INSTITUTO TECNOLÓGICO AUTÓNOMO DE MÉXICO

SISTEMAS DISTRIBUIDOS

PROYECTO ALPHA

Evaluación experimental de diseño

Integrantes
Santiago Flores Larrondo – 196864
Mariel Sofía Gutiérrez Zapién – 195525

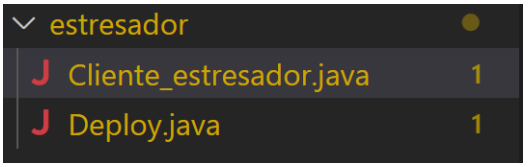
Fecha de entrega del proyecto
13 de Marzo de 2024

Definición del experimento

El propósito del presente reporte es documentar el desempeño de nuestro proyecto cliente-servidor “Proyecto Alpha”. El funcionamiento esperado del proyecto fue mostrado al profesor durante la clase pero si hay interés de parte del lector de probar el proyecto en su modalidad interactiva también se puede consultar de forma local. Para realizar esto último sería necesario clonar el repositorio de Github[1] y ejecutarlo.
El propósito de este documento es elaborar un análisis cuantitativo del funcionamiento. Teniendo ese propósito en mente, definiremos el “estresamiento” del servicio, con el fin de poder reportar los tiempos de respuesta del juego, como los tiempos de respuesta en el registro del juego.

El estresamiento en el contexto de servicios cliente-servidor se refiere a la simulación de cargas o demandas extremas sobre el sistema, con el fin de evaluar su capacidad para manejar una gran cantidad de solicitudes simultáneas o una carga de trabajo intensiva. En otras palabras, implica poner bajo presión al servidor al enviarle un gran número de solicitudes en un período de tiempo determinado para determinar su capacidad máxima de respuesta y su estabilidad bajo condiciones de estrés.

En particular, para estresar al servidor, se define un nuevo tipo de cliente. Dicho cliente tiene exactamente la misma funcionalidad que el cliente “normal” a excepción de que el "cliente nuevo" no cuenta con una interfaz gráfica, sino que funciona manera automática (no espera que el usuario de click a un elemento de la cuadrícula, más bien la respuesta del cliente es aleatoria).
Una vez definida la clase Cliente_estresador, se define también la clase Deploy, desde donde estaremos generando a varios clientes para simular la demanda extrema sobre el servidor.



Como el Cliente_estresador no cuenta con una interfaz gráfica, sino que manda mensajes al servidor de manera automática, se hicieron unos cambios con respecto al Cliente original (MainClient). Dichos cambios se muestran a continuación:

En primer lugar, al declarar un nuevo cliente, se tiene como parámetro una variable tipo String nombre, la cual será el nombre con el que se hará el login o el registro al servidor. Como el servidor asigna automáticamente una clave única concatenada al

[1] https://github.com/marielsgtzz/ProyectoAlpha

nombre, no tendremos problema al declarar a varios clientes con el mismo nombre, pues en el servidor se tratarán de usuarios distintos[2].

```
public Cliente_estresador(String nombre){
    this.nombre = nombre;
    connectToServer();
}
```

En segundo lugar, una vez que se instancia un nuevo cliente, se manda a llamar el método connectToServer(), la cual inicia la conexión TCP con el servidor. Con esta conexión el cliente manda 'automáticamente' lo que sería el login con la línea out.writeUTF(nombre). Luego, espera la respuesta del servidor, el cual regresa el nombre con el que se identificará dentro del mismo. El tiempo que pasa durante el registro del cliente se registra y, posteriormente, se imprime dentro de la misma ejecución del cliente[3].

```
public void connectToServer() {

    try {
        // Se conecta al servidor en el puerto especificado para comunicacion TCP
        int serverPort = 49152;
        socket = new Socket(host:"localhost", serverPort);
        in = new DataInputStream(socket.getInputStream());
        out = new DataOutputStream(socket.getOutputStream());

        //Se cuenta el tiempo de respuesta del registro
        long startTime = System.currentTimeMillis();
        out.writeUTF(nombre);
        nombre = in.readUTF();
        long spentTime = System.currentTimeMillis() -startTime;
        System.out.println("TR: "+spentTime);
```

En tercer lugar, una vez que se genera un registro exitoso, el cliente recibe del servidor las coordenadas en las que se encuentra en ese momento el "Monstruo". En vez de presentarlas en una interfaz gráfica (que no existe en esta versión del cliente), se espera un tiempo aleatorio entre 0 y 500 milisegundos para mandar su respuesta, que sería la misma coordenada que recibió. En el servidor se tiene un manejo de las solicitudes First Come - First serve: el primer mensaje que llegue con

[2] Los puertos, adicionalmente, se asignan dinámicamente, por lo que nunca se trata de la misma solicitud.

[3] Se decide imprimir la información del tiempo de registro como TR: (el tiempo transcurrido en milisegundos), pues se notará que cada vez que se inicia un nuevo cliente, se vuelve un hilo distinto, por lo que el main contenido en el Deploy no tiene control sobre dichos hilos. Resulta más difícil aún tener una variable global que registre los tiempos, exactamente por los 'problemas' que podría traer la concurrencia. Al imprimir la información de ejecución, no se requiere ningún tipo de sincronización, y es más fácil copiar los resultados a un archivo xlsx para su posterior análisis.

las coordenadas correctas, se le otorga un punto, las demás se desechan. Se decide tener un tiempo aleatorio de respuesta, para ver simular el tener distintos usuarios conectados desde distintas direcciones y distintos tiempos de respuesta. También se hace con la finalidad de que se tenga un ganador. Este segundo tiempo mide el tiempo que dura el juego, proceso que también se va a imprimir, en vez de guardarlo en una variable global.

```
DatagramPacket messageIn = new DatagramPacket(buffer, buffer.length);
socketM.receive(messageIn);
String data = new String(messageIn.getData(),offset:0,messageIn.getLength());
//System.out.println("Received: "+data+ " from: "+ messageIn.getAddress());

if(!msjWinner(data)){
    int tiempoDelay = r.nextInt(bound:501);
    Thread.sleep(tiempoDelay);
    out.writeUTF(data);
}
else{
    long spentT = System.currentTimeMillis() -startTime;
    System.out.println("TJ: "+spentT);
    stopClient();
}
```

Por cada mensaje recibido, verifica si este indica el fin del juego. Si no es el caso, simula un retraso aleatorio antes de reenviar el mensaje al servidor. Por el contrario, si el mensaje indica que hay un ganador, registra el tiempo total desde la conexión via Multicast hasta este punto como TJ (tiempo de juego), notifica al objeto callback que el juego ha terminado y procede a cerrar la conexión.

El Deploy y la interfaz ClienteCallback trabajan juntos para orquestar una serie de juegos, donde múltiples instancias de clientes (Cliente_estresador) interactúan con el servidor. La clase `Deploy` se encarga de iniciar múltiples juegos secuencialmente, controlando la creación y coordinación de varios clientes que participan en cada juego. Para lograr lo anterior, se define un ciclo que se repite 30 veces, correspondientes a 30 juegos diferentes. Por cada juego, imprime un mensaje indicando el inicio de y resetea el contador clientesTerminados[4] a 0, además de iniciar 10, 15, 20, 30[5] nuevas instancias de Cliente_estresador, pasándoles un nombre para el registro y un objeto callback[6].

[4] Basado en la interfaz ClienteCallback que define un único método. Dicho método se implementa para actualizar el contador de clientes terminados y notificar que todos hayan terminado su tarea en el juego actual.
[5] Se configura manualmente que se intente con 10, 15,... clientes; cada configuración se intenta 30 veces.
[6] Este objeto se invoca cuando un cliente determina que el juego ha terminado. Al ser llamado por un cliente incrementa clientesTerminados

Después de iniciar todos los clientes para un juego, Deploy entra en un bloque synchronized en el que espera hasta que todos los clientes hayan notificado su terminación mediante el callback. Cuando todos los clientes hayan terminado, notifica para salir de la espera.

Adicionalmente, utiliza un objeto `lock` para sincronizar el acceso al contador clientesTerminados y asegurar que la espera y notificación entre hilos (el principal y los de los clientes) se maneje de manera segura y coordinada.

Se muestra a continuación varias imágenes de código, explicadas anteriormente:

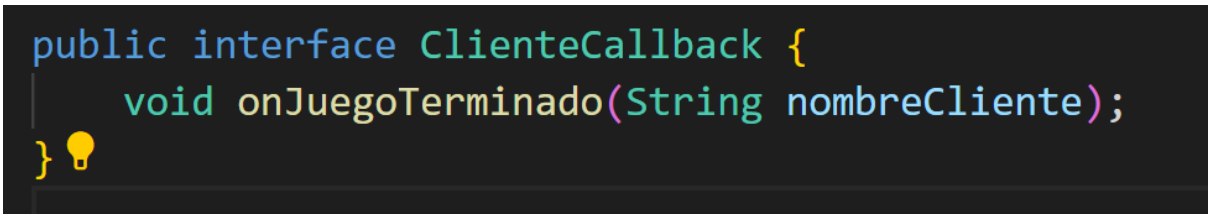
```java
public class Deploy {
    private static int clientesTerminados = 0;
    private static final Object lock = new Object();


    Run | Debug
    public static void main(String[] args) {
        ClienteCallback callback = nombreCliente -> {
            synchronized (lock) {
                clientesTerminados++;
                if (clientesTerminados == 10) {
                    lock.notify(); // Notifica que todos los clientes han terminado
                }
            }
        };

        for (int j = 1; j <= 30; j++) {
            System.out.println("Iniciando juego número: " + j);
            clientesTerminados = 0; // Restablecer el contador para el nuevo juego

            for (int i = 0; i < 10; i++) {
                new Cliente_estresador(nombre:"Santi", callback);
            }
```

```java
synchronized (lock) {
    while (clientesTerminados < 10) {
        try {
            lock.wait(); // Espera a que todos los clientes terminen
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        }
    }
}
```



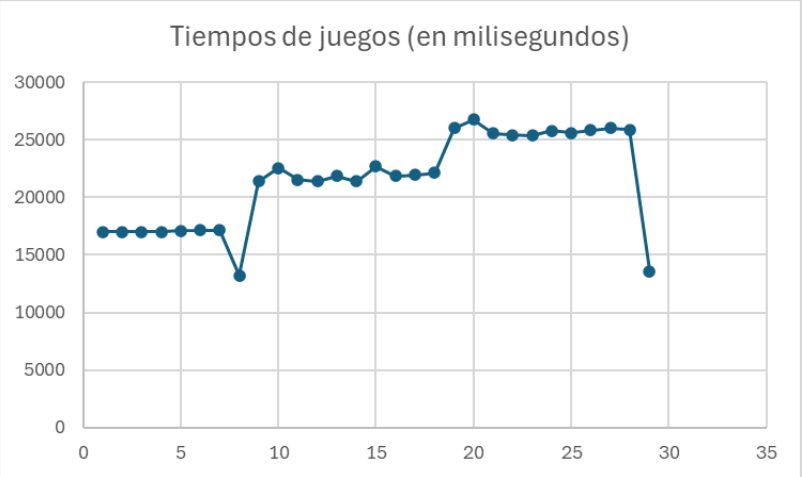
```java
public interface ClienteCallback {
    void onJuegoTerminado(String nombreCliente);
}
```

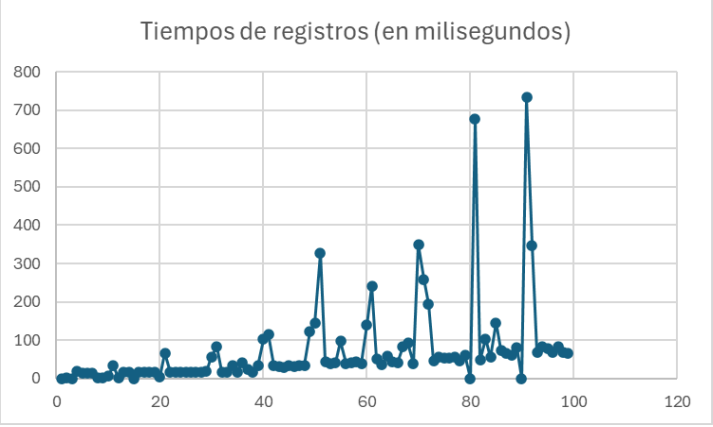
Resultados del experimento

A continuación están las gráficas con los datos de rendimiento para cada iteración del experimento. Se recopilan principalmente dos datos; el tiempo que tomó registrar en el servidor a cada uno de los clientes, y el tiempo que duró cada partida, es decir, cuánto tiempo tomó ganar cada partida.

Configuración 10 clientes:

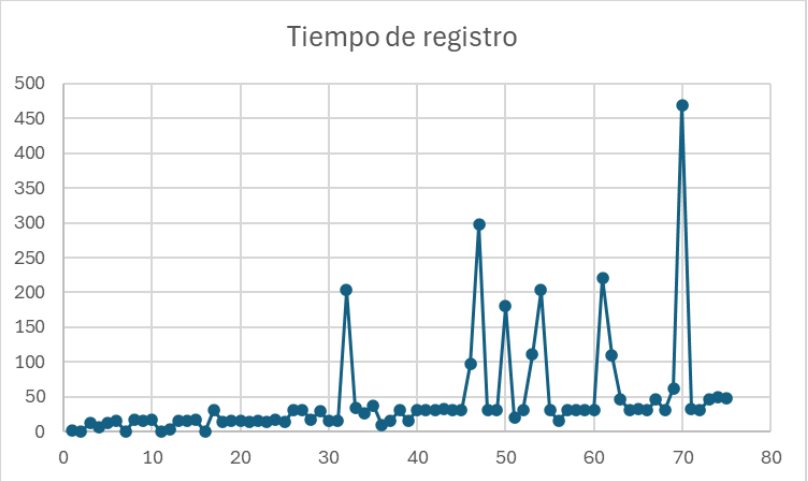
Métricas Tiempo de registro	
Tiempo promedio	71.5959596
Desviación estándar	114.282493
% de registros exitosos	33%
Métricas tiempo de juego	
Tiempo promedio	21485.8621
Desviación estándar	4053.3162

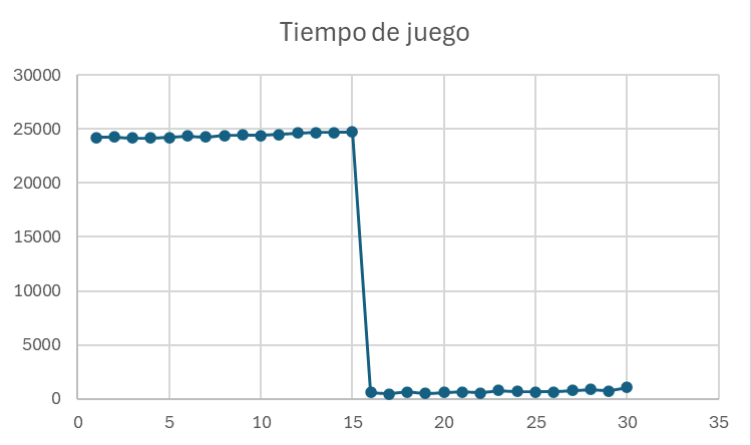




Configuración 15 clientes:

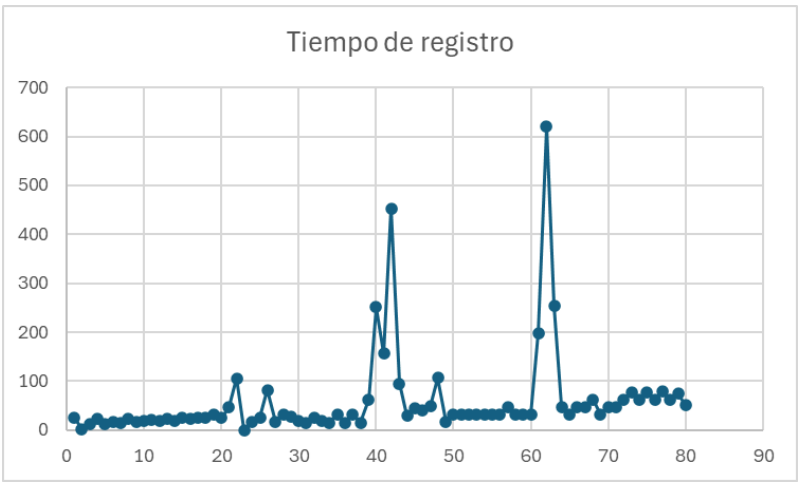
Métricas Tiempo de registro	
Tiempo promedio	46.2533333
Desviación estándar	72.9581689
% de registros exitosos	17%
Métricas tiempo de juego	
Tiempo promedio	12538.6
Desviación estándar	12066.7328

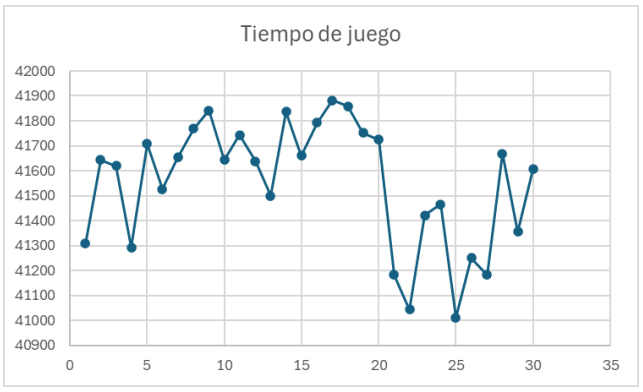




Configuración 20 clientes:

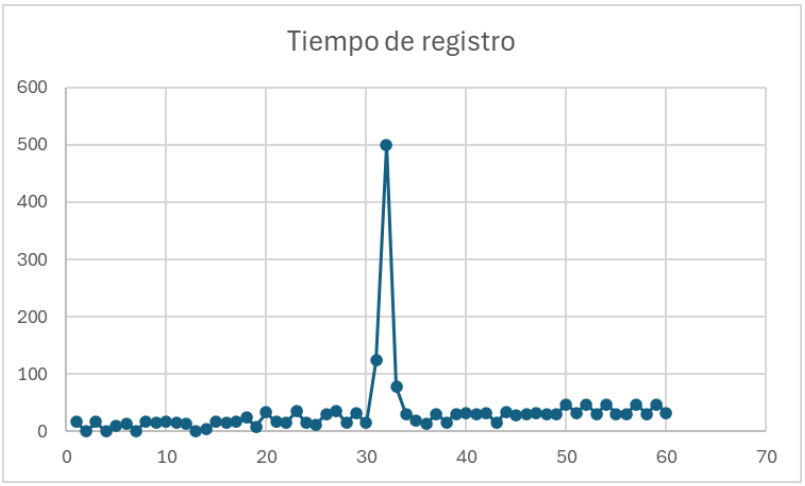
Métricas Tiempo de registro		
Tiempo promedio		58.075
Desviación estándar		90.6605541
% de registros exitosos		13%
Métricas tiempo de juego		
Tiempo promedio		41552.6333
Desviación estándar		247.704842

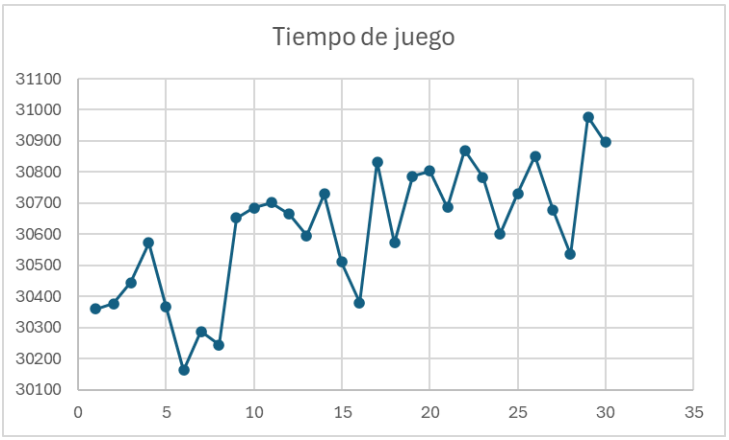




Configuración 30 clientes:

Métricas Tiempo de registro	
Tiempo promedio	34.0833333
Desviación estándar	64.0535506
% de registros exitosos	6.667%
Métricas tiempo de juego	
Tiempo promedio	30611.2333
Desviación estándar	208.641366





Al analizar las tablas proporcionadas, observamos el impacto del número de clientes concurrentes en el tiempo de registro y juego del servidor. A medida que el número de clientes disminuye de 30 a 10, vemos un aumento en el porcentaje de registros exitosos, lo cual indica que el servidor maneja mejor cargas más ligeras.

Para la configuración de 30 clientes, el porcentaje de registros exitosos es del 6.67%, con una desviación estándar alta en el tiempo de registro, lo que sugiere una gran variabilidad e inestabilidad. Por otro lado, con 10 clientes, el porcentaje de éxito

aumenta al 33%, indicando una mayor estabilidad y confiabilidad del servidor. Es interesante el dato de que entre menos clientes había, hubieron más anomalías (picos) en los tiempos de registro.

En cuanto al tiempo de juego, aunque no se ve un patrón claro en el tiempo promedio conforme cambia el número de clientes, las desviaciones estándar altas en las configuraciones de 15 y 30 clientes indican variabilidad en la respuesta del servidor, posiblemente debido a una carga excesiva.

Estos resultados sugieren que el servidor es más efectivo y estable con menos clientes simultáneos, lo que podría deberse a limitaciones de la estructura de datos para manejar conexiones o limitaciones de hardware. Sería beneficioso realizar pruebas adicionales en diferentes hardware para separar las limitaciones del software de las del hardware.

Conclusiones del del experimento

Como se evidencia en las gráficas incluidas en este documento, la eficiencia de nuestro proyecto "Proyecto Alpha" varía significativamente dependiendo del número de clientes conectados de manera concurrente. Inicialmente, el objetivo del experimento era llevar a cabo el experimento múltiples veces pero incrementando el número de clientes desde 50 hasta 100, y así sucesivamente hasta 500, o hasta que el servidor alcanzara su punto de colapso. En nuestras pruebas, observamos que el servidor colapsaba al intentar manejar más de 30 clientes simultáneamente. Para todas las pruebas la variable dependiente era la cantidad de clientes, y la variable independiente eran la cantidad de juegos realizados (30).

Curiosamente, notamos que el colapso del servidor no ocurría durante la conexión de usuarios ni en la acumulación de puntos durante el juego. De forma sorprendente, el fallo se presentaba al final de cada partida, específicamente en el proceso de desconexión de los clientes. Esta peculiaridad nos lleva a considerar que la causa subyacente podría estar relacionada con la estructura de datos empleada para gestionar las conexiones (y desconexiones) de los clientes.

Ante esta situación, hubiera sido ideal dedicar más tiempo a la limpieza y optimización del código para mejorar su eficiencia y, posteriormente, evaluar los cambios en los tiempos de ejecución. Sin embargo, dada la complejidad y el alcance del proyecto, se optó por priorizar el desarrollo de una funcionalidad más completa, sacrificando potencialmente algunos aspectos de eficiencia.

Este hallazgo sugiere la importancia de una revisión detallada del manejo de conexiones y desconexiones dentro del servidor, apuntando a posibles mejoras en la estructura de datos y en la lógica de gestión de conexiones. Para futuros proyectos, sería prudente considerar un balance más equitativo entre la

implementación de funcionalidades y la optimización del rendimiento, asegurando así la escalabilidad y la estabilidad del sistema bajo cargas de trabajo intensivas.

Otro aspecto intrigante que hubiese enriquecido nuestro entendimiento sobre el rendimiento del "Proyecto Alpha" habría sido la realización de pruebas de estresamiento en una diversidad de dispositivos. Al llevar a cabo las pruebas en distintos tipos de hardware, se podría haber evaluado el impacto que tienen los procesadores y otros componentes del sistema en la capacidad de respuesta y estabilidad del servidor. Esta aproximación nos permitiría discernir hasta qué punto las limitaciones observadas durante las pruebas de estresamiento son atribuibles a deficiencias en el código del proyecto, versus las inherentes al hardware utilizado.

Inicialmente, optamos por realizar las pruebas en un solo dispositivo para minimizar la variabilidad en los resultados. Sin embargo, realizar comparativas en distintos dispositivos habría proporcionado información adicional valiosa, permitiendo una mejor comprensión de cómo el hardware afecta el rendimiento del proyecto.

Incluir esta variable en nuestro análisis nos hubiera ofrecido una perspectiva más completa sobre los factores que inciden en el rendimiento del sistema, permitiendo identificar si el colapso del servidor bajo condiciones de carga elevada es principalmente una cuestión de optimización de software o si, por el contrario, se ve significativamente influenciado por las capacidades del hardware. Tal comprensión es crucial para tomar decisiones informadas sobre posibles optimizaciones, tanto en el ámbito del código como en la selección de la infraestructura adecuada para desplegar aplicaciones de alta demanda.

Por tanto, para futuras investigaciones o desarrollos, se recomienda considerar la implementación de pruebas cruzadas en una gama más amplia de dispositivos y configuraciones. Esto no solo ayudaría a mejorar la robustez y eficiencia del proyecto desde el punto de vista del software sino que también proporcionaría valiosos insights sobre las especificaciones mínimas de hardware requeridas para asegurar una experiencia de usuario satisfactoria, incluso bajo condiciones de uso intensivo.