METODOLOGÍA DE SELECCIÓN DE PROTOCOLOS WEB BASADO EN PARÁMETROS DE INTERÉS

Autor:
Bryan Steven Perez Salas

Director:
M.Sc. Dario Alejandro Segura Torres

Universidad Santo Tomás
División de Ingeniería
Facultad de Ingeniería Electrónica
Bogotá D.C.

2024

Tesis presentada en cumplimiento de los requisitos
para el grado de ingeniero electrónico

Periodo de Escritura

05/ 02/ 2024 – 05/ 11/ 2024

Director

M.Sc. Dario Alejandro Segura Torres

NOTA DE ACEPTACIÓN

El trabajo de grado titulado ***METODOLOGÍA DE SELECCIÓN DE PROTOCOLOS WEB BASADO EN PARÁMETROS DE INTERÉS****, realizado por el estudiante* ***Bryan Steven Perez Salas****, cumple con los requisitos exigidos por la* ***UNIVERSIDAD SANTO TOMÁS*** *para optar al título de* ***INGENIERO ELECTRÓNICO****.*

Dario Alejandro Segura _____

Carlos Javier Mojica _____

David Alejandro Martinez _____

DEDICATORIA

A mi abuela: Te presto mis ojos para que veas lo hermoso que es el mundo.

AGRADECIMIENTOS

Agradezco a mi director de proyecto de grado, el Ingeniero Dario Alejandro Segura Torres, por su tiempo y consejo; a mis amigos por hacer de este viaje uno increíble; a mi familia escogida por ser la seguridad que me permitió avanzar y a mi mismo, por soñar con un mundo mejor.

Índice general

Índice de figuras

Índice de tablas

1 Resumen

El proyecto de grado titulado *Metodología de selección de protocolos web basado en parámetros de interés* plantea de forma detallada una estructura de investigación, selección de parámetros de entrada, variables medibles, un modelo para la ejecución de pruebas y un formato para la presentación de resultados de tal forma que se evidencie una hoja de ruta para llegar a una toma de decisiones para la correcta selección del protocolo de comunicación que mejor se adapte a las necesidades de una solución particular.

El desarrollo del proyecto se divide en tres etapas: la primera de ellas consiste en la investigación de trabajos similares sobre metodologías comparativas entre protocolos desglosando los parámetros en entrada, las variables medidas, protocolos evaluados, metodología de pruebas e infraestructura implicada para cada uno de los antecedentes; generando un compendio de información que será utilizada posteriormente como insumo para la elaboración de la planteamiento metodológico.

La segunda etapa consiste en el planteamiento metodológico, este argumenta la selección y exclusión de mediciones, parámetros de entrada escogidos y los protocolos a evaluar; posteriormente describe el entorno de pruebas a utilizar en términos de hardware e implementaciones tanto del cliente que tiene por responsabilidad el de gestionar las pruebas de carga, como de cada uno de los servidores REST, Websocket y gRPC.

La tercera etapa es la presentación de resultados donde se agrupan los productos de la ejecución de las pruebas por variable medida (latencia, uso de CPU y uso de memoria) al ser sometidos a un barrido incremental en el tamaño de la carga útil entre 5kB-500kB y un incremento en el número de peticiones en simultáneo entre 1-50 clientes concurrentes. Los resultados anteriormente descritos se presentan por medio de gráficos de área que permiten dimensionar mejor los rangos de trabajo de cada uno de los protocolos al compararse entre sí.

2 Introducción

La arquitectura de microservicios ha captado la atención tanto de la academia como de la industria, ya que permite desarrollar sistemas más eficientes al dividir las aplicaciones en servicios pequeños y autónomos, además de ofrecer ventajas como escalabilidad, aislamiento de fallas y mantenibilidad [1]. Estos beneficios han impulsado a la migración de las arquitecturas monolíticas a arquitecturas basadas en microservicios a compañías como Uber, Spotify y Netflix [2]. Pero la segmentación de responsabilidades resultantes en programas de menor envergadura (microservicios) significa también un aumento en la relevancia los protocolos de comunicación utilizados para el traslado de información, acrecentando de igual forma los problemas inherentes a la comunicación por red como lo son la latencia, la pérdida de paquetes y el rendimiento.

Para abordar la demanda cada vez mayor en el volumen de información a transmitir y sumado a la evolución en las capacidades tecnológicas, han proliferado a lo largo de la historia una amplia variedad de protocolos para la transferencia de información entre servicios que abordan de manera particular la problemática de la comunicación; generando una nueva oportunidad de investigación: ¿Qué metodología se podría aplicar para la correcta selección de un protocolo de comunicación capaz de utilizar de mejor manera los recursos disponibles en términos de latencia, uso de CPU y uso de memoria, bajo una arquitectura determinada?

Aunque la literatura que aborda la problemática de la selección de protocolos de comunicación web es amplia, abordando el consumo de energía producto del uso de los protocolos [3], la afectación en latencia por la ejecución en entorno local y sobre Docker para diferentes protocolos [4], la relación entre carga útil y latencia en los protocolos [2], la afectación en el número de solicitudes por segundo relacionada al lenguaje de programación que implementa esto patrones de arquitectura [5], entre otros; aún presenta diferentes métodos de evaluación sin haber llegado a consensos sobre los factores de interés a medir ni de los parámetros de entrada relevantes para la prueba; evidenciando un área de investigación aún por abordar [6] para un proyecto que analice metodologías existentes para la extracción procesos de interés que puedan ser aplicados a un amplio espectro de protocolos de manera efectiva sin limitarse a un protocolo en particular,

permitiendo la obtención de conocimiento sólido por medio de la recolección y análisis de datos.

Con la finalidad de abordar esta problemática este trabajo plantea de forma detallada una estructura de investigación, selección de parámetros de entrada, variables medibles, un modelo para la ejecución de pruebas y un formato para la presentación de resultados de tal forma que se evidencie una hoja de ruta para llegar a una toma de decisiones para la correcta selección del protocolo de comunicación que mejor se adapte a las necesidades de una solución particular.

3 Planteamiento del Problema

En los últimos años se ha iniciado un proceso de migración de las arquitecturas monolíticas a arquitecturas basadas en microservicios debido a las ventajas que ofrecen estas últimas, como lo son: la escalabilidad, aislamiento de fallas y mantenibilidad[1]. Adquiriendo especial relevancia los protocolos de comunicación utilizados para el traslado de información entre los microservicios. Debido a la masificación en la necesidad de comunicación al interior de la aplicación, de igual forma, se acrecientan los problemas inherentes a la comunicación por red como lo son la latencia, la pérdida de paquetes y el rendimiento.

Debido a la evolución en las capacidades tecnológicas y a la demanda cada vez mayor en el volumen de información a transmitir, han proliferado a lo largo de la historia una amplia variedad de protocolos para la transferencia de información entre servicios que abordan de manera particular la problemática de la comunicación haciendo uso de sockets, la codificación en formato binario o patrones de diseño flexible, entre otros [7].

Tomando como base lo descrito anteriormente, este proyecto de grado busca dar respuesta a la pregunta de investigación:

¿Qué metodología se podría aplicar para la correcta selección de un protocolo de comunicación capaz de utilizar de mejor manera los recursos disponibles en términos de latencia, uso de CPU y uso de memoria, bajo una arquitectura determinada?

4 Justificación

4.1. Justificación práctica

Con el objetivo de abordar la problemática en la escogencia del mejor protocolo de comunicación entre servicios web que mejor se adapte a las particularidades que exige cada solución, teniendo en cuenta el amplio espectro de protocolos existentes en la actualidad [7], se evidencia necesaria una solución de carácter técnico, resultante en información cuantitativa sobre el rendimiento en términos de tiempos de latencia, el uso de CPU y el uso de RAM; de cada protocolo sometido a este proceso que permita la selección adecuada para un desarrollo en particular.

4.2. Justificación del planteamiento de una nueva metodología

La literatura que aborda la problemática de la selección de protocolos de comunicación web presenta diferentes métodos de evaluación sin haber llegado a consensos sobre los factores de interés a medir ni de los parámetros de entrada relevantes para la prueba; evidenciando una oportunidad de investigación [6] para un proyecto que analice metodologías existentes para la extracción procesos de interés que puedan ser aplicados a un amplio espectro de protocolos de manera efectiva sin limitarse a un protocolo en particular, permitiendo la obtención de conocimiento sólido por medio de la recolección y análisis de datos.

4.3. Justificación económica

En la actualidad las aplicaciones web no solo buscan la efectividad en los procesos que realizan, sino que ha adquirido gran relevancia la experiencia de usuario en los mismos. Haciendo de los tiempos de respuesta en los servicios un factor que afecta de manera directa la usabilidad del sitio web [8], de tal manera que la escogencia del protocolo web que haga mejor uso de los

recursos disponibles influye directamente en la experiencia de usuario sin requerir mayor inversión económica en infraestructura.

5 Impacto social

En Colombia las industrias relacionadas con las tecnologías de la información (TI) se posicionan como uno de los sectores que ha venido adquiriendo más fuerza en los últimos años, específicamente la exportación de servicios relacionados con software ha tenido un alza del 4.7 % al alcanzar en el 2021 la cifra de los US$299 millones. Además, se calcula que entre 2003 y marzo de 2022 se crearon 24.200 empleos relacionados con esta área [9]. La metodología de selección de protocolos de comunicación web propuesta beneficiaría a estos nuevos empleos y empresas, facilitando las tareas de diseño de infraestructura en roles como la arquitectura de software y el desarrollo web.

Por otro lado, la metodología de selección de protocolos sirve de insumo para futuras investigaciones, que tengan relación con problemas que se puedan solucionar por medio del uso de servicios web, ya que la metodología estará pensada de tal manera que pueda ser utilizada para el desarrollo de diferentes proyectos relacionados con esta área, siendo un tema de investigación novedoso para la Facultad de Ingeniería Electrónica y de considerable auge en la industria.

6 Objetivos

6.1. Objetivo General

Proponer una metodología de selección de protocolos de comunicación web para la elección informada del protocolo que mejor se adapte a las necesidades de un proyecto, a partir de parámetros clave como la latencia, el uso de CPU y la memoria, aplicado en Websocket, REST y gRPC.

6.2. Objetivos Específicos

1. Analizar metodologías existentes para extraer procesos de interés que puedan ser aplicados en una metodología para la selección de protocolos.
2. Desarrollar un ambiente de pruebas capaz de hacer uso de los protocolos de comunicación web Websocket, REST y gRPC; para realizar las pruebas necesarias que permitan la correcta selección del protocolo.
3. Realizar pruebas de carga sobre el ambiente de pruebas a los protocolos Websocket, REST y gRPC para recolectar la información de latencia, el uso de CPU y la memoria.
4. Analizar los resultados de las pruebas aplicando la metodología propuesta para corroborar su efectividad.

7 Antecedentes

7.1. Autonomía resultante del uso de diferentes protocolos

En [3] se evalúa el consumo de energía producto del uso de los protocolos REST, SOAP, Socket y gRPC, al procesar algoritmos de diferentes tamaños, complejidades y variando el tipo de dato de entrada; alternando entre ejecuciones locales y la descarga de cálculos. Finalizando con la condensación de la información en una tabla que expresa el protocolo utilizado, el método de ejecución y el consumo de energía.

El artículo justifica la relevancia en la escogencia del protocolo de comunicación sobre el marco de aplicación del Internet de las cosas (IoT), ampliando el rango sobre el cuál una metodología de selección del protocolo es relevante y significativa. De igual forma, expone que la variación en los tipos de datos de entrada afecta directamente a las capacidades del equipo que hace uso del protocolo.

El objetivo de los autores es presentar cómo los cuatro protocolos evaluados afectan a la autonomía de un dispositivo móvil, sin tener en cuenta factores como la variación en la carga útil del mensaje o los tiempos de respuesta cambiantes entre los diferentes protocolos, enfatizando principalmente en la ejecución de diferentes tipos de algoritmos de clasificación.

7.2. Comparativa protocolos en AWS EC2

Weerasinghe y Perera presentan en [10] una metodología con la finalidad de contrastar algunos de los protocolos de comunicación más populares (gRPC, HTTP y Web Socket). Sometiéndolos a pruebas de carga para determinar los tiempos de respuesta de cada uno, bajo un ambiente de pruebas implementado sobre servicios EC2 de AWS.

El artículo manifiesta la importancia de la obtención de información sobre la latencia como un

parámetro clave para la selección del método de comunicación y la diferencia significativa sobre los tiempos de respuesta para la transferencia de valores mínimos en el mensaje.

El trabajo en cuestión está soportado por la infraestructura de AWS, dotando de un carácter muy específico a los resultados ya que dependen de la infraestructura de esta empresa en términos de velocidad de red entre servicios, afectada por la ubicación geográfica de los servidores que realizan los roles de servidor-cliente, evidenciando una oportunidad de exploración en el testeo de los protocolos sobre valores diferentes a los mínimos del mensaje, como los son la variación en el tamaño de la carga útil o el número de peticiones soportadas.

7.3. Evaluación de protocolos realizando operaciones sobre tablas en docker

En [4] se ejecuta una serie de pruebas automatizadas en las que se busca evaluar los protocolos REST, WebSocket, gRPC y GraphQL; al realizar operaciones de creación y lectura de elementos sobre una tabla en un ambiente en ejecución local y virtualizado utilizando Docker, con el fin de obtener los tiempos de respuesta para un número de peticiones en aumento.

El paper ejemplifica la importancia de las pruebas de carga como método de obtención de resultados significativos a la hora de evaluar los diferentes métodos de comunicación, reflejando el comportamiento esperado ante el aumento de carga durante los diferentes picos y valles en el tráfico de un aplicativo web.

El artículo evalúa los protocolos sobre una misma máquina que aloja los clientes, los servidores y la base de datos. Esta arquitectura permite la competencia entre servicios por los recursos de la máquina, generando resultados que se ven afectados por más de una variable a la hora de evaluar la latencia sobre los diferentes métodos de comunicación.

7.4. Comparativa protocolos con carga útil variante

Olivos y Johansson en [2] realizan pruebas de rendimiento en términos de la latencia modificando el tamaño del mensaje en la carga útil para los protocolos REST y gRPC. Estas pruebas se realizan sobre un ambiente local para cuatro tamaños de mensaje diferente: Pequeño no mayor a 11 caracteres, Típico de 32 caracteres, Enorme con 10300 caracteres con un cálculo a partir de

la información enviada y Enorme de 10300 caracteres sin realizar ningún cálculo.

El trabajo presenta un tipo de prueba en donde la variación en la carga útil del mensaje bajo un flujo constante de peticiones, arroja no sólo una latencia distinta dependiendo del tamaño del mensaje sobre el mismo protocolo, sino que se evidencia fuertemente las diferencias existentes entre los protocolos en términos de rendimiento al momento de la transferencia de información.

El artículo se centra en los resultados al modificar el número de caracteres en la carga útil del mensaje sin contemplar la variación en la respuesta dependiendo del aumento en las peticiones sobre el servicio, al igual que deja abierta la posibilidad de investigación sobre otro tipo de pruebas como lo son el uso de la memoria y del CPU.

7.5. Eficiencia de protocolos dependiendo del lenguaje de programación

En [5] se comparan los protocolos REST y gRPC bajo pruebas de estrés variando la carga útil del mensaje en cuatro niveles: XS sin carga útil, S aproximadamente 1KB, M con un tamaño de 50KB y L con 500KB de información. Todo lo anterior se testea sobre los lenguajes de programación Java, Python y Rust; utilizando como servidor una Raspberry PI 400 y un computador de escritorio como cliente bajo la lógica de tener el grueso del procesamiento en el cliente y así, estar siempre en la disposición de enviar peticiones al servidor y que esto no afectará la evaluación en el rendimiento del protocolo y el lenguaje.

El trabajo expone un tipo de prueba que prioriza la disponibilidad y la estabilidad del servicio al ejecutar pruebas de estrés sobre los protocolos y manifiesta la importancia en la escogencia del lenguaje de programación que hace uso del protocolo de comunicación, ya que el método de ejecución propio de cada uno de los lenguajes, afecta las capacidades de procesamiento en el número de peticiones posibles de atender por parte del servidor.

Berg y Mebrahtu presentan resultados del número de peticiones soportadas por los tres lenguajes testeados pero se ven limitados en el envío de peticiones de gran tamaño por el ancho de banda disponible para la prueba con un máximo de un 1 Gb/s para el envío y recepción del mensaje; sugiriendo así un área de trabajo futura la comparativa entre comunicación servidor-cliente sobre una misma máquina, eliminando a su mínima expresión la intervención de factores externos como

el medio por el que transita el mensaje.

7.6. Comparativa Protocolo uso de CPU y memoria

Oona Laitamäki [11] con el fin de proponer una arquitectura capaz de transferir información en tiempo real, evalúa los protocolos HTTP polling, WebSocket y envío de eventos por parte del servidor en términos de cantidad de datos transmitidos por segundo, uso de CPU y uso de memoria. Desplegados desde los navegadores Microsoft Edge y Google Chrome.

El trabajo presenta una evaluación más completa de los factores relacionados con el rendimiento de los protocolos de comunicación, como lo es el uso de memoria y CPU. Descarta la existencia de una diferencia significativa entre la ejecución del protocolo desde alguno de los buscadores Google Chrome y Microsoft Edge, haciendo de este un punto de poca relevancia a la hora de seleccionar un protocolo para una arquitectura determinada.

Los métodos de evaluación de los protocolos, específicamente la evaluación de la cantidad de datos enviados por segundo responde al espíritu de selección del protocolo idóneo para la transferencia en tiempo real pero no aplica para la evaluación de escogencia de la generalidad de los protocolos existentes.

7.7. Comparativa protocolos para comunicación en tiempo real

En [12] se evalúan los métodos de comunicación web de HTTP Short polling, HTTP Long polling, WebSocket y envío de eventos por parte del servidor, todos enfocados en la comunicación en tiempo real. Presentando una ecuación para el cálculo de la puntuación de velocidad de la aplicación (Spd) al dividir en tres etapas los diferentes momentos de la comunicación entre servicios web.

El trabajo presenta una estructura minuciosa para la evaluación de la latencia entre la comunicación servidor-cliente al dividir esta en tres etapas: Tiempo de respuesta del servidor, Tiempo de procesamiento del cliente y tiempo de ida y vuelta del mensaje. Por último, menciona al protocolo Websocket como el protocolo mejor librado entre los protocolos de comunicación en tiempo real evaluados.

El escrito basa su evaluación en mensajes sencillos donde solo se contempla los tiempos de respuesta para ese tipo de mensajes y con una cantidad baja en el número de las peticiones. Se omiten otros criterios de evaluación relevantes presentes en trabajos similares como los son el uso de CPU, uso de memoria y la variación sobre la carga útil.

7.8. Evaluación protocolos REST y gRPC en latencia y flujo

En [13] se comparan los protocolos REST y gRPC, sobre un ambiente de pruebas basado en C# donde su objetivo último es la persistencia en un base de datos soportada por SQL Server. Se evalúan sobre análisis de flujo y latencia a partir de percentiles de los valores encontrados de flujo máximo.

El artículo aborda la comparación de los protocolos desde el análisis de flujo, en donde se escoge una cantidad importante de peticiones para ser atendidas, se gestionan de tal manera que se evalúa el tiempo que le toma al servidor responder a todas las peticiones de manera exitosa. No se busca evaluar la latencia de cada una de las peticiones ni estresar al sistema para conocer su límite.

El trabajo se centra en la variación del tiempo total que le toma al sistema procesar un determinado número de peticiones, modificando la cantidad de peticiones. Estas peticiones mantienen un valor constante en el tamaño de la carga útil de los mensajes, sin explorar la respuesta de los protocolos a un aumento paulatino en el número de peticiones y el cambio en el rendimiento en los tiempos de respuesta al someter el sistema a pruebas de carga.

8 Marco Teórico

8.1. HTTP/1.1

Hypertext Transfer Protocol (HTTP) es un protocolo de la capa de aplicación para la transmisión de datos. Fue diseñado para la comunicación entre los navegadores y servidores web. Siguiendo el modelo cliente-servidor, en el que un cliente establece una conexión con el servidor, realiza una petición y espera hasta que recibe una respuesta [14].

La comunicación entre el cliente y el servidor para el intercambio de información se realiza por medio de mensajes HTTP, los cuales se dividen en 2 tipos: mensajes de solicitud y mensajes de respuesta (ver Figura 1). Los mensajes de solicitud son una petición de acción por parte de un servidor web. Los mensajes de respuesta llevan los resultados de una solicitud a un cliente. Tanto los mensajes de solicitud como los de respuesta tienen la misma estructura de mensaje básica:

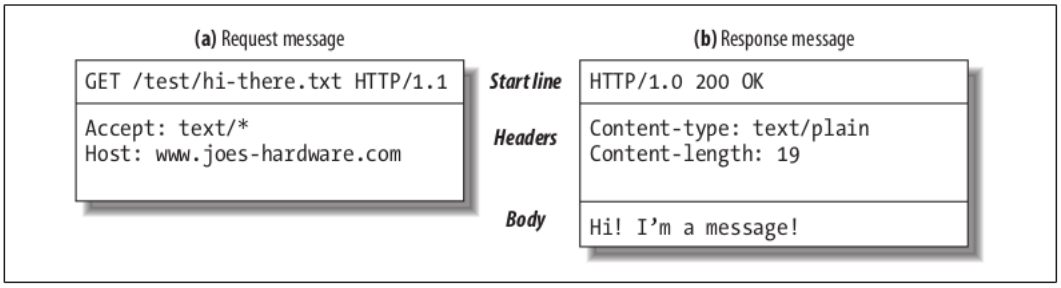

Figura 1: Estructura de solicitud y respuesta extraída de [15].

A continuación se desglosa la sintaxis de los mensajes HTTP:

8.1.1. Start line

- **Request line:**

<method><request-URL><version>

La línea de petición, contiene un método que describe la operación a realizar por el servidor, una URL de solicitud que describe el recurso en el que realizar el método y una versión HTTP que le indica al servidor qué dialecto de HTTP está hablando el cliente. Todos estos campos están separados por espacios en blanco.

- **Response line:**

<version><status><reason-phrase>

La línea de respuesta, contiene la versión HTTP que utiliza el mensaje de respuesta, un código de estado numérico y un frase de motivo que describe el estado de la operación. Todos estos campos están separados por espacios en blanco.

Methods

HTTP admite varios comandos de solicitud diferentes, denominados métodos HTTP. Cada mensaje de solicitud HTTP tiene un método. Este método le informa al servidor la acción a realizar; la Tabla 1 presenta los cinco métodos más utilizados:

Método HTTP	Descripción
GET	Solicita un recurso en específico (no deben incluir datos).
PUT	Crea o reemplaza un elemento con los datos de la petición.
DELETE	Elimina un recurso.
POST	Envía datos al servidor y resulta en un cambio.
HEAD	Solicita los headers que serían devueltos para un GET pero sin el body.

Tabla 1: Métodos HTTP más utilizados, basado en [15].

Version

El campo de la versión HTTP proporciona un medio para que las aplicaciones se informen entre sí en qué versión del protocolo se están comunicando. Los números de versión están destinados a proporcionar a las aplicaciones que hablan HTTP una pista sobre las capacidades de cada uno y el formato del mensaje. El formato para la comunicación de la versión HTTP es el siguiente:

HTTP/<major>.<minor>

Status Codes

El status code proporciona información rápida al cliente sobre el resultado de su petición. En la primera versión del protocolo HTTP/0.9 no existía el concepto de los códigos de respuesta, por lo que era necesario buscar en el cuerpo del mensaje para evidenciar si la petición fue errónea o exitosa. Con el surgimiento de HTTP/1.0 nacen los códigos de respuesta y con el paso de los años se han estandarizado una amplia variedad de códigos, los más comunes se mencionan a continuación en la Tabla 2:

Categoría	Valor	Descripción	Detalle
2xx (exitoso)	200	OK	Respuesta estándar para una solicitud exitosa.
	201	Created	Código de respuesta para una solicitud POST.
	202	Accepted	La solicitud se está procesando.
4xx (error del cliente)	400	Bad request	La solicitud no pudo ser interpretada.
	401	Unauthorized	Carece de credenciales válidas de autenticación.
	404	Not found	No se pudo encontrar el recurso.
5xx (error del servidor)	500	Internal server error	Solicitud incompleta por problema del servidor.
	502	Not implemented	El servidor no soporta la funcionalidad requerida.
	503	Service unavailable	El servidor no puede cumplir con la solicitud.

Tabla 2: Status codes más utilizados, basado en [15].

8.1.2. Headers

<key>:<value>

Los campos de encabezado HTTP agregan información adicional a los mensajes de solicitud y respuesta. Cada campo consta de un nombre, dos puntos (:), un valor y termina con un CRLF. Un CRLF es la combinación de dos códigos de control: CR (retorno de carro) y LF (salto de línea).

Los encabezados terminan con una línea en blanco (CRLF), que marca el final de la lista de encabezados y el comienzo del body de la entidad. Algunas versiones de HTTP, como HTTP/1.1, requieren que ciertos encabezados estén presentes para que el mensaje de solicitud o respuesta sea válido [15]. Los encabezados HTTP se clasifican en:

- **General headers:** Puede aparecer tanto en mensajes de solicitud como de respuesta.
- **Request headers:** Proporcionar más información sobre la solicitud.

- **Response headers:** Proporcionar más información sobre la respuesta.
- **Entity headers:** Describir el tamaño del cuerpo y el contenido, o el recurso en sí.
- **Extension headers:** Nuevos encabezados que no están definidos en la especificación.

8.1.3. Body

La tercera parte de un mensaje HTTP es el body, el cual contiene la carga útil de mensajes HTTP. Son las cosas para las que HTTP fue diseñado para transportar, aunque son de carácter opcional, por lo que a veces un mensaje termina con un CRLF simple. Los mensajes HTTP pueden transportar muchos tipos de datos digitales: imágenes, vídeos, documentos HTML, aplicaciones de software, transacciones con tarjetas de crédito, correo electrónico, etc.

El cuerpo de la entidad solo contiene la carga sin procesar. Cualquier otra información descriptiva está contenida en los encabezados. Debido a que la carga útil son solo datos sin procesar, los headers son necesarios para describir el significado de esos datos. Por ejemplo, el encabezado Content-Type indica cómo interpretar los datos (imagen, texto, etc.) y el encabezado Content-Encoding indica si los datos se comprimieron o recodificaron de alguna otra manera [16].

8.2. HTTP/2

En mayo de 2015, se aprobó formalmente como RFC 7540 por la IETF (Internet Engineering Task Force) el protocolo HTTP/2 [17]. Basado principalmente en el protocolo SPDY creado por Mike Belshe y Robert Peon de Google; esta nueva versión busca solucionar algunos de los problemas fundamentales de HTTP/1.1, particularmente en la obtención de múltiples recursos. Con los años se han generado diferentes soluciones para estos problemas de rendimiento (múltiples conexiones, fragmentación, sprites, etc.), pero tienen sus propias desventajas. Bajo la primicia de solucionar estos problemas de rendimiento, HTTP/2 se basa en los siguientes conceptos:

- Protocolo binario.
- Multiplexado.
- Control de flujo.
- Compresión de encabezados.
- Server push.

8.2.1. Binario

El protocolo HTTP/2 es un protocolo binario basado en paquetes, a diferencia de su antecesor HTTP/1.1 que se basa completamente en texto. El uso de comunicación a través de texto es de fácil entendimiento para los seres humanos pero requiere de mayor procesamiento por parte de las computadoras, convirtiéndose en un limitante para el internet moderno [18].

HTTP/2 pasa a ser un protocolo binario completo, en el que los mensajes HTTP se dividen y se envían en tramas claramente definidas, a esta práctica se le conoce como codificación fragmentada. Aunque la representación binaria se utilice para el envió y recepción de mensajes, los mensajes mantienen una estructura semejante al utilizado en HTTP/1.1.

8.2.2. Multiplexado

HTTP/1 es un protocolo síncrono de solicitud y respuesta única, donde el cliente enviaba un mensaje HTTP/1 y el servidor recibía una respuesta HTTP/1, ver Figura 2. Por otra parte, HTTP/2 utiliza múltiples tramas binarias para enviar solicitudes y respuestas HTTP a través de una única conexión TCP, utilizando un flujo multiplexado (Figura 3).

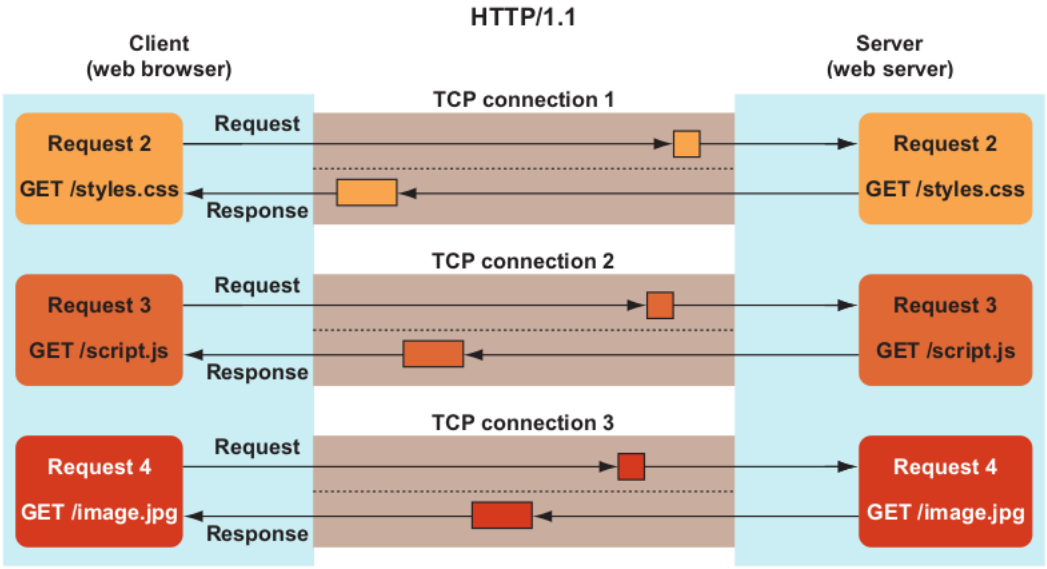

Figura 2: Múltiples peticiones HTTP/1 requieren de múltiples conexiones TCP, extraído de [18].

En las conexiones HTTP/2 a cada solicitud se le asigna un nuevo ID de flujo con valor incremental (5, 7 y 9 en la Figura 3) y las respuestas se envían de vuelta con el mismo ID de flujo; estos flujos se cierran cuando finaliza la respuesta. Con el fin de evitar un conflicto al entre IDs de flujo, a las solicitudes iniciadas por el cliente se les asignan números impares y a las solicitudes iniciadas por el servidor se les asignan identificadores de flujo pares.

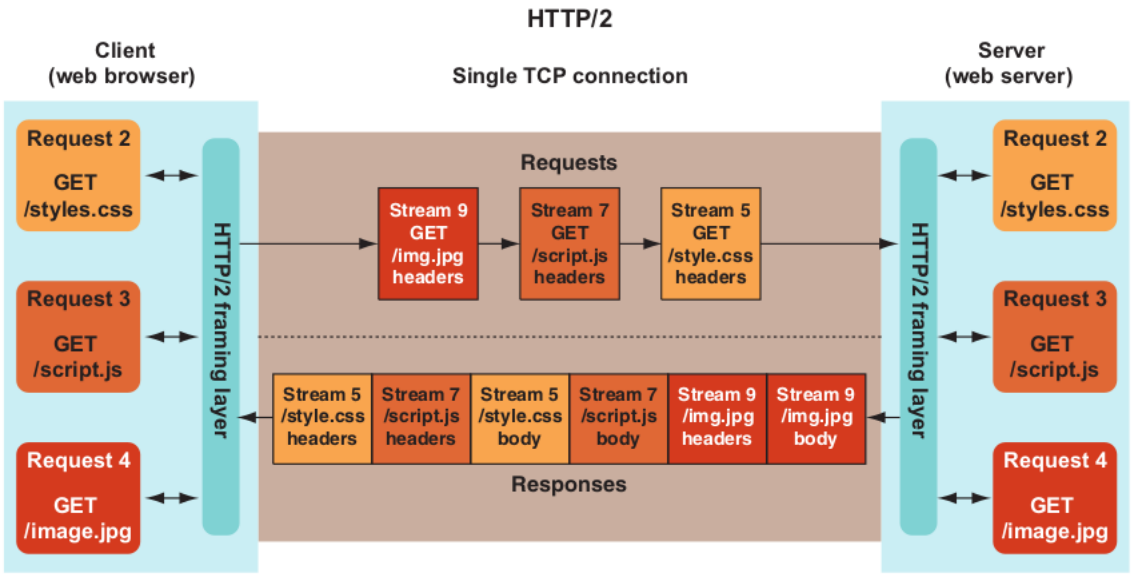

Figura 3: Tres peticiones a través de una conexión HTTP/2 multiplexada, extraído de [18].

8.2.3. Priorización de flujos y control de flujo

HTTP/1 es un protocolo de solicitud y respuesta única, por lo que no era necesario establecer prioridades dentro del protocolo. El cliente decidía la prioridad al decidir en qué orden generar las peticiones al servidor, solicitando primero recursos críticos como (HTML, CSS y JavaScript crítico) y posteriormente recursos como imágenes y JavaScript asíncrono. Al usar HTTP/2 con sus múltiples peticiones en paralelo (alrededor de 100 transmisiones activas), pueden competir por respuesta recursos con baja prioridad con recursos de prioridad alta por ser pedidos al mismo tiempo, por este motivo HTTP/2 posibilita en señalar la prioridad de los recurso generando que el servidor envíe más frames para solicitudes de mayor prioridad al momento de existir una cola de frames esperando ser enviada [18].

8.2.4. Compresión de encabezados

Los encabezados HTTP se utilizan para enviar información adicional sobre las solicitudes y respuestas entre el cliente y el servidor. A continuación, se desglosan alguno de los campos de las cabeceras que comúnmente se repiten entre las peticiones:

- **Cookie:** Las cookies se envían con cada solicitud hacia ese dominio y suelen ser particularmente grandes.
- **User-Agent:** Generalmente indica el navegador web que se está utilizando. Nunca cambia durante la sesión, pero aún así se envía con cada solicitud.
- **Host:** Se utiliza como campo de información de la URL a la que se realiza la solicitud, siendo siempre el mismo para cada solicitud al mismo host.
- **Accept:** Este campo define el formato de la respuesta que se espera de la petición. Los formatos que admite un navegador normalmente no cambian sin una actualización del navegador, por lo que este valor se repite constantemente.
- **Accept-Encoding:** Este campo define los formatos de compresión (gzip, deflate, br, entre otros). Este encabezado no cambia durante la sesión.

Los campos mencionados anteriormente tienden a duplicarse y en casos de encabezados de políticas de seguridad pueden llegar a ser más grandes que el contenido de la respuesta del servidor. Aunque HTTP/1 permite la compresión de la carga útil, no lo permite para los encabezados; a diferencia de HTTP/2 que incorpora el concepto de compresión de cabeceras en el protocolo [18].

8.2.5. Server push

HTTP/2 agrega el concepto de server push, el cual permite al servidor responder a una petición con más de una respuesta, a diferencia de HTTP/1 que realiza una petición y recibe una respuesta. Bajo este concepto también se crea la necesidad de una buena estructuración de las respuestas con el fin de no desperdiciar el ancho de banda con recurso que ya fueron enviados por solicitudes anteriores y están disponibles en memoria caché [18].

8.3. REST

REST no es un protocolo, ni un formato de archivo, ni un framework desarrollo. Es un conjunto de restricciones de diseño, denominadas restricciones de Fielding, debido a que fueron planteadas por primera vez en la tesis doctoral de Roy T. Fielding sobre arquitectura de software del año 2000, que las reunió bajo el nombre de *representational state transfer* (REST) [19]. REST adopta las características propias del protocolo HTTP y agrega una capa de estructura y estandarización para una masificación en su uso, en este trabajo se mencionara a REST como protocolo bajo la claridad de que se refiere a la implementación de REST sobre el protocolo de transferencia de hipertexto HTTP/1.1.

Las restricciones que describen REST están agrupadas en las siguientes categorías:

- Cliente-Servidor.
- Interfaz uniforme.
- Sistema en capas.
- Caché.
- Stateless.
- Código bajo demanda.

8.3.1. Cliente-Servidor

La separación de responsabilidades es el tema central de las restricciones cliente-servidor en la Web. Ambas partes pueden ser implementadas en lenguajes o tecnologías distintas y desplegadas de forma independiente, siempre que acuerden los métodos de comunicación entre las partes y se ajusten a la interfaz uniforme de la Web [19].

8.3.2. Interfaz uniforme

Las interacciones entre los componentes de la Web (clientes, servidores e intermediarios) dependen de la uniformidad de sus interfaces. Si alguno de los componentes se aparta de los estándares establecidos, el sistema de comunicación entre las partes deja de funcionar [20]. Para evitar fallas en la comunicación entre componentes Fielding identificó cuatro restricciones en la interfaz uniforme:

- **Identificación de recursos:** Cada concepto distinto basado en la Web se conoce como recurso y se puede acceder a él mediante un identificador único, como un URI.
- **Manipulación de recursos a través de representaciones:** El mismo recurso puede presentarse de distintas maneras y formatos a cada cliente sin cambiar de identificador.
- **Mensajes autodescriptivos:** El estado deseado de un recurso se puede representar dentro del mensaje de solicitud de un cliente. El estado actual de un recurso se puede representar dentro del mensaje de respuesta que llega desde el servidor.
- **Hipermedia como motor del estado de la aplicación (HATEOAS):** La representación del estado de un recurso incluye enlaces a recursos relacionados.

8.3.3. Sistema en capas

Los intermediarios basados en la red, como proxies y gateways, deben ser implementados de manera transparente entre un cliente y un servidor aplicando los conceptos de la interfaz uniforme. Un intermediario de red intercepta la comunicación entre las partes con el fin de aplicar ciertas medidas de seguridad, almacenamiento en caché de respuestas y realizar acciones como balanceador de carga [20].

8.3.4. Caché

Las restricciones de caché indican a un servidor web que debe declarar la capacidad de almacenamiento en caché de los datos de cada respuesta. El almacenamiento en caché de los datos de respuesta puede ayudar a reducir la latencia percibida por el cliente, aumentar la disponibilidad y la confiabilidad generales de una aplicación y controlar la carga de un servidor web [19].

8.3.5. Stateless

No se requiere que un servidor web memorice el estado de sus clientes. Como resultado, cada cliente debe incluir toda la información contextual necesaria en cada interacción con el servidor, esta asignación de responsabilidad le permite al servidor dar gestión a una cantidad mucho mayor de clientes [21].

8.3.6. Código bajo demanda

Es una restricción que permite a los servidores web transferir temporalmente programas ejecutables, como scripts o complementos, a los clientes. El código a demanda tiende a establecer un acoplamiento tecnológico entre los servidores web y sus clientes, ya que el cliente debe ser capaz de comprender y ejecutar el código que descarga del servidor. Por esta razón, el código a demanda es la única restricción REST que se considera opcional [20].

8.4. Websocket

En diciembre de 2011, se aprobó formalmente como RFC6455 por la Internet Engineering Task Force (IETF) el protocolo WebSocket. El protocolo busca dar solución al abuso en el número de llamados HTTP por parte del cliente por aplicaciones que necesitan de constante actualización del servidor como lo son juegos y programas de mensajería instantánea. Como solución Websocket ofrece un canal de comunicaciones bidireccional y full-duplex que opera sobre HTTP mediante un único socket [22].

La transferencia de información sobre el protocolo WebSocket se basa en los siguientes cuatro eventos:

- **Open:** Cuando el servidor WebSocket responde a la solicitud de conexión y se completa el protocolo de enlace, se activa el evento Open y se establece la conexión.
- **Message:** Una vez que haya establecido una conexión con el servidor WebSocket, el servidor está listo para enviar y recibir mensajes de su aplicación cliente.
- **Error:** Al momento de presentarse un error, se activa el controlador asociado al evento de error. Se puede suponer que la conexión WebSocket se cerrará disparando un evento de cierre. Debido a que el evento de cierre ocurre poco después del evento de error, los datos de entrada al evento suele contener información relevante sobre la naturaleza del fallo.
- **Close:** El evento close puede ser disparado por cualquiera de las partes a través del método close() o posterior a la ejecución del evento error, terminando así la conexión entre las partes.

Si bien WebSocket utiliza HTTP como mecanismo de transporte inicial, la comunicación no finaliza después de que el cliente recibe una respuesta (Figura 4), superando las limitaciones del

ciclo típico de solicitud/respuesta HTTP. Como consecuencia, mientras la conexión permanezca abierta, el cliente y el servidor pueden enviar mensajes libremente de forma asincrónica sin necesidad de realizar una nueva conexión [23].

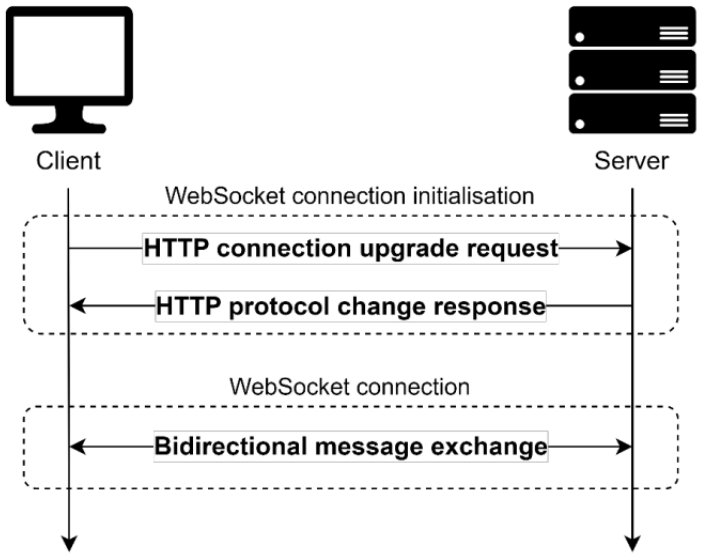

Figura 4: Esquema de comunicación por Websocket, extraído de [11].

8.5. gRPC

En 2001 Google crea un framework RPC de propósito general llamado Stubby para conectar miles de microservicios que se ejecutan en centros de datos distribuidos y desarrollados en diversas tecnologías. Aunque Stubby poseía un gran desempeño para el manejo de grandes cantidades de solicitudes, se encontraba demasiado acoplado a la infraestructura propia de la empresa. Es así que en 2015 se lanza gRPC como un framework RPC de código abierto, estandarizado, de propósito general y multiplataforma [24]. Entre sus capacidades se destacan las siguientes:

- **Eficiencia en la comunicación entre procesos:** En lugar de utilizar un formato de texto como JSON o XML, gRPC utiliza un *protocol buffer* basado en binario para la comunicación entre servicios y clientes de gRPC, ademas, gRPC implementa este *protocol buffer* sobre HTTP/2, lo que representa una mayor eficiencia para la comunicación entre procesos.

- **Esquema e interfase de servicio bien definidos:** gRPC promueve un enfoque de contrato primero al momento del desarrollo de aplicaciones. Siendo necesario iniciar por la definición de las interfaces de servicio y posteriormente se da paso a la implementación, ofreciendo un enfoque simple, consistente y confiable.

- **Fuertemente tipado:** Debido a la utilización del *protocol buffer* para la definición de los servicios gRPC, los contratos de servicio gRPC definen claramente el tipo de información que se utilizará para la comunicación entre las aplicaciones. Lo anterior se refleja en mayor estabilidad en el desarrollo de aplicaciones distribuidas, debido a que la tipificación estática ayuda a evitar la mayoría de los errores de interoperabilidad y tiempo de ejecución comunes al momento de desarrollar aplicaciones nativas de la nube que abarcan diferentes equipos y tecnologías.

- **Políglota:** gRPC cuenta con un amplio rango de lenguajes de programación en los que es posible hacer uso del protocolo, debido a que las definiciones del *protocol buffer* son independientes del lenguaje. Por lo tanto, puede interoperar con cualquier servicio o cliente gRPC existente sin importar el lenguaje de programación en el que se ejecutan las partes.

- **Transmisión dúplex:** gRPC cuenta con soporte nativo de streaming entre el cliente y el servidor, configurable desde la definición del servicio. Y la capacidad de alternar entre mensajería de tipo solicitud-respuesta convencional y streaming es una ventaja clave sobre el estilo de mensajería REST convencional.

- **Funciones básicas incorporadas:** gRPC ofrece soporte integrado para funciones básicas como autenticación, cifrado, resiliencia (deadlines y timeouts), intercambio de metadatos, compresión, balance de carga, entre otros.

- **Integración con ecosistemas nativos de la nube:** Al ser parte gRPC del Cloud Native Computing Foundation (CNCF), la mayoría de los frameworks y tecnologías modernas ofrecen soporte nativo para gRPC de manera inmediata.

En la Figura 5 se ilustra la comunicación entre dos microservicios utilizando gRPC. A partir de la definición del *protocol buffer* ProductInfo.proto, tanto el servidor como el cliente lo utilizan como estructura/acuerdo para la comunicación entre las partes sin importar que estén desarrollados en lenguajes diferentes (Java y Go). Aunque la comunicación entre el cliente y el servidor se realiza por una implementación del protocolo HTTP/2 que binariza la información a transmitir

[24], a nivel del programa la implementación se realiza por medio del llamado de los métodos previamente definidos en el protobuf.

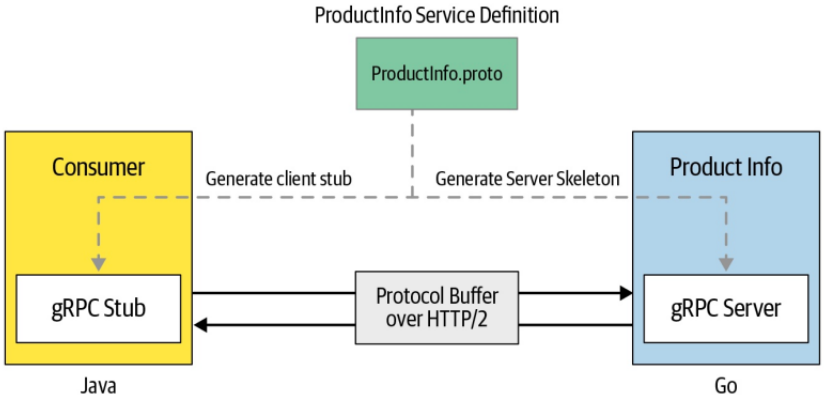

Figura 5: Esquema de comunicación por gRPC, extraído de [24].

9 Diseño Metodológico

A continuación, se describirán las etapas y los diferentes pasos a seguir para la realización del proyecto. El presente proyecto consta de 4 etapas que corresponden a: Investigación, planteamiento metodológico, elaboración de pruebas y construcción del documento. Cada una de estas etapas posee una sucesión de pasos a seguir, descritos a continuación.

9.1. Investigación

En la etapa de Investigación se realiza el estudio del sistema de reglas que permite la comunicación web para la transmisión de información entre dos o más entidades, teniendo en cuenta el estándar que define la sintaxis, semántica y sincronización de la comunicación de cada uno de los siguientes protocolos:

- Estructura de comunicación protocolo REST
- Estructura de comunicación protocolo Websocket
- Estructura de comunicación protocolo gRPC

9.2. Planteamiento de la metodología

Se describe de forma detallada la estructura propuesta en términos de investigación, diseño y evaluación de los protocolos incluyendo los métodos de medición y el procesamiento de los resultados de tal forma que se evidencie una hoja de ruta para llegar a una toma de decisiones para la correcta selección del protocolo de comunicación que mejor se adapte a las necesidades de una solución particular.

9.3. Elaboración de pruebas

9.3.1. Desarrollo del ambiente de pruebas

Se configura un entorno controlado capaz de suplir las necesidades en términos hardware, software y demás dependencias, con tal de garantizar la correcta ejecución de las pruebas y la fiabilidad de los datos obtenidos por la mismas.

9.3.2. Ejecución de las pruebas de carga

Se realiza la ejecución de pruebas de carga como un método que permite evaluar el rendimiento de un aplicativo web ante el crecimiento proporcional del número de peticiones sobre un servicio, variando a su vez, la carga útil de las peticiones.

9.4. Construcción del documento

Etapa final del proyecto, en la elaboración del documento se realizó el registro de todo el trabajo elaborado, para este fin se deben cumplir las siguientes actividades:

9.4.1. Compendio de información

El proceso de investigación sobre las particularidades en el envió y recepción de información para los protocolos REST, Websocket y gRPC, el planteamiento de la metodología propuesta y los resultados de las pruebas de carga realizadas se diligencian en el documento final del proyecto de grado.

9.4.2. Análisis de resultados

Finalmente, se incluye el análisis de resultados a través de lo obtenido en las pruebas.

10 Desarrollo Conceptual

10.1. Antecedentes Metodológicos

10.1.1. Autonomía resultante del uso de diferentes protocolos

En [3] se evalúa el consumo de energía producto de la ejecución de algoritmos de forma local o remota por medio de los patrones de arquitectura de la comunicación REST, SOAP, gRPC y Socket en un dispositivo móvil bajo los parámetros expuestos en la Figura 6:

Parámetros variables:

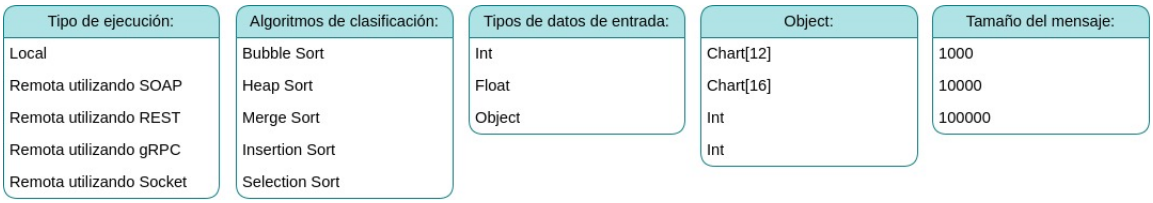

Figura 6: Parámetros variables basados en [3].

Metodología:

La metodología propuesta consta de la generación de ejecuciones sucesivas de la combinación de parámetros seleccionada hasta que la batería decaiga en 1 %. Entre más ejecuciones se puedan realizar antes de que baje ese 1 %, menor es el consumo de batería de esa configuración.

Con la intención de medir el porcentaje de descarga en todo momento, se opta por una herramienta proporcionada por terceros, el Android Battery Manager. Esta aplicación tiene como valor de cambio mínimo medible el 1 % en el porcentaje de batería. Con este porcentaje y sabiendo las especificaciones del equipo (Galaxy S5), se puede calcular que un 1 % de la batería equivale a 388,1 Joules; este valor dividido por el número de ejecuciones representa el consumo de energético

por petición.

Esta metodología pretende proponer información adicional a las comparativas entre el procesamiento de datos local o de forma externa, presentando el consumo de energía relacionado como un valor relevante y a los diferentes patrones de arquitectura de la comunicación como un factor influyente en el consumo de energía del dispositivo. La infraestructura implicada en la prueba se expone a continuación en la Figura 7.

Diagrama Arquitectura:

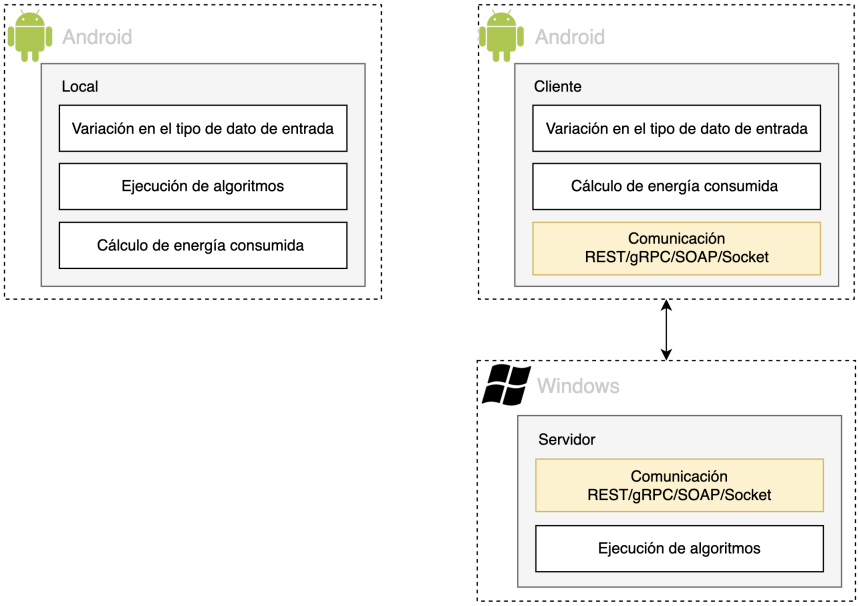

Figura 7: Diagrama de arquitectura basado en [3].

10.1.2. Comparativa protocolos en AWS EC2

En [10] analizan los tiempos de respuesta entre microservicios, alojando el programa en la nube de AWS (Amazon Web Services), en una instancia de EC2. La prueba contempla los siguientes parámetros expuestos en la Figura 8:

Parámetros variables:

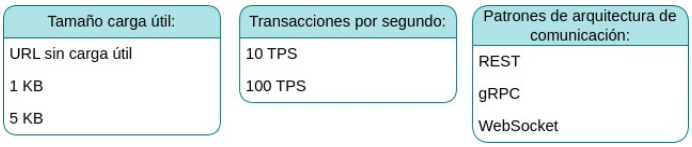

Figura 8: Parámetros variables basados en [10].

Metodología:

A partir de la configuración descrita en la Figura 9, la cual se ejecuta sobre el lenguaje de Java Spring boot, se proponen dos tipos de pruebas de carga:

- Variación en el tamaño de la carga útil mientras se limita el número de transacciones por segundo (TPS) para cada uno de los protocolos.
- Tamaño constante de la carga útil en 1KB sin limitaciones de TPS.

En ambos casos se mide el tiempo de comunicación entre los microservicios, siendo registrado por el microservicio A por medio del formato de registros Log4j, para su posterior recepción, procesamiento y visualización por medio de los programas Elasticsearch, Kibana, Beats y Logstash (ELK Stack).

Diagrama Arquitectura:

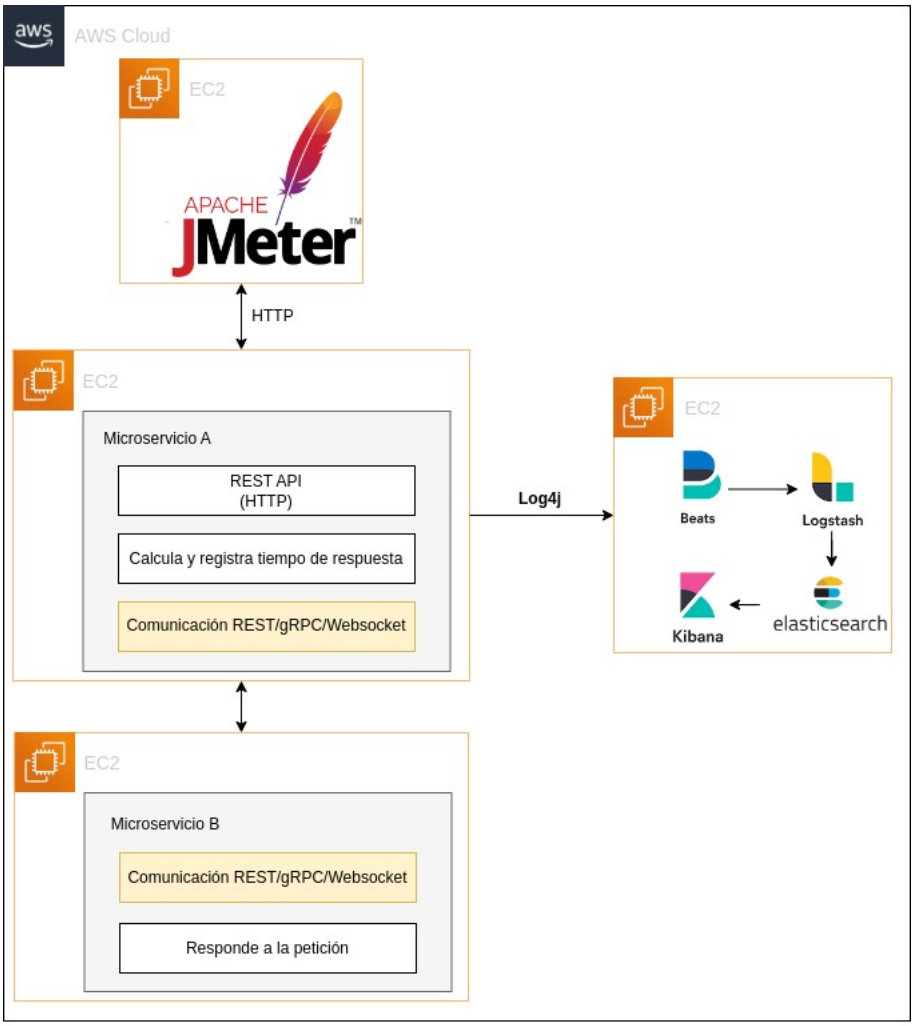

Figura 9: Diagrama de arquitectura basado en [10].

10.1.3. Evaluación de protocolos realizando operaciones sobre tablas en docker

En [4] se mide el tiempo que tardan en realizarse tres tipos de funciones básicas en una base de datos NoSQL. Los programas que realizan dicha prueba son ejecutados de forma local sobre Windows y Docker, con la finalidad de evidenciar algún tipo de afectación relacionada al entorno de ejecución. La prueba contempla los siguientes parámetros expuestos en la Figura 10:

Parámetros variables:

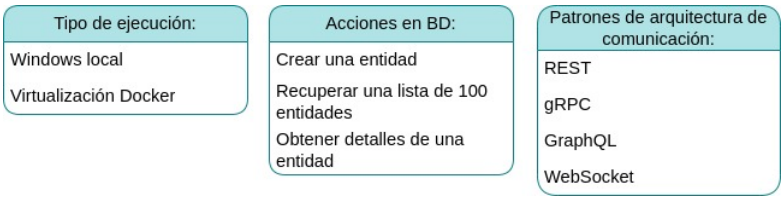

Figura 10: Parámetros variables basados en [4].

Metodología:

A partir de la configuración descrita en la Figura 11, se realizan cuatro tipos de pruebas diferentes:

- Comparación de rendimiento en la creación de una entidad en la base de datos.
- Comparación de rendimiento en la lectura de una lista de 100 entidades en la base de datos.
- Comparación de rendimiento en la lectura de los detalles de una entidad en la base de datos.
- Comparación de memoria: Número de bytes transferidos por cada tipo de arquitectura de comunicación durante para cada tipo de acción en la BD.

Para cada uno de los escenarios de pruebas se registran los tiempos mínimos, máximos y el promedio de la ejecución tanto local en Windows como en Docket. Además, con el fin de realizar la medición del número de bytes transferidos por cada tipo de arquitectura de programación, se utiliza el programa WireShark. Los programas del cliente y el servidor se ejecutan sobre el lenguaje de programación Python.

Diagrama Arquitectura:

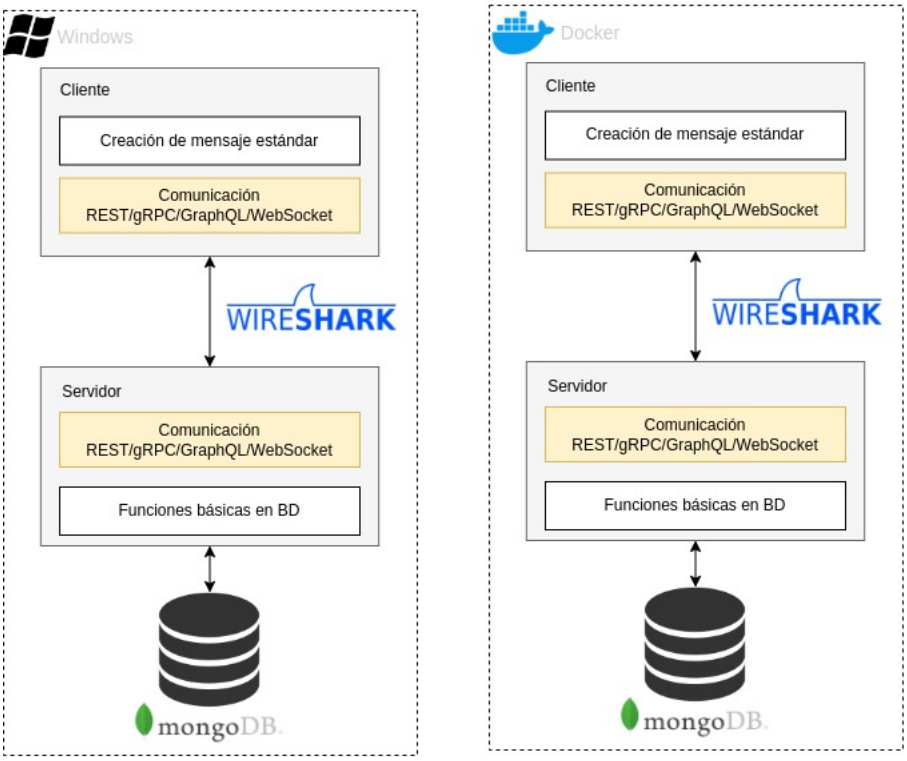

Figura 11: Diagrama de arquitectura basado en [4].

10.1.4. Comparativa protocolos con carga útil variante

En [2] se mide el tiempo promedio de respuesta para solicitudes con diferentes tamaños en su carga útil. Tanto el script de pruebas como los microservicios se ejecutan en un ambiente local. La prueba contempla los siguientes parámetros expuestos en la Figura 12:

Parámetros variables:

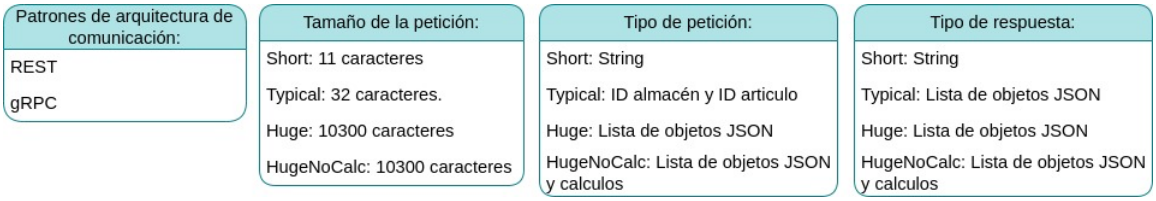

Figura 12: Parámetros variables basados en [2].

Metodología:

A partir de la configuración descrita en la Figura 13, se realizan cuatro tipos de pruebas diferentes:

- Short: Se envía una cadena de texto de 11 caracteres y recibe otra cadena de texto.
- Typical: Se envía un ID almacén y un ID artículo con un total de 32 caracteres. El servidor realiza búsquedas en la base de datos, además de cálculos, antes de responder una lista de objetos JSON.
- Huge: Se envía una lista de objetos JSON de 10300 caracteres. El servidor realiza cálculos similares a la prueba Typical pero realiza menos búsquedas en la base de datos.
- HugeNoCalc: Se envía una lista de objetos JSON de 10300 caracteres. No realiza cálculos ni búsquedas en la base de datos, respondiendo la misma lista de objetos que recibió.

Las pruebas se ejecutaron utilizando un script en Locust, realizando llamados consecutivos durante 500 segundos para cada uno de los escenarios de prueba, tanto con REST como con gRPC. El programa mide el tiempo entre la petición al servicio de Communication Testing Service y la respuesta del mismo. El servicio de Supply Chain Planning es estándar para ambos protocolos y al igual que el servicio de Communication Testing Service se crean utilizando el framework Spring Boot de Java.

Diagrama Arquitectura:

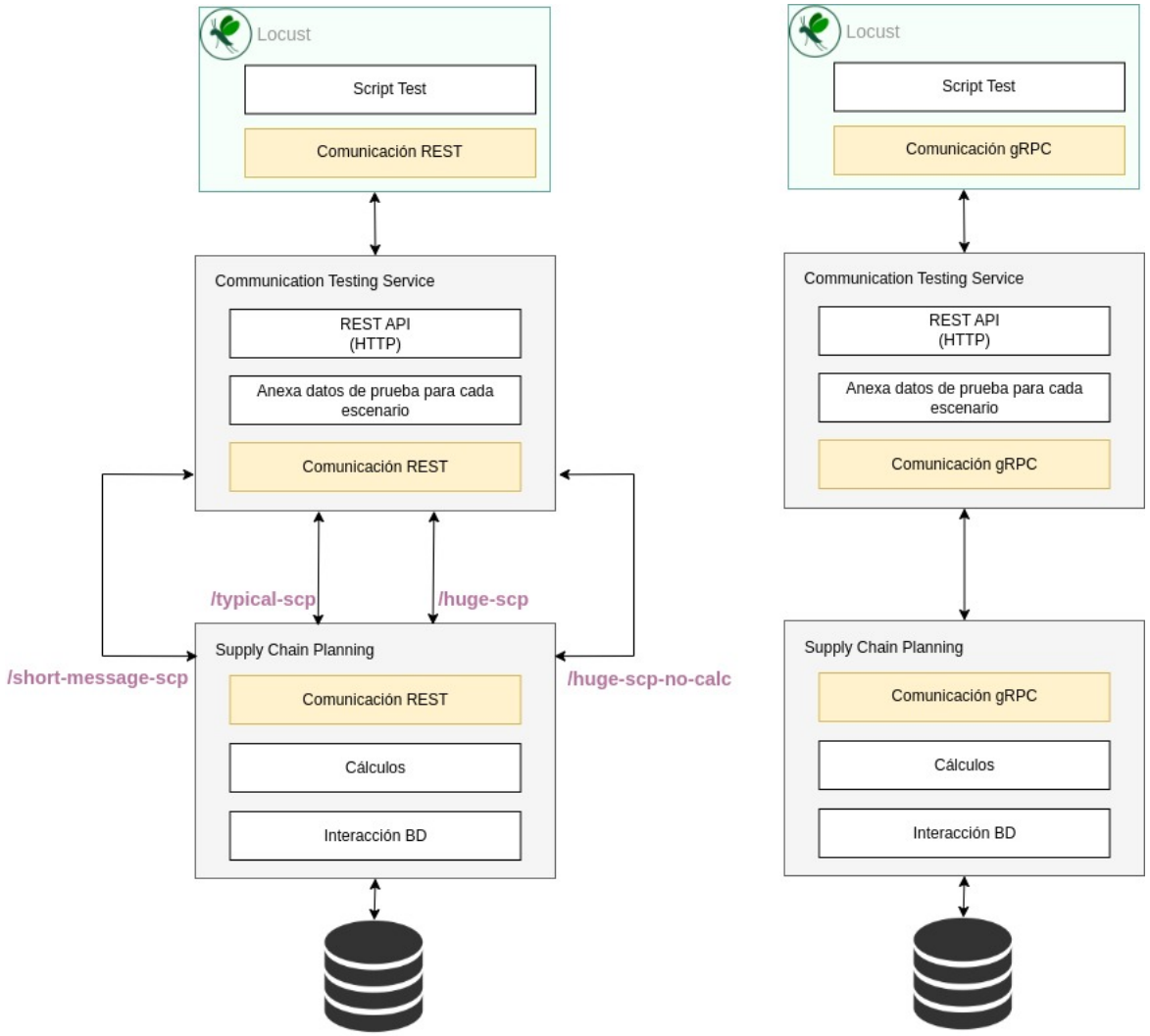

Figura 13: Diagrama de arquitectura basado en [2].

10.1.5. Eficiencia de protocolos dependiendo del lenguaje de programación

En [5] se mide la cantidad promedio de solicitudes por segundo para los patrones de arquitectura REST y gRPC. Los programas que realizan dicha prueba fueron realizados sobre tres lenguajes de programación diferentes (Java, Python y Rust), con la finalidad de evidenciar algún tipo de

afectación relacionada al lenguaje de programación que implementa estos patrones de arquitectura. La prueba contempla los siguientes parámetros expuestos en la Figura 14:

Parámetros variables:

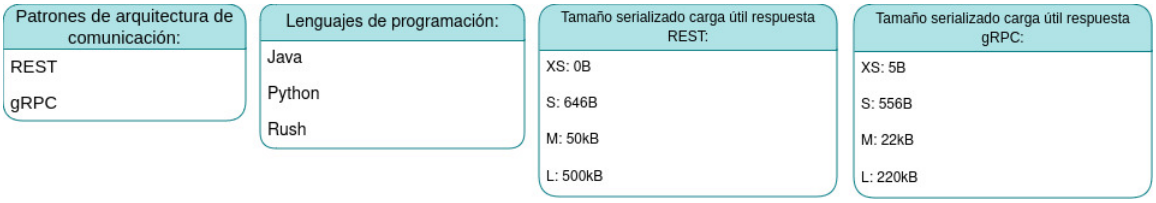

Figura 14: Parámetros variables basados en [5].

Metodología:

A partir de la configuración descrita en la Figura 15, se realizan cuatro tipos de pruebas diferentes para cada uno de los patrones de arquitectura:

- XS: Se envía una petición sin carga útil y se responde del mismo modo. Esto con el fin de obtener un valor base sobre el rendimiento de los servicios.
- S: Se envía una petición sin carga útil y se recibe un objeto que contiene números y caracteres.
- M: Se envía una petición sin carga útil y se recibe una lista de objetos.
- L: Se envía una petición sin carga útil y se recibe una lista de objetos con mayor información que la prueba L.

Anterior a la ejecución de las pruebas se plantea una "fase de descubrimiento"[5] donde el cliente va aumentando la cantidad de hilos que envían peticiones hasta alcanzar el punto de saturación donde sin importar el aumento en el número de subprocesos, el número de peticiones a las que responde el servidor se mantiene igual.

Las pruebas se ejecutan desde el cliente (Computador), el cual envía peticiones por un periodo de 100 segundos hacia el servidor (Raspberry Pi) mientras calcula el tiempo promedio de latencia de las peticiones. El test consta de tres servidores, cada uno con un lenguaje de programación

distinto con el que fue desarrollado el servidor: Java, Python y Rust; siendo testeados para cada tamaño de carga útil (XS, S, M, L), sobre cada patrones de arquitectura (REST, gRPC), resultando en un total de 24 pruebas.

Diagrama Arquitectura:

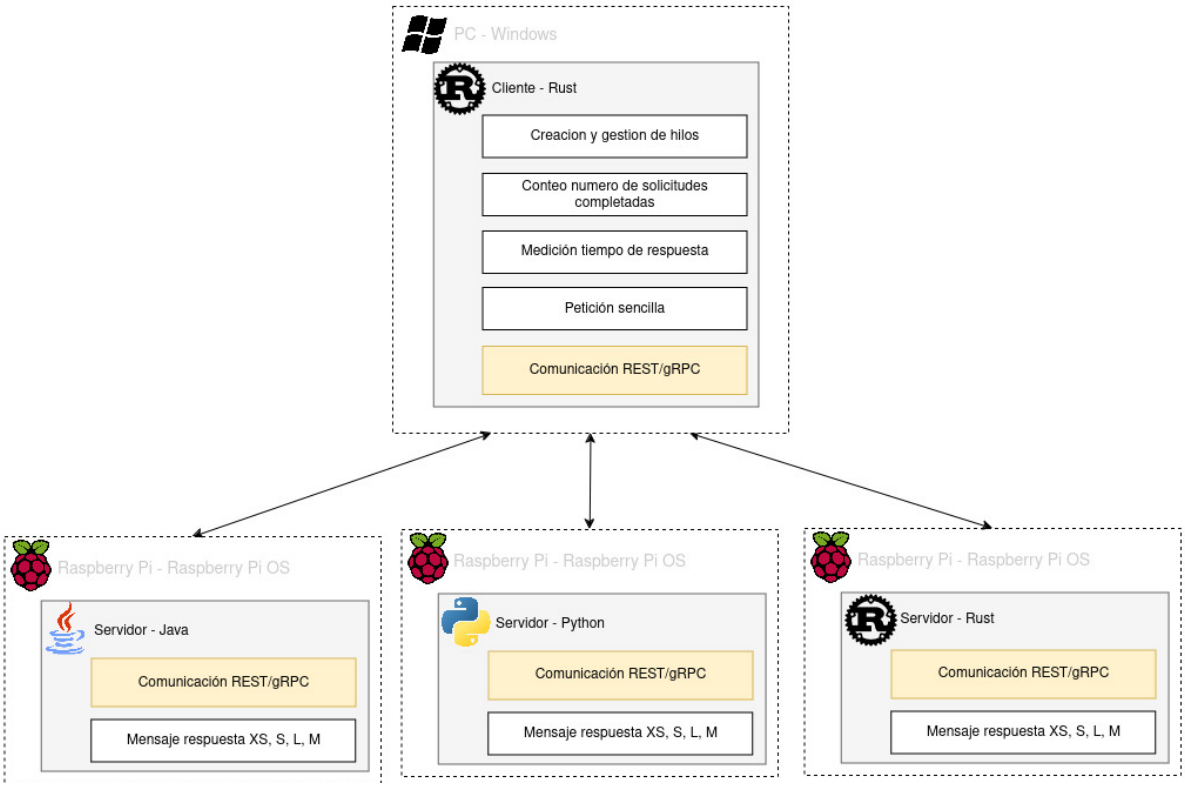

Figura 15: Diagrama de arquitectura basado en [5].

10.1.6. Comparativa Protocolo uso de CPU y memoria

En [11] se realiza una comparativa entre tres patrones de arquitectura de comunicación: Websocket, HTTP Polling y eventos del servidor; con la finalidad de proponer una arquitectura que utilice de la mejor manera los recursos disponibles. La prueba contempla los siguientes parámetros expuestos en la Figura 16:

Parámetros variables:

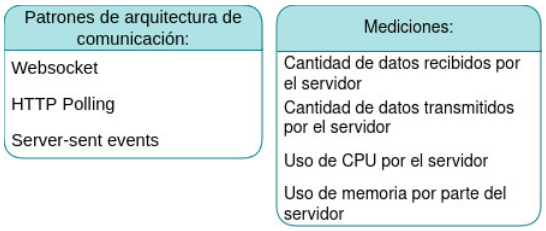

Figura 16: Parámetros variables basados en [11].

Metodología:

A partir de la configuración descrita en la Figura 17, se realizan cuatro tipos de pruebas diferentes para cada uno de los patrones de arquitectura:

- Cantidad de datos recibidos por el servidor.
- Cantidad de datos transmitidos por el servidor.
- Uso de CPU por el servidor
- Uso de memoria por parte del servidor

Las pruebas se realizan al generar un flujo constante de peticiones desde el frontend y ser resueltas por el servidor con un mensaje estándar. La diferencia entre la cantidad de datos enviados por el cliente y recibidos por el servidor permite identificar la cantidad de información extra, diferente a la carga útil, que envía cada uno de los patrones de arquitectura. Del mismo modo, la medición en el uso del CPU evidencia las particularidades en la gestión de la comunicación de cada patrón de arquitectura, aunque en la práctica estos valores sean opacados por el procesamiento, cálculos y demás procesos de la aplicación.

El test de medición en el uso de la memoria en el servidor, se presenta como una prueba relevante que es altamente influenciada por la implementaciones de las librerías que permiten el uso de estos patrones de arquitectura de la comunicación.

Diagrama Arquitectura:

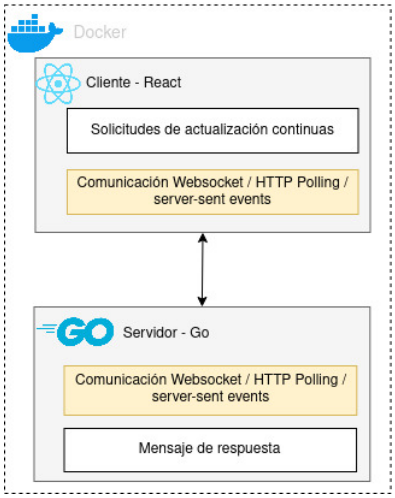

Figura 17: Diagrama de arquitectura basado en [11].

10.1.7. Comparativa protocolos para comunicación en tiempo real

En [12] se proponen tres momentos relevantes para la medición del tiempo que tarda una petición: Ida y vuelta, respuesta del servidor y procesamiento del cliente. A estas mediciones se someten los patrones de arquitectura de comunicación Websocket, Short polling, Long polling y Server-sent event. La prueba contempla los siguientes parámetros expuestos en la Figura 18:

Parámetros variables:

Patrones de arquitectura de comunicación:	Mediciones:	Tipo de ejecución:
Websocket	Tiempo de ida y vuelta del mensaje	Inicial
Short Polling	Tiempo de respuesta del servidor	A largo plazo
Long Polling	Tiempo de procesamiento del cliente	
Server-sent events		

Figura 18: Parámetros variables basados en [12].

Metodología:

A partir de la configuración descrita en la Figura 19, se realizan tres mediciones diferentes para cada uno de los patrones de arquitectura:

- Tiempo de respuesta servidor: Lo que tarda el servidor en recibir, procesar y responder a la petición.
- Tiempo de procesamiento del cliente: Hace referencia al tiempo que tarda el cliente en recibir, procesar y mostrar el mensaje al usuario.
- Tiempo ida y vuelta del mensaje: Se refiere al tiempo que tarda en enviarse el mensaje del usuario desde el cliente, en ser recibido y respondido por el servidor y que se muestre la respuesta de la petición al usuario.

Para las mediciones mencionadas anteriormente, se realiza una ejecución inicial y ejecución a largo plazo. El valor en la ejecución inicial es obtenido en el primer lanzamiento del test, también conocido como arranque en frío; esto para comprender la respuesta del sistema ante la llegada de peticiones desde cero. Las ejecuciones a largo plazo son mediciones realizadas después de varias iteraciones en las peticiones sobre el sistema; este es el comportamiento común que presentaría la aplicación.

Con los tres valores anteriores se procede a calcular el puntaje de velocidad en la aplicación, a través de la siguiente ecuación:

$$Spd(app) = \frac{Mrt(app) + Srt(app) + Cpt(app))))}{3} \tag{1}$$

Donde *Mrt* es el tiempo de ida y vuelta, *Srt* el tiempo de respuesta del servidor y *Cpt* es el tiempo de procesamiento del cliente.

Diagrama Arquitectura:

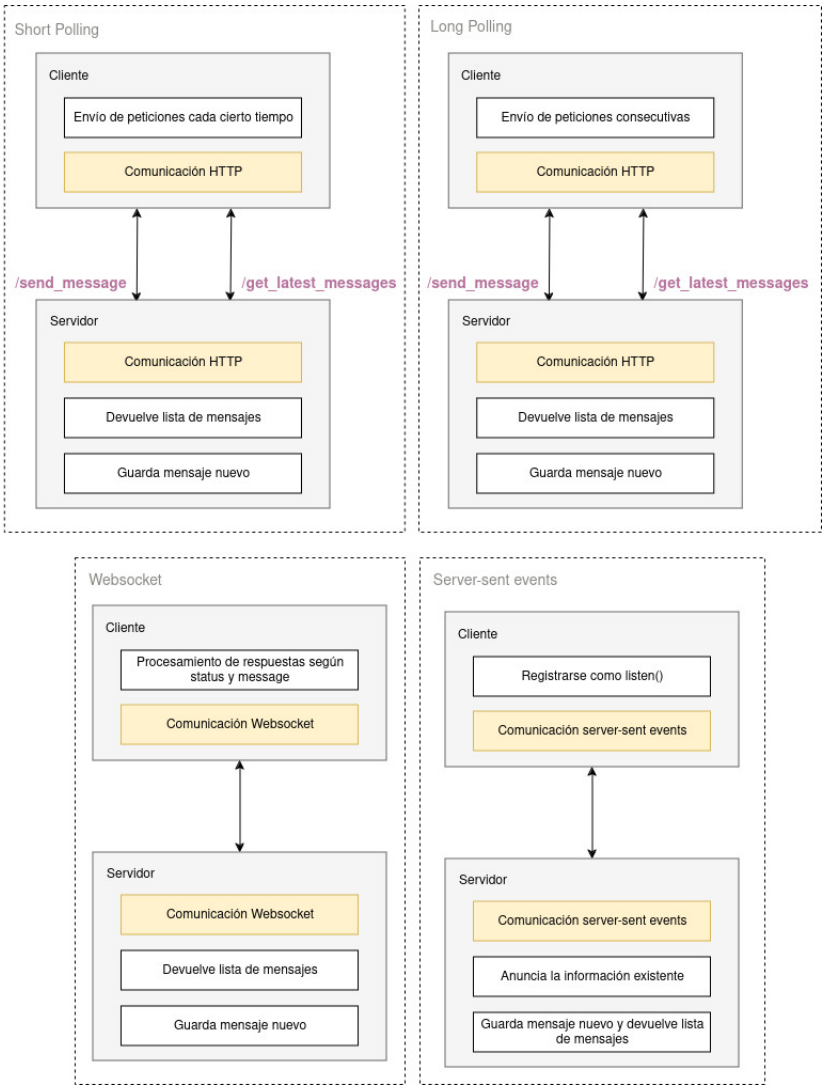

Figura 19: Diagrama de arquitectura basado en [12].

10.1.8. Evaluación protocolos REST y gRPC en latencia y flujo

En [13] se realiza la comparativa entre dos patrones de arquitectura: REST y gRPC, con la finalidad de identificar cual tenía el mejor rendimiento. Estas pruebas se ejecutan en un ambiente local y contemplan los siguientes parámetros expuestos en la Figura 20:

Parámetros variables:

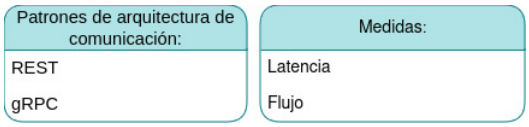

Figura 20: Parámetros variables basados en [13].

Metodología:

A partir de la configuración descrita en la Figura 21, se realizan dos mediciones diferentes para cada uno de los patrones de arquitectura:

- Latencia: Consta de medir el retraso en la llegada de un paquete al destino.
- Flujo: Hace referencia al número máximo de solicitudes que pueden ser atendidas por el servidor durante la ejecución.

Las pruebas constan del envío de 200.000 mensajes entre el servidor y el cliente. La información relacionada tanto al flujo como a la latencia fue obtenida por medio del programa Prometheus que realiza las funciones de interceptor en la comunicación entre las partes.

Diagrama Arquitectura:

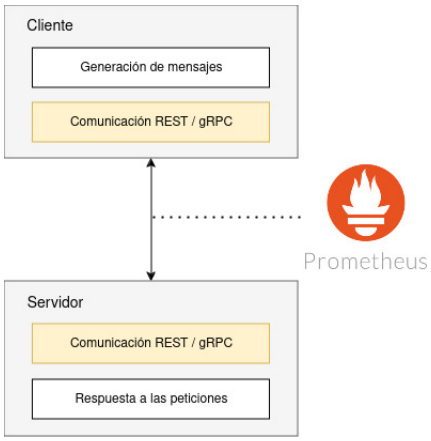

Figura 21: Diagrama de arquitectura basado en [13].

10.2. Metodología propuesta basada en metodologías existentes

A partir de la investigación de antecedentes metodológicos que abarcan la comparativa entre protocolos, se procede a la construcción de una nueva metodología que permita la comparativa y selección informada del protocolo que mejor se adapte a las condiciones particulares de hardware y latencia pertinente para una solución específica. Con esta finalidad, se procede con el descarte y justificación de las mediciones que no están directamente relacionadas con el objeto de la metodología a proponer:

10.2.1. Exclusión de mediciones

Consumo de energía

En [3], se propone la medición del consumo energético en un celular resultado del uso de los diferentes protocolos que fueron sometidos al test. Esta medición en particular se presenta como un valor de interés para proveedores de servicios en la nube y áreas afines al diseño de hardware en general pero no se alinea con el objeto de esta metodología enfocada a presentar medidas más generales que puedan proveer información útil a una porción más grande del sector de desarrollo web y afines.

Análisis de flujo

En [13] definen el análisis de flujo como el número máximo de solicitudes realizadas al servidor durante un tiempo determinado; presentando a esta medición como un valor de gran relevancia en la comparativa de protocolos enfocados en la transmisión de datos en tiempo real. En busca de no limitar la metodología propuesta únicamente a patrones de arquitectura de comunicación que puedan ser incluidos en esta categoría, se decide prescindir de esta medición.

Tiempo de procesamiento del cliente

En [12] se presenta la medición del tiempo que tarda el cliente en recibir, procesar y mostrar el mensaje al usuario. Esta medida proporciona un valor muy específico que depende netamente del procesamiento y presentación en el componente frontend de cada aplicación en particular, por lo que se descarta como medida para una metodología que busca mantener un carácter de uso más general.

Tiempo de respuesta del servidor

En [12] se plantea medir el tiempo que tarda el servidor en recibir, procesar y responder a la petición. En el caso de la metodología a proponer, el servidor se limita a la recepción y respuesta de las peticiones generadas por el cliente, ya que el procesamiento de las peticiones y otras acciones como la interacción con bases de datos, consulta a otros servicios, cálculos numéricos y demás posibles actividades, no cumplen directamente con el objeto de la metodología que es la comparativa de protocolos; evidenciando la medición del tiempo de respuesta del servidor como un valor del que prescindir.

Cantidad de datos recibidos por el servidor

En [11] se presenta la medición del rendimiento en la comparativa de los protocolos al realizar el envió de una carga útil fija desde el cliente y comparar la cantidad de datos que llegan al servidor. Este aumento en la cantidad de datos que recibe el servidor se produce por factores como cabeceras, el uso de *heartbeats*, entre otros. Dado a que el aumento en el tamaño de las peticiones producto de las particularidades propias de cada protocolo, no representa un valor relevante en relación con el ancho de banda en crecimiento constante, proporcionado por los servicios en la nube y en la web en general [15] [11], se decide excluir esta medida de la metodología.

Cantidad de datos transmitidos por el servidor

En [11] se presenta la medición de la variación en la cantidad de datos transmitidos por cada uno de los protocolos al enviar la misma carga útil como respuesta a una petición hacia el cliente. Como resultado de la prueba, se entiende que la diferencia entre la carga útil y la información adicional consecuencia del formato de transmisión de cada protocolo, es aún menor a resultante de la medición de la *cantidad de datos recibidos por el servidor*. Se descarta como medida para la metodología propuesta por los mismos motivos que la medición anterior.

10.2.2. Descripción de la metodología

Superada la etapa de investigación y descarte, se procede con la presentación de los parámetros de entrada, mediciones y los patrones de arquitectura que constituyen el marco de trabajo de esta metodología. Los parámetros escogidos se presentan a continuación en la Figura 22:

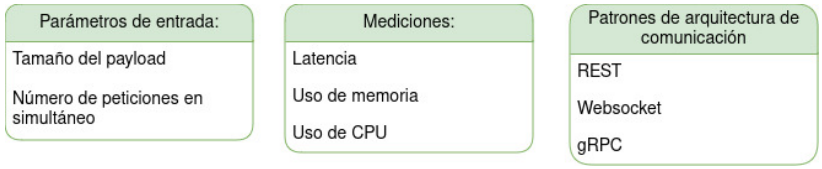

Figura 22: Parámetros metodología propuesta (autoría propia).

Parámetros de entrada:

Tanto el número de peticiones en simultáneo que llegan al servidor, como el tamaño en la carga útil que se transmite entre el cliente y el servidor, son parámetros que afectan directamente el rendimiento de la aplicación [5], reflejándose esta afectación en el aumento en los tiempos de respuesta y en un mayor consumo de recursos de procesamiento y memoria [2].

Mediciones:

En [12] se presenta la latencia como una medida relevante debido a que largos tiempos de espera afectan directamente en la experiencia del usuario, influyendo en la percepción satisfacción y reducción en la participación del usuario. Tanto el uso de memoria como de CPU por parte de los protocolos, son medidas que influyen en la toma de decisiones de arquitectura al momento de requerir de una escalabilidad de recursos en los procesos [11].

Patrones de arquitectura de comunicación:

La escogencia de estos 3 patrones de arquitectura de la comunicación: REST, Websocket y gRPC, se debe a que son formas bien establecidas de interoperar datos entre sistemas [3]. Siendo REST uno de los protocolo de comunicación de mayor popularidad entre los desarrolladores de software [4] [10], Websocket es mencionado como el protocolo revolucionario para la comunicación en tiempo real [10] y gRPC debido a las características propias de un protocolo basado en HTTP/2 que se perfila como una opción de amplia acogida en el futuro [13].

10.2.3. Entorno de pruebas

Dado que el core de la metodología propuesta es de evaluar la afectación en los tiempos de latencia de la comunicación cliente-servidor al verse expuesta a cambios en la carga útil del mensaje y al aumento en la concurrencia de las peticiones en simultáneo, se planteó un entorno

de pruebas controlado que pudiese mitigar cuellos de botella producto de hardware disparejos, valores de latencia aumentadas por conexiones de red lentas o competencia por recursos al ejecutar el cliente y el servidor en la misma máquina.

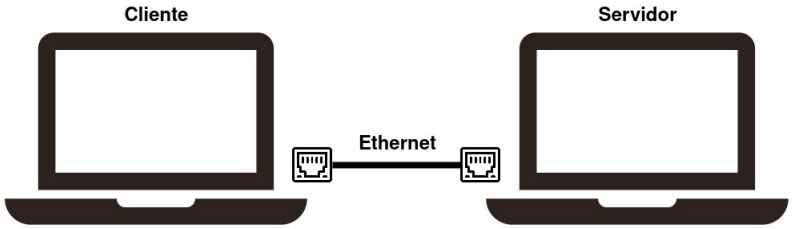

Figura 23: Esquema de red para el entorno de pruebas (autoría propia).

Como se muestra en la Figura 23, se utilizan dos computadoras portátiles que cumplen las funciones de cliente y servidor. Estas máquinas están conectadas por un cable Ethernet sin intermediarios, reduciendo el retraso en la comunicación de la red. Dado a que las tecnologías modernas permiten la negociación entre dispositivos sin la necesidad de enrutadores, módems y conmutadores, se puede hacer uso de la máxima velocidad en la comunicación por red a la que los computadores seleccionados para la prueba pueden llegar, que en este caso es de 1000Mb/s.

Las especificaciones técnicas de las máquinas seleccionadas se exponen en la Tabla 3. Tanto el cliente como el servidor son dispositivos idénticos en sus características de hardware y sistema operativo; la elección de dos sistemas idénticos se debe a la búsqueda de mitigar los cuellos de botella producto de hardware disparejos que no pueda cumplir con la demanda de peticiones de parte del cliente o genere tiempos de inactividad importantes al servidor producto de la capacidad de procesamiento del cliente.

	Cliente	Servidor
CPU	Intel Core i7-4710HQ	Intel Core i7-4710HQ
Freq.	2.50GHz	2.50GHz
Arq.	x86_64	x86_64
Cores	4	4
Threads	4	4
RAM	16GB DDR3-1600MHz	16GB DDR3-1600MHz
OS	Linux Mint 20.3	Linux Mint 20.3

Tabla 3: Especificaciones cliente y servidor (autoría propia).

Acorde con la estructura establecida para los antecedentes metodológicos, la Figura 24 presenta un diagrama de arquitectura donde se presentan las responsabilidades del cliente y el servidor. Cabe resaltar que ambas partes tienen la responsabilidad de la medición del uso de CPU y memoria en sus respectivas máquinas. El programa del cliente está en la capacidad de comunicarse con el servidor por medio de los protocolos REST, gRPC y Websocket; de igual forma, la máquina restante aloja un servidor por protocolo, los servidores solo están en funcionamiento al momento de la ejecución de su respectiva prueba y apagados en la ejecución de las demás para que no existan interferencias o competencia por recursos.

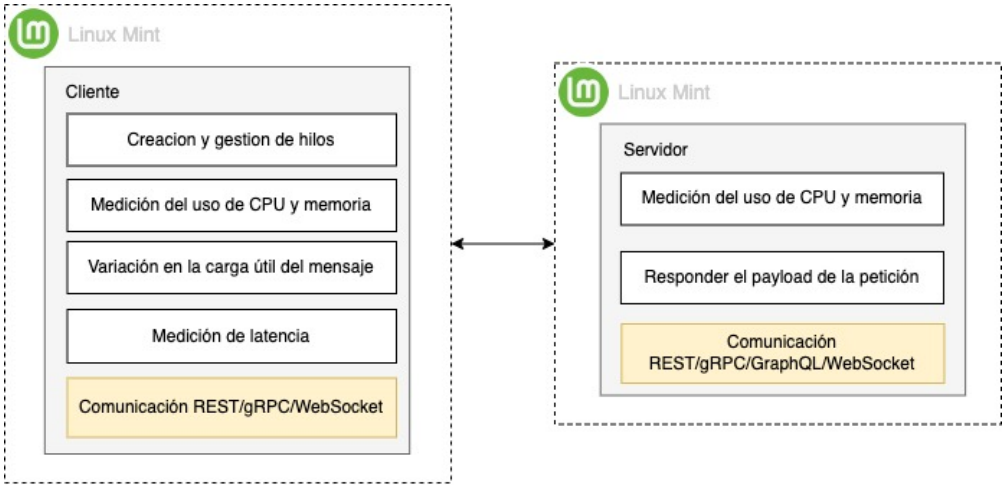

Figura 24: Diagrama de arquitectura de la metodología propuesta (autoría propia).

10.2.4. Implementación del cliente

El programa del cliente fue diseñado en el entorno de ejecución de Node.js basado en JavaScript para el envío de peticiones a cada uno de los servidores: REST, Websocket y gRPC. El software del cliente tiene la responsabilidad del aumento en la carga útil, el incremento en el número de hilos que emulan el número de usuarios en simultáneo que generan las peticiones, la medición del uso de CPU del cliente, el uso de memoria del cliente y la latencia en la comunicación con el servidor.

El proceso del cliente inicia con el envío de peticiones con una carga útil de 0kB, durante 2 segundos; superado este tiempo, se realizan incrementos de 5kB hasta alcanzar los 500kB, ver Figura 25. Ambos valores de tamaño, tanto inicial como final para la carga útil están basados en el artículo [5].

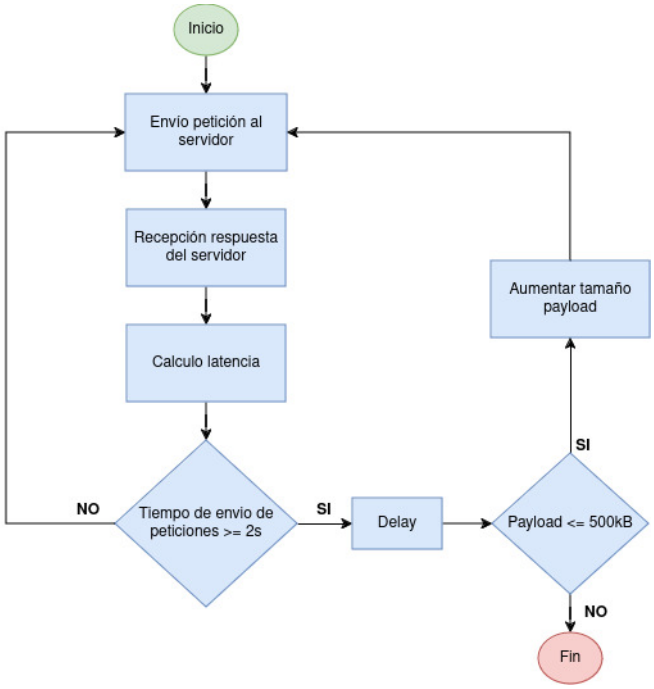

Figura 25: Flujo simplificado variación en el payload (autoría propia).

Al proceso de la Figura 25 se le denominará variación en el payload para futuras referencias. En este flujo se mide la latencia de cada una de las peticiones que se realizan en lo que dura un ciclo (2s); al terminar cada ciclo, el hilo pasa a un tiempo de espera (delay) de 2s. La finalidad

de este tiempo de espera es la de facilitar la identificación visual de los cambios entre ciclos del uso de CPU al momento de analizar los resultados.

El flujo de la variación en el payload, Figura 25, está contenido por un proceso más grande como se puede apreciar en la Figura 26. El proceso descrito en la Figura 26 consiste en el aumento progresivo de hilos que emulan a un número creciente de usuarios que realizan peticiones con una carga útil cada vez mayor. El proceso se repite hasta alcanzar el número máximo de hilos, en el caso de las pruebas este número máximo es de 50 procesos en simultáneo.

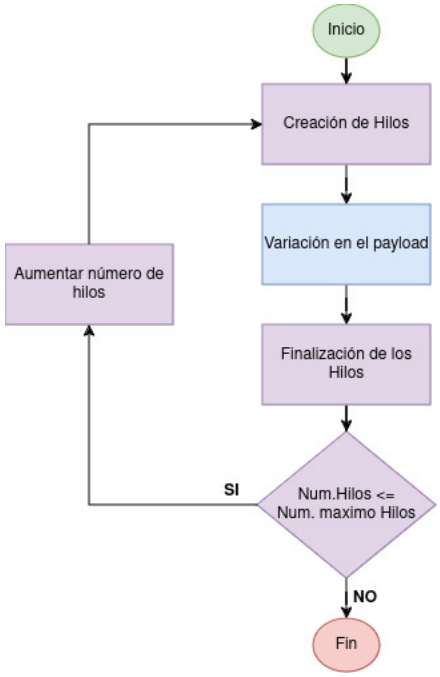

Figura 26: Flujo simplificado variación en el número de usuarios (autoría propia).

Al proceso de la Figura 26 se le denominará variación en el número de usuarios para futuras referencias. En este flujo se gestionan los datos de latencia generados por cada uno de los hilos para cada uno de los tamaños en la carga útil del mensaje. Cabe mencionar que el programa espera a la finalización de cada uno de hilos para iniciar nuevamente el proceso con una cantidad de hilos mayor.

Todos los flujos anteriormente descritos están contenidos en el macro flujo del cliente, Figura 27. El macro flujo contiene 2 procesos que se ejecutan en paralelo: La variación en el número de

usuarios y la medición del uso de CPU y memoria. El tiempo de vida de ambos procesos es el mismo y al terminar se procede al guardado de la información recopilada por parte del servidor en términos de latencia, uso de CPU y memoria, en un archivo JSON. Se busca que la escritura de los archivos no afecte las mediciones por lo que se realiza al final del programa cuando todos los flujos han terminado.

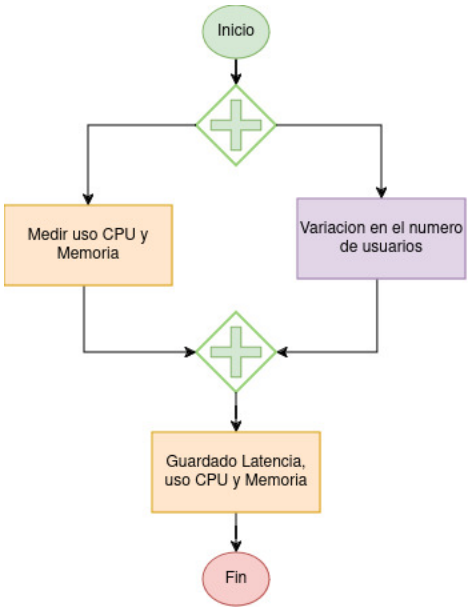

Figura 27: Flujo simplificado del cliente (autoría propia).

La medición del uso de CPU se realiza por medio de la librería del sistema "os" con el fin de registrar el promedio de carga del CPU. De igual forma, se utiliza esta librería para la obtención del valor de RSS (Resident Set Size) que es la cantidad de espacio de memoria física (RAM) que tiene un proceso. Estos valores, sumados a la latencia son guardados en un archivo de texto para su posterior procesamiento y graficación.

10.2.5. Servidor REST

El servidor REST fue diseñado en el entorno de ejecución de Node.js utilizando la librería Express como framework para la recepción de las peticiones HTTP/1.1 del cliente y dar respuesta a la misma. El servidor recibe las peticiones POST que genera el cliente y responde status 200 con la misma carga útil de la petición. El programa no realiza ninguna validación o transformación

a los datos enviados por parte del cliente. En paralelo al proceso de recepción y respuesta del cliente, se recopila información del uso de CPU y memoria por parte del servidor.

10.2.6. Servidor Websocket

El servidor Websocket se ejecuta bajo el entorno de JavaScript de Node.js utilizando la librería "ws" como framework para la comunicación por medio del protocolo de websocket. El servidor recibe las petición de conexión con el cliente, al aceptarla se establece una conexión para la comunicación bidireccional, pero para propósitos del que el protocolo sea comparable con los demás (REST y gRPC) se controla la generación de respuesta sólo como resultado a un mensaje de parte del cliente. En este caso la responsabilidad del cierre de la conexión la tiene el cliente al finalizar el flujo macro del cliente expuesto en la Figura 27. De igual forma, además de la tarea de comunicación con el cliente, el servidor recopila información del uso de CPU y memoria por parte del servidor.

10.2.7. Servidor gRPC

El servidor gRPC se ejecuta bajo el entorno de Node.js utilizando la librería "grpc/grpc-js" como framework para comunicar cliente-servidor por medio del protocolo de gRPC. Para hacer uso del protocolo gRPC se crea un archivo de definición protobuf (Listing 10.1); el cual consta del método *transferInformation()* que tiene como parámetro y retorno la variable payload de tipo string. De igual forma se puede apreciar que tanto el parámetro como el retorno son Unary RPCs, lo que significa que el cliente envía una sola solicitud al servidor y obtiene una respuesta posterior, al igual que una petición normal de HTTP/1.1, esto con la finalidad de que la metodología de comunicación entre los 3 protocolos sea comparable.

Listing 10.1: Definición del archivo .proto del servicio de comunicación

```
service CommunicationService {
  rpc transferInformation(Data) returns (Data);
}
message Data {
  string payload = 1;
}
```

11 Resultados y Discusión

11.1. Resultados

A continuación se presentan los resultados de la ejecución de las pruebas utilizando la metodología planteada en la sección anterior. La Figura 28 expone los resultados de la ejecución de la prueba sobre el protocolo gRPC en términos de latencia; debido a que la cantidad de información presente en la figura dificulta la interpretación de los datos y la comparación entre los diferentes protocolos. Con el propósito de que esta presentación de resultados faculte al intérprete para una toma de decisiones informada, se propone el uso de gráficos de área en el que se comparen los resultados de latencia, uso de CPU y uso de memoria para cada uno de las pruebas.

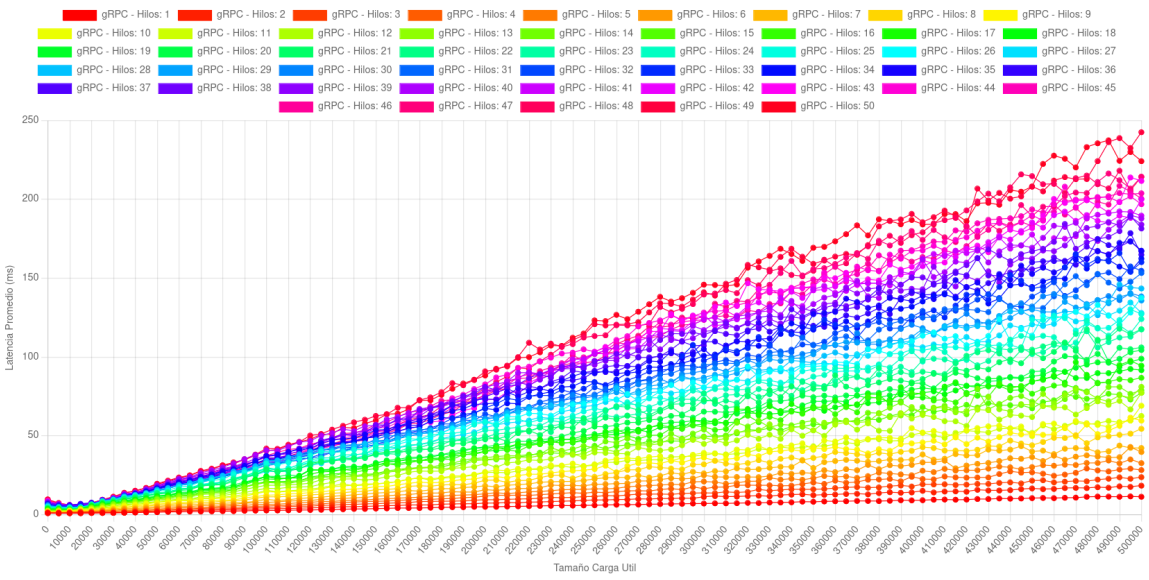

Figura 28: Resultados sin procesar latencia gRPC (autoría propia).

Los gráficos de área permiten entender los rangos de respuesta de los protocolos, además de la relación del resultado (latencia, uso de CPU y uso de memoria) con la variación en los parámetros de entrada: Tamaño del payload entre 5kB-500kB y Número de peticiones en simultáneo entre 1 hilo-50 hilos. A su vez, que se agrupen los resultados de la evaluación de los diferentes protocolos por variable medida, permite dimensionar mejor los rangos de trabajo de cada uno de los protocolos al compararse entre sí.

11.1.1. Comparación latencia cliente-servidor

Como se presenta en la Figura 29, los tiempos entre la petición de información de parte del cliente y la respuesta por parte del servidor se ven claramente afectados tanto por el tamaño de la carga útil como por la cantidad de peticiones en simultáneo, siendo el protocolo REST el que ostenta el menor cambio ante las variaciones en entrada con un rango de resultados mucho menor en comparación con gRPC y Websocket.

De igual forma, se puede apreciar en la Figura 29 como entre el rango de variación en la carga útil entre 0B y 25kB no se presentan diferencias significativas en los tiempos de latencia entre los protocolos, pero al alcanzar un payload de 500kB con 50 clientes generando peticiones en simultáneo Websocket se alcanza tiempos de 12.7 veces el de REST y gRPC con tiempos de 13.9 veces mayores que el de REST. Por lo que la latencia en cargas útiles mayores a 25kB el protocolo REST es claramente el que mejor desempeño demuestra.

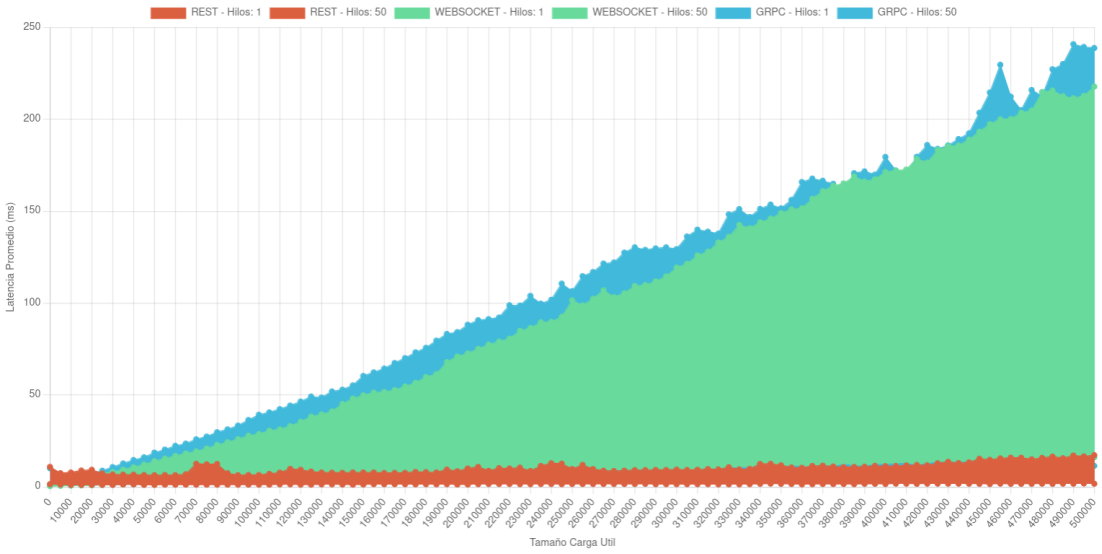

Figura 29: Resultados comparativos de latencia (autoría propia).

11.1.2. Comparación uso de CPU en el servidor

En la Figura 30 se presentan los resultados comparativos del porcentaje de uso de CPU en el servidor de cada uno de los protocolos (REST, Websocket y gRPC) al verse sometidos a una variación en el tamaño de la carga útil de los mensajes y en el número de clientes realizando peticiones en simultáneo. Se aprecia como el protocolo REST es el que mejor hace uso de los recursos de procesamiento con una rango de respuesta en el porcentaje de uso de CPU entre 0.78 %-1.3 % que prácticamente no cambia ante la variación en la carga útil y solo se ve afectado por el número de clientes que generan peticiones en simultáneo.

Para los protocolos Websocket y gRPC el aumento en la cantidad de clientes generando peticiones en simultáneo influye directamente en el incremento en el porcentaje de uso del CPU del servidor de forma importante, evidenciando un crecimiento de 3.7 veces entre porcentaje con un cliente: 1.64 % y con 50 clientes en simultáneo: 6.08 % para una carga útil de 5kB para Websocket y un crecimiento de 3.2 veces el inicial de 1.85 % para un solo cliente y de 6 % para 50 clientes en simultáneo para un payload de de 5kB, siendo el protocolo gRPC el peor desempeño en esta categoría.

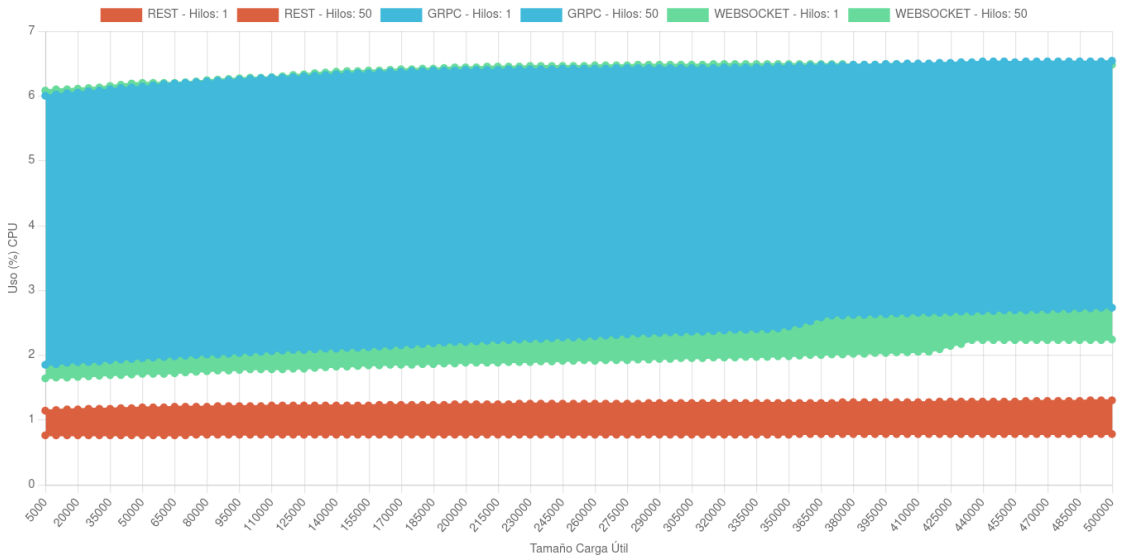

Figura 30: Comparativa uso de CPU en el servidor (autoría propia).

11.1.3. Comparación uso de memoria en el servidor

La Figura 31 expone los resultados producto de la comparación del uso de memoria (RSS) en el servidor por parte de los protocolos evaluados al ser sometidos a la variación en el tamaño de la carga útil y en el número de clientes generando peticiones en simultáneo. Resultando en un menor uso de memoria por parte del protocolo Websocket para peticiones menores o iguales 285kB en su carga útil para un solo cliente, para para cargas mayores a 285kB en un solo cliente y los demás casos, se evidencia un mejor desempeño del protocolo REST.

En la Figura 31 se observan las diferencias más marcadas en los resultados en el uso de memoria (RSS) al evaluar la respuesta producto del aumento en el número de clientes que generan peticiones en simultáneo, teniendo como valores para REST: 113.8MB, Websocket: 180.9MB y gRPC: 215.2MB para un payload de 5kB y 50 clientes en simultáneo, siendo el protocolo gRPC el que mayor cantidad de memoria utiliza en toda la prueba.

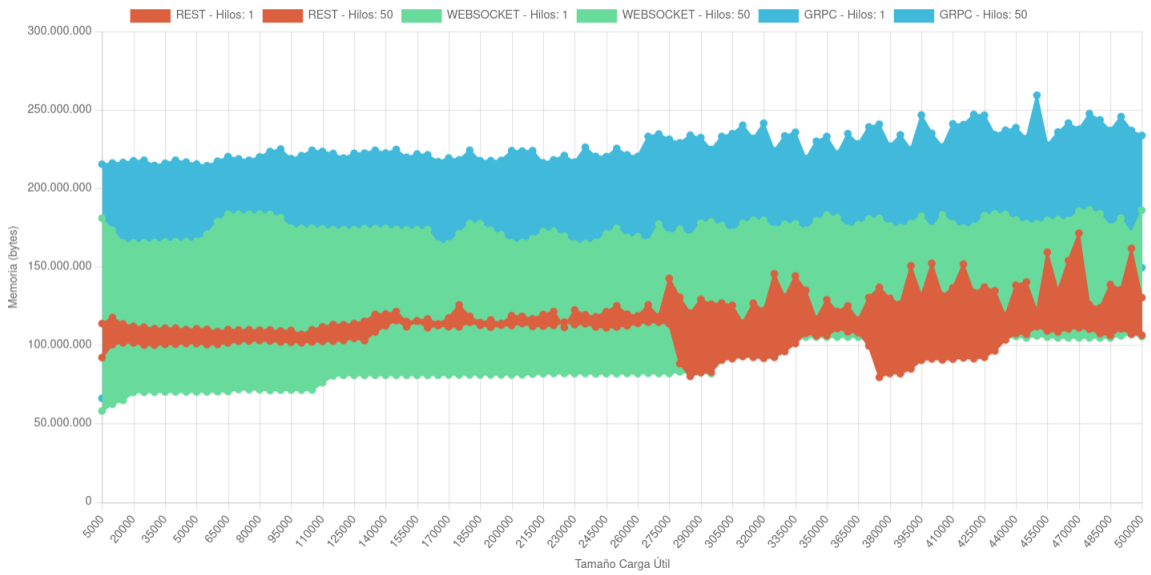

Figura 31: Comparativa uso de memoria en el servidor (autoría propia).

11.2. Caso de Uso

Para probar la capacidad de toma de decisión sobre la utilización del protocolo que mejor se adapte a una situación particular, se presenta el siguiente caso de uso:

- **Descripción:** Una empresa pyme en Colombia cuenta con un aplicativo de ecommerce desarrollado bajo una arquitectura de monolito, el cual desea migrar a una instancia de EC2 en la nube de Amazon Web Services (AWS).
- **Requerimientos:**
  - **Usuarios simultáneos:** 50 usuarios concurrentes.
  - **Payload:** El promedio en la carga útil de la comunicación entre el cliente y el servidor es de 250kB, con un tope aproximado de 300kB.
  - **Latencia:** El aplicativo tiene un tiempo promedio de procesamiento y respuesta a las peticiones de 100ms. Se desea que posterior a la migración las solicitudes sean respondidas en un tiempo menor a 250ms.

- **Uso de CPU:** El servidor utiliza aproximadamente el 10 % del CPU en momentos pico de demanda sobre el aplicativo.
- **Uso de memoria:** El servidor necesita de al menos 0.5GB de memoria para el correcto funcionamiento de la aplicación.

En busca de seleccionar la instancia de EC2 que mejor se adapte a las necesidades del caso de uso, se procede a la investigación de latencias y costos en relación a la región de disponibilidad de los servicios de AWS. La Tabla 4 se expone la latencia aproximada resultado de la red entre la región del servidor y la ubicación geográfica de donde se espera que se originen el mayor de peticiones (Colombia), con su respectiva variación en los costos ejemplificada con la instancia más pequeña de propósito general *t3.nano*.

Región	Latencia aprox.	Tipo instancia EC2	Costo hora (USD)
us-east-1 (Virginia)	84 ms	t3.nano	$0.0052
us-west-1 (California)	121 ms	t3.nano	$0.0062
ca-central-1 (Canada Central)	102 ms	t3.nano	$0.0058

Tabla 4: Latencia y costos por región, basado en [25] [26] [27].

A partir de la información de la Tabla 4 se procede a seleccionar la región *us-east-1 (Virginia)* debido a su menor latencia agregada y su menor costo por hora. De igual forma, teniendo en cuenta los requerimientos de latencia y la información sobre el payload, se descartan los protocolos Websocket con una latencia de 119.205ms y gRPC con 129.236ms para la carga útil máxima de 300kB del caso de uso, debido a su valor de latencia mayor a los requerimientos:

$$Latencia = RetardoProtocolo + RetardoRed + RetardoProcesamiento$$

- **Websocket:**

$$Latencia = 119{,}205ms + 84ms + 100ms$$

$$Latencia = 303{,}205ms$$

- **gRPC:**

$$Latencia = 129{,}236ms + 84ms + 100ms$$

$$Latencia = 313{,}236ms$$

- **REST:**

$$Latencia = 9{,}28ms + 84ms + 100ms$$

$$Latencia = 193{,}28ms$$

Con la claridad en la selección de la región *us-east-1 (Virginia)* y del protocolo REST, se procede a la elección de la instancia de EC2 que mejor se adapta a los requerimientos. La Tabla 5 relaciona el tipo de instancia con su respectivas capacidades de uso de CPU, memoria y costos en la región *us-east-1 (Virginia)*.

Tipo instancia EC2	Rendimiento vCPU	Memoria	Costo hora (USD)
t3.nano	5 %	0.5 GiB	$0.0052
t3.micro	10 %	1 GiB	$0.0104
t3.small	20 %	2 GiB	$0.0208
t3.medium	20 %	4 GiB	$0.0416
t3.large	30 %	8 GiB	$0.0832

Tabla 5: Costos Instancia EC2 región Virginia (us-east-1), basado en [25] [26].

Teniendo en cuenta que el requerimiento de memoria es de 0.5GB, se procede a calcular el uso máximo del CPU aproximado:

$$CPUTotal = CPUAplicativo + CPUProtocolo$$

$$CPUTotal = 10\,\% + 1{,}26\,\%$$

$$CPUTotal = 11{,}26\,\%$$

En base a la Tabla 5 y los requerimientos del uso de CPU total de 11.26 %, se descartan las instancias *t3.nano* y *t3.micro*. Y se selecciona la instancia de EC2 *t3.small* la cual puede cumplir con los requerimientos planteados; esta elección está soportada en la información producto de la metodología propuesta en términos de latencia, uso de CPU y uso de memoria de los protocolos REST, Websocket y gRPC.

11.3. Discusión

Debido a que los enfoques propios de cada protocolo son diferentes, fue necesario realizar implementaciones de estos bajo una misma estructura de petición-respuesta. Este método de

implementación que es propio del protocolo HTTP/1.1 que es el protocolo sobre el que se utilizó REST, puede verse como limitantes para los protocolos que se basan en HTTP/2 como lo es gRPC y no saca provecho de la conexión bidireccional del Websocket. El motivo de la implementación de esta estructura tenía como finalidad el que fuesen comparables estos 3 protocolos con enfoques tan diferentes, por lo que un buen punto en consideración es la elección de protocolos que tengan enfoques similares como lo pueden ser la comunicación en tiempo real o la gestión de múltiples peticiones desde un mismo origen.

En términos generales los resultados de estas pruebas no deberían ser tomados como regla general de desempeño, si no que son resultados particulares condicionados por la infraestructura e implementación propia de las pruebas, por lo que se insta a la ejecución de las pruebas sobre entornos similares a los de la solución final. De igual forma, se debe tener en cuenta que la implementación de librerías para el uso de protocolos influye directamente en la gestión de recursos como CPU y memoria, por lo que la comparativa entre resultados sobre un mismo protocolo utilizando diferentes librerías se visualiza como un posible marco de trabajo para futuras investigaciones.

12 Conclusiones

La investigación de trabajos similares de metodologías comparativas sobre protocolos, proporciona herramientas que permiten construir un marco de trabajo propio al generar un catálogo de posibles entradas, variables medibles, métodos de implementación y de presentación de resultados. Originando una metodología robusta que pueda identificar los parámetros de interés a evaluar resultando en una selección informada del protocolo que mejor se adapte a las condiciones de hardware y latencia deseados.

La adecuación de un entorno de pruebas controlado donde no se presenten hardware disparejos que generen cuellos de botella donde no se pueda cumplir con la demanda de peticiones de parte del cliente o se generen tiempos de inactividad importantes al servidor producto de la capacidad de procesamiento del cliente, conexiones de red lentas que repercutan en valores de latencia aumentados o competencia por recursos al ejecutar el cliente y el servidor en la misma máquina; permitiendo así una obtención de resultados de mayor veracidad para la correcta selección del protocolo que mejor se adapta a una solución en particular.

La ejecución de pruebas de carga con valores incrementales pequeños permite el identificar puntos de inflexión donde los protocolos convergen o desde donde el comportamiento cambia de forma pronunciada; información que podría pasarse por alto al solo escoger determinados valores para evaluar. Al identificar estos puntos de inflexión se puede predecir con mayor precisión los valores resultantes de latencia, uso de CPU y memoria dentro de un rango delimitado.

Los resultados producto de la utilización de la metodología propuesta, permiten la toma de decisión del protocolo que mejor se adapta a los requerimientos particulares de una solución y produce información relevante a través de medidas clave como lo son la latencia, uso de CPU y uso de memoria, para la toma de decisiones de arquitectura como se evidencia en el caso de uso propuesto.

Bibliografía

[1] Fernando Doglio. *Rest API development with node.js: Manage and understand the full capabilities of successful rest development*. Apress, 2018.

[2] Martin Johansson y Olivos Isabella. *Comparative Study of REST and gRPC for Microservices in Established Software Architectures*. 2023.

[3] Carolina Luiza Chamas, Daniel Cordeiro y Marcelo Medeiros Eler. «Comparing REST, SOAP, Socket and gRPC in computation offloading of mobile applications: An energy cost analysis». En: *2017 IEEE 9th Latin-American Conference on Communications (LATINCOM)*. 2017, págs. 1-6. DOI: 10.1109/LATINCOM.2017.8240185.

[4] Lukasz Kamiński et al. «Comparative Review of Selected Internet Communication Protocols». En: *Foundations of Computing and Decision Sciences* 48.1 (2023), págs. 39-56. DOI: doi:10.2478/fcds-2023-0003. URL: https://doi.org/10.2478/fcds-2023-0003.

[5] Johan Berg y Daniel Mebrahtu Redi. *Benchmarking the request throughput of conventional API calls and gRPC: A Comparative Study of REST and gRPC*. 2023.

[6] Victor Hugo Fernández Bedoya. «Tipos de justificación en la investigación científica». En: *revista* 4.3 (jul. de 2020), págs. 65-76.

[7] Łukasz Kamiński et al. «Comparative review of selected Internet communication protocols». En: *arXiv preprint arXiv:2212.07475* (2022).

[8] D.A. Menasce. «Load testing of Web sites». En: *IEEE Internet Computing* 6.4 (2002), págs. 70-74. DOI: 10.1109/MIC.2002.1020328.

[9] Redacción Portafolio. *Industria de software, clave para atraer inversión extrajera al país*. Oct. de 2022. URL: https://www.portafolio.co/innovacion/fedesoft-industria-de-software-clave-para-atraer-inversion-extrajera-a-colombia-573053.

[10] L.D.S.B Weerasinghe e I Perera. «Evaluating the Inter-Service Communication on Microservice Architecture». En: *2022 7th International Conference on Information Technology Research (ICITR)*. 2022, págs. 1-6. DOI: 10.1109/ICITR57877.2022.9992918.

[11] Oona Laitamäki. «WEB APPLICATION ARCHITECTURE FOR REAL-TIME MOBILE NETWORK ANALYSIS». En: (2023).

[12] Nataliia Sharonova, Iryna Kyrychenko y Daria Shapovalova. «Comparative Analysis of Instant Messaging Protocols and Technologies for Effective Communication in Computer-Mediated Environments». En: ().

[13] Maicon Alcântara de Oliveira y Romualdo Monteiro de Resende Costa. «Análise da Eficiência da Transferência de Dados em uma Rede de Microserviços–Proposta de Comparação de Desempenho entre REST E GRPC». En: *Caderno de Estudos em Engenharia de Software* 4.2 (2023).

[14] *HTTP | MDN*. Jul. de 2023. URL: https://developer.mozilla.org/es/docs/Web/HTTP.

[15] Brian Totty et al. *HTTP: The Definitive Guide*. Sep. de 2002. URL: http://ci.nii.ac.jp/ncid/BA62153412.

[16] Clinton Wong. *HTTP Pocket Reference*. .º'Reilly Media, Inc.", jun. de 2000.

[17] Mike Belshe, Roberto Peon y Martin Thomson. *Hypertext Transfer Protocol Version 2 (HTTP/2)*. RFC 7540. Mayo de 2015. DOI: 10.17487/RFC7540. URL: https://www.rfc-editor.org/info/rfc7540.

[18] Barry Pollard. *HTTP/2 in Action*. Simon y Schuster, mar. de 2019.

[19] Leonard Richardson y Michael Amundsen. *RESTful web apis*. O'Reilly Media, 2013.

[20] Mark Masse. *REST API design rulebook*. .º'Reilly Media, Inc.", oct. de 2011.

[21] Leonard Richardson y Sam Ruby. *RESTful web services*. .º'Reilly Media, Inc.", dic. de 2008.

[22] Alexey Melnikov y Ian Fette. *The WebSocket Protocol*. RFC 6455. Dic. de 2011. DOI: 10.17487/RFC6455. URL: https://www.rfc-editor.org/info/rfc6455.

[23] Andrew Lombardi. *WebSocket*. .º'Reilly Media, Inc.", sep. de 2015.

[24] Kasun Indrasiri y Danesh Kuruppu. *gRPC: Up and Running*. O'Reilly Media, ene. de 2020.

[25] *Instancias T3 de Amazon EC2 – Amazon Web Services (AWS)*. URL: https://aws.amazon.com/es/ec2/instance-types/t3/.

[26] *Precios de las instancias bajo demanda de EC2 – Amazon Web Services*. URL: https://aws.amazon.com/es/ec2/pricing/on-demand/.

[27] *Cloud services latency calculator*. URL: https://cloudping.info/.