Chatty Open Questions

a) ¿Esta implementación escala cuando el número de clientes aumenta?

No, esta implementación no escala bien a medida que aumenta el número de clientes. En primer lugar, la estructura de datos para gestionar los **clientes** es una **lista** (en process_request) **que crece linealmente**, en la cual para buscar o eliminar un cliente (list:delete) se tiene una complejidad O(n). En segundo lugar, el **servidor maneja todas las conexiones**, siendo un cuello de botella porque maneja todas las solicitudes en tan solo un proceso, el cual podría colapsar por la sobrecarga de mensajes, dejando el servicio caído.

b) ¿Qué pasa si el servidor se desconecta?

El sistema **maneja todos los mensajes en un único servidor** SPOF, por tanto, si el servidor se desconecta, todos los clientes perderán la conexión y dejarán de recibir mensajes. No hay resiliencia en el sistema, una vez caído los clientes no pueden encontrar otro servidor de respaldo en la red.

c) ¿Los mensajes de un solo cliente están garantizados a ser entregados en orden?

Sí, los mensajes de un solo cliente se entregan en el mismo orden en que fueron enviados, ya que hay un único servidor que reenviará los mensajes al único cliente. Erlang garantiza que los mensajes enviados entre dos procesos en un instante determinado, sin interferencias de otros, se entregan en orden.

d) ¿Los mensajes concurrentes están garantizados a ser entregados en orden?

No hay garantía absoluta de orden entre diferentes clientes. Aunque Erlang garantiza el orden entre dos procesos específicos, no garantiza orden entre múltiples procesos. Si varios clientes envían mensajes al mismo tiempo, el servidor los recibe en el orden que el sistema operativo los entrega.

Las alternativas para garantizar esto, es agregar un **timestamp** a cada mensaje y ordenarlos antes de enviarlos o bien **priorizar mensajes de determinados clientes**, programando un orden de **receive**.

e) ¿Es posible que C reciba la respuesta de B antes que el mensaje de A?

No, no es posible. Esto se debe a que Erlang garantiza el orden de entrega de los mensajes entre dos procesos FIFO, y que tan solo hay un servidor: si un proceso A envía dos mensajes a otro proceso B, estos serán recibidos en el mismo orden en que fueron enviados.

f) Si un cliente entra o sale del chat mientras el servidor está emitiendo un mensaje, ¿recibirá ese mensaje?

No, el cliente que se una después de que el servidor haya enviado un mensaje no lo recibirá. El servidor para enviar mensajes tiene que registrar previamente a todos los que se unen, y no lo puede hacer a la hora que realiza un broadcast. Es decir, el servidor realiza un broadcast, que envía el mensaje solo a los clientes que están en la lista *Clients* en ese momento.

Sí que recibirá el broadcast si hace leave, puesto que todavía está a la lista y la función process_requests del cliente está escuchando.

Una alternativa sería mantener un historial de mensajes y enviarlos a los nuevos clientes cuando se unan.

g) ¿Qué pasa si un servidor se desconecta?

Si un servidor se desconecta, el **sistema sigue funcionando mientras haya otros servidores activos**. Es decir los clientes que estén conectados al servidor caído, pierden la conexión y se desconectan del chat. Para poder reincorporarse, deben de conectarse al otro servidor restante disponible.

h) ¿Siguen siendo válidas las respuestas de las preguntas c), d) y e)?

Respecto a las preguntas c) i d), se sigue sin poder asegurar con Erlang el orden correcto de los mensajes entre dos nodos si existe más de un proceso, por tanto la respuesta para estas sigue siendo la misma.

Respecto a la pregunta e), sí que puede darse el caso si la latencia de respuesta es inferior a la latencia del mensaje inicial (latencia de red entre servidores). También puede deberse a los tiempos de procesamiento de cada nodo o proceso.

Ejemplo: Un cliente A y un cliente B están conectados a un servidor S1 y un cliente C (conectado a un servidor S2 envía un mensaje inicial que será respondido por B. El cliente A (más próximo a B que a C) vería antes la respuesta (hecha por B) que el mensaje inicial (hecho por C) si la latencia de C es superior que a la de su compañero B.

Para controlar esto, una solución es etiquetar respuestas con un identificador del mensaje original para que los clientes puedan reconstruir la secuencia.

i) ¿Qué podría pasar con la lista de servidores si varias solicitudes de unión o salida ocurren al mismo tiempo?

La lista de servidores puede quedar inconsistente debido a las condiciones de carrera, por ejemplo, cuando hay peticiones concurrentes de 'server_join_req' en servidores diferentes. Esto se debe porque **cada servidor mantiene su propia copia de la lista y la actualiza de forma independiente**, podrían producirse discrepancias temporales, como que algunos servidores no conozcan la existencia de otros o que intenten enviar mensajes a servidores que ya salieron. Esto podría generar fallos en la comunicación hasta que la lista se estabilice.

j) ¿Cuáles son las ventajas e inconvenientes de esta aplicación con respecto a la anterior?

La principal ventaja de server2.erl es que, al tener servidores replicados, **se mejora la tolerancia a fallos** y la **escalabilidad**, ya que si uno falla, los demás pueden seguir operando y se puede **distribuir mejor la carga**.

En cambio, su principal inconveniente es la **mayor complejidad** y **latencia**. La necesidad de mantener y sincronizar la lista de servidores (especialmente ante solicitudes concurrentes de unión o salida) puede generar inconsistencias temporales y aumentar la latencia debido a la comunicación extra entre servidores.

- Dos clientes conectados a un mismo servidor tendrán una latencia más baja que si un cliente y otro cliente a otro servidor se intercambian mensajes

La implementación de un chat como primera práctica de la asignatura hace que se pueda ver como funcionan a pequeña escala las redes distribuidas y sus principales aspectos y ventajas de su utilización, así como el papel de cliente-servidor y todos los pasos que siguen para poderse comunicar.