.

Se pretende con este proyecto ahondar en los conceptos de la capa de Aplicación y los *sockets*, por medio de los cuales es posible que esta recibe y envíe datos. Además, se profundiza en los conceptos de concurrencia al manejar hilos (*threads*) y persistencia de datos.

# Índice general

<b>1. Objetivo</b>	<b>2</b>
1.1. Fin y misión . . . . .	2
<b>2. Detalles Técnicos</b>	<b>3</b>
2.1. Implementación del protocolo . . . . .	3
2.1.1. Explicación . . . . .	3
2.1.2. Tabla de estados . . . . .	4
2.1.3. Códigos de mensajes transmitidos . . . . .	4
2.1.4. <i>FSM</i> . . . . .	7
2.2. Capturas de tráfico en <i>wireshark</i> . . . . .	7
<b>3. Interpretación, ejecución y documentación</b>	<b>23</b>
3.1. Instrucciones . . . . .	23
3.2. Documentación del código . . . . .	23
<b>4. Conclusiones</b>	<b>24</b>

# Capítulo 1

## Objetivo

### 1.1. Fin y misión

La empresa pretende entregar un protocolo de calidad que sirva a los usuarios frente a la cada vez menos usada Aplicación *Pokémon Go*® con tal de adentrarse en el mundo de los *pokémons* a la par que profundiza en una comunicación más directa con un servidor. Por medio del pago de una módica cantidad de \$9,99 pesos mensuales podrá formar parte de los cientos de entrenadores *pokémon* que se están sumando a esta nueva aventura, además de que no tendrá que pagar para poder competir y ser el primero en capturarlos todos.

Estamos comprometidos con proporcionar experiencias nuevas, rediseñando los juegos más populares de los últimos tiempos en protocolos de calidad y orientados a conexión, sirviendo a nuestros usuarios y estando a su disposición en cualquier momento.



# Capítulo 2

## Detalles Técnicos

### 2.1. Implementación del protocolo

#### 2.1.1. Explicación

En este protocolo de la capa de aplicación se pretende presentar una interacción en la comunicación entre varios clientes (conurrencia) y un servidor. Este último recibirá solicitudes de juego por parte de usuarios, mismos que deberá iniciar sesión con un ID y una contraseña. Luego, el servidor le ofrecerá aleatoriamente un *pokémon* a capturar que los clientes deberán decidir si quieren o no intentar atrapar. En caso de que la respuesta sea positiva y no se cuenta aún con el *pokémon*, se procede a intentar varias veces capturarlo hasta que se tenga éxito —en cuyo caso el servidor debe transferir la imagen al cliente del *pokémon* capturado— o bien se agoten los intentos y en ambos casos se procede luego al cierre de la sesión y terminación de la conexión.

El protocolo fue implementado con el módulo de `socket` con el lenguaje de programación orientado a objetos Python en su versión 3.7; además, se optó por el uso de un diccionario y el módulo de `pickle` para hacerlo persistente como mecanismo para ir guardando el estado global del juego. Debido a que se implementó un servidor multi-hilo, varios usuarios pueden conectarse para jugar al mismo tiempo y se extendieron las funcionalidades básicas presentadas en el PDF del proyecto con nuevos mensajes y códigos —tanto de error como de flujo normal de ejecución— para permitir el inicio de sesión y cierre de esta luego de cerrar sesión, no permitiendo que dos clientes jueguen así con el mismo usuario al mismo tiempo.

En caso de querer ahondar en los detalles del código, recomendamos ver la documentación generada con Sphinx o bien adentrarse de lleno en el código fuente de:

- `src/client.py`: Funciones e implementación de los estados para el cliente.
- `src/server.py`: Funciones e implementación de los estados para el servidor.
- `src/message_codes.py`: Especificación de los mensajes y sus códigos únicos.
- `src/init_db.py`: Script de inicialización de la base de datos implementada con un diccionario de Python, mismo que ya está protegido para la concurrencia.

### 2.1.2. Tabla de estados

Estado	Descripción
$s_0$	Estado inicial. Desde aquí se inicia la conexión del protocolo de la capa de aplicación y el servidor manda al cliente un mensaje de bienvenida. El cliente despliega en pantalla el banner del juego.
$s_1$	El servidor manda al cliente la pregunta acerca de si se desea participar en el juego.
$s_2$	El cliente responde la pregunta al servidor.
$s_3$	Terminando la sesión.
$s_4$	Cierre de sesión.
$s_5$	Se recibe la respuesta afirmativa a jugar.
$s_6$	El cliente recibe la lista de entrenadores pokémon registrados para proceder a elegir el ID del que le corresponde.
$s_7$	El servidor recibe el ID por parte del cliente y verifica si es válido dentro de su BD.
$s_8$	El cliente recibe la alerta de que el ID no fue encontrado en la BD.
$s_9$	El cliente recibe la alerta de que el usuario cuyo ID fue elegido ya se encuentra activo.
$s_{10}$	El cliente recibe la confirmación de que el ID solicitado es válido y todavía no estaba activo. El servidor especifica que ahora dicho cliente está activo.
$s_{11}$	El servidor recibe la contraseña del usuario por parte del cliente y verifica que coincida de acuerdo a la información de la BD.
$s_{12}$	El cliente recibe la información de que la contraseña provista no coincide con la del usuario de la BD de acuerdo al ID proporcionado antes.
$s_{13}$	El cliente recibe la confirmación acerca de la contraseña provista; se ha iniciado sesión exitosamente.
$s_{14}$	El cliente recibe la lista de los pokémones que ha capturado al momento presente y la despliega en pantalla.
$s_{15}$	El servidor recibe la solicitud del cliente de que quiere capturar un pokémon.
$s_{16}$	El servidor elige aleatoriamente un pokémon a capturar y manda la información al cliente.
$s_{17}$	El cliente decide si quiere o no capturar al pokémon sugerido.
$s_{18}$	El servidor se percata de que el cliente ya tiene todos los pokémones existentes en esta versión del juego.
$s_{19}$	El servidor se da cuenta de que el cliente ya tiene al pokémon sugerido.
$s_{20}$	El servidor ve que el cliente no cuenta con el pokémon sugerido. El servidor inicia un contador $\{n\}$ como el máximo número de intentos. Aleatoriamente indica si se capturó el Pokémon o no.
$s_{21}$	El cliente da respuesta para reintentar captura de pokémon.
$s_{22}$	Se recibe el pokémon capturado.

### 2.1.3. Códigos de mensajes transmitidos

Código	Descripción
WELCOME (1)	Servidor al cliente. Mensaje de bienvenida.
PLAYING_QUESTION (2)	Servidor al cliente. Pregunta para ver si quiere jugar o no.
TRAINER_LIST (3)	Servidor al cliente. Lista de los entrenadores pokémon registrados. Se presenta su ID y nombre.
CHOSEN_ID (4)	Cliente al servidor. ID del entrenador pokémon elegido.
PASSWORD (5)	Cliente al servidor. Envío de contraseña del usuario previamente elegido.
DO_NOT_HAVE_POKEMON (6)	Servidor al cliente. Notifica que no cuenta con el pokémon sugerido.
REQUEST_CAPTURING (10)	Cliente al servidor. Solicita captura de un pokémon.
CAPTURING_VERIFICATION (20)	Servidor al cliente. Envío de la información del pokémon que se sugiera para la captura.
REMAINING_ATTEMPTS (21)	Servidor al cliente. Intentos que le quedan para capturar al pokémon en cuestión.
CAPTURED_POKEMON (22)	Servidor al cliente. Envía pokémon (imagen) capturado.
NO_MORE_ATTEMPTS (23)	Servidor al cliente. Indica que ya se le agotaron los intentos para la captura.
YES (30)	Mensaje común para cliente y servidor. Sí.
NO (31)	Mensaje común para cliente y servidor. No.
TERMINATED_SESSION (32)	Servidor al cliente. Terminando sesión.
ID_NOT_FOUND (41)	Servidor al cliente. El ID del usuario solicitado no está en la BD.
PASS_NO_MATCH (42)	Servidor al cliente. La contraseña solicitada no corresponde al usuario elegido.
ALREADY_HAVE_POKEMON (43)	Servidor al cliente. El cliente ya cuenta con el pokémon sugerido en su pokédex.
ALREADY_HAVE_ALL (44)	Servidor al cliente. El cliente ya cuenta con todos los pokémones registrados en la BD.
ACTIVE_USER (45)	Servidor al cliente. El usuario elegido está ya jugando.
TIMEOUT (46)	Servidor al cliente. Se agotó el tiempo de espera por la respuesta del cliente.
IMAGE RECEIVED (33)	Cliente al servidor. Indica que recibió exitosamente la imagen proporcionada del pokémon capturado.

La conformación de los mensajes es la siguiente (en todos se antepone el código del mensaje correspondiente):

- En general (1, 2, 6, 10, 23, 30, 31, 32, 33) y para todos los mensajes de error (41, 42, 43, 44, 45, 46) los mensajes transferidos constan de un solo *byte* (ocho *bits*) que son la representación en binario en ocho *bits* del código en cuestión de cada caso. Esto es:

código
1 byte

- Para el mensaje de envío de la lista de entrenadores pokémon que sirve para que el cliente pueda elegir el ID del cual se iniciará sesión, la longitud del mensaje evidentemente depende de la longitud de la representación en cadena de dicha lista. Por ende, el mensaje sería:

código	Cadena de lista de entrenadores pokémon
1 byte	$k$ bytes

- Para el mensaje que envía el cliente al servidor luego de elegir un ID, se envía el código y un *byte* con el binario del ID elegido. Esto quiere decir que en esta versión se soporta a un máximo de  $2^8 = 256$  entrenadores pokémon registrados. Así sería:

código	ID del entrenador pokémon
1 byte	1 byte

- De manera similar al código 4, cuando se envía la contraseña al servidor, el mensaje estaría conformado así:

código	Contraseña del entrenador pokémon
1 byte	$k$ bytes

- En el caso en que el servidor envía al cliente el pokémon sugerido para la captura, envía también el ID del pokémon<sup>1</sup> y su nombre codificado en *bytes*, de forma tal que el mensaje sería:

código	ID del pokémon	Nombre del pokémon
1 byte	1 byte	$k$ bytes

- Cuando el servidor avisa al cliente de cuántos intentos le quedan para la captura del pokémon, el mensaje posee la siguiente estructura:

código	ID del pokémon	Intentos que quedan
1 byte	1 byte	1 byte

- Finalmente, cuando se envía la imagen del pokémon capturado al cliente, se tiene el siguiente esqueleto en el mensaje transmitido:

código	ID del pokémon	Tamaño de la imagen	Imagen del pokémon capturado
1 byte	1 byte	4 bytes	$k$ bytes

Nótese que en ningún caso fue necesario añadir al mensaje con el código o identificador del entrenador pokémon que ha iniciado sesión, pues una vez iniciada ésta, el servidor guardará como parte de la conexión dicho valor y sabrá ya en qué entrada de la BD tendrá que ir actualizando los valores al cerrar la sesión o añadir a la lista de pokémones capturados el que se atrape en caso de éxito en el juego.

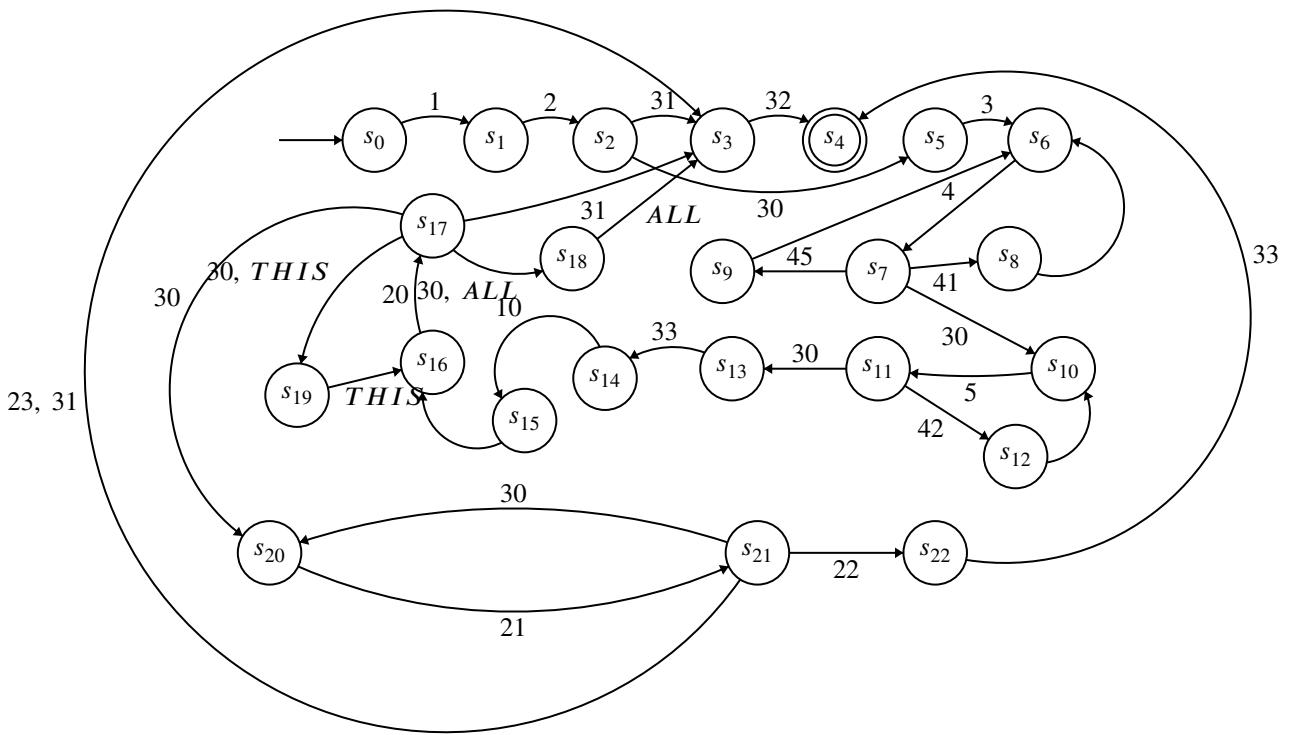
---

<sup>1</sup>Se tiene así un soporte por ahora para un máximo de  $2^8 = 256$  pokémones.

### 2.1.4. FSM

Las transiciones contienen los códigos de los mensajes que hacen pasar de un estado a otro, siendo el cierre de la conexión (estado  $s_4$ ) el estado final y  $s_0$  el inicial. En el estado  $s_{17}$  se presenta una bifurcación dependiendo de si el cliente cuenta o no con el pokémon presentado —comprobación llevada a cabo por el servidor—, haciendo que dependiendo de este hecho se transite al estado  $s_{18}$ ,  $s_{19}$  o bien al  $s_{20}$  para continuar con la captura.

Dado que en cualquier momento puede ocurrir un **timeout** en la conexión, se revisa en todo momento el código de respuesta del servidor al cliente, viendo si corresponde al código de error 46, lo cual le indica al cliente que la conexión fue cerrada por el *timeout* en la espera de su respuesta. Dichas transiciones no fueron incluidas en el autómata por claridad, pero las transiciones a dicho estado de error y el posterior término de sesión y cierre de conexión están implícitas.



## 2.2. Capturas de tráfico en wireshark

En esta sección, mostramos como se realiza el intercambio de mensaje entre el cliente y el servidor durante una ejecución de la aplicación a través de Wireshark. Para ello se realizará una simulación de ejecución y colocaremos un filtro en el puerto 9999 de la máquina en la que se localiza el servidor (en este caso, tanto el cliente como el servidor son ejecutados de manera local). Cada una de las imágenes tendrá seleccionado (y por ende remarcado en

color azul) el paquete relevante del que se habla en ese momento. Lo primero que se realiza durante el intercambio de paquetes es el three way handshake una vez que el cliente comienza la conexión con el servidor, lo cual se muestra en los primeros tres paquetes en la siguiente captura de wireshark.

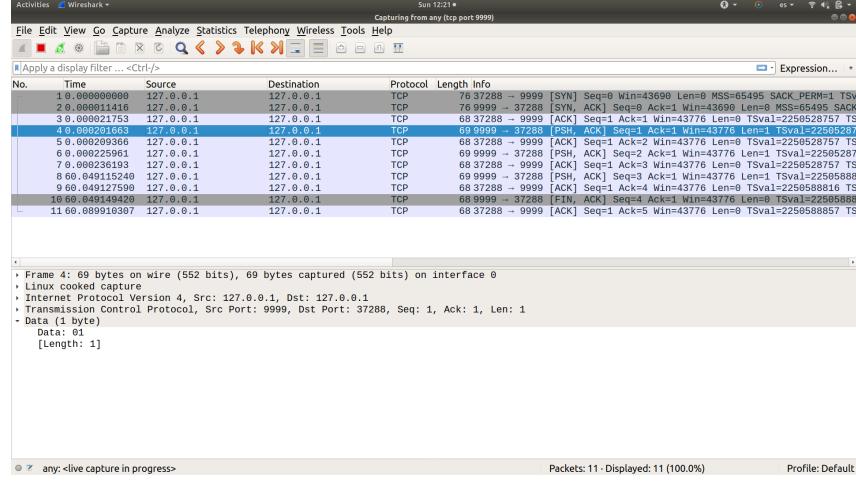


Figura 2.1: Inicio de la conexión: three way handshake.

Una vez establecida la conexión, el servidor envía un mensaje de bienvenida al cliente, indicando que la interacción con la aplicación ha comenzado. Este paquete de bienvenida está codificado como 01, en su valor en bytes. Esto se puede observar en el paquete número 4 de la siguiente captura.

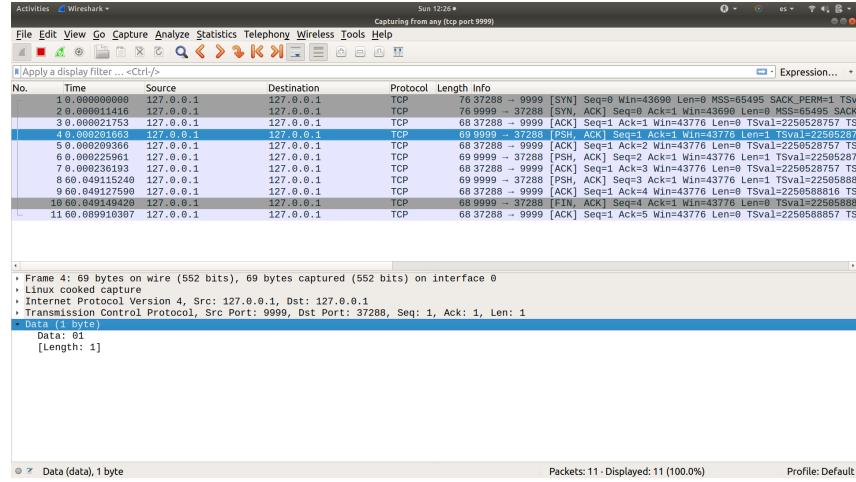
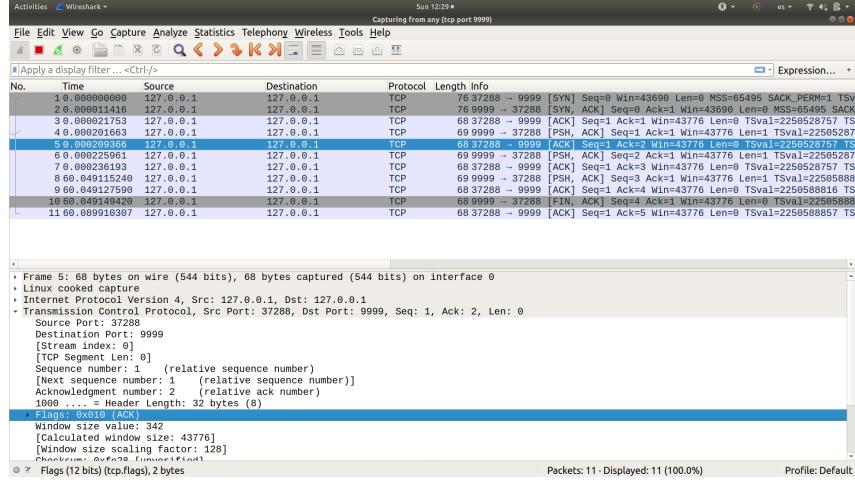


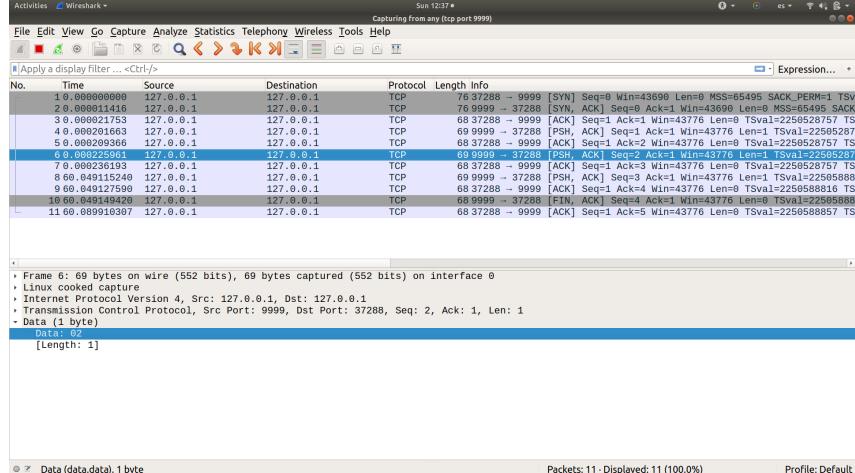
Figura 2.2: El servidor envía mensaje de bienvenida al cliente.

Una vez que el cliente recibe el paquete, envía una ACK al servidor para confirmar el recibimiento correcto del paquete. Esto se puede observar en el paquete número 5 de la captura de wireshark.



**Figura 2.3:** El cliente realiza una confirmación de recibo correcto a través de un ACK.

A continuación, en el inicio de la aplicación el servidor pregunta al cliente si desea atrapar un pokémon. Dicha pregunta se le hace al cliente a través del código 02, lo cual capturamos en wireshark en el paquete número 6.



**Figura 2.4:** El servidor pregunta al cliente si desea jugar.

Y ahora el cliente, una vez recibido el código, confirma al servidor a través de un ACK que el mensaje fue recibido correctamente, lo cual se muestra en el paquete número 7 de la captura.

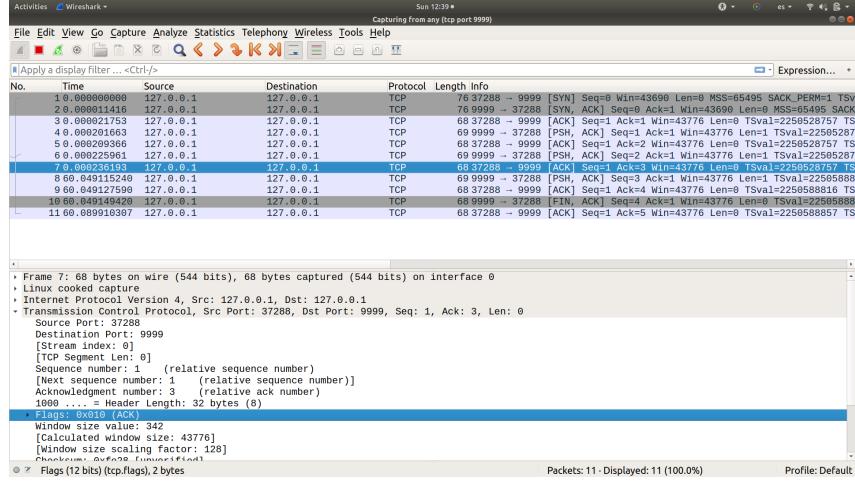


Figura 2.5: El cliente confirma la recepción del código con un ACK.

Ahora, el servidor espera una respuesta del cliente que será o bien que sí desea atrapar un pokémon o que no. Nuestra aplicación implementa el manejo de timeouts a través de envío de paquetes, para dar prueba de ello, en la simulación hicimos que el cliente no diera respuesta al servidor, lo cual provocó un timeout debido a que el servidor no recibe respuesta por parte del cliente en un tiempo predeterminado. Lo cual se muestra a continuación, pues el siguiente paquete capturado de la conexión, el paquete número 8, corresponde a un mensaje enviado por parte del servidor al cliente con el código 2e, correspondiente a un timeout.

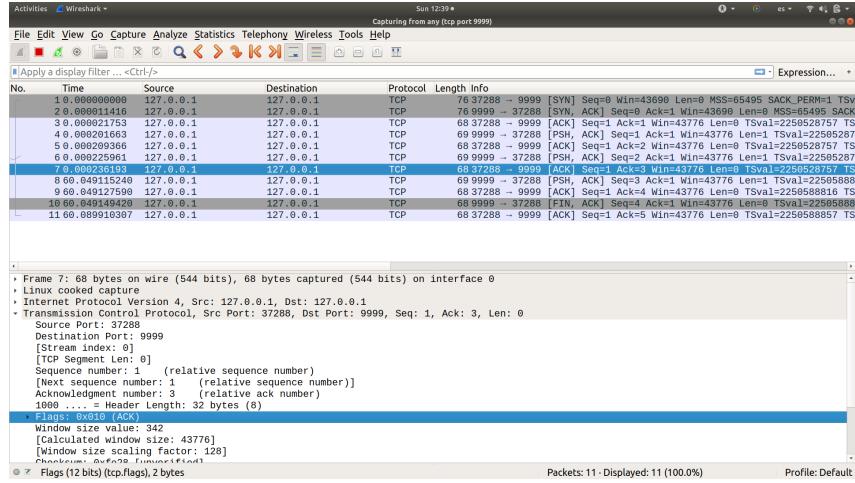


Figura 2.6: El servidor envía un código de timeout al cliente.

```

aflores@aflores-inspiron:~/Documents/7moSemestre/Redes/Proyecto2/NetworkPokemon
File Edit View Search Terminal Help
Conexión cerrada.
aflores@aflores-inspiron:~/Documents/7moSemestre/Redes/Proyecto2/NetworkPokemon$ make run_pokemon_client
Socket creation complete
Connected to localhost
Quieres capturar un pokémon? (y/n): 
Quieres capturar un pokémon? (y/n): n
buscar b
Sesión terminada: timeout
Conexión cerrada.

```

Figura 2.7: Ocurre un timeout durante la ejecución de la aplicación.

Una vez recibido el mensaje correspondiente al timeout, se comienza con el intercambio de mensajes para llevar a cabo la finalización de la conexión, como se puede mostrar en la captura de los paquetes restantes en la conexión en los paquetes 10 y 11.

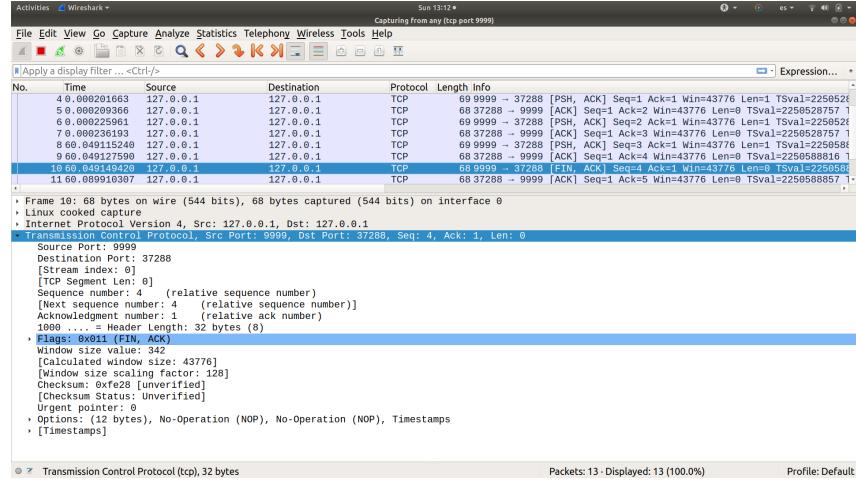
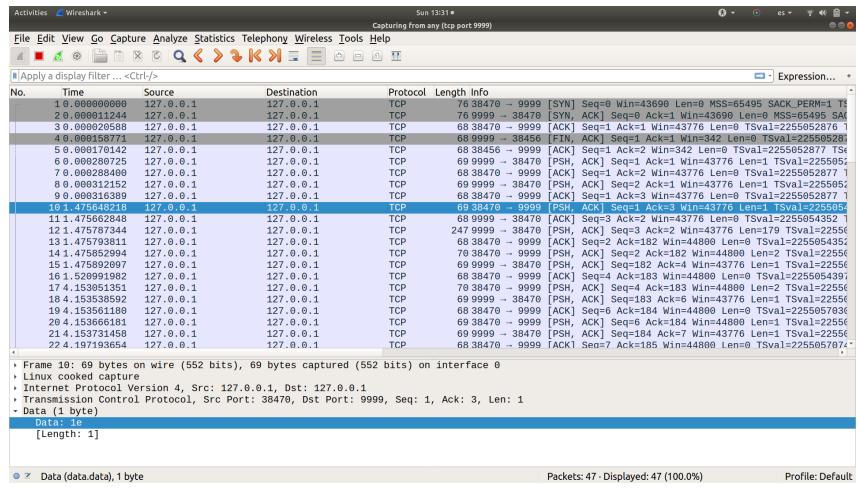


Figura 2.8: Se realiza la finalización de la conexión.

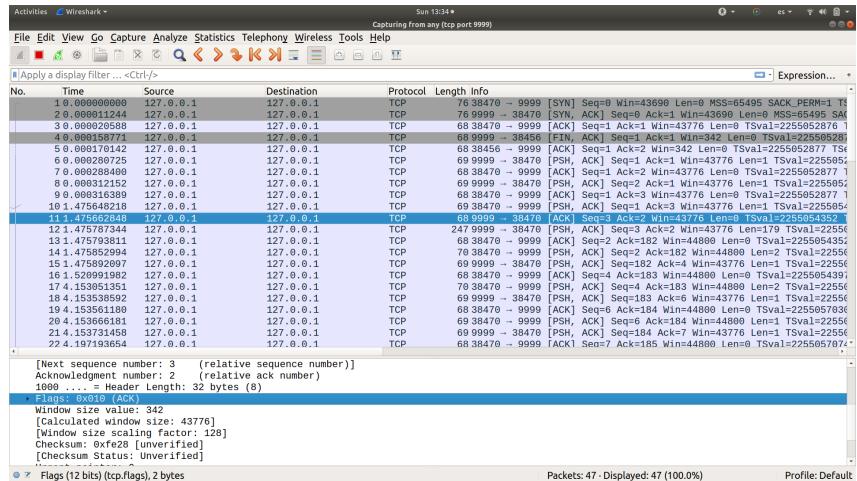
Lo anterior fue el caso en el que en cualquier punto de la conexión ocurre un timeout a lo largo de la interacción entre el servidor y alguno de sus clientes. Ahora llevamos a cabo la simulación del resto de la aplicación sin timeouts para ver como se comporta la conexión. Reanudando (en una conexión nueva) en el momento en el que el servidor le pregunta al cliente si desea atrapar un pokémon, a lo que el cliente confirma con un ACK. A continuación, el cliente envía la respuesta a la pregunta, que en este caso es sí<sup>2</sup> codificado con el número 30, correspondiente a 1e en hexadecimal.

<sup>2</sup>en caso contrario, el servidor iniciaría con el proceso de desconexión como se mostró anteriormente cuando hay un timeout



**Figura 2.9:** El cliente responde a la pregunta enviada por el servidor con el código 1e, afirmando que desea atrapar un pokémon.

El servidor recibe la respuesta del cliente y se lo confirma a través de un ACK.

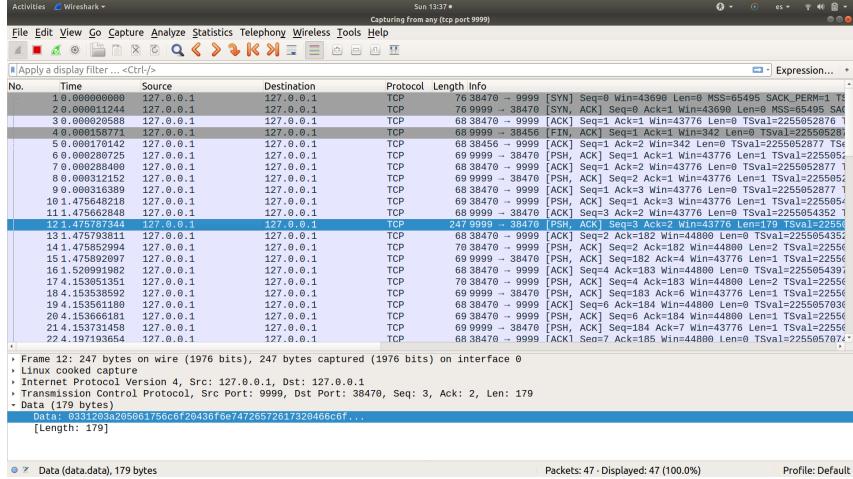


**Figura 2.10:** El servidor confirma la recepción del SÍ del cliente mediante un ACK.

Ahora, dado que el servidor recibió que el cliente sí desea atrapar un pokémon, entonces procede con el inicio de sesión del parte del cliente, y para ello primero envía una lista de los maestros pokémon disponibles. El paquete enviado para esto contiene en el primer byte el código 03, que indica al cliente que la información que contiene el paquete es la lista de entrenadores; el resto de bytes que contiene el paquete es la información de los maestros pokémon. Esto se puede apreciar en el paquete número 12 de la captura.

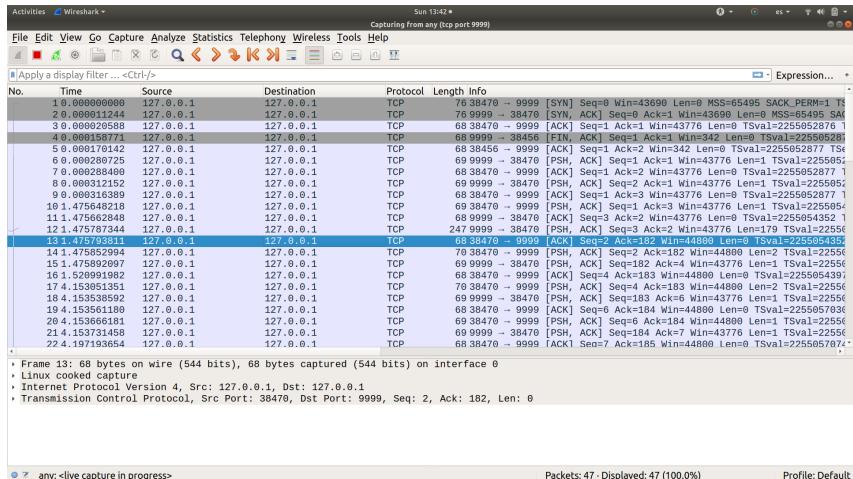
## 2.2. Capturas de tráfico en wireshark

13



**Figura 2.11:** El servidor envía la lista de maestros pokémon disponibles al cliente. Se puede comprobar que el primer byte es 03.

Ahora, el cliente confirma a través de un ACK la recepción del paquete.



**Figura 2.12:** El cliente confirma la recepción del paquete.

Lo siguiente, es que el cliente decide el id del pokémon que desea usar. Para ello, envía un mensaje con el código 04 que indica al servidor que el resto de la información corresponde al id seleccionado.

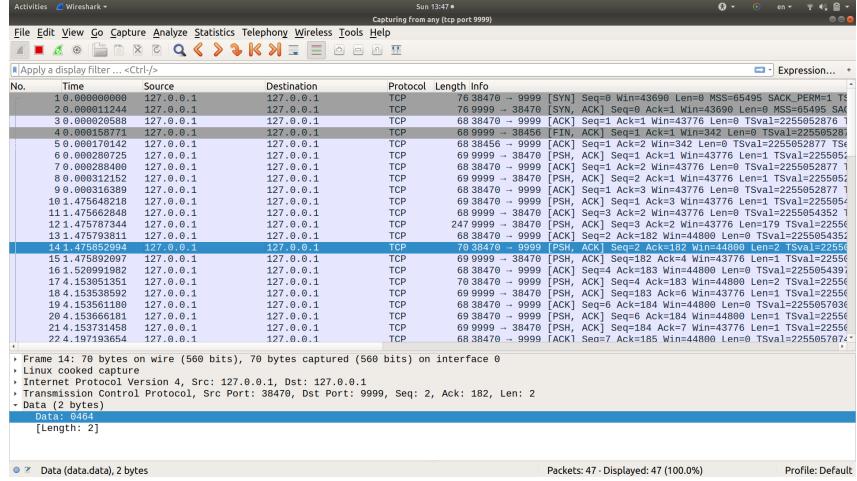


Figura 2.13: El cliente envía el id del maestro pokémon con el que desea iniciar sesión.

Dado que el id enviado por el cliente no está dentro de los disponibles, entonces el servidor le indica a través de un paquete con código 41 (hexadecimal 29) de ID\_NOT\_FOUND que el id enviado por parte del cliente no fue encontrado, como se puede observar en la siguiente captura.

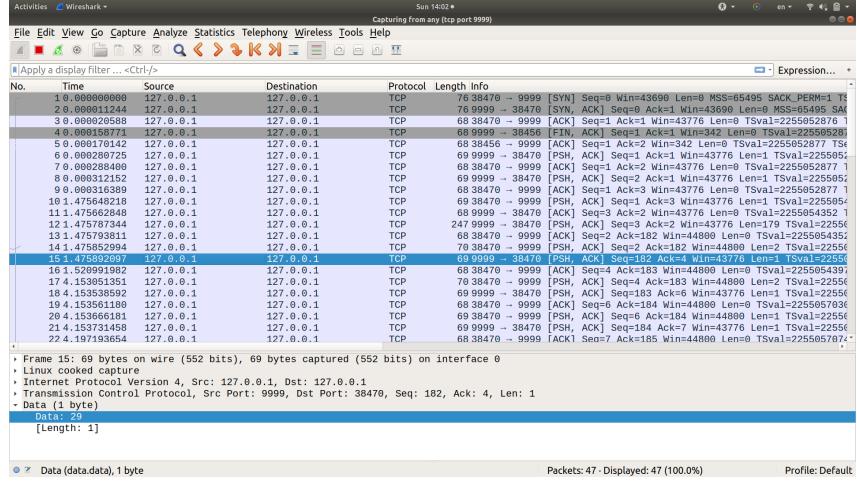


Figura 2.14: El indica al cliente que el id no fue encontrado.

Una vez que el cliente recibe el dicho mensaje entonces envía un ACK de que recibió ese mensaje y en otro paquete envía el nuevo id con el que desea iniciar sesión, como muestran los siguientes dos paquetes.

## 2.2. Capturas de tráfico en wireshark

15

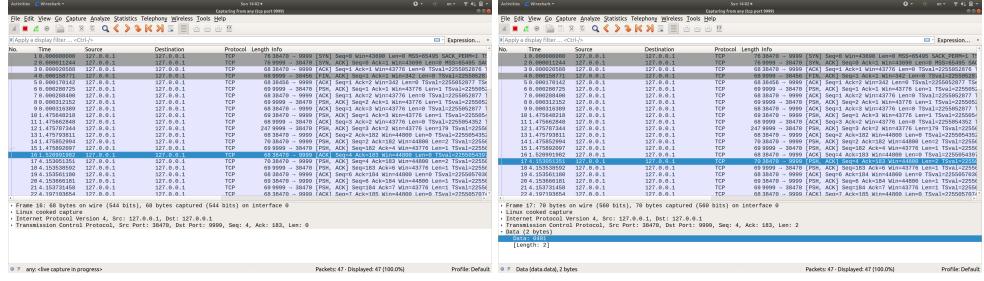


Figura 2.15: El cliente envía un ACK confirmando el mensaje de id no encontrado, y escoge un nuevo id y lo envía.

Ahora, el servidor sí encuentra el id como disponible y envía un mensaje con código 30 indicando que sí encontró el id.

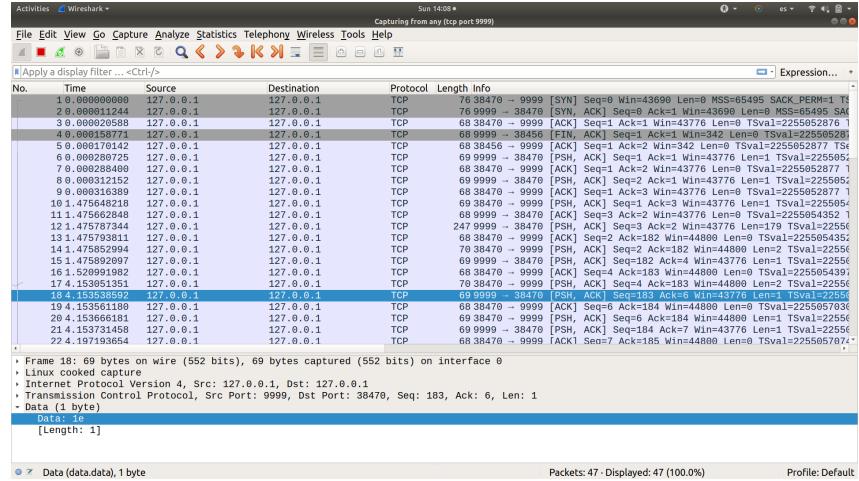
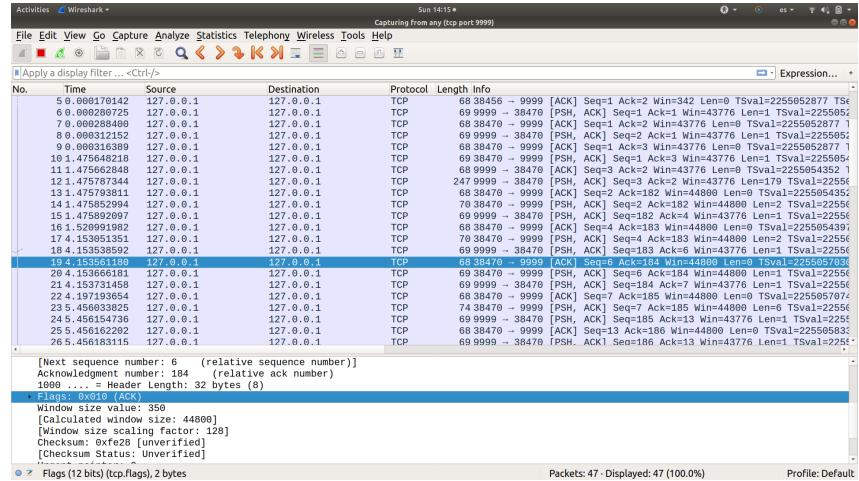


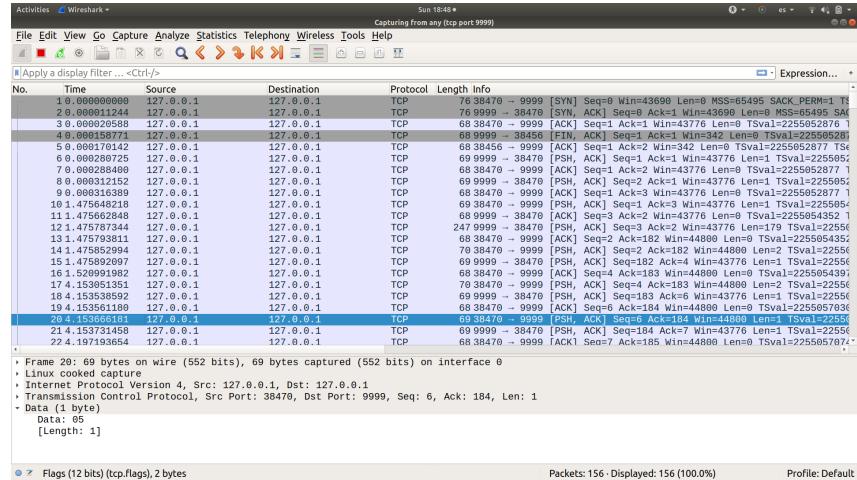
Figura 2.16: El servidor indica al cliente que el id fue encontrado.

A continuación, el cliente envía una confirmación mediante un ACK.



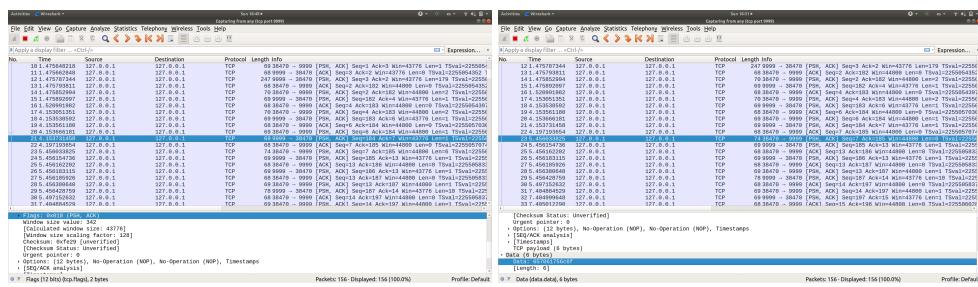
**Figura 2.17:** El cliente confirma la recepción del mensaje de id encontrado por parte del servidor.

A continuación, el cliente debe de ingresar una contraseña para poder ingresar, para ello, envía un paquete con el código 05 correspondiente a PASSWORD al servidor indicándole que enviará la contraseña seguido de la contraseña en el mismo paquete, que en este caso el cliente deja vacío (de forma que el servidor le indicará que hubo un error).



**Figura 2.18:** El cliente informa al servidor que enviará la contraseña.

A continuación, el servidor le indica al cliente que la contraseña no coincide; por lo cual despues el cliente vuelve a enviar un paquete de contraseña ahora con la contraseña correcta.



**Figura 2.19:** El servidor indica que hubo un error con la contraseña y el cliente la vuelve a enviar.

Dado que ahora la contraseña sí fue la correcta, el servidor envía un mensaje con el código 30 (1e en hexadecimal) indicando que la contraseña ingresada fue correcta, a su vez el cliente manda una ACK para confirmar que recibió dicho mensaje.

## 2.2. Capturas de tráfico en wireshark

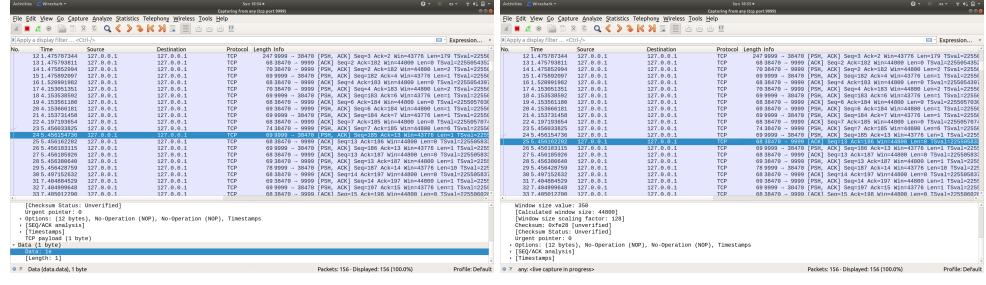


Figura 2.20: El servidor indica que la contraseña fue correcta, y el cliente confirma la recepción del mensaje.

Una vez iniciada la sesión, el servidor le envía al cliente la lista de los pokemones capturados por el usuario con el que inicio sesión (que en este caso es vacía porque es la primera vez que se inicia sesión con ese usuario), una vez que el cliente recibe el paquete, envía un ACK para confirmar la recepción.

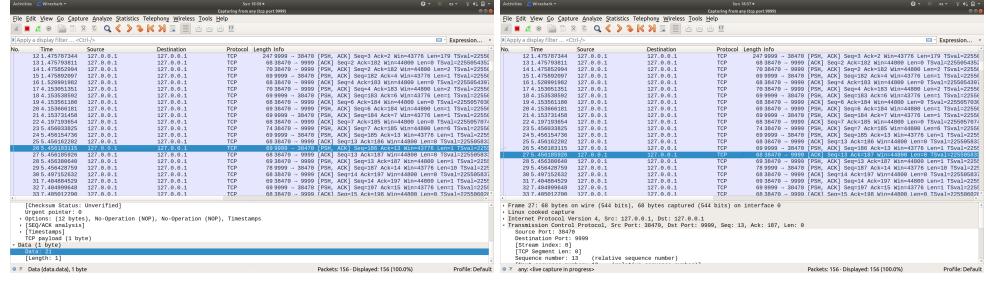


Figura 2.21: El servidor envía la lista de pokemones, y el cliente confirma la recepción del mensaje.

A continuación, el cliente hace una solicitud al servidor para comenzar la captura de un pokémon. Esto lo hace a través del código 10 REQUEST CAPTURING (0a en hex).

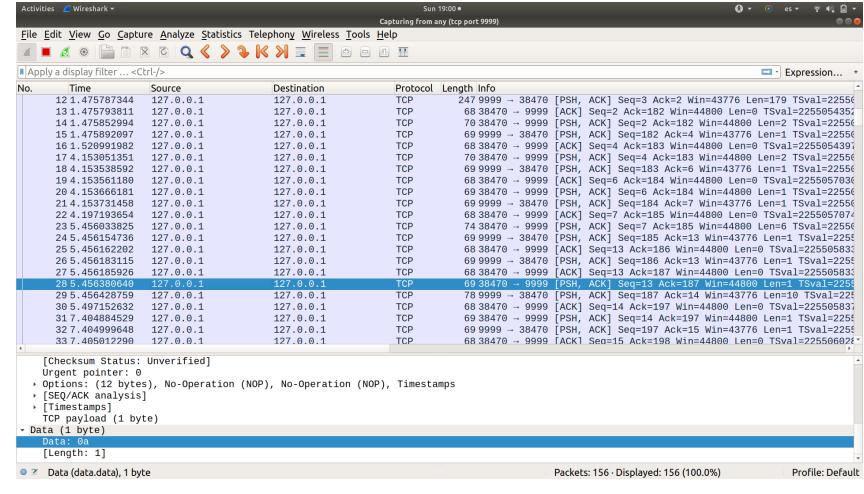


Figura 2.22: El cliente solicita atrapar pokémon.

Una vez que el servidor recibe la solicitud, procede a enviar la información de una sugerencia de pokémon para atrapar al cliente a través del código 20 CAPTURING VERIFICATION.

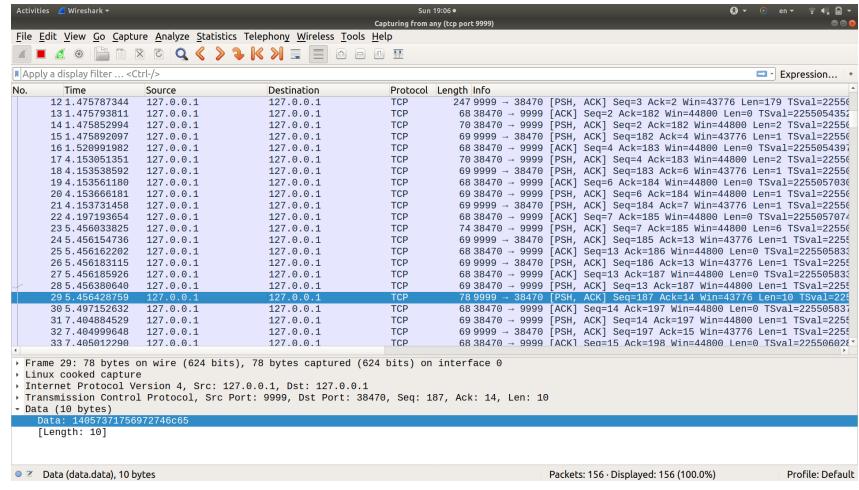


Figura 2.23: El cliente solicita atrapar pokémon.

El cliente confirma la recepción del mensaje e indica que sí desea atrapar el pokémon sugerido.

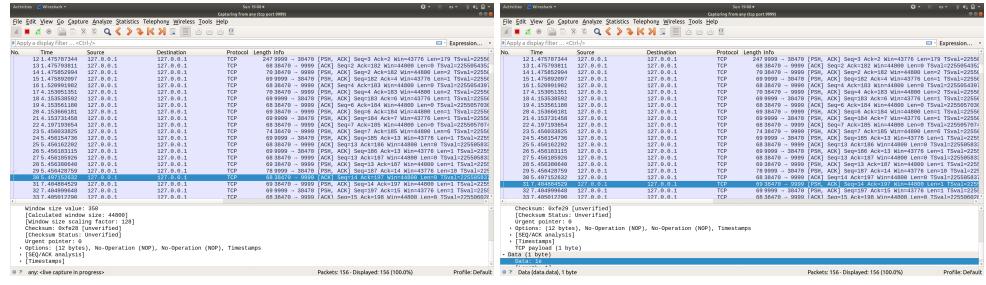


Figura 2.24: El cliente envía confirmación con ACK y acepta atrapar el pokémon.

Ahora, el servidor le indica al cliente que el pokémon que aceptó intentar capturar el cliente no ha sido atrapado por el usuario, y el cliente confirma que recibió dicho mensaje.

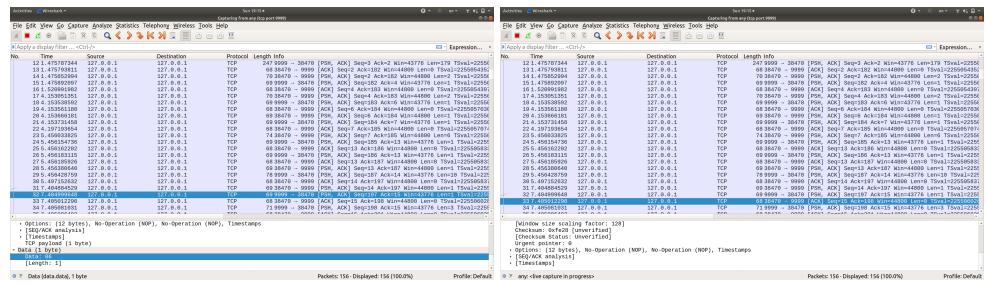


Figura 2.25: El cliente envía confirmación con ACK y acepta atrapar el pokémon.

El servidor ahora envía un paquete con código 21 REMAINING ATTEMPTS indicando que el pokémon no fue capturado e incluye la cantidad de intentos restantes que le quedan al cliente para intentar atrapar al pokémon.

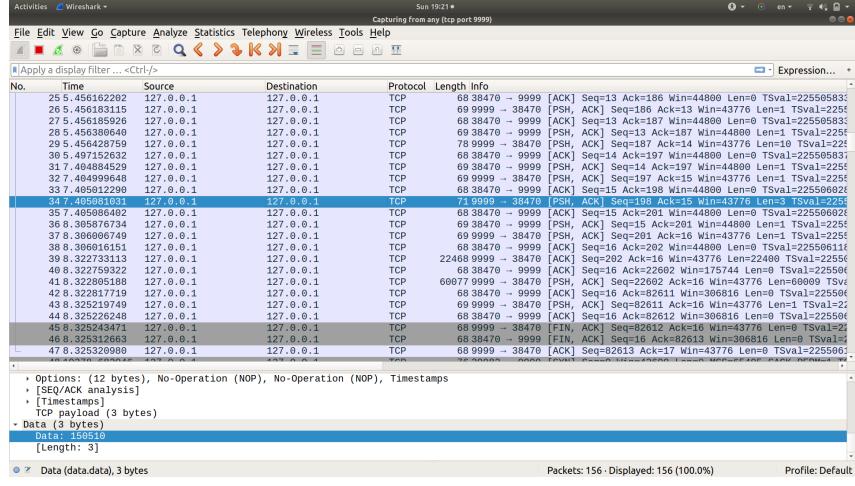


Figura 2.26: El cliente solicita atrapar pokémon.

Una vez que el cliente recibe el paquete realiza la confirmación a través de un ACK. Una vez enviado el ACK, el cliente (dado que le quedan más de 0 intentos) decide que quiere intentar atrapar de nuevo, por lo cual responde con un SI a la pregunta del servidor de si desea seguir intentando.

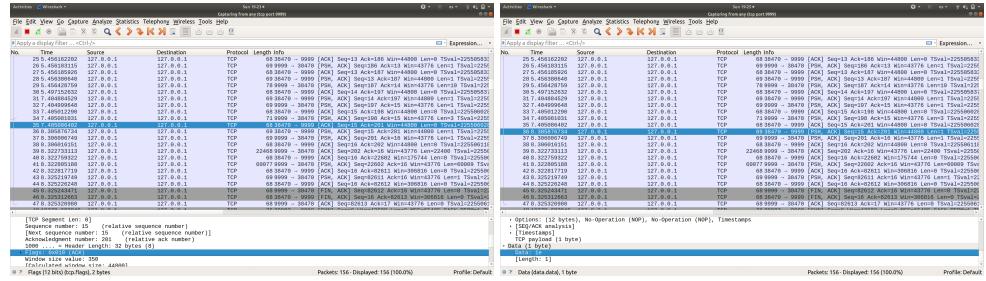


Figura 2.27: El cliente envía confirmación con ACK e indica al servidor que desea seguir intentando.

A continuación, el servidor envía un código 22 CAPTURED POKEMON para indicar al cliente que en su segundo intento ha logrado atrapar al pokémon

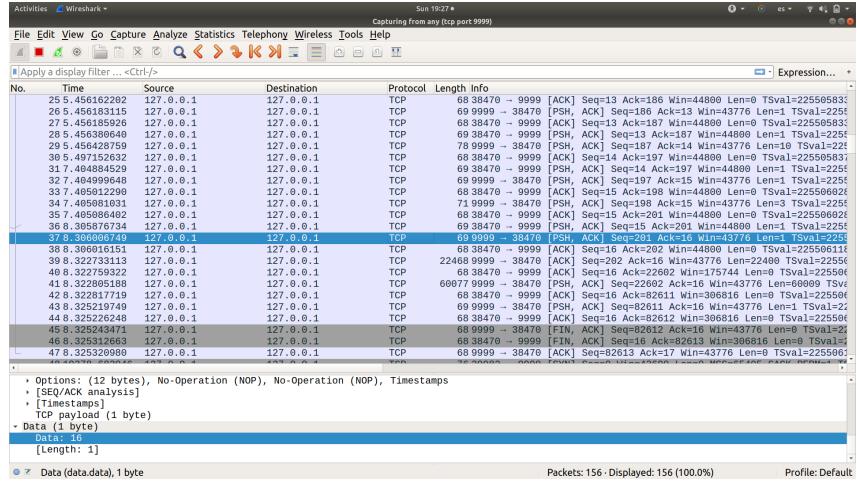


Figura 2.28: El le hace saber al cliente que el pokémon ha sido atrapado.

El cliente realiza la confirmación de recepción del mensaje.

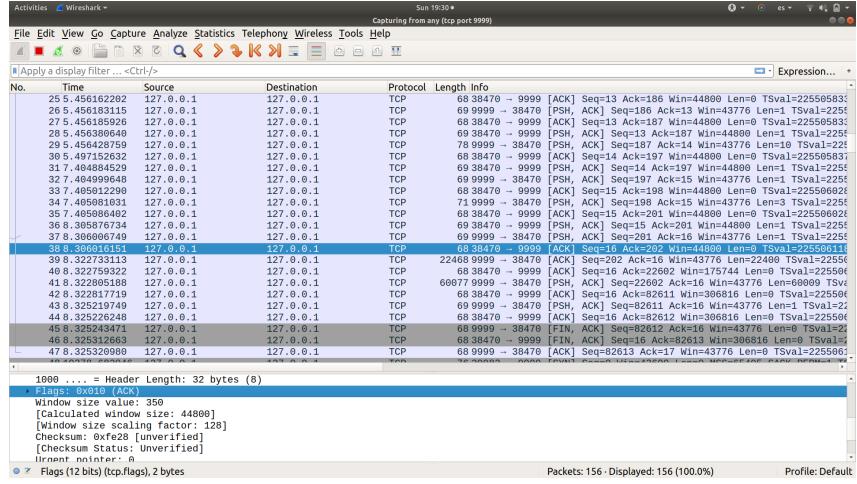
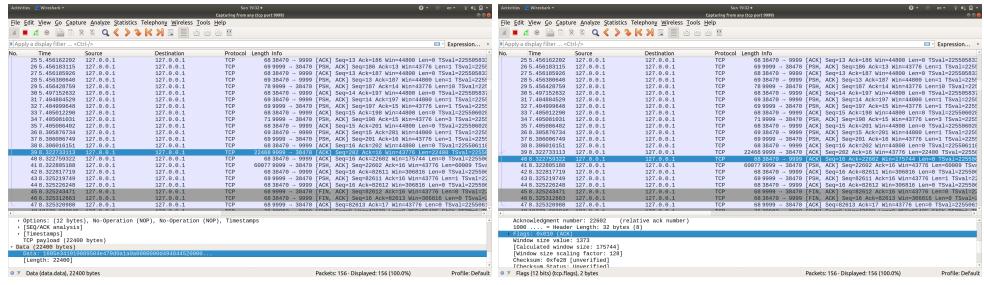
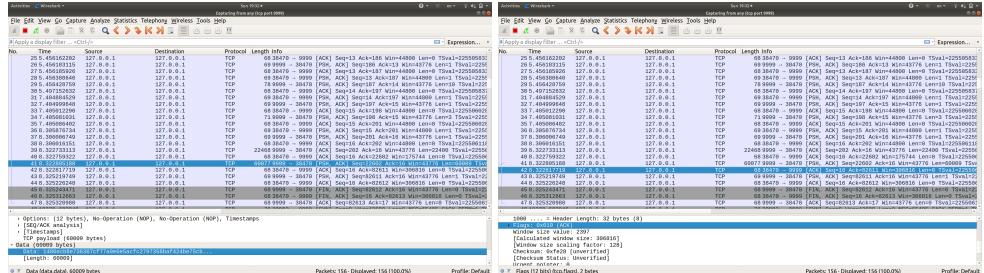


Figura 2.29: El cliente hace ACK.

Dado que el pokémon ha sido atrapado, el servidor procede a enviar la imagen correspondiente al pokémon atrapado. Debido al tamaño de la imagen enviada, la imagen debe ser dividida en dos paquetes.

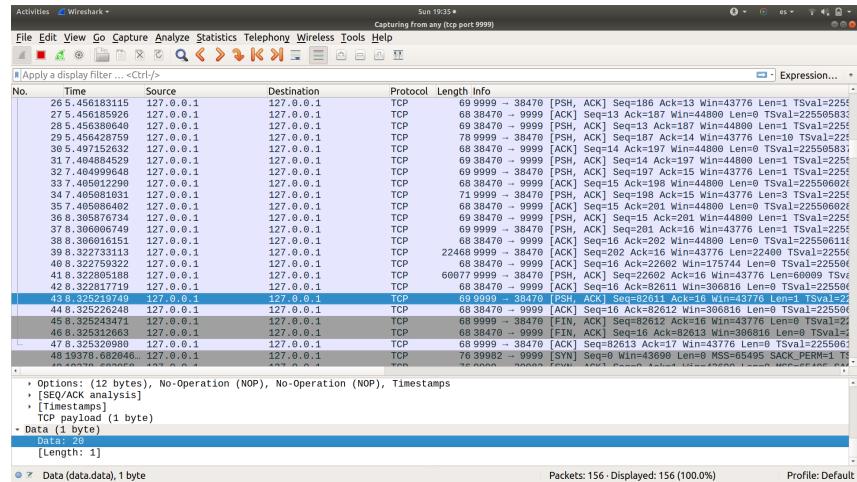


**Figura 2.30:** El servidor envía la primera parte de la imagen al cliente, y el cliente una vez recibido el paquete realiza una confirmación a través de un ACK.



**Figura 2.31:** El servidor envía la segunda parte de la imagen al cliente, y el cliente una vez recibido el paquete realiza una confirmación a través de un ACK.

La imagen ya ha sido enviada en su totalidad, por lo tanto el servidor procede a finalizar la conexión con el usuario. Para ello envía un código 32 TERMINATED SESSION al cliente, una vez que el cliente recibe el código manda un ACK.



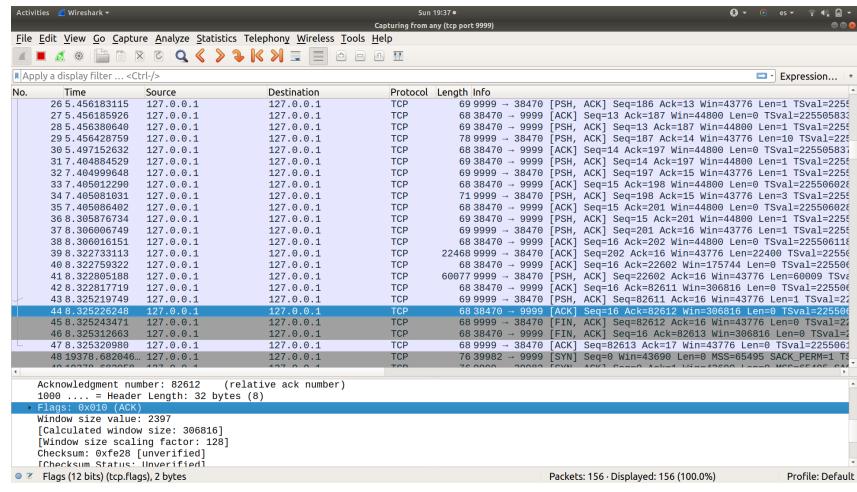


Figura 2.32: El servidor inicia el proceso de finalización y el cliente confirma la recepción del paquete.

Por último se lleva a cabo la finalización empleando el protocolo TCP.

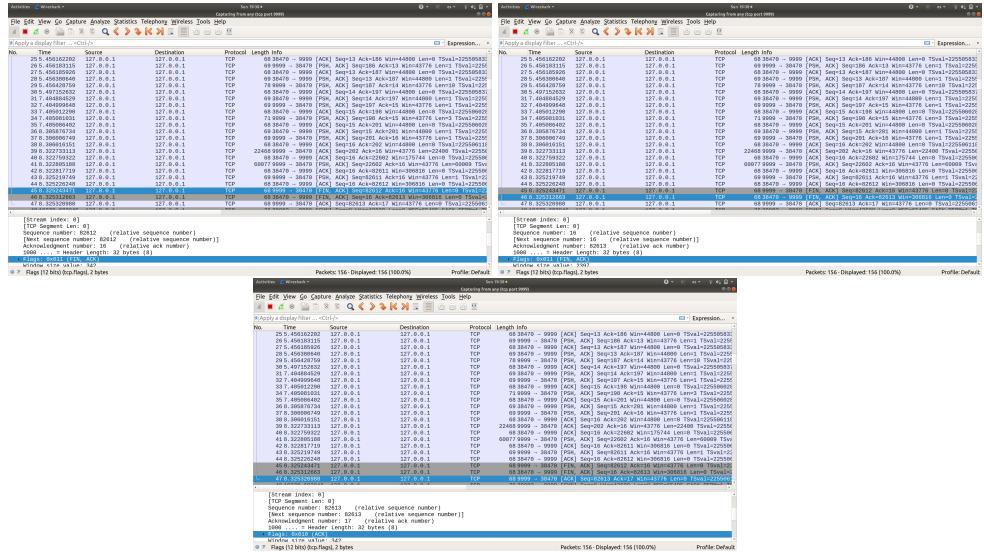


Figura 2.33: El servidor y el cliente finalizan su conexión.

## Capítulo 3

# Interpretación, ejecución y documentación

### 3.1. Instrucciones

En el repositorio de *GitHub* se detalla en el archivo `README.md` las instrucciones necesaria para la interpretación y ejecución de la aplicación presentada. Además, el *makefile* adjunto servirá para instalar todas las dependencias, paquetes y bibliotecas necesarias. Se instará también la documentación en el manual de *Linux* (*manpages*).

### 3.2. Documentación del código

La documentación del código fue generada con la herramienta *Sphinx*. Se generó un *html* con toda la documentación en el directorio `doc`; dentro de éste solo se debe abrir el archivo llamado `index.html`. Aquí se listan y detallan todas las funciones implementadas y requeridas por el protocolo, cada una con la descripción de para qué es usada, qué parámetros recibe y cuáles devuelve.

## Capítulo 4

# Conclusiones

A lo largo del desarrollo de este proyecto nos fue posible notar la importancia que radica en la correcta planeación de un protocolo previo a su implementación concreta. Por medio de la tabla de estados, mensajes y el *FSM* fue más sencillo programarlo, además de que sirve eventualmente como guía para los usuarios de un protocolo de este estilo. Pudimos presenciar de primera mano el papel de los *sockets* como interfaz entre la capa de Aplicación que permite el envío y recepción de datos y la capa de Transporte. Además, gracias a los códigos de mensajes y la formación correcta de estas cabeceras para los mensajes es posible que los clientes y servidores tengan conocimiento del estado actual del sistema para poder actuar conforme a ello.

En nuestro caso optamos por una extensión parcial de la aplicación; claro que el protocolo presentado puede ser extendido todavía más por futuros programadores y el esquema de persistencia mejorado con la incorporación de un Sistema Manejador de Bases de Datos (*DBMS*), pues por ahora usamos los diccionarios de Python y el módulo de `pickle` para guardar el estado del mismo en un objeto persistente.

A lo largo de todo el curso nos fue posible entender lo relacionado al funcionamiento general de las redes de computadoras, desde el aspecto físico, pasando por el enrutamiento de los paquetes y hasta la transferencia de datos entre *hosts*. Sin lugar a dudas se trata de un campo de la computación y de la ingeniería trascendental para el mundo moderno cada vez más conectado.