ALUMNO: José María Fernández

Asignatura: Programación de Sistemas Distribuidos

Curso: 2023/2024 Fecha: 22-05-2024
Semestre: 2º

PRACTICA 5:

CHAT SINCRONIZADO

CON PERSISTENCIA Y METRICAS

INDICE:

Presentación

Se ha tomado como base del proyecto la práctica que se hizo por grupos sobre el chat empleando websockets, añadiendo modificaciones y nuevos servicios relacionados con conceptos vistos en la asignatura.
Por otra parte, se ha querido trabajar con otras tecnologías que no se han visto en clase, pero sí se han mencionado y se ha establecido una relación con los sistemas distribuidos.

En este documento se analizará la arquitectura del servicio, completo, se analizarán los parámetros de diseño y se explicarán las tecnologías empleadas.

Caso Inicial

Como se ha mencionado previamente, como proyecto base se ha tomado la práctica del chat de los websockets, la cual era un aplicativo que a través de una arquitectura modelo cliente servidor, levantaba un servidor en NodeJS y un endpoint público a través del cual variosclientes
podían acceder a este y enviar mensajes instantáneamente, todo esto a través de una red local y enviando nada más que una cadena de caracteres.

REF: https://github.com/Josojmf/Practica_4_Distribuidos

Caso de Uso

Para poder añadir más funcionalidades a este servicio se han analizado los conceptos vistos en clase y se ha concluido que una buena modificación podría ser intentar sincronizar de alguna forma el chat para poder observar el sesgo que diferentes métodos de sincronización tendrían dentro de un aplicativo, teniendo así 4 relojes observables por mensaje del chat. Cada vez que un mensaje es enviado se hace una llamada a una API externa, simulando así el caso de sincronización por servidor central, también se hace una llamada a un endpoint interno, en el que está levantado un servidor NTP, posteriormente explicaré, y por último tenemos dos relojes locales, que simplemente llaman a los relojes propios del servidor, uno actualizándose constantemente, y otro añadiendo su tiempo local a cada mensaje, pudiendo ver así el sesgo que presentan estos métodos de sincronización entre ellos.

Tecnologías empleadas

WEBSOCKETS

En la práctica anterior, ya se trabajó con websockets, que son un canal full duplex de comunicación que corre sobre la capa de transporte y aplicación.

```
const io = require("socket.io")(http);
form.addEventListener('submit', function (e) {
  e.preventDefault();
  if (input.value) {
    socket.emit('chat message', input.value);
    input.value = '';
  }
});
```

NODEJS

Como framework de programación y runtime se ha usado nodejs con html y css, sin ningún framework extra ni que englobe funcionalidades extra, para poder observar bien como se hace únicamente empleo de llamadas REST,ntp y ws para la comunicación entre los distintos servicios

```
"scripts": {
  "startchat": "node src/index.js",
  "startntp": "node src/ntpserver.js",
  "dev": "nodemon ./CHAT/index.js -watch './CHAT/**/*.html' -watch './CHAT/**/*.js'",
  "devntp": "nodemon ./NTP/ntpserver.js",
  "start": "npm run startchat & npm run startntp ",
  "build-local": "docker push josojmf55/practica_5_distribuidos:upload & docker compose up --build ",
  "monitor": "node -r newrelic ./CHAT/index.js"
},
```

LLAMADAS A APIS EXTERNAS

Para emular una sincronización a través de un servidor central se ha hecho una llamada a una API pública ajena a todo el proyecto, lo cual vuelve a aportar más peso l parámetro de diseño de heterogeneidad y transparencia, ya que se desconoce completamente la arquitectura, sistema operativo y demás servicios que corren tras esta API externa, pero por medio del uso de intercambio de mensajes, en este caso http y rest se puede lograr una comunicación sólida con este servicio.
La hora local se ha codificado para que figure la de Madrid, para modificar la hora local a otra franja horaria habría que cambiar simplemente el código donde figuran esta localización.

```
async function fetchTime() {
  var timeAPIURL = "http://worldtimeapi.org/api/timezone/Europe/Madrid"
  const response = fetch(timeAPIURL).then(response => response.json()).then(data => {
    alert(data.datetime);
    console.log(data.datetime);
    return data;
  });
}
```

DOCKER

Para facilitar la implementación, gestión de dependencias, portabilidad y mantener la independencia de todo el sistema del sistema operativo subyaciente, ya sea de las máquinas servidores, o de los posibles clientes, se ha implementado el despliegue de todo el aplicativo mediante el uso de contenedores docker

```
ARG NODE_VERSION=20.11.1

################################################################################
# Use node image for base image for all stages.
FROM node:${NODE_VERSION}-alpine as base

# Set working directory for all build stages.
WORKDIR /usr/src/app


################################################################################
# Create a stage for installing production dependecies.
FROM base as deps

# Download dependencies as a separate step to take advantage of Docker's caching.
# Leverage a cache mount to /root/.npm to speed up subsequent builds.
# Leverage bind mounts to package.json and package-lock.json to avoid having to copy them
# into this layer.
RUN --mount=type=bind,source=package.json,target=package.json \
    --mount=type=bind,source=package-lock.json,target=package-lock.json \
    --mount=type=cache,target=/root/.npm \
    npm ci --omit=dev

################################################################################
# Create a stage for building the application.
FROM deps as build
```

SINCRONIZACIÓN LOCAL

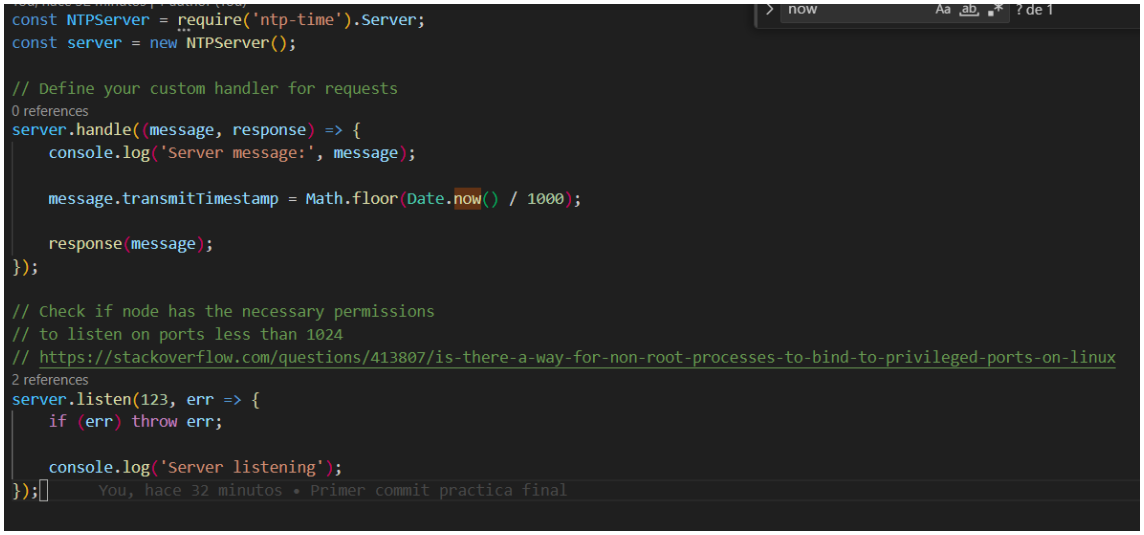
```
function startTime() {
  const today = new Date();
  let h = today.getHours();
  let m = today.getMinutes();
  let s = today.getSeconds();
  m = checkTime(m);
  s = checkTime(s);
  document.getElementById('txt').innerHTML = "Local Time: " + h + ":" + m + ":" + s;
  setTimeout(startTime, 1000);
}

function checkTime(i) {
  if (i < 10) { i = "0" + i };  // add zero in front of numbers < 10
  return i;
}
```

Se ha hecho uso de un reloj contador iterativo que va actualizando el tiempo segundo a segundo, este reloj es llamado con el hook html onload

```
<body onload="startTime()">
```

SERVIDOR NTP



Aquí se muestra el servidor NTP que se levanta junto a todo el programa, este será el encargado de suministrar la hora que el procesa.

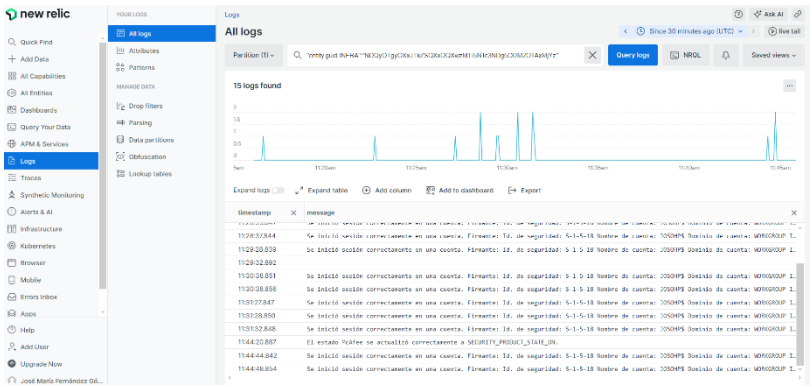
CLIENTE NTP

Aquí tenemos la contraparte del cliente NTP, simplemente crea una instancia de un cliente NTP, y hace una llamada al servidor para que este le devuelva el tiempo, luego simplemente se formatea para obtener una cadena de texto claro que podamos transmitir y leer por el websocket.

NEWRELIC MONITOR

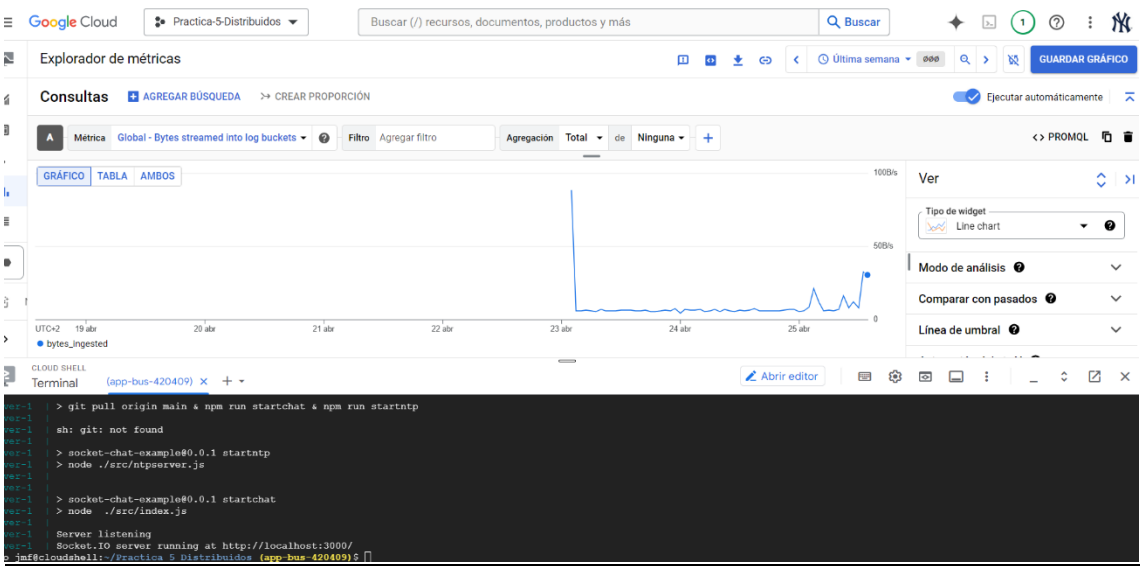
Se han monitorizado la aplicación de diferentes maneras, la mayoría de los hostings ya proveen de métodos de monitorización de datos, carga y servicios, pero se ha hecho uso de este servicio ya que de manera externa se puede acceder a datos de diagnóstico locales, y se pueden pasar pruebas de carga y diferentes puntos de fallo, a través de este script se corre el aplicativo con monitoreo por newrelic

```
"monitor": "node -r newrelic ./CHAT/index.js"
```

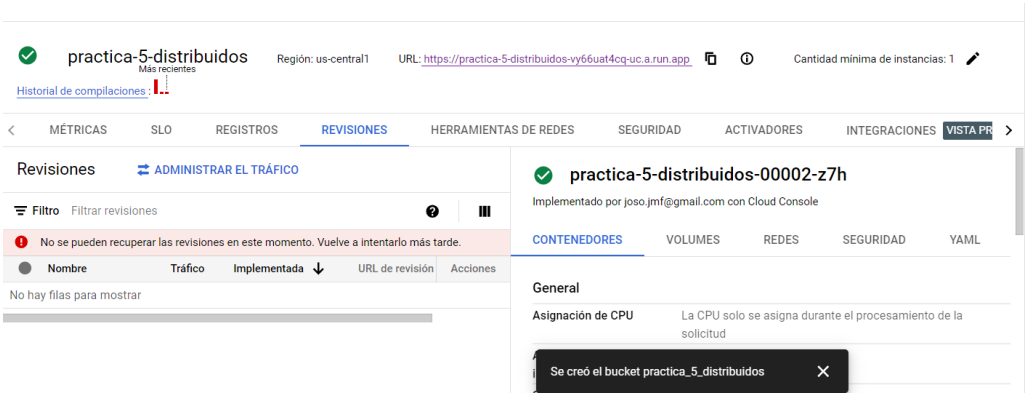


GOOGLE CLOUD

Se han probado diferentes maneras de alojar el servicio en proveedores web y SAAS, Google cloud siendo uno de ellos, el cual ofrecí muchas herramientas de monitoreo, pero fui incapaz de desplegarlo de manera que fuera público a todo el mundo y no solo a personas con invitación directa

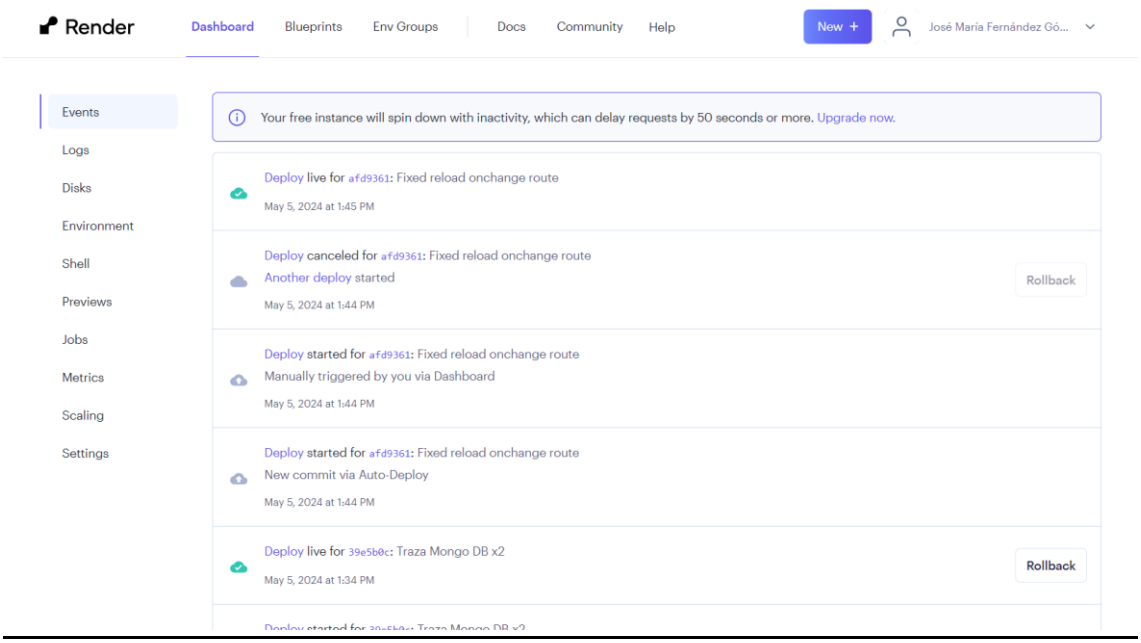


Aquí se puede observar el panel de Google cloud, el cual fue capaz de mostrar las métricas para algunos tests que pude realizar de forma local donde se veía la carga y las peticiones realizadas en una gráfica.

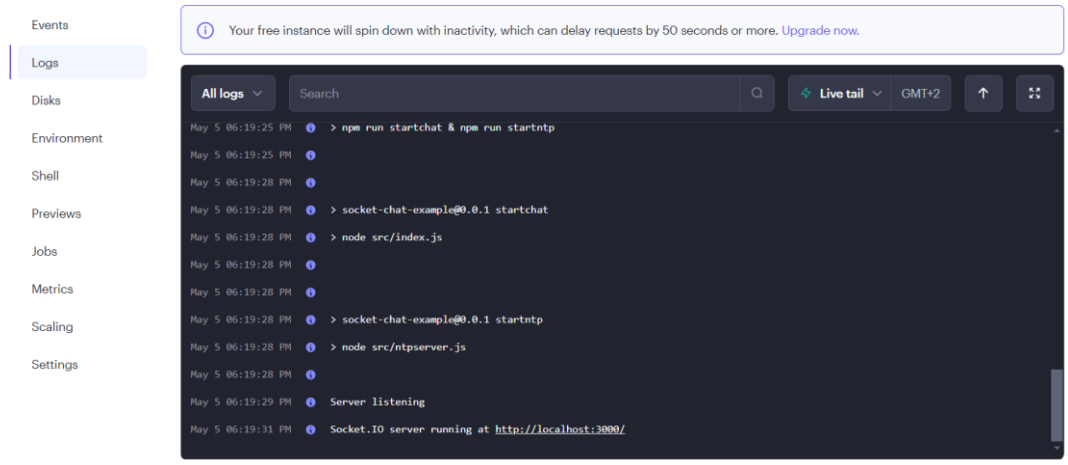


ONRENDER HOSTING

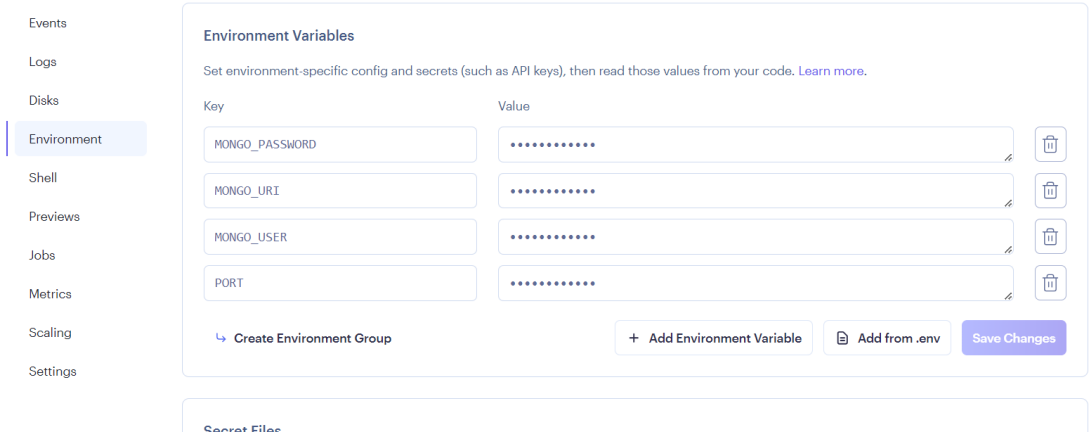
Deploys:



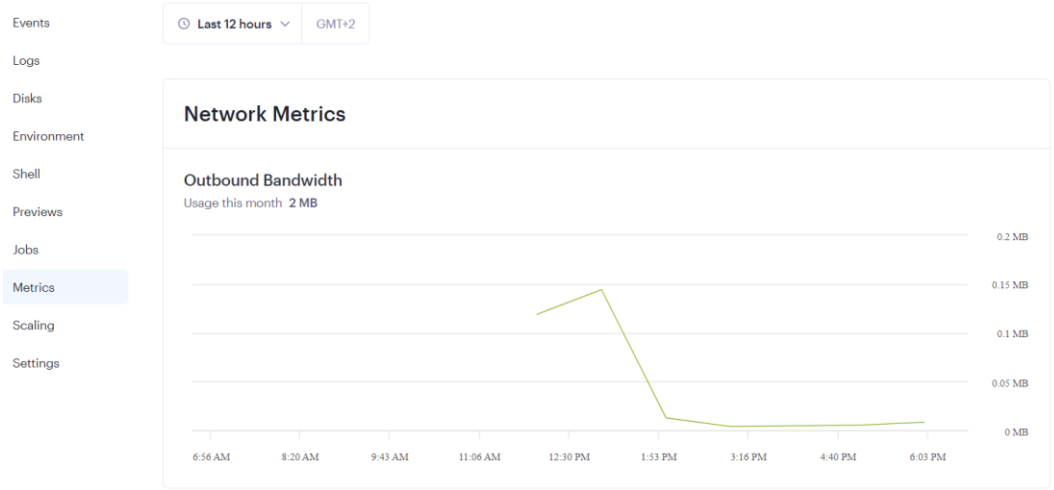
LOGS:



ENV:



METRICAS:



Se han tenido que configurar variables de entrono como el puerto sobre el que el aplicativo va a correr, el puerto sobre el que el servidor ntp va a estar ofreciendo servicio, la url de la base de datos y demás datos de acceso a esta, ya que los clientes necesitan usarlas pero no han de estar disponibles visiblemente para estos, por motivos de seguridad, y para que no accedan directamente a esta. Por otro lado se aprecia un gráfico con métrica sd e acceso a la red, al ser una versión gratuita, si se quieren ver más, habría que suscribirse a un plan de pago de este proveedor.

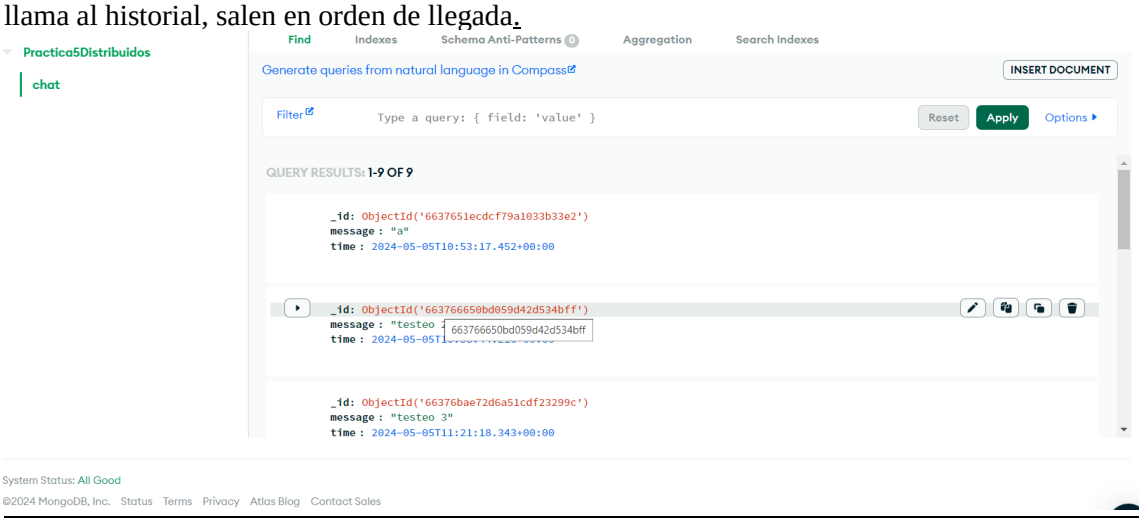
DESPLIEGUES ONRENDER

Se han creado reglas para que se realicen despliegues del aplicativo cada vez que se detecte un nuevo commit en github, por lo que la aplicación puede ser actualizada y verse reflejados los cambios en cuestión de minutos.

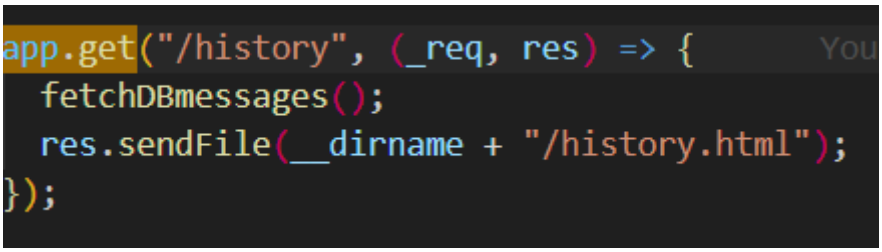
```
app.get("/history", (_req, res) => {
  fetchDBmessages();
  res.sendFile(__dirname + "/history.html");
});

0 references
io.on("connection", (socket) => {
  0 references
  socket.on("chat message", (msg) => {
    const message = { message: msg, time: new Date() };
    const dbName = "Practica5Distribuidos";
    const collectionName = "chat";
    const uri =
      "mongodb+srv://josojmf:yk6zucBZhK9CGsRT@practica5distribuidos.ryqbuhp.mongodb.net/?retryWrites=true&w=majority&appName=Practica5Dis
    const mongoclient = new MongoClient(uri);
    0 references
    mongoclient.connect().then(() => {
      const database = mongoclient.db(dbName);
      const collection = database.collection(collectionName);
      0 references
      collection.insertOne(message).then(() => {
        io.emit("chat message", msg);
      });
```

Se ha diseñado una base de datos no relacional en mongodb para el almacenamiento de los mensajes y los timestamps de cada uno, los mensajes se añaden por orden de llegada al ser enviados, y se hacen fuera del websocket de manera asíncrona, esto para facilitar la comunicación y que no existan cuellos de botella en el envío y recepción de mensajes, los mensajes se añaden a posteriori en la base de datos
Aquí se puede observar como también tenemos métricas de acceso, lectura y escritura en la base de datos, los mensajes se almacenan en orden de llegada, por lo que al extraerlos cuando se llama al historial, salen en orden de llegada.



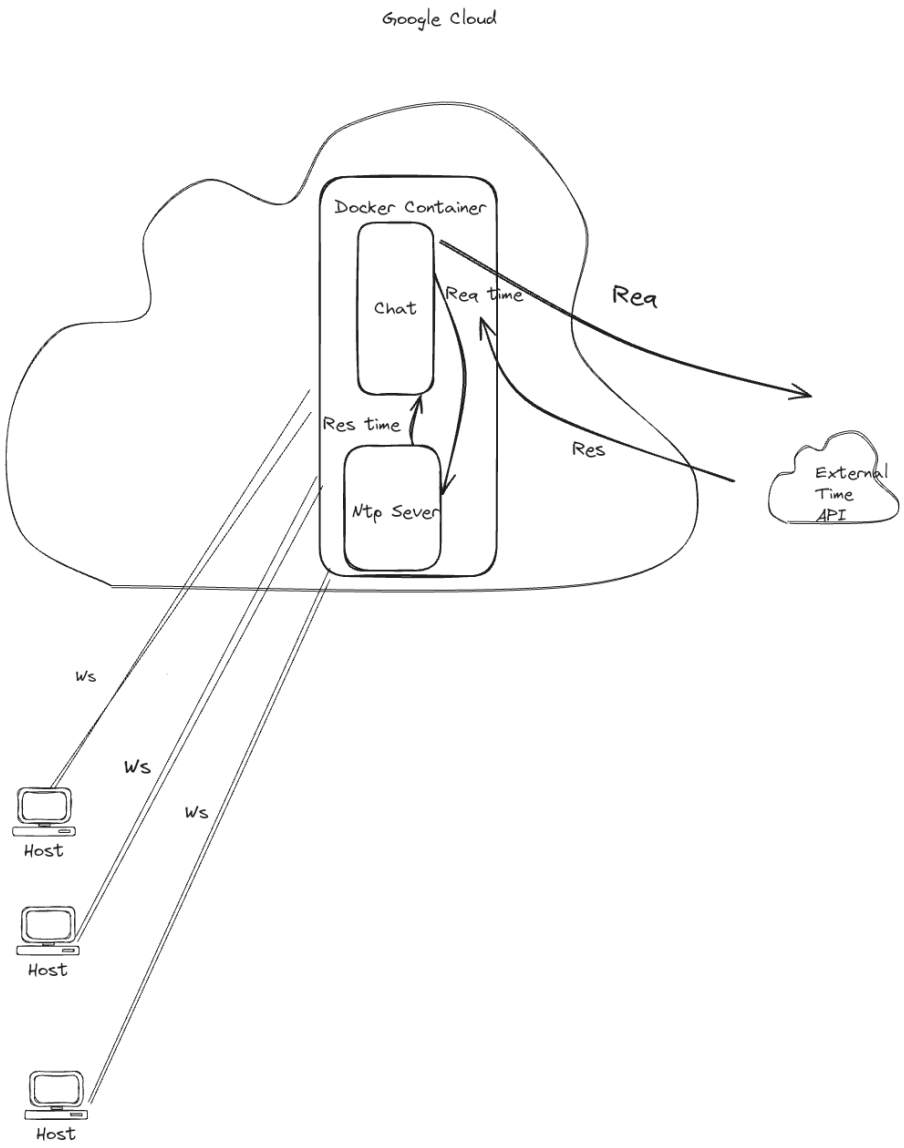
Se ha creado una ruta distinta a la del chat para mantener la navegabilidad y poder hacer la llamada a la base de datos automáticamente al acceder a esta ruta, este sistema de ruteo se hace usando un servidor oak.



```
app.get("/history", (_req, res) => {
  fetchDBmessages();
  res.sendFile(__dirname + "/history.html");
});
```

DIAGRAMA DE ARQUITECTURA

Se proporciona una imagen de la arquitectura ideada



CONCLUSIONES Y POSIBLES MEJORAS

Se ha diseñado un sistema de comunicaciones a través de canales websockets usando distintos microservicios para aportar una capa de complejidad extra, como lo pueden ser el servidor NTP local. Se ha tratado de usar nuevas tecnologías no explicadas en clase como lo son los contenedores en Docker y dockerhub, nodejs y sus dependencias, y servicios de hosting online. Se ha automatizado el despliegue de aplicaciones por medio de la detección de nuevas implementaciones a través del servicio de control de versiones de git y Github, se han visto varios servicios de despliegue de aplicaciones en la nube, como lo son Onrender, google cloud o Hostinger, se ha implementado un servicio de monitorización externo, a parte de los ya incluidos en los previamente mencionados.

POSIBLES MEJORAS:

- Implementar canales privados a los que subscribirse con una clave para poder crear múltiples chats.
- Cachear mensajes en una base de datos REDIS, ya que para la persistencia se haría más eficiente.
- Tener un control de quien envía cada mensaje y poder asignar un formato distinto en función de quién sea el remitente.
- Controlar mejor el historial, ya que es poco consistente.