PCS: Diseño Chat con JAutomata Y Modificación Algoritmo Berkeley (ZeroMQ)

Programación en sistemas Cloud

Alumno

José Javier Gutiérrez Gil

jogugi@posgrado.upv.es| jogugil@gmail.com

27 novembre 2024

Profeso/i

Bataller Mascarell, Jordi (batallerdsic.upv.es)

Índice

CAPÍTULO 1

Sistema Chat

1.1 introducción

Objetivos

El objetivo de este proyecto es diseñar e implementar un sistema de chat funcional y sencillo, permitiendo a los usuarios interactuar en tiempo real con un enfoque en simplicidad y comodidad tanto en el cliente como en el servidor. El presente documento detalla las especificaciones funcionales para el desarrollo de este sistema, el cual permitirá a los usuarios comunicarse mediante mensajes de texto en tiempo real, asegurando una experiencia de usuario óptima y accesible.

Alcance

El sistema de chat permitirá a los usuarios iniciar sesión, enviar y recibir mensajes en tiempo real. En esta primera versión solo existirá una única sala, un único tipo de usuario y no existirá comprobaciones de sesión. La pantalla principal sólo mostrará el historial de mensajes y la posibilidad de desconectarse de la sesión de usuario.

1.1.1 Primer tentativa del diseño del sistema chat

1.1.2 Diagrama de clases

Se muestra la primera versión del diagrama de clases principales que conlleva el diseño e implementación del chat, tanto el módulo cliente como servidor. Notar que no se han añadido las clases asociadas al control de formularios y gestión de sesión del usuario. Se incluye en el esquema las clases base de usuario, sala y mensaje que son las principales. Además se añaden las clases y módulos presentes en el servidor. Notar que no se han añadido en el esquema las funciones get/set para los atributos de las clases. Un resumen del esquema de clases para la implementación de nuestro sistema chat se puede ver en la imagen 1.1

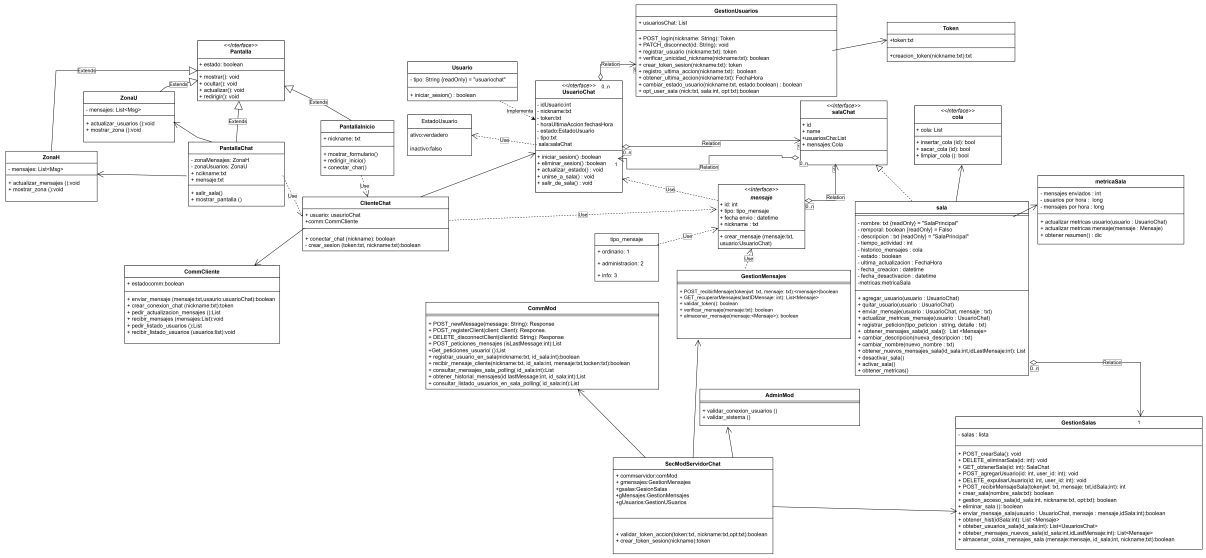

Figura 1.1. Diagrama Clases principales

1.1.3 Diagramas de Interacción

Inicio de Sesión

En la figura 1.2 podemos ver un diagrama de interacción el proceso de entrada de un usuario al chat. Un resumen de los pasos son:

1. El usuario introduce su nombre.
2. El cliente envía la solicitud al servidor para validarlo y crea un token de sesión JWT.
3. El servidor registra el usuario en la sala principal y responde con éxito o error.
4. Si responde con éxito. El cliente crea un usuario y lo registra en el polling para la sala principal. Envía peticiones a través del polling para refrescar la pantalla del chat (histórico de mensajes en al sala, usuarios activos en la sala).

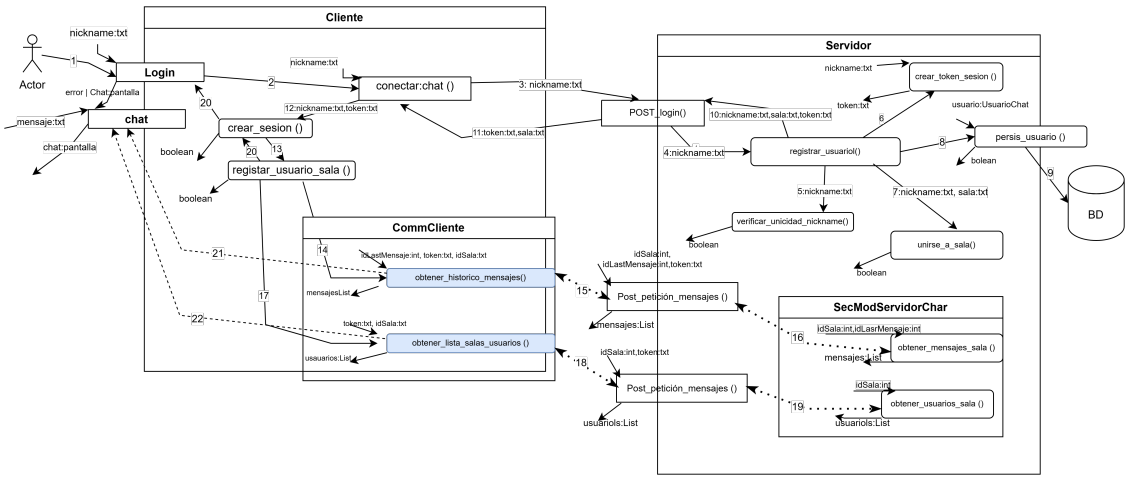

Figura 1.2. Inicio sesión en el cliente chat

Envío de Mensajes

En la figura 1.3 podemos ver un diagrama de interacción el proceso de envío de mensajes de un usuario a la sala del chat. Un resumen de los pasos son:

1. El usuario escribe un mensaje.
2. El cliente envía el mensaje al servidor.
3. El servidor comprueba y valida el token, el formato del mensaje, crea un objeto mensaje y lo almacena en la base de datos si todo ha ido bien.
4. El servidor almacena el objeto mensaje en al cola de mensajes de la sala correspondiente (sólo existe una única sala)
5. El servidor devuelve true o false se se ha podido enviar correctamente el mensaje
6. El cliente refresca la pantalla chat Si ha ido mal le sale un mensaje de error que sólo puede ver él.
7. El cliente cada cierto tiempo refresca la pantalla del chat con el histórico de mensajes de la sala ordenado por fecha d envío y el listado de usuarios presentes. Esto lo realiza bien pro petición al servidor o si el servidor emite un evento de actualización de datos.

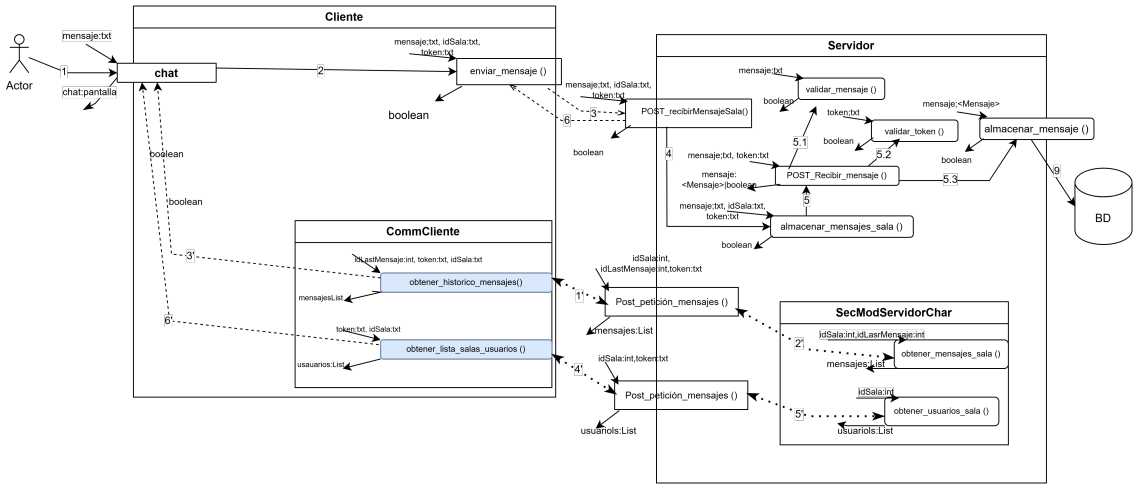

Figura 1.3. Flujo envío mensaje del usuario a la sala del chat

Mecanismos Comunicación

A continuación, se presentan dos diagramas de interacción que ilustran los procesos de comunicación en el sistema de chat. El primer diagrama describe el uso de API REST para gestionar la conexión al chat y el envío de mensajes nuevos por parte del usuario, complementado con sockets para recuperar el historial de mensajes de la sala y la lista de usuarios presentes en ella, implementando el patrón polling (1.4).

El segundo diagrama muestra una alternativa basada en API REST combinada con tecnología SSE (Server-Sent Events), que implementa el patrón push para una transmisión más eficiente de datos en tiempo real (1.5).

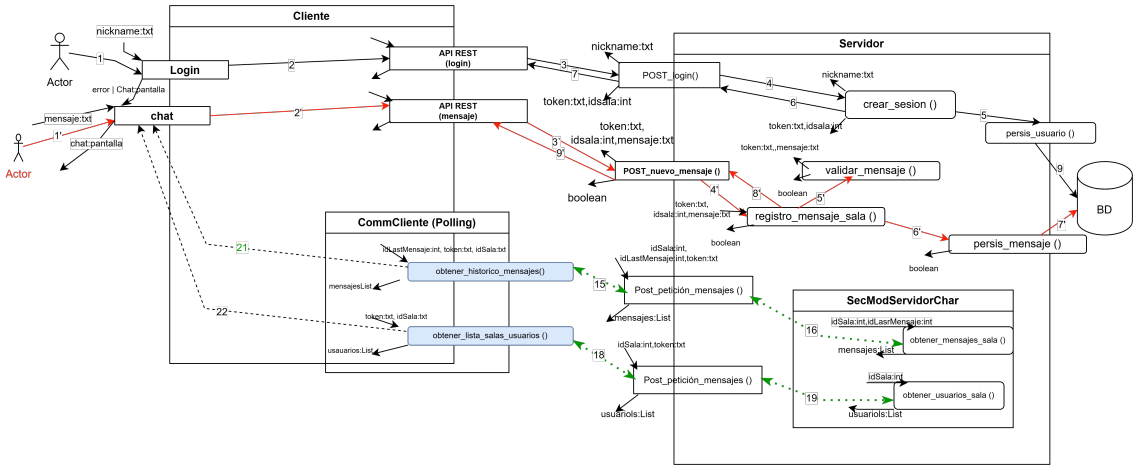

Figura 1.4. Flujo Autenticación y comunicación APIREST + Sockets (Polling)

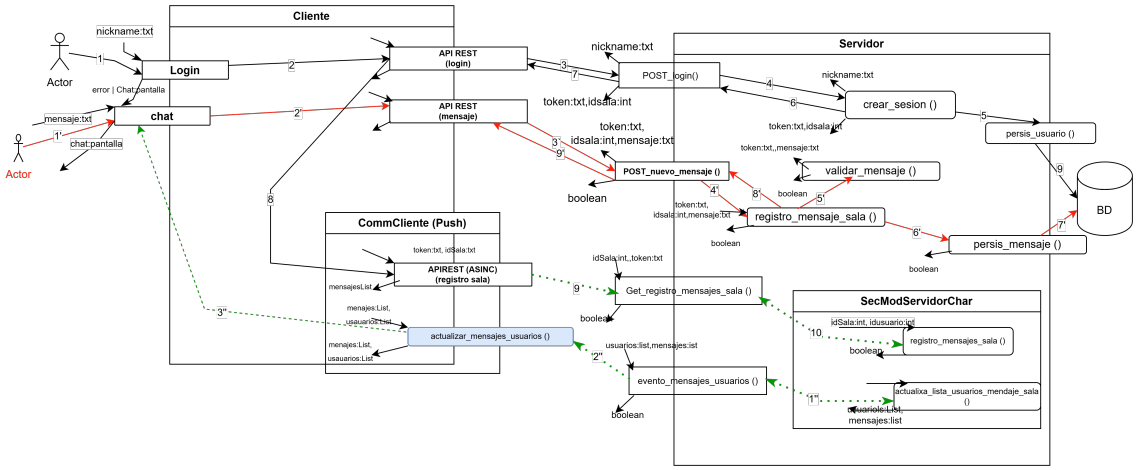

Figura 1.5. Flujo Autenticación y comunicación APIREST + SSE

1.2 Implementación del sistema Chat

Tecnologías Utilizadas

- **Frontend:** Node.js, typescript con React.js o React Native (según la plataforma deseada: web o móvil). El diseño será responsive utilizando styled-components o Material-UI.
- **Backend:** Go. Ambos soportan APIs RESTful y autenticación basada en JWT.

1.2.1 Diseño de las Comunicaciones

1.2.2 Detalles del Patrón Polling

El patrón de polling se implementa mediante las siguientes acciones y requisitos:

Solicitud de Nuevos Mensajes (Polling del Cliente)

Los clientes deben hacer una solicitud periódica para obtener los nuevos mensajes de una sala. Cada vez que un cliente hace una solicitud GET, el servidor debe devolver únicamente los mensajes que hayan

sido enviados después del último mensaje conocido por el cliente. Esto se logra mediante el parámetro `lastMessageId` que los clientes incluirán en la solicitud para indicar hasta qué mensaje se ha leído.

Colas Cíclicas para Mensajes

Cada sala tiene su propia cola cíclica que almacena los mensajes. El tamaño máximo de la cola está limitado a N mensajes. Cuando se envía un nuevo mensaje y la cola está llena, el mensaje más antiguo es eliminado para dar espacio al nuevo. Esto permite que las salas no sobrecarguen la memoria con mensajes antiguos que ya no son relevantes.

Actualización de la Cola Cíclica

Cada vez que un mensaje nuevo es recibido, se agrega a la cola cíclica de la sala correspondiente. Los mensajes deben ser entregados a los clientes registrados en la sala, garantizando que todos los mensajes sean distribuidos en orden de llegada.

Manejo de Sesiones Activas y Registro de Clientes

Los clientes deben estar registrados en una sala específica para recibir mensajes. Si un cliente no está registrado en la sala, no podrá recibir los mensajes. El servidor debe mantener un listado de usuarios y salas donde se encuentran activas cada usuario.

Lista d esalas y usuarios

Al igual que el histórico de mensajes, el cliente chat debe mantener un listado de salas activas y existentes en el chat. cuando el usuario entra al sistema chat, es el cliente quién obtendrá del servidor la salas existentes y los usuarios presentes en cada sala. Al igual que con el histórico de mensajes, el cliente utiliza el polling para actualizar dicha información. Si se opta por no usar polling y si SSE, es el cliente el que debe emitir una petición de actualización mediante un APIREST.

La diferencia entre usar sockets dedicados sobre los cuales realizar peticiones de actualización del histórico de mensajes y usar tecnología SSE es quién mantiene el listado de salas activas del usuario. En el caso de usar SSE, al ser una comunicación unidireccional, el servidor debe mantener un registro de qué clientes están activos en cada sala. Por tanto, cada vez que el usuario sala o entra de una sala debe enviar una petición de registro o de salida de sala al servidor, esto se realizará mediante un apirest (endpoint /joinRoom<salas>). Si por el contrario se usan sockets dedicados, es el mismo cliente chat quien mantiene la salas donde el usuario está activo. Es en la petición de actualización de histórico de mensajes donde el cliente le envía junto al token de sesión JWT, el listado de salas activas.

Tiempo de Polling y Control de Solicitudes

El servidor debe definir un intervalo de tiempo mínimo entre cada solicitud de polling para evitar sobrecargar el sistema. Los clientes deben realizar las solicitudes de polling de manera eficiente y no con demasiada frecuencia para evitar una carga innecesaria en el servidor.

En el caso de usar SSE, el cliente refresca cada cierto tiempo pero no realiza ninguna petición al servidor. Es el servidor quien actualiza los históricos de mensajes en el cliente, cuando en ese tiempo refresque el frontend, le aparecería al cliente el listado de mensajes actualizado.

Respuestas a Solicitudes de Polling

El servidor debe responder a las solicitudes de polling de manera rápida, enviando solo los mensajes nuevos y asegurándose de que los mensajes se entreguen correctamente a los clientes correspondientes.

Integración con el Módulo de Procesamiento de Mensajes

El servidor debe recibir los mensajes de los clientes a través de la API REST (endpoint: /messagesnew?<mensaje>) y una vez comprobado si el token de sesión es válido, pasarlos al módulo de procesamiento de mensajes, donde se validarán, procesarán, y luego almacenarán en las colas cíclicas de la sala. Una vez procesado el mensaje, se distribuye a los clientes registrados en la sala mediante el patrón de polling o el sistema SSE actualizará las estructuras del cliente que mantenga el histórico de mensajes.

Eliminación Automática de Mensajes Antiguos

El servidor debe eliminar automáticamente los mensajes antiguos cuando se alcance el límite de N mensajes en la cola cíclica, lo que garantiza que la memoria no se llene innecesariamente y que solo los mensajes recientes sean accesibles.

CAPÍTULO 2

Algoritmo de Berkeley

El algoritmo de Berkeley es un método de sincronización de relojes en sistemas distribuidos. Se basa en la existencia de un líder (coordinador) que ajusta los relojes de los nodos en el sistema para mantener la coherencia temporal.

2.1 Objetivos

Refinar el diseño del "Algoritmo de Berkeley" que aparece en `"Apuntes_NotacionDiseño.pdf"` basándolo en sockets de ZeroMQ. A continuación, se explica paso a paso el algoritmo:

- **Inicio del proceso de sincronización:** El líder toma la iniciativa para sincronizar los relojes en el sistema y envía una petición a todos los nodos seguidores solicitando sus marcas temporales actuales.
- **Seguidores responden al líder:** Cada seguidor responde al líder con su marca temporal actual y, opcionalmente, con el tiempo que tardó en procesar la solicitud (latencia de ida y vuelta).
- **Líder calcula los ajustes:** Una vez que el líder recibe todas las marcas temporales, calcula el tiempo promedio ajustado del sistema (considerando posibles retrasos) y determina la diferencia de tiempo entre el promedio y el reloj de cada nodo.
- **Líder envía ajustes:** El líder comunica a cada nodo un ajuste temporal que especifica cuánto adelantar o retrasar su reloj.
- **Seguidores aplican ajustes:** Cada seguidor ajusta su reloj local según el valor recibido. En algunos casos, los ajustes se aplican de forma gradual para evitar cambios bruscos.
- **Finalización:** Los relojes de los nodos quedan sincronizados dentro de los límites de error permitidos por la red.

El algorimo de berkeley representado en IOAutomata se muestra en al figura 2.1. Adaptaremos eta representación usando zeroMQ para las comunicaciones.

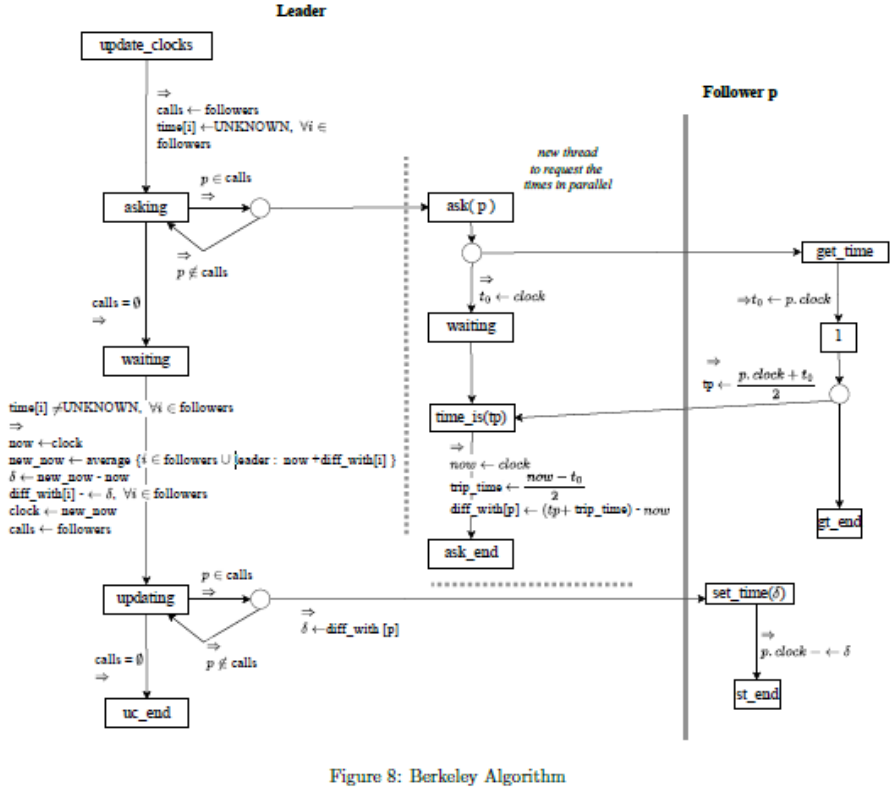

Figure 8: Berkeley Algorithm

Figura 2.1. Algoritmo Berkeley JAutomata (ref. Jordi Ballester «notacion.pdf»)

2.1.1 Cambios en el diagrama para representar ZeroMQ

Se proponen los siguientes cambios en el diagrama del algoritmo Berkeley para integrar sockets ZeroMQ:

- **Líder (Request/Reply):**

  El lider mantiene una nueva variable $req[i]$ que permite verificar que un nodo seguidor ha respondido correctamente a la solicitud de tiempo del líder. Dicha variable se inicializa a false para cada follower ($req[i] = false$ y $time[i] = UNKNOWN$)

  En el flujo para cada hilo donde el lider obtiene el tiempo de cada nodo p se realia lso sigueitnes cambios:

  - Cambiar el nodo *asking* para reflejar el envío de un mensaje ZeroMQ con el patrón REQ (Request) para pedir los tiempos de los seguidores. Notar que cada nodo p tiene un REP para esperar las peticiones del lider. (El líder abrirá un socket REQ para cada nodo y enviará un mensaje de solicitud de tiempo.)
  - Eliminar el nodo *waiting* ya que en el anterior ya espera la respuesta REP (Reply) del nodo p. Además, si se llega el tiempo t_p del nodo p actualizamos el valor de $req[p] = True$ para controlar que followers devuelven correctamente su tiempo.

  Una vez el lider obtiene todos los tiempos de cada follower, modificamos las sigueintes operaciones;

- Calculamos el tiempo promedio pero sólo con el número de followers que han contestado correctamente, es decir, aquellos que tienen true en la variable req. Por tanto, modificamos el cálculo:

$$\texttt{new\_now} \leftarrow \text{average}\,(i \in \text{followers} \wedge \text{req}[i] = \text{true} \cup \text{leader} : \text{now} + \text{diff\_with}[i])$$

- Modificamos la variable `diff_with` solo para los nodos que han contestado con su tiempo:

$$\texttt{diff\_with}[i] \leftarrow \delta, \forall i \in \text{followers}, \text{req}[i] = \text{true}$$

- Creamos el conjunto `call` sólo con estos nodos que tienen la variable `req` a true:

$$\texttt{call} \leftarrow \{i \in \text{followers} : \text{req}[i] = \text{true}\}$$

  De esta forma, el líder sólo enviará el delta para que modifique su tiempo local solo a los nodos que han contestado.

- **Seguidores (Request/Reply):**
  - Cambiar el nodo *get_time)* para representar la recepción de una solicitud REQ del líder.
  - Añadir un nodo para que cada seguidor envíe una respuesta REP con el tiempo calculado o el ajuste solicitado.
  - Modificar el nodo set_time añadiendo un nodo en cada seguidor que indique que está esperando la solicitud de ajuste y que recibirá el ajuste del líder mediante una respuesta REP.

Con esta corrección, el cálculo de `new_now` ahora solo toma en cuenta los seguidores que han respondido correctamente (aquellos donde `req[i] = true`) y el líder, asegurando que solo se promedien los tiempos válidos. Esto es crucial para sincronizar los relojes de manera correcta en un sistema distribuido. Esta nueva variable ayuda a filtrar los nodos que han respondido correctamente. Si el líder envía una solicitud de tiempo (`REQ`) y no recibe respuesta (por algún fallo en la red o en el nodo), entonces `req[i]` se mantiene como `false`, y ese nodo no se incluye en el cálculo del promedio de tiempo (`new_now`). Mejora la robustez del algoritmo y facilita la comprensión y el manejo de errores. Asegura que solo los nodos que han respondido exitosamente sean tomados en cuenta para el cálculo del nuevo tiempo ajustado. Esto ayuda a evitar errores de sincronización si algún nodo no puede ser alcanzado por el líder.

El esquema del algoritmo, adaptado a ZeroMQ utilizando REQ/REP y con la utilización de la variable $req[i]$ se refleja en la imagen 2.2.

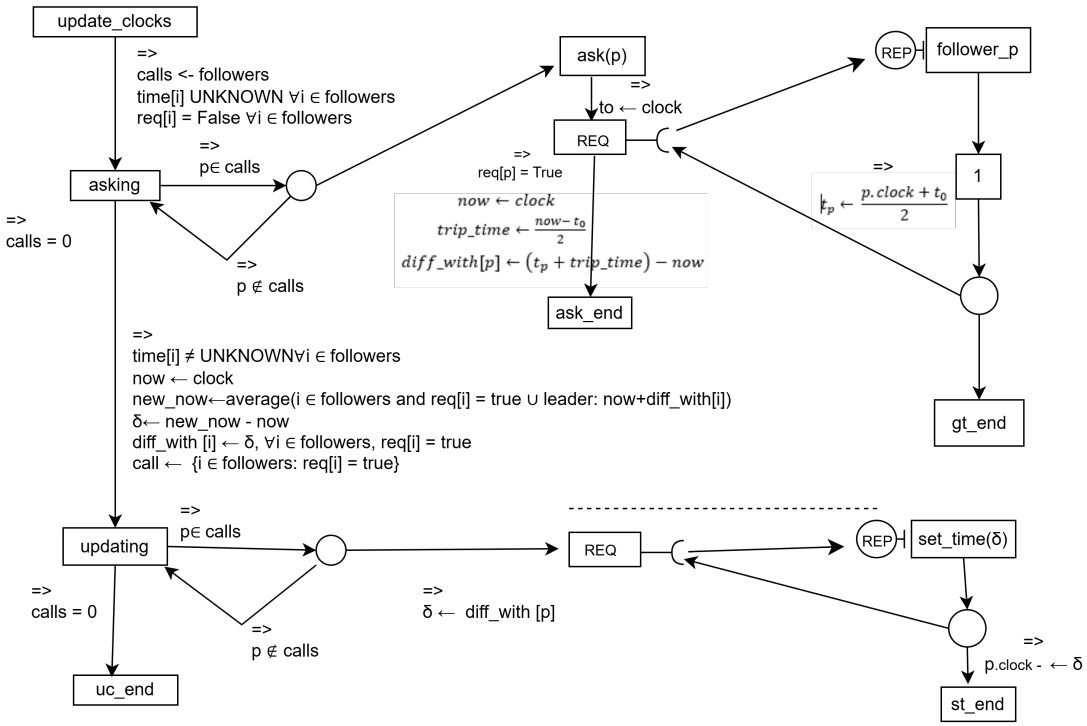

Figura 2.2. Algoritmo Berkeley con Sockets REQ/REP ZeroMQ