ANÁLISIS DEL USO DE WEBSOCKETS PARA IMPLEMENTAR APLICACIONES WEB EN TIEMPO REAL

José Guillermo Fierro Mendoza

Tecnológico Nacional de México en Celaya

guillermo.fierro@itcelaya.edu.mx

Francisco Gutiérrez Vera

Tecnológico Nacional de México en Celaya

francisco.gutierrez@itcelaya.edu.mx

Julio Armando Asato España

Tecnológico Nacional de México en Celaya

julio.asato@itcelaya.edu.mx

Claudia Cristina Ortega González

Tecnológico Nacional de México en Celaya

claudia.ortega@itcelaya.edu.mx

Eduardo Alejandro Noria Juárez

Tecnológico Nacional de México en Celaya

eduardo.noria@gmail.com

Resumen

En este artículo, en base al análisis del comportamiento del protocolo para la transferencia de datos de una aplicación web que utiliza WebSockets (WS), se establecen los beneficios de integrarlo en las aplicaciones que requieren ejecutarse en tiempo real o en las que incluso el inicio de la comunicación puede originarse en el lado del cliente o en el lado del servidor, como es el caso de monitoreo remoto de procesos, juegos interactivos, chats con retroalimentación del estado de los usuarios, entre otras aplicaciones. Para propósitos de pruebas se ha

implementado una aplicación web de chat, donde el servidor recibe mensajes de clientes que se conectan mediante un navegador y que al recibir un nuevo mensaje de un usuario emite a todos los usuarios conectados un mensaje para que sea desplegado en un bloque de la página del navegador. Para establecer puntos de evaluación se analiza el comportamiento del protocolo durante la transferencia de datos, se analizan parámetros como el tiempo de carga y la latencia para evaluar el comportamiento del protocolo, mediante pruebas de stress utilizando una herramienta de software como JMeter. Las pruebas realizadas demuestran que la latencia utilizando el protocolo es baja y que una gran cantidad de usuarios conectados no afecta el desempeño del protocolo WS durante la transferencia de datos.

Palabra(s) Clave: Aplicaciones web, Latencia, Protocolo WebSockets, Tiempo real.

Abstract

This article, based on the analysis of the behavior of the protocol for the transfer of data from a web application using WebSockets (WS), the benefits of integrating it in applications that require real-time execution or in which even the start is established Communication can originate on the client side or on the server side, as is the case of remote monitoring of processes, interactive games, chats with feedback on the status of users, among other applications. For testing purposes a web chat application has been implemented, where the server receives messages from clients that are connected through a browser and upon receiving a new message from a user sends all the connected users a message to be displayed in a block of the browser page. To establish evaluation points, the behavior of the protocol is analyzed during the transfer of data, parameters such as loading time and latency are analyzed to evaluate the behavior of the protocol, through stress tests using a software tool such as JMeter. The tests carried out show that the latency using the protocol is low and that a large number of connected users do not affect the performance of the WS protocol during data transfer.

Keywords: *Latency, Real time, Web applications, WebSockets protocol.*

1. Introducción

Una aplicación de software en tiempo real está caracterizada porque la respuesta a un estímulo o evento se produce en un intervalo o plazo de tiempo determinado, limitado a la exigencia que demanda el entorno, [Sommerville, 2011]. Ejemplos de ellas son un tablero de despliegue de resultados de juegos en línea, que son actualizados de acuerdo a como se mueve el marcador, un almacén en línea mostrando la existencia o inventario de productos, un sitio web de noticias en línea, donde la última noticia se despliega primero desplazando a las anteriores, un sistema de publicación de precios de las acciones de acuerdo a su cotización mundial, un chat o sistema de mensajería entre múltiples usuarios un juego interactivo remoto e incluso operado por dispositivos móviles y sistemas de monitoreo de señales vía web conectados a equipos o maquinaria, entre otros. La referencia [Kaazing, 2017], contiene ejemplos diversos que ilustran de muy buena forma el alcance de este tipo de aplicaciones.

La terminología común en todas estas aplicaciones es que la información debe ser desplegada oportunamente desde la ocurrencia del evento y la entidad que está escuchando para el despliegue de los datos. En este sentido, la evolución tecnológica de los sistemas web y la demanda de situaciones que precisan que la información se despliegue con la exigencia de tiempo real, es que se ha venido utilizando o adoptando diversas técnicas o métodos para resolverlo. El protocolo de comunicación http, utilizado para la comunicación a nivel de la aplicación, es un protocolo que no tiene estado, un cliente se conecta a un servidor a través de una conexión y hace una petición a un servidor y éste le responde con el recurso solicitado o con un código de error. Sin embargo, una nueva solicitud, requiere de una nueva conexión y no conoce información del estado de la conexión anterior.

La transmisión en el protocolo http, es *Half duplex*, es decir, aunque es bidireccional, sólo se puede transmitir en un sentido a la vez, lo que limita a que el cliente tenga que esperar a que el servidor responda para realizar otra petición. AJAX fue una tecnología que pretendía hacer que Internet pareciera más dinámico. Sin embargo, aún todas las comunicaciones HTTP eran dirigidas por el cliente, lo que requería la interacción del usuario o había que preguntarle

periódicamente cada vez que se cargaban nuevos datos del servidor, Ubl, M. & Kitamura, E. [2010]. Las técnicas para implementar esquemas de comunicación entre el cliente y el servidor con la finalidad de mejorar el tiempo de respuesta evolucionaron desde el método conocido como "*polling*", que con algunas mejoras derivó en el método "*long pollig*". Otro método interesante fue el que se conoce como "http streaming" o transmisión por secuencias. Sin embargo, estas tecnologías no resolvieron la problemática de enviar más de un mensaje a la vez desde el cliente en una conexión o mantener una "conversación" interactiva entre el cliente y el servidor, Ubl, M. & Kitamura, E. [2010]. Otra gran necesidad actual, sobre todo las aplicaciones de Internet en las cosas (IoT), es que la comunicación no necesariamente la debe iniciar el cliente, sino que producto del evento producido por una señal de un dispositivo externo, el servidor recibe la información y entonces debe comunicarle la información al cliente para su despliegue en el navegador. Como solución, surge el concepto WebSockets (WS), un protocolo en la capa de transporte, TCP. Con la característica de que es bi-direccional, es decir, el servidor puede enviar mensajes también y enviar un mensaje en cualquier momento al cliente. En la figura 1 se muestra gráficamente el funcionamiento del protocolo. Para iniciar una conexión, el cliente hace una petición al servidor, en realidad es una negociación de tipo *handshake*, para validar la comunicación.

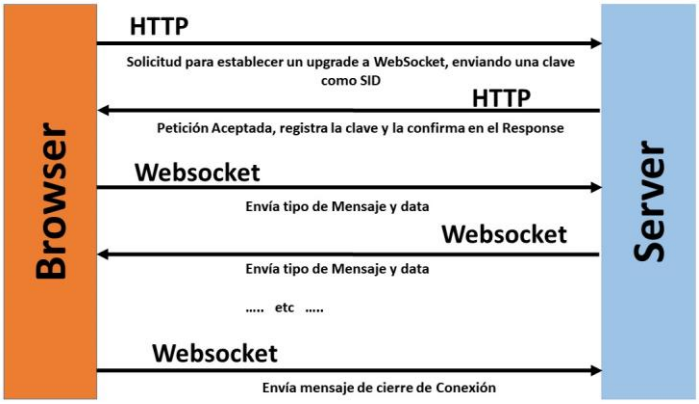

Figura 1 Protocolo WebSocket.

Luego, si el servidor responde aceptando la conexión se produce un *upgrade* de la conexión, o sea que dejan de utilizar HTTP y pasan al intercambio o transferencia

de datos mediante WS. En la figura 2 se muestra un ejemplo de la estructura del *handshake*, tanto del lado del cliente como del servidor.

Figura 2 Ejemplo de la estructura del WebSocket en la etapa de negociación.

Una vez que el cliente y el servidor han enviado su *handshake*, y si resultó exitoso, comienza la parte de transferencia de datos, consistente en un canal de comunicación bidireccional donde cada lado puede independientemente del otro. Además de proporcionar una conexión bi-direccional, el WS, elimina lo que se denomina, *payload* de la conexión, es decir, el encabezado del protocolo HTTP, o *header*, donde se encapsulan la información para que el servidor conozca quien se la envía y qué tipo de conexión se está utilizando. Este trabajo tiene como propósito, a través de la puesta en práctica de una aplicación basada en WS, determinar a través de pruebas de rendimiento demostrar objetivamente los beneficios del uso del protocolo de WS.

2. Metodología

Primero, se construyó una aplicación cliente-servidor, con la funcionalidad de un chat simple. El servidor recibe mensajes de una gran cantidad de usuarios, los almacena internamente y los despliega en los clientes que estén conectados. En la figura 3 se muestra el flujo de información entre los clientes y el servidor de la aplicación web para el caso de estudio.

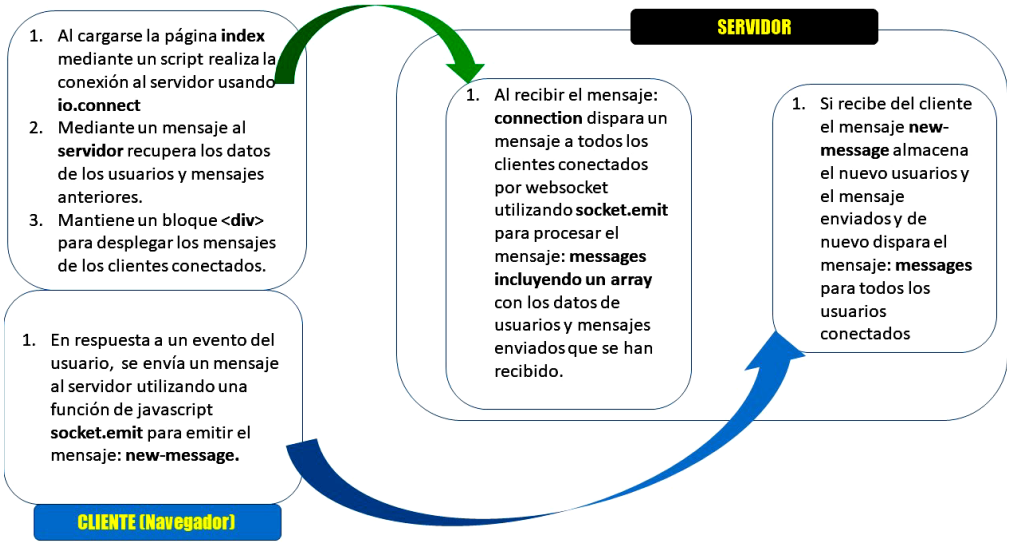

Figura 3 Esquematización de la comunicación en la aplicación chat.

Para implementar el servidor se utilizó node.js y como alternativas para programar la comunicación bidireccional se tienen el uso de WS estándar o las librerías socket.io de javascript que están basadas en WS, pero que también aportan otras alternativas justo en el momento de la ejecución [Clements, 2014], como el caso de reconexión automática en caso de falla de la red.

El análisis del comportamiento del protocolo se hizo utilizando JMeter, herramienta para realizar simulaciones sobre cualquier recurso de Software, por ejemplo, sitios web o componentes basados en http o https, bases de datos, sitios ftp y otros.

De esta manera fue posible también evaluar el tiempo de respuesta de la comunicación entre el cliente y el servidor bajo el protocolo WS implementando pruebas para enviar mensajes entre el cliente y el servidor. Una segunda forma de evaluar el comportamiento de los mensajes y sobre todo para efectos de evaluar la recuperación del protocolo a fallas por desconexión se realizó mediante el uso de las herramientas de los navegadores como el complemento de Chrome de herramientas del desarrollador.

Para simular una cantidad significativa de clientes conectados a la vez al servidor del chat, utilizamos JMETER, lo que nos permite construir un plan de prueba con este tipo de funcionalidad y evaluar el desempeño del servidor al tiempo que verificamos el "*payload*" o carga del *header* para el intercambio de mensajes.

Finalmente se evalúa el desempeño del servidor, analizando la información, de cada solicitud-respuesta, mediante la identificación de los elementos del protocolo que hacen que la comunicación de paquete transmitidos entre cliente y servidor tenga un mejor rendimiento.

El servidor se implementó utilizando las librerías expressJS y socket.io. Los clientes se conectan mediante un navegador y envían mensajes al servidor que los almacena y en respuesta a cada evento un mensaje con el historial actualizado de todos los mensajes recibidos en la sesión. La petición inicial es una solicitud a una página HTML que despliega un formulario para edición de los datos y que una vez capturados se envían al servidor mediante funciones de Javascript, que hacen llamadas al API de WS para establecer la conexión y enviar el mensaje respectivo, que es procesado del lado del servidor.

Para analizar el protocolo se utilizó Jmeter, una aplicación de software libre que permite realizar diferentes pruebas de rendimiento y stress a las aplicaciones, simulando la funcionalidad del navegador. Para el caso de las pruebas de WS, utilizó un generador de muestras, identificado como: JMeter WebSocket Sampler, plugin desarrollado por Maciej Zaleski y hasta el momento de esta investigación, era la única extensión compatible con RFC6455 que admite la reutilización de una sesión TCP y es fácil de instalar y usar. Se simuló un escenario durante algunos minutos, pretendiendo que tenemos un pico de usuarios que envían información a la aplicación. Para ejecutar la misma prueba de rendimiento una y otra vez, mediante JMeter se implementó un esquema de repetición de eventos, simulando la interacción del usuario. En este caso, mediante JMeter se modeló el hecho de que cada usuario debe tener un nombre único y envía su mensaje mediante la generación de cadenas aleatorias con un determinado patrón.

3. Resultados

La primera prueba consistió en la validación de que el protocolo permite la comunicación bidireccional. A través de JMeter y de las herramientas del desarrollador en los navegadores, pudimos comprobar que efectivamente se establece un canal que permanece abierto y tanto el cliente como el servidor

pueden emitir mensajes. Un cliente envía su mensaje al servidor y este responde enviando el mensaje a todos aquellos que se hayan conectado, y en el navegador se actualiza el bloque donde se está desplegando el mensaje sin necesidad de actualizar la página. Esta prueba se realizó con el servidor montado en una computadora y desde otros equipos conectados a la red se hicieron las peticiones correspondientes, el esquema de comunicación implementado se muestra en la figura 4. Es importante destacar que como el protocolo inicia con una petición http, los firewalls no afectan la conexión WS.

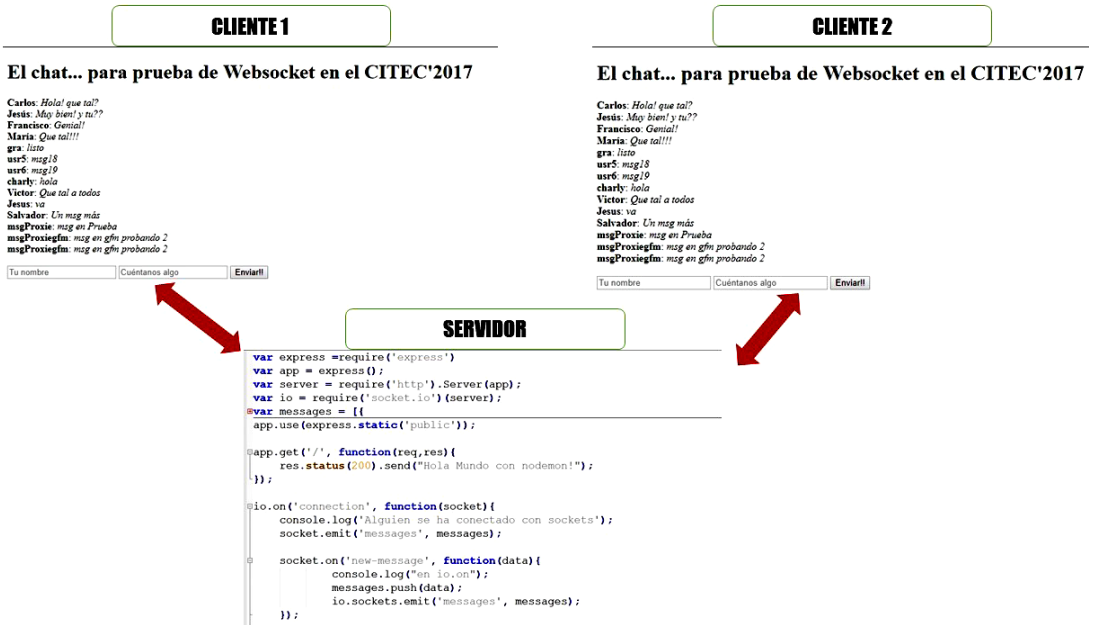

Figura 4 Configuración de la aplicación web.

Una segunda serie de pruebas fueron relativas a la recuperación de la comunicación que usa WS cuando se pierde la comunicación entre el cliente y el servidor. Se hicieron pruebas de pérdida de conexión que duraron desde unos cuantos instantes de tiempo hasta varias horas y en todos los casos el protocolo sockets.io se recuperó satisfactoriamente. Esto se debe en gran medida al mecanismo de *long polling* que incorpora la librería socket.io y que se mantiene enviando peticiones de tipo Xhr hasta que hay una respuesta o en su defecto sea abortada por el usuario. En la figura 5 se muestra como se recuperó la conexión después de una desconexión.

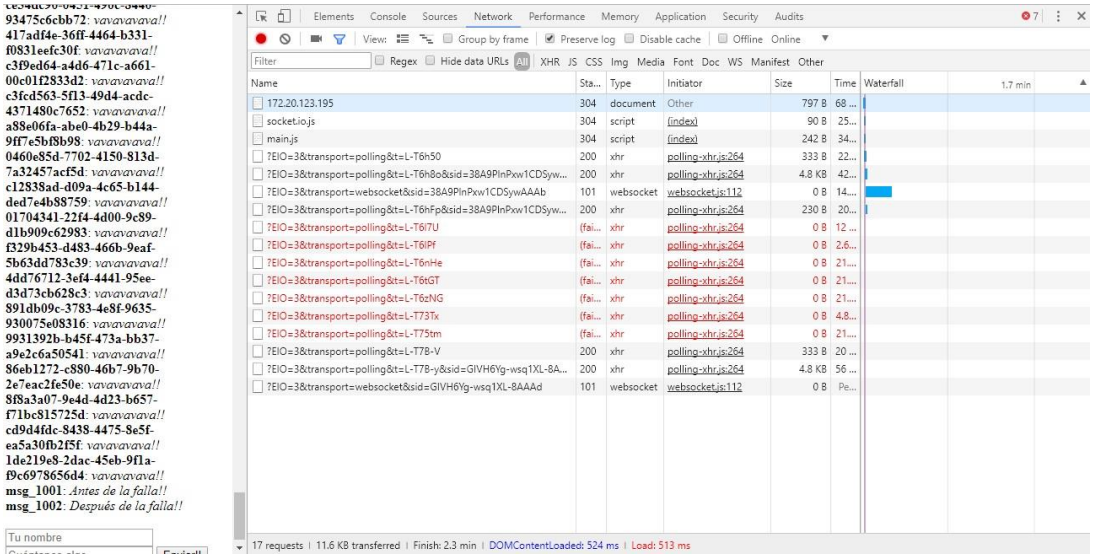

Figura 5 Recuperación de la conexión después de un evento de falla de la conexión.

Respecto al análisis que demuestra el impacto producida por el aumento del tamaño de la data transferida en una aplicación web en tiempo real, se simuló un escenario en que 1000 clientes en un segundo envían un mensaje cada dos segundos. La cantidad de bytes enviados aumentó debido a la acumulación del historial de mensajes en la sesión en cada petición/respuesta desde el cliente al servidor, en este caso solo se afectó el tiempo de carga de los datos, pero no hubo efectos en otros parámetros como la latencia. Quedó demostrado que una vez que la conexión se establece, todos los mensajes siguientes se realizan sobre la conexión abierta en lugar de realizar una nueva petición/respuesta a sólo que haya una pérdida de conexión.

En las pruebas de JMeter se registró un valor de latencia de cero en la mayoría de los casos y el parámetro de "*load time*" o tiempo de carga se mantuvo por bajo de 20 milisegundos para la prueba de carga del servidor, donde el tamaño del cuerpo del mensaje va desde 16 bytes al inicio de la prueba hasta 14099 bytes. El tamaño del *header* en una petición de tipo http, para el caso del ejercicio fue de 293 a este valor le tendríamos que sumar la cantidad de bytes enviados en el cuerpo del mensaje. El WS una vez establecido no requiere *header*, por lo tanto, se asume que los WS pueden enviar y recibir más mensajes por unidad de tiempo que otras conexiones, los resultados obtenidos se presentan en la tabla 1.

Tabla 1 Resultados obtenidos con cada caso de prueba.

Nombre	Status	Type	Initiator	Size (B)	Time (ms)	TFB	Stalled
localhost	304	document	other	797	414		
Socket.io.js	304	script	index	90	287	11	74.5
main.js	304	script	index	242	284	4.37	79
http://localhost:8081/socket.io/?EIO=3&transport=polling&t=L-E-7Lh	200	Xhr	Polling-xhr.js:264	333	312	6.48	11.82
http://localhost:8081/socket.io/?EIO=3&transport=polling&t=L-E-7Qu&sid=EoPmx70VYY4B4jqIAAEz	200	xhr	Polling-xhr.js:264	989	420	1.5	9.55
ws://localhost:8081/socket.io/?EIO=3&transport=websocket&sid=EoPmx70VYY4B4jqIAAEz	101	Websocket	Websockets.js:112	0	-	-	-
http://localhost:8081/socket.io/?EIO=3&transport=polling&t=L-E-7Xb&sid=EoPmx70VYY4B4jqIAAEz	200	xhr	Polling-xhr.js:264	230	239	48	47

Un beneficio más que pudimos probar mediante JMeter es el efecto de la latencia del protocolo. Al mantenerse la conexión persistente entre el servidor y el cliente la latencia solamente refleja los efectos que la red pudiera generar para hacer más lento el envío de los paquetes. Cuando las pruebas se realizaron en la misma máquina (localhost), la latencia casi siempre arrojó valores de cero y en todos los casos siempre fue menor a cuatro milisegundos. En la configuración real, donde el servidor y el cliente no son la misma computadora contrario a lo que pudiera suponerse, la latencia se mantuvo en cero. Entendiendo que en realidad sólo pudiera afectarse por las características de la infraestructura de red.

Para evaluar el performance o rendimiento del protocolo, se ejecutó una prueba en JMeter configurando una simulación en el esquema de conexión entre el cliente y el servidor con el protocolo WS estableciendo un parámetro de 100 clientes enviando 50 mensajes de cada uno.

El parámetro que pudimos observar que aumenta es el “tiempo de carga”, y es razonable por la cantidad de datos que se están manejando en el historial. Coincidiendo con lo que se ha escrito en las diferentes referencias bibliográficas, ya que en el caso de Node.js las peticiones no se realizan como sistema multihilo estándar, sino que se realiza con un sistema monohilo, esto implica que solo atiende a un cliente a la vez, cuando el cliente solicita un recurso al hilo, el hilo delega la obtención de ese recurso al Sistema Operativo por una interfaz POSIX, y mientras el recurso está siendo obtenido, el cliente libera el hilo y se va a la sala

de espera hasta ser llamado. Cuando el hilo ha obtenido el recurso, el cliente vuelve a la cola de clientes, para que cuando llegue su turno, recibir el recurso solicitado. Esto pudiera parecer que ralentiza la comunicación, pero en realidad las referencias establecen que el protocolo soporta millones de clientes, claro dependiendo de la infraestructura de cómputo en el lado del servidor.

4. Discusión

Por medio de la aplicación construida, el análisis teórico suministrado de la especificación del protocolo de WS y la información extraída de las pruebas realizadas, podemos concluir que efectivamente se comprueba que para las aplicaciones de tiempo real que demandan baja latencia entre el cliente y el servidor, como las aplicaciones de juegos en línea, los sistemas de monitoreo de señales que proceden de dispositivos o de información que cambia continuamente como las páginas de noticias o despliegue de marcadores de juegos, el protocolo de WS es el que actualmente ofrece mejores tiempos de respuesta para interactuar en un contexto de exigencia en la demanda de respuesta.

De acuerdo con los resultados obtenidos compartimos lo que establece Madam [2015], respecto a que la comunicación con WS presenta un protocolo adecuado para el entorno de IoT donde los paquetes de datos se transmiten de forma continua en múltiples dispositivos y genera un gran valor al estar basado en eventos. Siendo tecnología relativamente nueva, se entiende que no esté implementada totalmente en todos los navegadores. Chrome y Firefox son los que ofrecen las funcionalidades de este protocolo por completo. Por último, producto de la investigación documental, recabamos algunas referencias que incluyen ejemplos que proporcionan un panorama sobre la utilidad de este protocolo: www.achex.ca/rpg, www.achex.ca/games/chess, www.websocket.org/echo.html y demos.kaazing.com/portafolio-web/index.html.

Por otra parte, mediante la realización de las pruebas, se encontró que JMeter es una herramienta que efectivamente es útil para identificar y evaluar características de rendimiento de diferentes componentes basados en protocolos http y

websockets, con la excepción de que no aplica cuando la conexión es mediante un proxy.

En trabajos futuros es deseable ampliar la aplicación de chat, para incluir un protocolo eficiente para el tratamiento de mensajes asíncronos utilizando tecnologías como JMS (Java Service Messages) o incluso contemplar un mecanismo mediador de mensajes como ActiveMQ y adicionalmente considerar MongoDB o algún otro motor de bases de datos noSQL para optimizar el acceso a datos.

5. Bibliografía y Referencias

[1] Cantelon, M., Harter, M., Holowaychuk, T. J. & Rajlich, N. (2014). Node. js in Action. E.U.A.: Manning Publications Co.

[2] Clements, D. M. (2014). Node Cookbook (2a Ed.). Birmingham, U.K.: Packt Publishing Ltd.

[3] Rai, R. (2013). Socket. IO Real-time Web Application Development. Birmingham, U.K.: Packt Publishing Ltd.

[4] Eldritch, M. (2017, 9 marzo). Escribiendo aplicaciones con WebSockets: https://goo.gl/nXf4ym.

[5] Hanson, J. (2014). What is HTTP Long Polling? De PubNub: https://www.pubnub.com/blog/2014-12-01-http-long-polling/

[6] Kaazing Co. (2017). Demos: https://kaazing.com/demos/.

[7] Madan, D. (2015, April 17). Unleashing the power of HTML5 WebSocket for Internet of Things. De HCL Blogs: https://goo.gl/bfNoZm.

[8] Sommerville, I. (2011). Ingeniería de Software (9ª Ed.). United States: Pearson Education, Inc.

[9] Tikhanski, D. (2016). WebSocket Testing With Apache JMeter: https://www.blazemeter.com/blog/websocket-testing-apache-jmeter.

[10] Ubl, M. & Kitamura, E. (2010). Introducción a los WebSockets: incorporación de sockets a la Web: https://www.html5rocks.com/es/tutorials/websockets/basics/.