Laboratorio 1
INF-256 Redes de Computadores

Integrantes: Zarko Kuljis (201823523-7), Sofía Palet (201873570-1)

1) Sea J la cantidad de jugadas hechas:
Según el esquema presentado en el enunciado se deberían enviar:

$$4[msg] + 4*J[msg]$$

Nuestro programa funciona de acuerdo al esquema de la Figura 1:

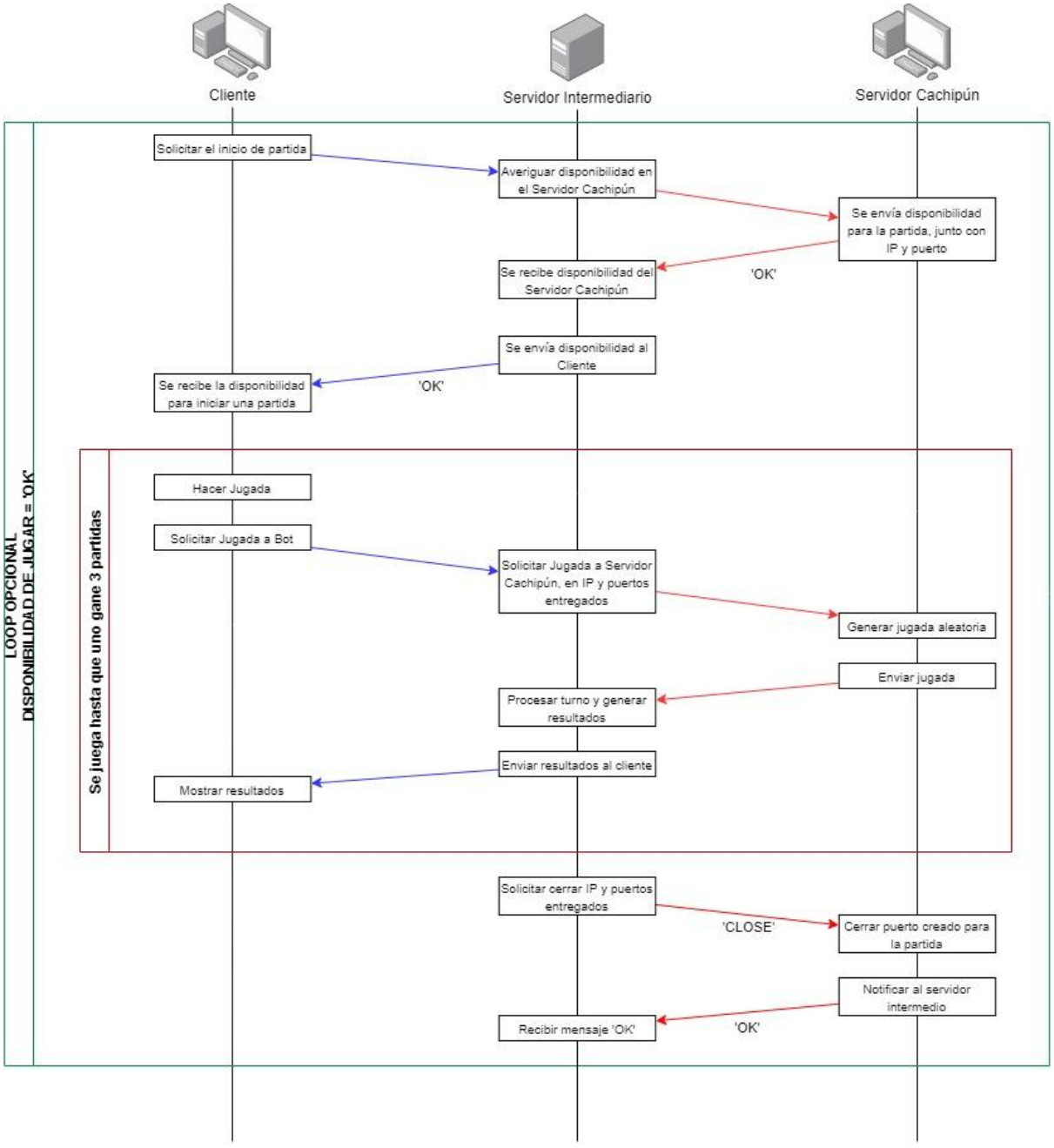

Figura 1: Diagrama donde se piden partidas junto con los dos mensajes finales para cerrar el socket de puerto aleatorio.

Donde se agregan 2 mensajes extras (‘CLOSE’ y ‘OK’) para cerrar el puerto que se abrió cuando comenzó la partida. Por ende esperabamos:

$$4[msg] + 4*J[msg] + 2[msg]$$

Utilizando Wireshark encontramos 59 mensajes a lo largo de 8 jugadas, diferente a los 36 mensajes esperados según el enunciado y a los 38 esperados de acuerdo al funcionamiento de nuestro programa.
Para explicar esto debemos ser conscientes de un par de cosas:

1. Cuando se envía un mensaje a través del protocolo TCP Wireshark detecta 2 mensajes.

```
TCP        55 54327 → 49152 [PSH, ACK] Seq=1 Ack=1 Win=65495 Len=11
TCP        44 49152 → 54327 [ACK] Seq=1 Ack=12 Win=65484 Len=0
```

Figura 2: Mensajes enviados por TCP obtenidos en Wireshark.

El primer mensaje que se observa es el que envía el cliente al servidor intermedio. Suponemos que el segundo mensaje es una especie de confirmación por parte del servidor intermedio al cliente oriunda del protocolo TCP.
Por ende, un mensaje a través del protocolo TCP equivale a dos mensajes.

2. Un mensaje enviado por UDP es contabilizado como un solo mensaje.

```
UDP        43 52200 → 49153 Len=11
UDP        40 49153 → 52200 Len=8
```

Figura 3: Mensajes enviados por UDP obtenidos en Wireshark.

El primer mensaje corresponde a ‘REQUESTGAME’ y el segundo a ‘OK’, por lo que notamos que efectivamente un mensaje UDP es contabilizado como un sólo mensaje en Wireshark.

Es por esto que nuestra ecuación cambia:

$$6[msg] + 6*J[msg]+2[msg]$$

Los primeros 6 mensajes corresponden a solicitar el inicio de la partida (2 TCP y 2 UDP), los siguientes 6*J mensajes corresponden a procesar una jugada y entregar los resultados (2 TCP + 2 UDP) que se repiten J veces. Y los últimos 2 mensajes corresponden a cerrar el puerto que se abrió para la partida.

Según nuestros cálculos deberíamos generar 56 mensajes. Notamos que los 3 primeros mensajes que muestra Wireshark no corresponden a ningún mensaje que hubiésemos mandado:

```
TCP        52 54327 → 49152 [SYN] Seq=0 Win=65495 Len=0 MSS=65495 SACK_PERM=1
TCP        52 49152 → 54327 [SYN, ACK] Seq=0 Ack=1 Win=65495 Len=0 MSS=65495 SACK_PERM=1
TCP        44 54327 → 49152 [ACK] Seq=1 Ack=1 Win=65495 Len=0
```

Figura 4: Primeros 3 mensajes TCP con Len = 0.

En la figura 4 se puede ver que el ‘Len’ de los mensajes es 0. Asumimos que estos 3 mensajes son una especie de Handshake. Completando así los 59 mensajes registrados por Wireshark.

2) Se espera ver protocolos TCP y UDP porque mandamos solicitudes por medio de sockets TCP y UDP.

Los protocolos que encontramos utilizando Wireshark corresponden a TCP y UDP cuando aplicamos el filtro correspondiente a cada uno, para así poder ver los mensajes que nos interesan, que corresponden al Cliente, Servidor Intermediario y Servidor Cachipún.

Sin aplicar el filtro encontramos protocolos como TCP, SSDP, UDP y IGMPv2.

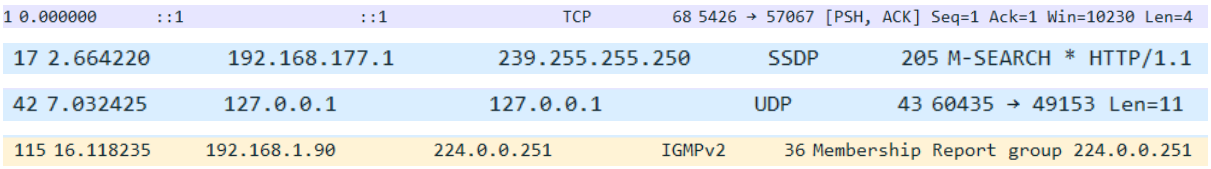

Figura 5: Protocolos encontrados usando Wireshark.

3) Sí, se deben ocupar los mismos puertos a lo largo del tiempo ya que el socket TCP del servidor intermedio nunca varía y la conexión TCP es constante, de forma que nunca se rompe la conexión y por ende los puertos nunca cambian.
Filtrando por ‘(ip.src==ip.dst) and (tcp.port == 49152)’ (donde 49152 es el puerto en el cual se aloja la conexión TCP del servidor intermedio) vemos:

```
TCP        44 54327 → 49152 [ACK] Seq=1 Ack=1 Win=65495 Len=0
TCP        55 54327 → 49152 [PSH, ACK] Seq=1 Ack=1 Win=65495 Len=11
TCP        44 49152 → 54327 [ACK] Seq=1 Ack=12 Win=65484 Len=0
TCP        46 49152 → 54327 [PSH, ACK] Seq=1 Ack=12 Win=65484 Len=2
TCP        44 54327 → 49152 [ACK] Seq=12 Ack=3 Win=65493 Len=0
TCP        49 54327 → 49152 [PSH, ACK] Seq=12 Ack=3 Win=65493 Len=5
TCP        44 49152 → 54327 [ACK] Seq=3 Ack=17 Win=65479 Len=0
TCP        67 49152 → 54327 [PSH, ACK] Seq=3 Ack=17 Win=65479 Len=23
TCP        44 54327 → 49152 [ACK] Seq=17 Ack=26 Win=65470 Len=0
TCP        49 54327 → 49152 [PSH, ACK] Seq=17 Ack=26 Win=65470 Len=5
TCP        44 49152 → 54327 [ACK] Seq=26 Ack=22 Win=65474 Len=0
TCP        67 49152 → 54327 [PSH, ACK] Seq=26 Ack=22 Win=65474 Len=23
TCP        44 54327 → 49152 [ACK] Seq=22 Ack=49 Win=65447 Len=0
TCP        49 54327 → 49152 [PSH, ACK] Seq=22 Ack=49 Win=65447 Len=5
TCP        44 49152 → 54327 [ACK] Seq=49 Ack=27 Win=65469 Len=0
TCP        70 49152 → 54327 [PSH, ACK] Seq=49 Ack=27 Win=65469 Len=26
TCP        44 54327 → 49152 [ACK] Seq=27 Ack=75 Win=65421 Len=0
TCP        49 54327 → 49152 [PSH, ACK] Seq=27 Ack=75 Win=65421 Len=5
TCP        44 49152 → 54327 [ACK] Seq=75 Ack=32 Win=65464 Len=0
TCP        65 49152 → 54327 [PSH, ACK] Seq=75 Ack=32 Win=65464 Len=21
TCP        44 54327 → 49152 [ACK] Seq=32 Ack=96 Win=65400 Len=0
TCP        49 54327 → 49152 [PSH, ACK] Seq=32 Ack=96 Win=65400 Len=5
TCP        44 49152 → 54327 [ACK] Seq=96 Ack=37 Win=65459 Len=0
TCP        65 49152 → 54327 [PSH, ACK] Seq=96 Ack=37 Win=65459 Len=21
TCP        44 54327 → 49152 [ACK] Seq=37 Ack=117 Win=65379 Len=0
TCP        49 54327 → 49152 [PSH, ACK] Seq=37 Ack=117 Win=65379 Len=5
TCP        44 49152 → 54327 [ACK] Seq=117 Ack=42 Win=65454 Len=0
TCP        70 49152 → 54327 [PSH, ACK] Seq=117 Ack=42 Win=65454 Len=26
TCP        44 54327 → 49152 [ACK] Seq=42 Ack=143 Win=65353 Len=0
TCP        49 54327 → 49152 [PSH, ACK] Seq=42 Ack=143 Win=65353 Len=5
TCP        44 49152 → 54327 [ACK] Seq=143 Ack=47 Win=65449 Len=0
TCP        67 49152 → 54327 [PSH, ACK] Seq=143 Ack=47 Win=65449 Len=23
TCP        44 54327 → 49152 [ACK] Seq=47 Ack=166 Win=65330 Len=0
TCP        49 54327 → 49152 [PSH, ACK] Seq=47 Ack=166 Win=65330 Len=5
TCP        44 49152 → 54327 [ACK] Seq=166 Ack=52 Win=65444 Len=0
TCP        66 49152 → 54327 [PSH, ACK] Seq=166 Ack=52 Win=65444 Len=22
TCP        44 54327 → 49152 [ACK] Seq=52 Ack=188 Win=65308 Len=0
TCP        48 54327 → 49152 [PSH, ACK] Seq=52 Ack=188 Win=65308 Len=4
TCP        44 49152 → 54327 [ACK] Seq=188 Ack=56 Win=65440 Len=0
```

Figura 6: Mensajes entre los puertos 49152 y 54327.

Notamos que todos los mensajes que llegan al puerto 49152 (es decir, los mensajes que envía el cliente) son a través del mismo puerto (el 54327).

4) Respecto a solicitar partida no, las partidas siempre se solicitan en el mismo puerto. Sin embargo cuando se acepta una partida la cosa cambia; en ese caso el Servidor Cachipún genera un nuevo puerto para esa partida que cambiará cada vez que se inicie una nueva.
Esto coincide con lo visto en Wireshark, podemos ver que las partidas siempre se solicitan al puerto 49153:

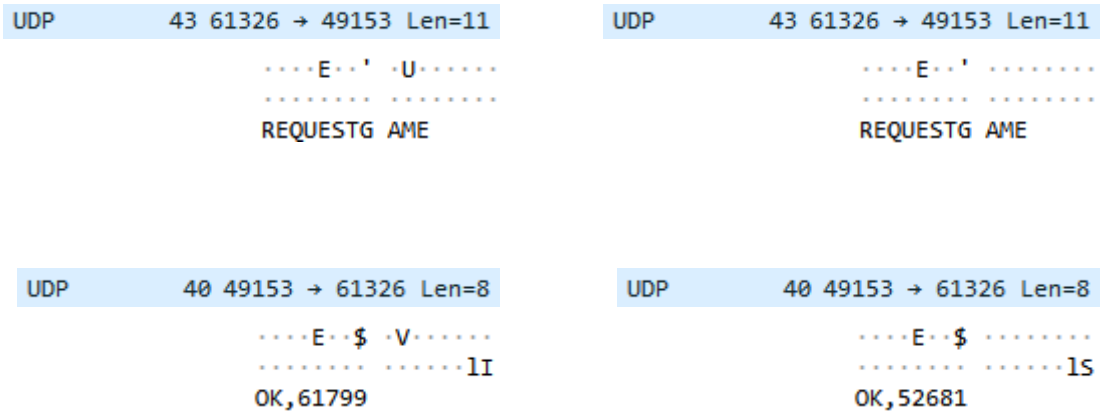

Figura 7: Mensajes donde el Servidor Intermediario envía ‘REQUESTGAME’ al socket principal del Servidor Cachipún y el Servidor Cachipún le responde con ‘OK’. (Nótese que el puerto 61326 corresponde al servidor intermedio)

En cambio cuando se solicita una jugada (es decir, ‘Piedra’, ‘Papel’ o ‘Tijeras’) se le pide a otro puerto,el cual cambia en cada partida:

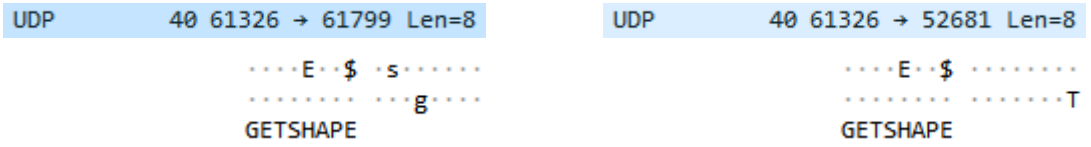

Figura 8: Mensajes donde el Servidor Intermediario envía ‘GETSHAPE’ al socket creado en un puerto aleatorio del Servidor Cachipún.

Notamos que el servidor intermedio (Puerto 61326) en la primera ocasión pide la jugada al puerto 61799, mientras que en la segunda ocasión se la pide al puerto 52681

5) La parte del mensaje en ASCII (es decir, los mensajes que enviamos y se encuentran al final del mensaje detectado por Wireshark) sí es legible, pero la gran parte del mensaje no es legible, esto se debe a que son Headers que podrían no estar codificados en ASCII.

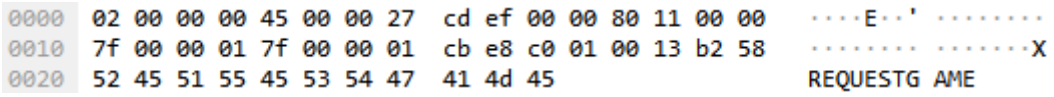

Figura 9: Mensaje que contiene ‘REQUESTGAME’ en su contenido.