ISSN:0975-9646

Improving the Transmission Performance of Fragmented Messages in Websocket

Raphael O Anumba
CTT Research Lab, 29626 Teasedale Place, Castaic, CA 91384, USA

***Abstract*— Websocket protocol is inherently based on data frame transmission, when fragmented can adversely affect the rate of data transmission. We have analyzed the frame structure of the Websocket protocol and provided a greedy approach of improving the transmission speed without modifying the underling frame structure. This approach will drastically improve the transmission of very large data, particularly music and video.**

***Keywords*- Websocket, Fragmentation, Performance.**

I. INTRODUCTION

Websocket was the first major HTTP upgrades ushered by the introduction of HTML5. The underlining WebSocket protocol [1] enables message exchange between clients and servers on top of a persistent TCP connection. The WebSocket detail created as a major aspect of the HTML5 activity, presented the WebSocket JavaScript interface, which characterizes a full-duplex single attachment association over which messages can be sent between the client and server. The WebSocket standard rearranges a great part of the unpredictability around bi-directional web correspondence and association administration.

WebSocket is intended to be actualized in client web programs and web servers, yet it can be utilized by any client or server application. The WebSocket protocol is a free TCP-built convention. It's association to HTTP is that its handshake is deciphered by HTTP servers as an Upgrade [2]. The WebSocket convention makes more communication between a client and a server conceivable, encouraging live substance and the production of ongoing amusements. This is made conceivable by giving an institutionalized path to the server to send messages to the a client program without being requested by the client, and taking into account messages to be sent forward and backward while keeping the communication open.

Along these lines a two-way (bi-directional) progressing discussion can occur between an application and the server. The correspondences are done over TCP port number 80, which is of advantage for those situations which square non-web internet associations utilizing a firewall. Comparable two-way program server correspondences have been accomplished in non-institutionalized ways utilizing stopgap advances, for example, Comet.

II. WEBSOCKET FRAME

After a successful handshake between the client and server will start transmitting messages of which at the protocol level are transmitted as contiguous frames. Frames can be categorized based on the value of the opcode. The major categories are non-control and control frames (Table 1). Non-control frames are mainly data frames (Text and Binary data) and control frames (continuation, close, ping and pong frames).

	HEX	Opcode		
Continuation Frame	0	0000		Non-Control Frames
Text Frame	1	0001		
Binary Frame	2	0010		
Reserved for further non-control frames	3	0011		
	4	0100		
	5	0101		
	6	0110		
	7	0111		
Close Frame	8	1000		Control Frames
Ping Frame	9	1001		
Pong Frame	A	1010		
Reserved for further control frames	B	1011		
	C	1100		
	D	1101		
	E	1110		
	F	1111		

Table 1- Categories of Opcode

A. Control Frames

The Control frames are recognized by opcodes where the most critical piece of the opcode is 1. As of now characterized opcodes for control frames incorporate 0x8 (Close), 0x9 (Ping), and 0xA (Pong). Opcodes 0xB-0xF are saved for further control frames yet to be characterized.

Control frames are utilized to convey state about the WebSocket. Control frames can be contributed amidst a fragmented message. All control frames must have a payload length of 125 bytes or less and must not be fragmented.

B. Data Frame

The information can be fragmented into numerous frames. The primary frame that transmits the information has an opcode on it that demonstrates what kind of information is being transmitted [3]. This is just truly important in light of the fact that JavaScript has essentially non-existing backing for binary data at the time the detail was begun.

Payload information can be part up into numerous individual frames. The less than desirable end should buffer them up until the fin is set. So one can transmit the string

Hello World in 11 bundles of 6 (header length) + 1 byte each if that is what floats. Discontinuity is not taken into consideration control bundles. However the particular needs to have the capacity to handle interleaved control frames.

Fragmentation works in this way that a WebSocket message may be part into numerous WebSocket frames - furthermore combine at whatever time by the sender, as well as any intermediaries while in transit to the receiver. The last WebSocket edge of a succession of edges for a given message will have the FIN bit set.

The rationale for joining frames is generally in this form: get first frame, recollect opcode, link outline payload together until the fin bit is set. Affirm that the opcode for every bundle is zero. The explanation behind fragmentation is probably that one can create odds and ends of data on the server and send them to the client to buffer up rather than the server and the other way around. It's hard to handle on both client and server yet makes it less demanding for a JavaScript software engineer to handle [5]. The messy bits of the transport protocol are totally concealed away. Since the server's local convention was totally in view of the idea of streaming JSON there is zero quality in the messages. It is a typical thing for individuals adding Websocket backing to servers that beforehand placed some other convention on top of TCP.

III. FRAGMENTATION

A. *Rational behind fragmentation*

The main role of fragmentation is to permit communicating specific message that is of obscure scope in term of size when transmission is done without buffering. In the event that a message couldn't be divided, then an endpoint would need to buffer the whole message so its distance could be numbered before the first byte is sent [4]. With discontinuity, a server or middle person may pick a sensible size buffer and, when the support is complete, compose a section to the system.

A subordinate use-case for fragmentation is for multiplexing, where it is not alluring for an extensive message on one consistent network to corner the yield guide, so the multiplexing should be allowed to break the message into smaller fragments to better share the output channel [1].

Unless determined generally by an expansion, frames have no semantic importance. A mediator may mix and/or split frames, if no augmentations were arranged by the client and the server or if a few expansions were arranged, yet the delegate saw every one of the expansions arranged and knows how to blend and/or split edges in the vicinity of these expansions. One implications of this is without augmentations, senders and receivers must not rely on upon the vicinity of particular frame limits.

B. *Use Cases*

WebSocket is intended to be executed in web programs and web servers; however it can be utilized by any client or server application. The WebSocket Protocol is a free TCP-based protocol. The interchanges are done over TCP port number 80, which is of advantage for those situations which square non-web Internet associations utilizing a firewall. The main role of fragmentation is to permit communicating something specific that is of obscure size when the message is begun without buffering that message. On the off chance that messages couldn't be divided, then an endpoint would need to support the whole message so its length could be checked before the first byte is sent. A second use-case for discontinuity is for multiplexing, where it is not alluring for an extensive message on one coherent channel to corner the yield channel, so the multiplexing should be allowed to break the message into small fragments to better share the output channel.

IV. EXPERIMENT

The transmission of very large messages over websocket channel tends to pose some problem in terms of speed of transmission, particularly when the transmission calls for fragmentation whenever duplex transmission kicks in. In order to isolate this bottleneck we studied and analysed the whole message transmission from one endpoint to another.

This endpoint to endpoint transmission has been divided into three phases: the transmitter building phase, the over the wire phase and the receiver build phase.

The two building phases are similar and measurable within some acceptable margin of error. However, the over the wire transmission measurement will drastically vary from media to media depending on the maximum transfer unit (MTU) of the underlining protocol [6].

We found out that most websocket transmitters (Server and client) send their messages at a fixed frame sizes which are usually at small frame sizes, which result in transmitting so many frames and consequently taking much time.

We were able to improve on this bottleneck by using what we refer to as progressive frame size as transmission proceeds.

A. *Setup*

The total transfer time is:

Average Total transmission time (ATTT)	=	Server Frame build time (SFBT)	+	Over the wire transmission time (OTWTT)	+	Client Frame build time (CFBT)

$$ATTT = SFBT + OTWTT + CFBT$$

$$ATTT = \frac{\sum_{i=1}^{N} Ft_i + It_i}{N} + \frac{\sum_{i=1}^{N} Wt_i + Sd_i}{N} + \frac{\sum_{i=1}^{N} Ft_i + Ot_i}{N}$$

Where *N* is the number frame transmitted,

Ft is a frame build time,

Wt is over the wire transmission time,

It, Sd and Ot are respectively input stream perturbation time; over the wire system dependent time and client output perturbation time.

Our experiment has only considered the performance of *SFBT*, which is completely similar to *CFBT*. We left out *OTWTT* because of the unpredictable and uncontrollable dependency on many external systems.

For only *SFBT*, we considered various message sizes ranging from 128MB to around 132GB. We measured the frame build time, in Nano-seconds for various frame size ranging from 8,192KB to 131,072KB. Figure-1 depicts the performance of various frame sizes as the message sizes increase.

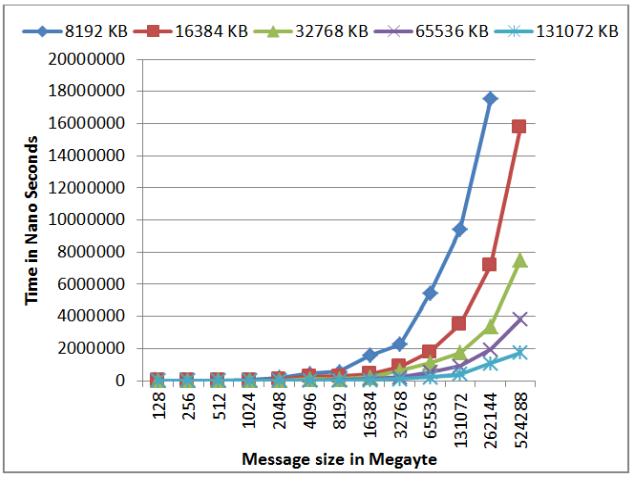

Figure-1: The Rate of transmission of various data size and message sizes

The result of the same experiment was transposed in Figure-2 to present additional smaller frame sizes 1,024KB, 2,948KB and 4,096KB. This shows the performance of message transmission in Nano seconds, against the various frame sizes.

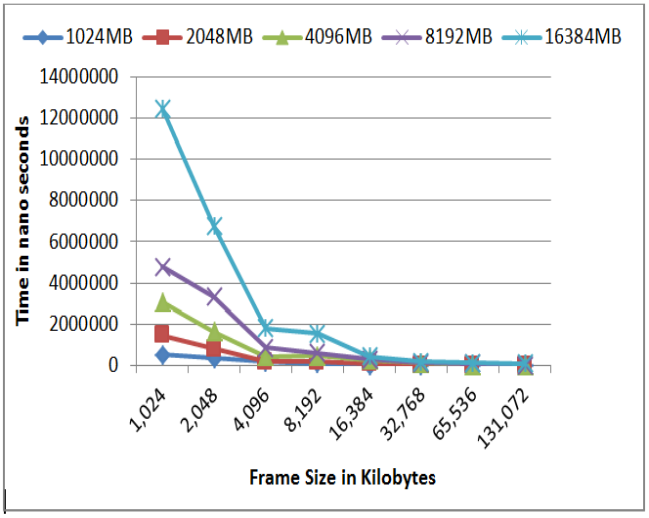

Figure-2: The Rate of transmission of various data size and frame sizes

B. Result

For the fact that the size of a message was not known at transmission time, we devised a method of increasing the frame sizes (8,192 KB, 16,384 KB, 32,768 KB, 65,536 KB and 131,072 KB) as the transmission progresses. At each frame size transmission, instead of remaining at a fixed frame size, it will switch into the next higher frame size. Figure-3 shows the drastic reduction in transmission time. Moreover, there were no much deviation in the performance at all frame sizes.

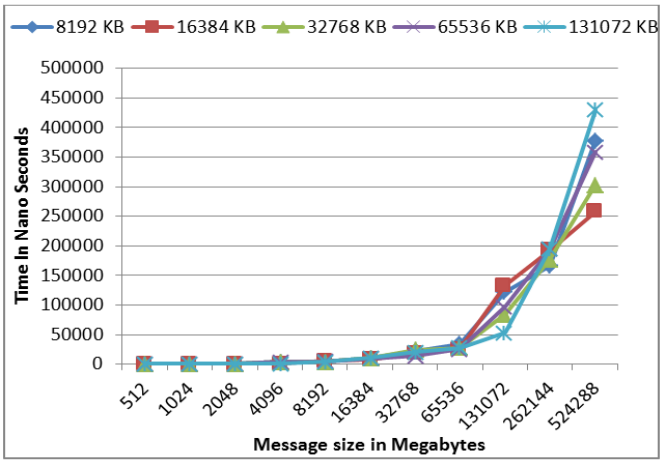

Figure-3: The Rate of transmission of various data size and progressive frame sizes

V. CONCLUSIONS

A study, design and implantation of a drastic improvement in frame based websocket message transmission was presented. Instead of transmission on a fixed frame size, a progressive increase in frame size was adopted. This has surprisingly reduced the transmission time.

We hope that our finding will reduce the transmission time over websocket duplex channel and assist in enhancing the quality of compressed data, music and video transmission over the internet.

REFERENCES

[1] Fette, I. and A. Melnikov, *The WebSocket Protocol*, RFC6455, December 2011.

[2] Erkkilä, J. P., 2012, *Websocket security analysis*, Aalto University School of Science, 2-3

[3] Gackenheimer, C., 2013, *Creating a WebSocket Server*. In Node. js Recipes (pp. 191-220), Apress.

[4] Simonsen, K. I. F., & Kristensen, L. M., 2014, *Implementing the WebSocket Protocol Based on Formal Modelling and Automated Code Generation*. In Distributed Applications and Interoperable Systems (pp. 104-118), Springer Berlin Heidelberg

[5] Wang, V., Salim, F., & Moskovits, P., 2013, *The WebSocket Protocol*. In The Definitive Guide to HTML5 WebSocket (pp. 33-60), Apress

[6] Murray, David; Terry Koziniec; Kevin Lee; Michael Dixon (2012). *Large MTUs and internet performance*. 13th IEEE Conference on High Performance Switching and Routing (HPSR 2012).