On the Use of Websockets to Maintain Temporal States in Stateless Applications

Josefa Gómez, Abdelhamid Tayebi, Juan Casado
Computer Science Department,
University of Alcalá
Alcalá de Henares, Spain
email: josefa.gomezp@uah.es, hamid.tayebi@uah.es, juan.casado@edu.uah.es

***Abstract*—This paper studies the use of Websockets to maintain temporal states in stateless applications. Concretely, it is used in a web-based application that calculates the propagation loss in outdoor environments. The reasons why Websockets are used and their limitations are discussed. A comparison with other similar technologies is also included.**

Keywords - web; WebSockets; communication protocols; real time web interfaces; stateless applications.

I. INTRODUCTION

Current web applications require fast communication between the server and the clients to produce close to real time updates on the web interface. If feedback is not received about the progress of the computations performed by the server, the user experience breaks apart. In [1], it is discussed that web page loading time increases user frustration and discomfort while browsing the web. This also is applicable to the waiting time between a user action and the webpage displaying the requested information.

The computational model that we will describe solves the waiting time problem for certain types of large computations. Once this computational model is described, REpresentational State Transfer (REST) and WebSockets will be analyzed, theoretically and experimentally, as the possible communication components to solve it.

Certain types of large computations need to hold a state in order to be performed or optimized. An algorithm that sorts a list of elements or looks for the shortest path on a graph needs those data structures as state. An algorithm that calculates propagation can be optimized by holding certain information as state and reuse it like if the state where a cache. But this state is temporal, once the computation has ended it is no longer needed to be held in the server.

In order to solve the waiting time problem, while performing a computation, the user must be informed of the progress of that computation. Ideally, by showing the current computation progress with as much detail as if it were the final result. The computation will be performed in the server and the representation update will be performed in the client, therefore, both processes will be concurrent. With this, solving the waiting time problem is transformed into finding the best way to hold that temporal state while the server informs the client about the progress of the computation.

Nowadays, lots of web applications and web services are based on the REST architectural style. REST has become very popular due to its simplicity and the fact that it builds upon the HyperText Transfer Protocol (HTTP), so developers are familiar with it. However, the REST architectural style has some disadvantages such as the lack of saving the stateful information between request-response cycles. In addition, there is no mechanism to send push notifications from the server to the client (to the web browser). This implies that it is hard to implement any type of services where the server updates the client without the use of client-side polling of the server or some other type of web hook. Consequently, every REST-based application is stateless, any state management tasks must be performed or initiated by the client.

Due to the fact that HTTP is half duplex, the server is not able to initiate a data transmission to a client as long as it has not been specifically asked for. Until now, web applications that needed bidirectional communication required an abuse of HTTP to poll the server for updates while sending upstream notifications as distinct HTTP calls. The disadvantages of using REST to maintain temporal states are mainly three. First, the server must use several TCP connections for each client: one for the client to send an initial request to the server where the operation to perform is described and a new one for each message that contains the progress or fraction of performed work. Second, the wire protocol has a high overhead, with each client-to-server message having an HTTP header. And third, the server is forced to maintain a mapping from the outgoing connections to the initial computation request to track which progress information must be sent to each client.

Combined, these disadvantages make REST hardly usable to perform computations that need temporal state on the server replicated context. In this context, multiple instances of the same server would run as separated process on the same or different machines, without one instance knowing the existence of the others. When a request is made it is handled by a load balancer that redirects it to one of the instances. Different connections from the same client may be answered by different instances of the server. Due to that, the server that received the initial computation request needs to share information with the other instances. This not only interfere with the idea of the replicated context, instances should not need to communicate, but also creates new synchronization problems. Some solutions make the client manage the temporal state and others make use of databases, both of which are slow and create an extra layer of complexity.

WebSockets [2]-[4] is a new protocol that provides a solution to overcome the aforementioned limitations. WebSockets uses a single TCP connection for traffic in both directions that allows a bidirectional, full-duplex, persistent socket connection between a web page and a remote server. Moreover, it is supported by all major web browsers.

Based on the two-way communication connection, the server can receive and process data, and can also send data back to the browser. Also, communication is more efficient than using HTTP if we focus on the size of the message and on the speed, especially for large messages, since in HTTP, for example, you have to send the headers in each request. This adds bytes.

According to the benchmarking done in [5] to compare the performance of HTTP vs WebSockets, the latter can be 50% faster than HTTP. This means that in many cases and depending on the needs of the project, WebSockets can be faster than traditional HTTP APIs. However, WebSockets is not the solution to all problems, other protocols perform certain tasks better than WebSockets does. In Section II, a comparison of several protocols vs. tasks is presented. A practical application of WebSockets in a real web-based simulation tool is described in Section III. Finally, conclusions are presented in Section IV.

II. EXPERIMENTAL RESULTS

To empirically test the performance differences between WebSockets and REST communication protocols, a demonstrative application was built. The design of this application aims to make the comparison as fair as possible, to let us inspect the strengths of both protocols.

If the application were to be built in the server replicated context, REST would have had major disadvantages. It would have been required to use a database to hold the temporal state or to hold it on the client and send it back and forth to the server in each request.

Therefore, the application will not be tested on this context. To hold the temporal state on the REST protocol, a random key is generated for each client. This key is provided to them as a response to their initial computation request. By providing this key, on each following request, the server will be able to know which temporal state belongs to each connection that requests a progress update on a computation. On the other hand, WebSockets will hold its temporal state on the single TCP connection that is created between the client and the server.

The developed demonstration takes as input a JSON document, which describes the computation to perform. Computations are a tree of actions, if one action is the child of another it will be performed after its father. On the other hand, if one action is in the same tree level of another they will be computed concurrently. Each computation has two steps, first a list of data is created according to the configuration on the JSON file and then the list is sorted with a certain criteria and sorting algorithm. The computations are performed on the server, while the client displays a graphical representation of the current position of the elements on the list. Multiple representations are available.

The server and the client communicate with REST or WebSockets allowing us to catch and dump the communication traces and inspect the transmitted packages with applications like Tcpdump or WhireShark.

Performing the same computations, the data shown in Table I have been collected with REST and WebSockets as communication protocols.

TABLE I. COMPARISON UNDER NOMINAL LOAD

	REST	WebSockets
Packets	135.098	44.976
Transmitted data	27.634.885 bytes	4.723.849 bytes
Communications	22.481	22.480
Mean Time	86s 778ms	35s 226ms

For a single computation, 22.479 intermediate results have been sent from the server to the client to perform a close to real time graphical representation of its state. In order to archive that using WebSockets, an extra communication was needed to send the computation description to the server. In the case of REST, two extra communication where needed. One communication was used to send the computation to the server and another to inform the client that the computation had ended. Using WebSockets the client can be informed of the finalization of the computation by the server closing the channel.

In average, using WebSockets, there are two packets exchange between the client and the server, which leaves a total of 16 packets for stablishing the connection (8 packets) and closing it (8 packets). With REST, an average of six packets are exchanged between the server and the client per communication. This quantity should have been eight packets, but the libraries used try to reuse TCP connections by not always sending 'FIN' packets in order to save resources. Only 36 connections where closed in average along this computation. Additionally, REST performs Cross-Origin Resource Sharing (CORS) checks from time to time using a HTTP OPTIONS request. For this computation an average of 17 CORS checks were made.

REST not only uses around three times more packets along the computation, those packets are also almost six times heavier than WebSockets ones. This is because they need to carry a longer header. WebSockets is able to use a smaller header because the header is sent once, when the connection is stablished, and since the connection is never closed there is no need to resend it on every data transmission along that same connection.

The benefits of WebSockets not only can be seen on the amounts of data and packets transmitted but also in the time taken. Using REST, the communication will take more than double the time than with WebSockets.

Performing the smallest computation that the demo program is able to make, the following statistics have been recorded. This computation requires two updates of the user interface and which means a total of three and four communications for WebSockets and REST respectively.

With this second comparison, an improvement of the REST performance against WebSockets is expected. The theory around these protocols suggests that WebSockets should take longer to initiate the communication channel but, in the long run, with multiple data transmissions between the server and the client, it should surpass REST efficiency as seen on the first comparison.

TABLE II. COMPARISON UNDER MINIMUM LOAD

	REST	WebSockets
Packets	44	22
Transmitted data	7.127 bytes	2.489 bytes
Communications	4	3
Mean Time	0s 048ms	0s 015ms

Nevertheless, this expectation has not been backed up by the experimentally recorded data (see Table II). REST has improved its efficiency in the transmitted packets department but has not in the transmitted data and mean time ones. Examples of the traces used to calculate these statistics are included on Figure 1 for REST and on Figure 2 for WebSockets. The reason for this difference is that REST expends almost the same resources as WebSockets does for the whole communication just for the CORS request (blue section of Figure 1). Looking just at the time column of one single REST communication (one green section of Figure 1 and the proportional part of the red section) it can clearly be seen that is more efficient than the same communication on WebSockets protocol (blue section, red section and one green section of Figure 2).

So, if the CORS request is not taken into account, the expectations about REST are met. REST is a more efficient protocol for single sporadic data transmissions than WebSockets. On the other hand, WebSockets is more efficient for multiple communications even if their number is small.

III. PRACTICAL USE OF WEBSOCKETS IN A WEB-BASED APPLICATION

WebSockets is a stateful protocol, while HTTP connections are stateless. This means that WebSockets creates a connection that is kept alive on the server until the socket is closed and messages are exchanged bidirectionally. This particular feature is very useful to overcome three frequent problems that arise in the use and development of a web-based simulation tool like the one presented in [6] by the authors:

• It is desirable to display a progress bar in order to inform the users about the state of the calculations performed to provide the propagation loss. If the progress bar is not displayed, the users do not know how long it will take to complete the requested task.

• It is desirable to obtain partial results of the request made while it is being completed without interrupting this process. This combined with a progress bar not only informs about the lasting time, but also lets visualize earlier some of the requested information. Additionally, it entertains the users creating better user experiences.

• On the context of replicated servers, which is the case of our application, it is desirable to communicate always with the same server. At least during a computation. This allows to perform better optimizations without adding extra layers of complexity like databases or other ways of sharing information between the server instances. Avoiding this complexity is not only desirable from a design point of view. It also makes the application cheaper, no data is saved on disk, wasted computation time is minimal and no more than the necessary data is sent to the web.

In addition, authentication is also simplified by using Websockets. When using WebSocket, authentication is performed when the connection is established, so future requests under the same channel do not need to be authenticated again. This method greatly simplifies the authentication process. Therefore, Websockets improves the security of the system because there is no need of passing user credential in every request.

A web-based simulation [6] has been developed by the authors. This application is able to predict propagation losses in urban and rural environments by applying a semi-empirical algorithm. Now, the authors are improving that simulation tool. Deterministic methods are being included. These methods provide results more accurate but they have the disadvantage of consuming lots of resources (time and memory), so Websockets are very useful to inform the client about the state of the computations that are carried out in the server.

IV. CONCLUSIONS

WebSockets is a great protocol that solves three communication problems: 1) Sending multiple packets of data between the server and the client with a single communication negotiation required. 2) Creating a channel between a client and a server through which the client can receive notification from the server without polling. 3) Granting a stable connection between a client and a single instance of a replicated server that is behind a load balancer.

Those characteristics are exploitable to achieve close to real time updates on the progress and current state of long-lasting computations without major complications. While the computations are been performed, the progress or new calculated portions or approximations to the final solution are been sent to the client with a minimum performance loss and minimum design considerations. At the same time, the clients will be displaying fresh and updated information to the users with each packet received, creating a better user experience.

ACKNOWLEDGMENT

This work is supported by the program “Programa de Estímulo a la Investigación de Jóvenes Investigadores” of Vice rectorate for Research and Knowledge Transfer of the University of Alcala and by the Comunidad de Madrid (Spain) through project CM/JIN/2019-028.

REFERENCES

[1] P. Saed and Y. Yahya, "Loading time effects: A case study of Malaysian Examination Syndicate web portal," Proceedings of the 2011 International Conference on Electrical Engineering and Informatics, Bandung, 2011, pp. 1-5, doi: 10.1109/ICEEI.2011.6021664.

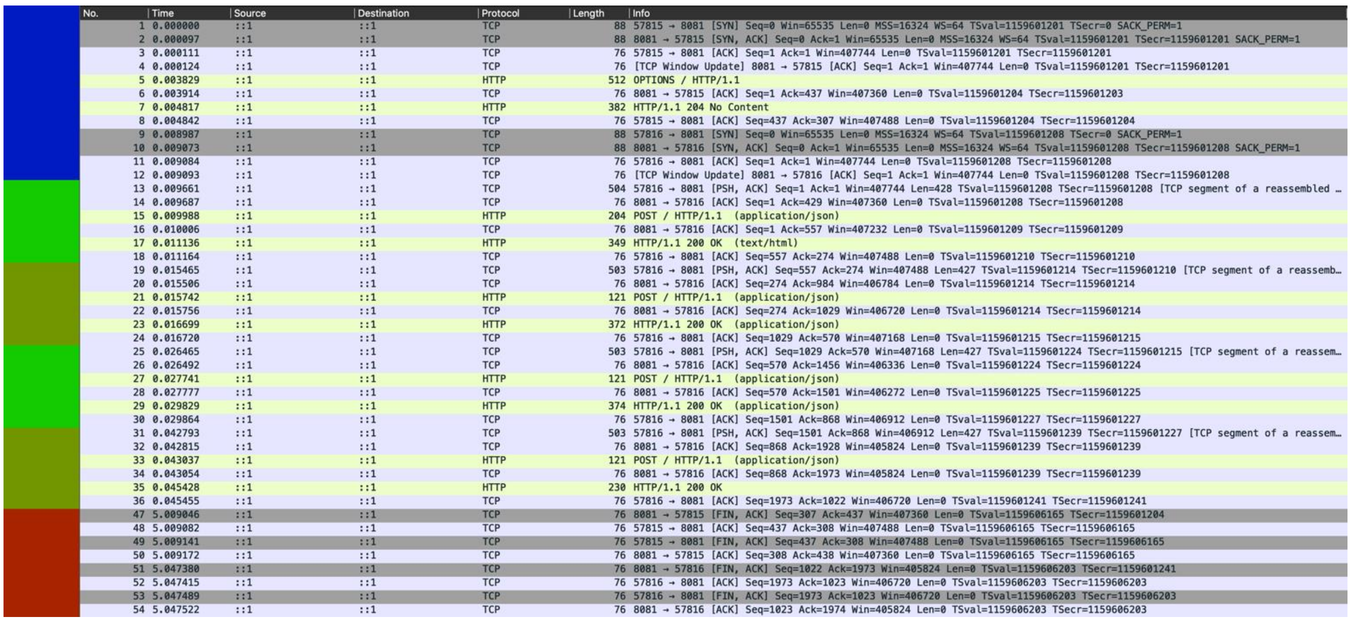
[2] I. Fette and A. Melnikov, The WebSocket Protocol, Internet Requests for Comments, RFC Editor, RFC 6455, World Wide Web Consortium, Cambridge, MA, USA, 2011, http://www.rfc-editor.org/rfc/rfc6455.txt [retrieved: September, 2020]

[3] B. Soewito, F. Christian, D. Gunawan, and I. Kusuma. “Websocket to Support Real Time Smart Home Applications”. Procedia Computer Science. 157. 2019. 560-566. 10.1016/j.procs.2019.09.014.

[4] Y. Wang, L. Huang, X. Liu, T. Sun, and K. Lei.. Performance Comparison and Evaluation of WebSocket Frameworks: Netty, Undertow, Vert.x, Grizzly and Jetty. 13-17. 2018. 10.1109/HOTICN.2018.8605989.

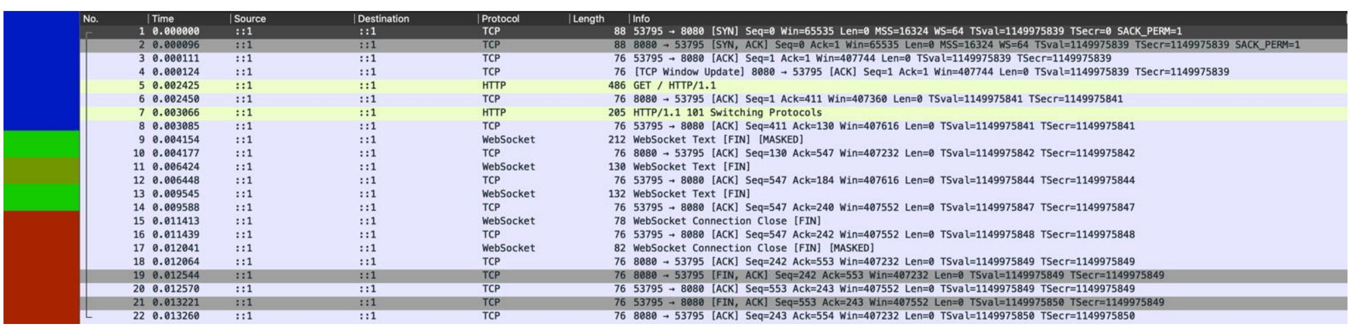
[5] HTTP vs Websockets: A performance comparison, available online at: https://blog.feathersjs.com/http-vs-websockets-a-performance-comparison-da2533f13a77[retrieved: September, 2020]

[6] A. Tayebi, J. Gomez, F. Saez de Adana, O. Gutierrez, and M. Fernandez de Sevilla, "Development of a Web-Based Simulation Tool to Estimate the Path Loss in Outdoor Environments using OpenStreetMaps [Wireless Corner]," IEEE Antennas and Propagation Magazine, vol. 61, no. 1, pp.123-129, Feb.2019.



No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000	::1	::1	TCP	88	57815 → 8081 [SYN] Seq=0 Win=65535 Len=0 MSS=16324 WS=64 TSval=1159601201 TSecr=0 SACK_PERM=1
2	0.000097	::1	::1	TCP	88	8081 → 57815 [SYN, ACK] Seq=0 Ack=1 Win=65535 Len=0 MSS=16324 WS=64 TSval=1159601201 TSecr=1159601201 SACK_PERM=1
3	0.000111	::1	::1	TCP	76	57815 → 8081 [ACK] Seq=1 Ack=1 Win=407744 Len=0 TSval=1159601201 TSecr=1159601201
4	0.000124	::1	::1	TCP	76	[TCP Window Update] 8081 → 57815 [ACK] Seq=1 Ack=1 Win=407744 Len=0 TSval=1159601201 TSecr=1159601201
5	0.003829	::1	::1	HTTP	512	OPTIONS / HTTP/1.1
6	0.003914	::1	::1	TCP	76	8081 → 57815 [ACK] Seq=1 Ack=437 Win=407360 Len=0 TSval=1159601204 TSecr=1159601203
7	0.004817	::1	::1	HTTP	382	HTTP/1.1 204 No Content
8	0.004842	::1	::1	TCP	76	57815 → 8081 [ACK] Seq=437 Ack=307 Win=407488 Len=0 TSval=1159601204 TSecr=1159601204
9	0.008987	::1	::1	TCP	88	57816 → 8081 [SYN] Seq=0 Win=65535 Len=0 MSS=16324 WS=64 TSval=1159601208 TSecr=0 SACK_PERM=1
10	0.009073	::1	::1	TCP	88	8081 → 57816 [SYN, ACK] Seq=0 Ack=1 Win=65535 Len=0 MSS=16324 WS=64 TSval=1159601208 TSecr=1159601208 SACK_PERM=1
11	0.009084	::1	::1	TCP	76	57816 → 8081 [ACK] Seq=1 Ack=1 Win=407744 Len=0 TSval=1159601208 TSecr=1159601208
12	0.009093	::1	::1	TCP	76	[TCP Window Update] 8081 → 57816 [ACK] Seq=1 Ack=1 Win=407744 Len=0 TSval=1159601208 TSecr=1159601208
13	0.009661	::1	::1	TCP	504	57816 → 8081 [PSH, ACK] Seq=1 Ack=1 Win=407744 Len=428 TSval=1159601208 TSecr=1159601208 [TCP segment of a reassembled …
14	0.009687	::1	::1	TCP	76	8081 → 57816 [ACK] Seq=1 Ack=429 Win=407360 Len=0 TSval=1159601208 TSecr=1159601208
15	0.009988	::1	::1	HTTP	204	POST / HTTP/1.1 (application/json)
16	0.010006	::1	::1	TCP	76	8081 → 57816 [ACK] Seq=1 Ack=557 Win=407232 Len=0 TSval=1159601209 TSecr=1159601209
17	0.011136	::1	::1	HTTP	349	HTTP/1.1 200 OK (text/html)
18	0.011164	::1	::1	TCP	76	57816 → 8081 [ACK] Seq=557 Ack=274 Win=407488 Len=0 TSval=1159601210 TSecr=1159601210
19	0.015465	::1	::1	TCP	503	57816 → 8081 [PSH, ACK] Seq=557 Ack=274 Win=407488 Len=427 TSval=1159601214 TSecr=1159601210 [TCP segment of a reassemb…
20	0.015506	::1	::1	TCP	76	8081 → 57816 [ACK] Seq=274 Ack=984 Win=406784 Len=0 TSval=1159601214 TSecr=1159601214
21	0.015742	::1	::1	HTTP	121	POST / HTTP/1.1 (application/json)
22	0.015756	::1	::1	TCP	76	8081 → 57816 [ACK] Seq=274 Ack=1029 Win=406720 Len=0 TSval=1159601214 TSecr=1159601214
23	0.016699	::1	::1	HTTP	372	HTTP/1.1 200 OK (application/json)
24	0.016720	::1	::1	TCP	76	57816 → 8081 [ACK] Seq=1029 Ack=570 Win=407168 Len=0 TSval=1159601215 TSecr=1159601215
25	0.026465	::1	::1	TCP	503	57816 → 8081 [PSH, ACK] Seq=1029 Ack=570 Win=407168 Len=427 TSval=1159601224 TSecr=1159601215 [TCP segment of a reassem…
26	0.026492	::1	::1	TCP	76	8081 → 57816 [ACK] Seq=570 Ack=1456 Win=406336 Len=0 TSval=1159601224 TSecr=1159601224
27	0.027741	::1	::1	HTTP	121	POST / HTTP/1.1 (application/json)
28	0.027777	::1	::1	TCP	76	8081 → 57816 [ACK] Seq=570 Ack=1501 Win=406272 Len=0 TSval=1159601225 TSecr=1159601225
29	0.029829	::1	::1	HTTP	374	HTTP/1.1 200 OK (application/json)
30	0.029864	::1	::1	TCP	76	57816 → 8081 [ACK] Seq=1501 Ack=868 Win=406912 Len=0 TSval=1159601227 TSecr=1159601227
31	0.042793	::1	::1	TCP	503	57816 → 8081 [PSH, ACK] Seq=1501 Ack=868 Win=406912 Len=427 TSval=1159601239 TSecr=1159601227 [TCP segment of a reassem…
32	0.042815	::1	::1	TCP	76	8081 → 57816 [ACK] Seq=868 Ack=1928 Win=405824 Len=0 TSval=1159601239 TSecr=1159601239
33	0.043037	::1	::1	HTTP	121	POST / HTTP/1.1 (application/json)
34	0.043054	::1	::1	TCP	76	8081 → 57816 [ACK] Seq=868 Ack=1973 Win=405824 Len=0 TSval=1159601239 TSecr=1159601239
35	0.045428	::1	::1	HTTP	230	HTTP/1.1 200 OK
36	0.045455	::1	::1	TCP	76	57816 → 8081 [ACK] Seq=1973 Ack=1022 Win=406720 Len=0 TSval=1159601241 TSecr=1159601241
47	5.009046	::1	::1	TCP	76	8081 → 57815 [FIN, ACK] Seq=307 Ack=437 Win=407360 Len=0 TSval=1159606165 TSecr=1159601204
48	5.009082	::1	::1	TCP	76	57815 → 8081 [ACK] Seq=437 Ack=308 Win=407488 Len=0 TSval=1159606165 TSecr=1159606165
49	5.009141	::1	::1	TCP	76	57815 → 8081 [FIN, ACK] Seq=437 Ack=308 Win=407488 Len=0 TSval=1159606165 TSecr=1159606165
50	5.009172	::1	::1	TCP	76	8081 → 57815 [ACK] Seq=308 Ack=438 Win=407360 Len=0 TSval=1159606165 TSecr=1159606165
51	5.047380	::1	::1	TCP	76	8081 → 57816 [FIN, ACK] Seq=1022 Ack=1973 Win=405824 Len=0 TSval=1159606203 TSecr=1159601241
52	5.047415	::1	::1	TCP	76	57816 → 8081 [ACK] Seq=1973 Ack=1023 Win=406720 Len=0 TSval=1159606203 TSecr=1159606203
53	5.047489	::1	::1	TCP	76	57816 → 8081 [FIN, ACK] Seq=1973 Ack=1023 Win=406720 Len=0 TSval=1159606203 TSecr=1159606203
54	5.047522	::1	::1	TCP	76	8081 → 57816 [ACK] Seq=1023 Ack=1974 Win=405824 Len=0 TSval=1159606203 TSecr=1159606203

Figure 1. REST packet trace.



No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000	::1	::1	TCP	88	53795 → 8080 [SYN] Seq=0 Win=65535 Len=0 MSS=16324 WS=64 TSval=1149975839 TSecr=0 SACK_PERM=1
2	0.000096	::1	::1	TCP	88	8080 → 53795 [SYN, ACK] Seq=0 Ack=1 Win=65535 Len=0 MSS=16324 WS=64 TSval=1149975839 TSecr=1149975839 SACK_PERM=1
3	0.000111	::1	::1	TCP	76	53795 → 8080 [ACK] Seq=1 Ack=1 Win=407744 Len=0 TSval=1149975839 TSecr=1149975839
4	0.000124	::1	::1	TCP	76	[TCP Window Update] 8080 → 53795 [ACK] Seq=1 Ack=1 Win=407744 Len=0 TSval=1149975839 TSecr=1149975839
5	0.002425	::1	::1	HTTP	486	GET / HTTP/1.1
6	0.002450	::1	::1	TCP	76	8080 → 53795 [ACK] Seq=1 Ack=411 Win=407360 Len=0 TSval=1149975841 TSecr=1149975841
7	0.003066	::1	::1	HTTP	205	HTTP/1.1 101 Switching Protocols
8	0.003085	::1	::1	TCP	76	53795 → 8080 [ACK] Seq=411 Ack=130 Win=407616 Len=0 TSval=1149975841 TSecr=1149975841
9	0.004154	::1	::1	WebSocket	212	WebSocket Text [FIN] [MASKED]
10	0.004177	::1	::1	TCP	76	8080 → 53795 [ACK] Seq=130 Ack=547 Win=407232 Len=0 TSval=1149975842 TSecr=1149975842
11	0.006424	::1	::1	WebSocket	130	WebSocket Text [FIN]
12	0.006448	::1	::1	TCP	76	53795 → 8080 [ACK] Seq=547 Ack=184 Win=407616 Len=0 TSval=1149975844 TSecr=1149975844
13	0.009545	::1	::1	WebSocket	132	WebSocket Text [FIN]
14	0.009588	::1	::1	TCP	76	53795 → 8080 [ACK] Seq=547 Ack=240 Win=407552 Len=0 TSval=1149975847 TSecr=1149975847
15	0.011413	::1	::1	WebSocket	78	WebSocket Connection Close [FIN]
16	0.011439	::1	::1	TCP	76	53795 → 8080 [ACK] Seq=547 Ack=242 Win=407552 Len=0 TSval=1149975848 TSecr=1149975848
17	0.012041	::1	::1	WebSocket	82	WebSocket Connection Close [FIN] [MASKED]
18	0.012064	::1	::1	TCP	76	8080 → 53795 [ACK] Seq=242 Ack=553 Win=407232 Len=0 TSval=1149975849 TSecr=1149975849
19	0.012544	::1	::1	TCP	76	8080 → 53795 [FIN, ACK] Seq=242 Ack=553 Win=407232 Len=0 TSval=1149975849 TSecr=1149975849
20	0.012570	::1	::1	TCP	76	53795 → 8080 [ACK] Seq=553 Ack=243 Win=407552 Len=0 TSval=1149975849 TSecr=1149975849
21	0.013221	::1	::1	TCP	76	53795 → 8080 [FIN, ACK] Seq=553 Ack=243 Win=407552 Len=0 TSval=1149975850 TSecr=1149975849
22	0.013260	::1	::1	TCP	76	8080 → 53795 [ACK] Seq=243 Ack=554 Win=407232 Len=0 TSval=1149975850 TSecr=1149975850

Figure 2. WebSockets packet trace.