Instituto Tecnológico Autónomo de México

Proyecto Alpha

Sistemas Distribuidos

Luis Felipe Landa Lizarralde

Andrea Marín Alarcón

15 de marzo de 2019

Índice

1. Descripción

El objetivo del experimento es implementar un juego de *¡Pégale al monstruo!* utilizando los protocolos de comunicación TCP, Multicast e Invocación de Métodos Remota.

Existen dos entidades en la descripción del juego: el cliente y el servidor. El trabajo del cliente es recibir estos "monstruos" por medio de mensajes Multicast, el usuario les "pega" y manda esta respuesta por el protocolo TCP al servidor. Por otro lado, el servidor es el encargado de emitir los "monstruos" y recibir las respuestas de los clientes. También se encarga de mantener el estado del juego, esto es, llevar la cuenta del número de monstruos que cada cliente ha "golpeado".

Un jugador gana cuando le ha pegado a 3 monstruos, cuando esto pasa el servido envía un mensaje por medio de Multicast avisando quién ganó y reinicia el juego. Además, el sistema debe de ser capaz de llevar un registro de jugadores, de manera que si un jugador sale del juego no pierda su puntaje. Este registro se hace por medio de RMI.

2. Diseño

El servidor implementa varios hilos para cumplir su función. El hilo principal se encarga de emitir los mensajes Multicast que representan a los "monstruosz espera a que algún cliente conteste o se acabe un límite de tiempo. Se genera un segundo hilo en el cual el servidor espera nuevas conexiones TCP, es decir, espera respuestas de los clientes. Finalmente, por cada cliente que se conecte vía TCP, se genera un hilo separado para recibir la respuesta del cliente particular; una vez que se recibe el mensaje del cliente y lo procesa, se cierra la conexión TCP.

Tanto el hilo principal como los hilos que corresponden a las conexiones con los clientes cuentan con la referencia a un objeto que mantiene el estado del juego. El hilo principal del servidor se mantiene en un ciclo infinito, emitiendo monstruos de forma constante hasta que se termine el proceso de manera externa.

Por el lado del cliente, se tienen dos hilos: uno en el que se corre la interfaz gráfica y otro con un ciclo infinito en el que se escucha por mensajes (monstruos o ganador) del servidor. Cuando el usuario le pega a un monstruo, la interfaz valida que haya sido en la posición correcta, en cuyo caso, regresa el id del usuario y la ronda en la que está jugando al servidor vía un mensaje TCP. En la Fig. 1 se puede ver la secuencia del sistema en una ronda del juego.

Del lado del servidor tenemos 3 clases importantes: *GameBoard*, *GameServer* y *ScoreManager*. *GameServer* es la clase principal. Aquí se despliega el servicio de *log in* al registro RMI del servidor, se crea un hilo que escucha por conexiones TCP y se mandan los mensajes Multicast. *ScoreManager* es el hilo que se crea con cada conexion TCP y es donde se procesa el mensaje recibido por el cliente y se decide si se debe de aumentar el puntaje del jugador o no. Por último,

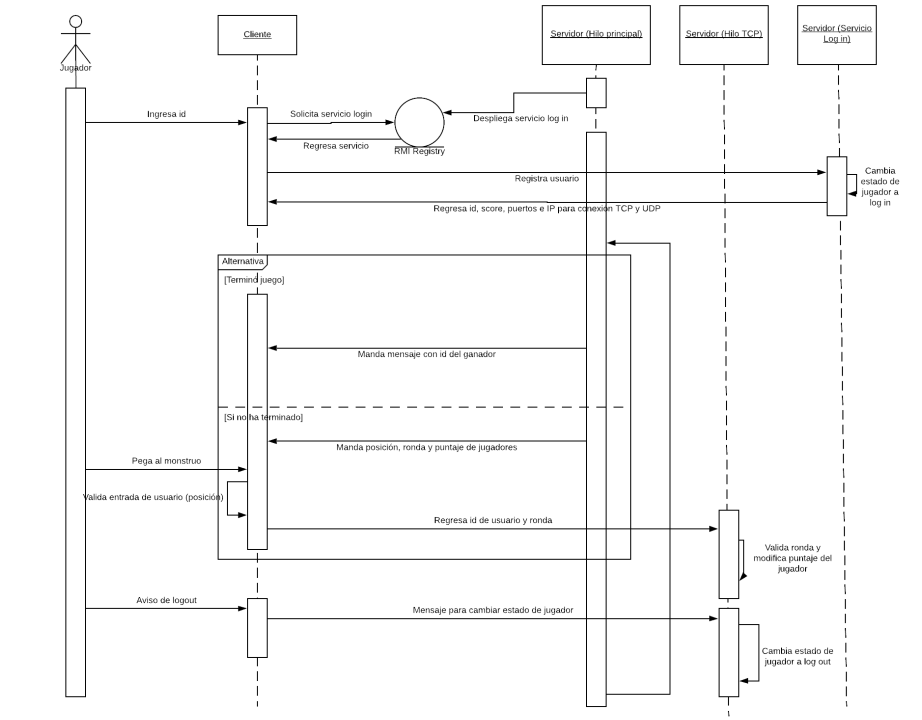


Figura 1: Diagrama de secuencia de una ronda del juego

GameBoard es la clase que mantiene el estado de juego. Tiene como atributos:

- **ArrayList de jugadores:** Cada jugador tiene ID, puntaje y su estado: *logged in* o *logged out*.
- **Número de ronda (*int*)**: Número de ronda actual para saber si los mensajes recibidos de los jugadores corresponden a la ronda actual o si son de una ronda anterior.
- **Fin (*boolean*)**: Bandera que indica si ya terminó el juego, es decir, si un jugador ya le pegó al número límite de monstruos.
- **Ganador (*Player*)**: Se guarda el jugador que ganó el juego
- **Max hits (*int*)**: Número de golpes necesarios para que un jugador gane.

GameBoard se encarga de agregar jugadores, aumentar el puntaje de un jugador, dado su ID, cambiar el estado de un jugador y reiniciar el juego (cambiar todos los puntajes a cero y la bandera de fin a *false*). Así, *GameBoard* es un atributo en común entre el servicio de *log in* que está desplegado en el RMI, el hilo que está esperando conexiones TCP, el hilo que se crea con cada conexión TCP y el hilo principal que manda los monstruos.

Del lado del cliente tenemos una única clase *GameClient*, ésta es el puente entre la lógica del juego y la interfaz gráfica. Esta clase es la encargada de obtener el servicio de *log in* del registro RMI del servidor, registrar al usuario, recibir los mensajes Multicast y mandar a la interfaz la posición del monstruo y mandar la respuesta del jugador por TCP.

2.1. Lógica

2.1.1. Registro de usuarios

Cuando comienza el juego se le pide al usuario que ingrese un ID, el cual se envía al servicio de *log in*. Si no existe el ID en el arreglo de jugadores del tablero, entonces se agrega un nuevo jugador con puntaje de 0 y estado *logged in*. Si ya existe, verifica que el jugador no haya iniciado sesión, es decir su estado es *logged out*, en cuyo caso se cambia a *logged in*. En los casos de éxito, el servicio regresa una instancia de la clase *Connection* la cual contiene el ID del jugador, su puntaje (0 si es nuevo o lo que tuviera guardado si ya estaba registrado), los puertos donde el servidor está mandando los mensajes Multicast y donde está escuchando las conexiones TCP, la IP del grupo multicast y la IP para hacer la conexion TCP.

En caso de que el jugador ya haya iniciado sesión se regresa un valor nulo y se le pide al usuario que ingrese un ID diferente.

2.1.2. Juego

La lógica del juego se divide en dos partes: los mensajes Multicast y los mensajes TCP.

Antes de mandar un mensaje multicast, se checa si la bandera *fin* está prendida. Si sí, se manda por multicast un mensaje con el ID del ganador y reinicia el juego; si no, se aumenta el número de ronda y se genera una posición del tablero de manera aleatoria. Así, el "monstruo" del mensaje multicast se ve de la siguiente manera:

$$x, y, \text{numRonda}, (\text{ID, puntaje: para todos los jugadores})$$

El servidor espera por 0.9 segundos o a que alguien gane la ronda, lo que pase primero, para mandar un mensaje nuevamente.

Luego, cuando *GameClient* recibe el mensaje manda la posición a la interfaz para que pinte el panel correspondiente y guarda la ronda actual. Después busca su ID y su puntaje para actualizar los valores correspondientes en la interfaz.

Cuando el jugador oprime un panel, la interfaz verifica que haya sido en la posición correcta y si es el caso, la clase *GameClient* realiza una conexión TCP con el servidor y le manda el ID del usuario y la ronda en la que está jugando.

El servidor recibe la conexión TCP y crea un hilo para cada cliente. La clase *ScoreManager* entonces verifica que la ronda del mensaje corresponda a la ronda actual del tablero. Tras la validación, aumenta el puntaje del jugador pertinente e incrementa el número de la ronda del tablero. De esta manera, el servidor se percata que hubo un ganador para la ronda. Si la ronda del mensaje no coincide con la del tablero, el mensaje se descarta. El método que se encarga de incrementar el puntaje del jugador, también se encarga de prender la bandera de fin y asignar un ganador en caso de que el nuevo puntaje del jugador sea igual al número máximo de intentos.

3. Estresamiento

El enfoque para estresar el servicio fue únicamente del lado del servidor. Se creó una versión de *GameClient* que responde de forma inmediata al mensaje Multicast emitido por el servidor. Asimismo, se limitó el número de rondas que juega cada cliente a 50.

Se corrieron pruebas con 20, 50, 80, 100, 150 y 200 clientes concurrentes. Cabe mencionar que se impuso un límite de 150 conexiones TCP simultáneas al servidor. De cada configuración de clientes se realizaron 10 repeticiones.

La métrica utilizada se da por el tiempo de un ciclo completo, entendiéndose como ciclo desde el momento en que el servidor emite el mensaje Multicast, hasta el momento en que se procesa la respuesta TCP del cliente. Además se contó el número de veces que se presentó un error del lado del cliente.

Tanto en las corridas de 20 (Fig. 2a) y 50 (Fig. 3b) clientes observamos que los promedios a lo largo de las corridas son casi constantes con intervalos de variación de 2 a 3 milisegundos. En el primer caso el promedio mínimo fue de 13.11ms y el máximo de 15.26ms; por el otro lado, con 50 se tuvo un promedio mínimo de 22.58ms y un máximo de 25.99ms. Esto nos indica que el servidor tuvo un desempeño promedio consistente.

Con 80 clientes (Fig. 4a) el intervalo de variación para el promedio fue mucho mayor, alcanzando los 10ms, que representa casi un 30 % del valor máximo (36.25ms). Con 100 y 150 clientes (Fig. 5a y Fig. 6a) el intervalo de variación aumentó todavía más a un valor de 22ms y 60 ms respectivamente. Sin embargo, nos percatamos que existe un intervalo de carga (número de clientes) en donde los tiempos promedio de respuesta varían poco entre sí, ejemplificado por los valores obtenidos en las corridas de 50 y 80 clientes.

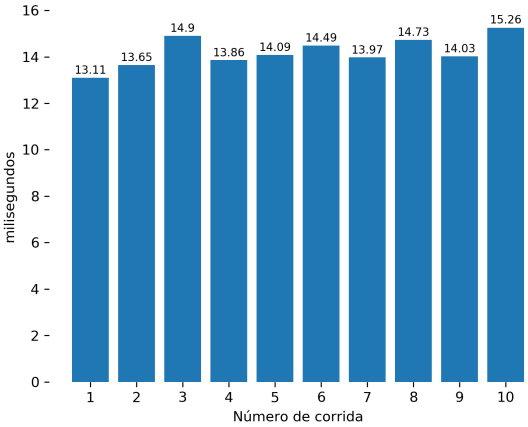
Finalmente, con 200 clientes (Fig. 7a) no hay una tendencia clara pues tenemos un intervalo de variación de 140ms, con un valor mínimo de 41.78ms y un máximo de 186.4 ms. Esto nos deja ver que con cargas de trabajo más grandes el servidor presenta un rendimiento variable.

Analizando las desviaciones estándar, percibimos que, para casi todos los casos (20, 50, 80, 100, 150), las desviaciones tomaban valores cercanos al 50 % del valor del promedio. Esto indica que los tiempos registrados están altamente dispersos. Dicha dispersión puede ser resultado de que las peticiones llegan en tandas, las primeras son atendidas de forma más rápida a las últimas, pues no se enfrentan a una demanda alta por los recursos. Otro factor influyente puede ser que el método que incrementa el puntaje de los jugadores es síncrono, por lo que se ejecuta una petición a la vez.

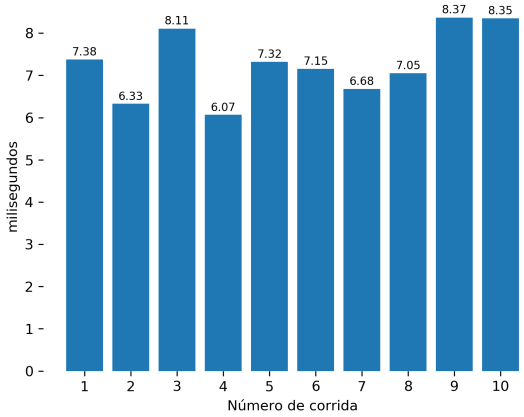
El único caso excepcional se presenta con 200 clientes (Fig. 7), cuando en las corridas 4, 6 y 7, la desviación supera en valor al promedio de respuesta lo que implica una variación aún mayor en los tiempos registrados.

Sólo se presentaron errores cuando el número de clientes era igual o mayor al número máximo de conexiones TCP simultáneas: 150. Los errores presentados tenían que ver con el hecho de que el cliente era incapaz de establecer una conexión con el servidor. Como era de esperar, el número de errores incrementa conforme aumenta el número de clientes. Además notamos que, en general, las corridas con más errores tienen un menor tiempo promedio de respuesta que aquellas con menos errores. Esto se puede explicar porque, al presentar el error, el cliente no puede enviar su respuesta, lo que representa menos carga para el servidor y libera recursos para otras peticiones. Finalmente, del comportamiento global (Fig. 8), vemos una tendencia esperada: entre más clientes, mayor el tiempo de respuesta promedio y desviación de éste. Y, en el caso de 200 clientes el promedio de respuesta es ligeramente menor al de 150, pero se explica con lo mencionado anteriormente.

En conclusión, el servidor ofrece un buen nivel de servicio a un número grande de clientes. Dado que tanto el servidor como los clientes se ejecutaron en una sola máquina los resultados vistos posiblemente sean peores a que si los clientes y el servidor estuvieran en máquinas separadas. Además, en un escenario de uso real, aún con 200 jugadores, es muy poco probable que todos contesten de forma simultánea, por lo que el servidor tendría un mejor rendimiento al presentado durante el estresamiento.



(a) Promedio



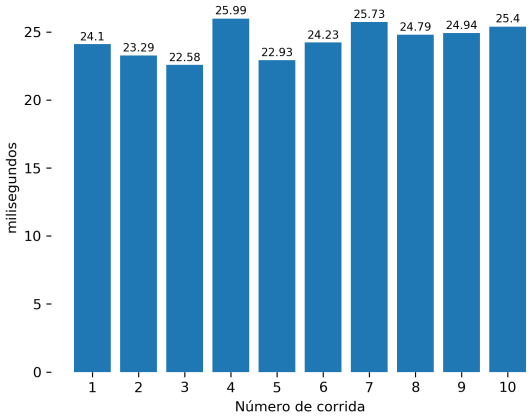
(b) Desviación

Figura 2: Estadísticas de rendimiento para 20 clientes en 10 corridas

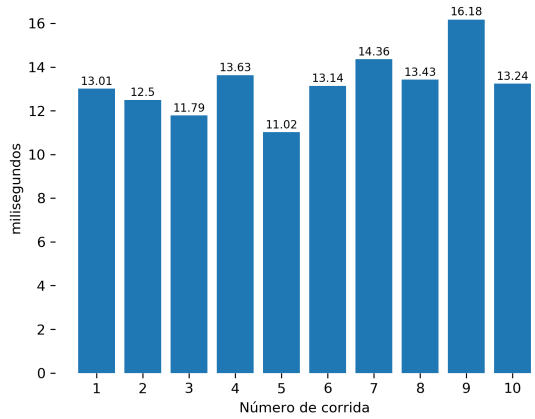
4. Recursos

El código fuente, el diagrama de secuencia, los archivos .csv con los datos del estresamiento, el script de Python para obtener las estadísticas y las gráficas se encuentran en el siguiente repositorio:

`https://github.com/andreamarin/ProyectoAlpha`

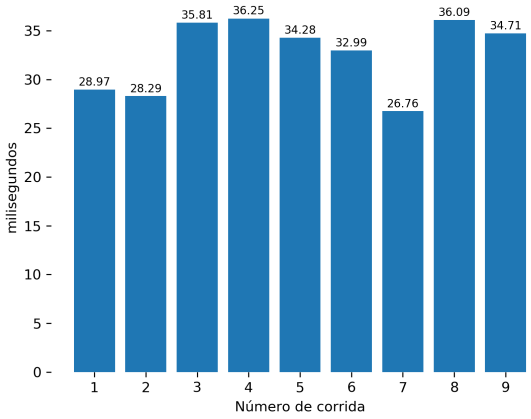


(a) Promedio

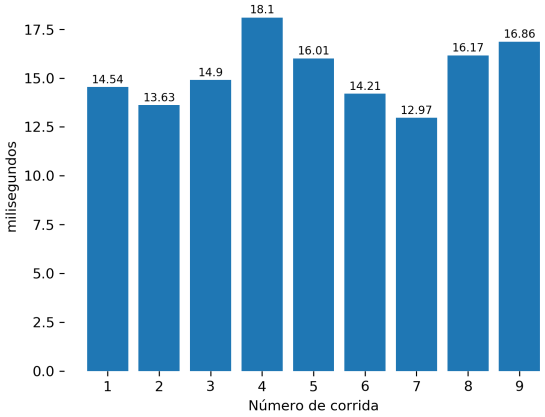


(b) Desviación

Figura 3: Estadísticas de rendimiento para 50 clientes en 10 corridas

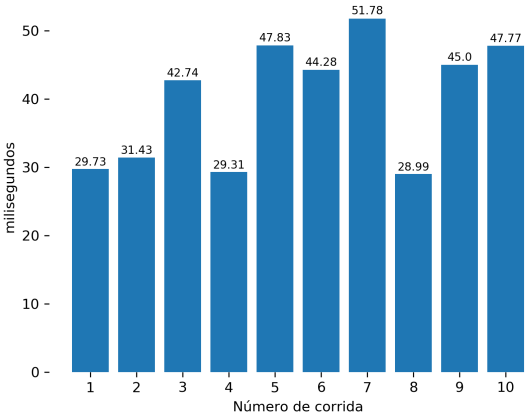


(a) Promedio

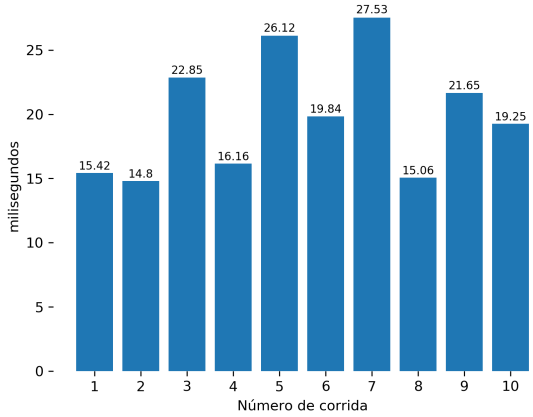


(b) Desviación

Figura 4: Estadísticas de rendimiento para 80 clientes en 10 corridas

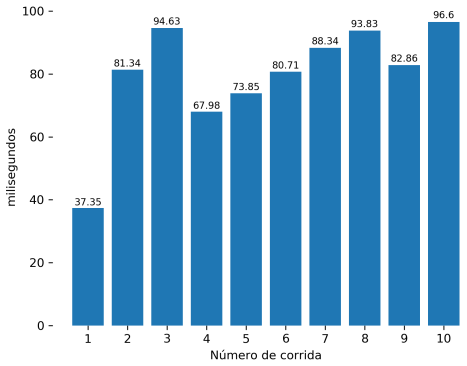


(a) Promedio

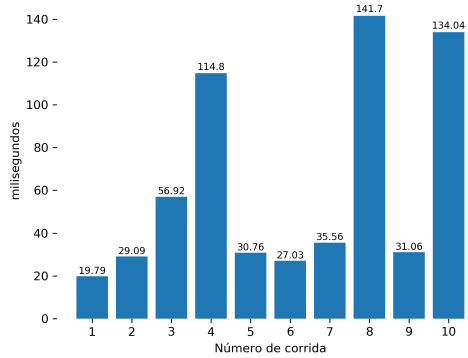


(b) Desviación

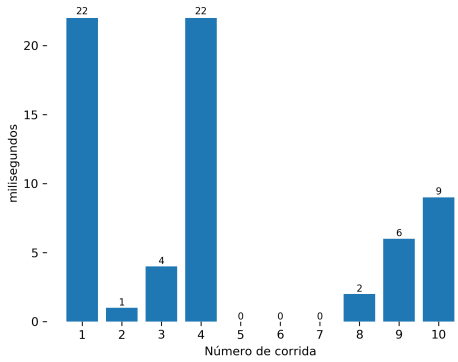
Figura 5: Estadísticas de rendimiento para 100 clientes en 10 corridas



(a) Promedio

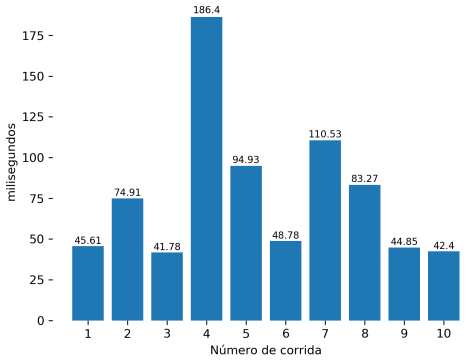


(b) Desviación

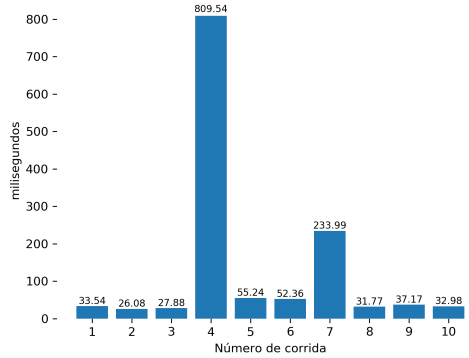


(c) Errores

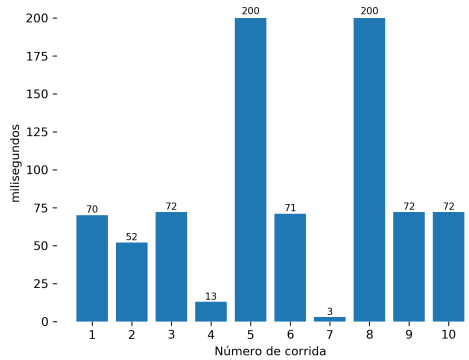
Figura 6: Estadísticas de rendimiento para 150 clientes en 10 corridas



(a) Promedio

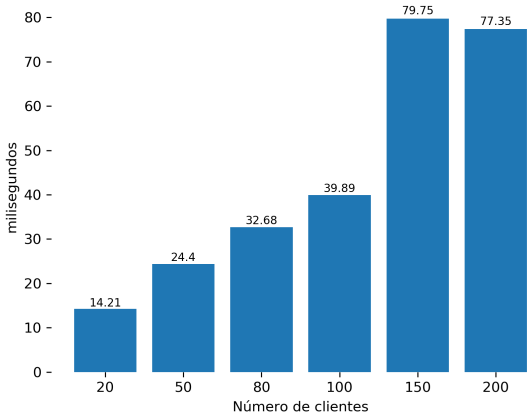


(b) Desviación

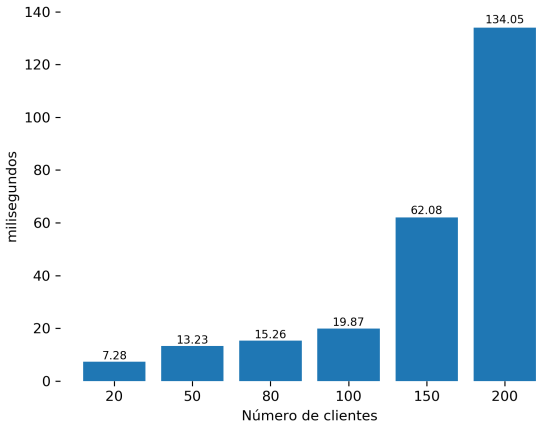


(c) Errores

Figura 7: Estadísticas de rendimiento para 200 clientes en 10 corridas



(a) Promedio de promedios



(b) Promedio de desviaciones

Figura 8: Estadísticas de rendimiento por número de clientes