Advances in Computer Science Research, volume 44
3rd International Conference on Wireless Communication and Sensor Network (WCSN 2016)

The Design and Implementation of CoAP Over WebSocket Proxy

Zhong-Yan Yuan, Geng-Yu Wei
School of Computer and Science, BUPT, Beijing, China
E-mail: yuanzhongyan@bupt.edu.cn, weigengyu@bupt.edu.cn

***Abstract*- CoAP is designed as an application layer protocol for IoT applications by IETF CoRE WG, and CoAP over WebSocket is currently of interest to researchers. This paper analyzes the features and defect of the HTTP/CoAP proxy proposed in RFC 7252, and explains the CoAP over WebSocket proxy as well as its advantage over the HTTP/CoAP proxy. Then a design and implementation of the WebSocket proxy based on Californium open-source framework is given. Performance tests and experiments results show that the WebSocket proxy has some significant advantages over HTTP/CoAP proxy in term of response time upon high concurrency requests.**

Keywords-internet of things; CoAP; restful; network proxy; WebSocket

I. INTRODUCTION

The Internet of Things (IoT) is envisioned as a global network of billions of smart devices. International Telecommunications Union (ITU) proposed the concept of IoT in 2005, and IoT has been high on research agenda for more than ten years. It is predicted that there will be millions of billions of IoT nodes around us by 2020 [1]. Cloud computing and big data analysis become very hot topics in recent years, and the Cloudification of IoT is sure to bring IoT application to a completely new stage. However, application and service strategies are still to be standardized.

Hypertext Transfer Protocol (HTTP), the protocol that runs on the World Wide Web, is supporting a myriad of Web applications. The main idea of HTTP protocol is Representational State Transfer (REST) architecture [2]: every object on the internet is viewed as a resource; every resource has its unique identifier; standard methods are defined for accessing the resources; all the methods are stateless. While HTTP performs well enough on the Web, it is too complex and computationally expensive for constrained environments where most IoT nodes lie. In order to provide RESTful operations for low-cost devices and IoT scenarios, IETF CoRE working group set out to design the Constrained Application Protocol (CoAP) [3]. Although CoAP protocol inherits a lot from HTTP protocol, there are two fundamental differences between them. First, CoAP is designed to naturally support duplex communication while HTTP is always client-initiate. Second, CoAP protocol uses a compact binary format and runs over UDP (or DTLS when security is required) which reduces package size while HTTP runs over TCP.

The intention of CoAP design is to make the constrained network an extension of World Wide Web. In the IoT vision [4], all kinds of sensors and actuators will integrate into the current Internet, allowing users to access the resources the way they browse the Web. Furthermore, Web technology can enable physical mashups for the IoT, that is, to combine services of different devices that belong to different application domains. It calls for a proxy to work between the Web that runs HTTP and the constrained network that runs CoAP. CoRE working group keep concerned with the design and functionality of the proxy, and they suggest in RFC 7252 that the proxy directly translate HTTP messages to CoAP messages field-by-field, and vice versa, as illustrated in Fig. 1. Such proxy is referred to as HTTP/CoAP proxy. However, the use of such "translating" proxies raises potential limitation when preserving CoAP protocol's features in HTTP. HTML5 WebSocket protocol [5] allows full-duplex communication between Servers and Clients, which exactly matches CoAP design. Therefore, in this paper we design and implement a CoAP over WebSocket proxy [6], which means using a WebSocket message to transport a CoAP message, overcoming defects of the "translating" proxy.

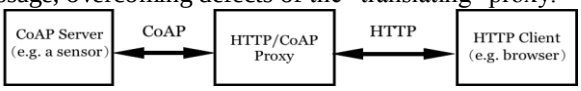

Figure 1. "Translating" proxy (HTTP/CoAP proxy).

The rest of paper is organized as follows: In section 2 the features of WebSocket proxy is introduced and discussed; in Section 3 the design and implementation of Websocket are demonstrated; in Section 4 the performance test and results are shown and discussed; and finally Section 5 concludes this work.

II. COAP OVER WEBSOCKET PROXY

A. CoAP protocol

CoAP protocol is a network protocol orienting networks and nodes that are resource-constrained. Resource-constrained nodes usually have only an 8-bit processor and a small ROM/RAM size, and they are battery-powered. Sensors are the most common examples. Resource constrained networks are those whose physical links suffer from high package loss and low throughput, such as 6LoWPAN [7].

CoAP protocol and HTTP [8] protocol share the main idea of resource abstraction, RESTful operations, and extensible headers (or options in CoAP). The RESTful architecture requires each CoAP resource to be attached to a Universal Resource Identifier (URI), a set of standard methods are defined to operate the resources, and that all these methods are stateless. Most frequently used request methods are GET, POST, PUT, and DELETE. When the

server receives a request, it will reply with respond code like 2.xx, 4.xx, 5.xx, and so on. Those codes have similar meaning to corresponding HTTP response codes.

CoAP can be logically divided into two sub-layers. The request/response layer enables RESTful interactions in accordance with the HTTP specifications. The messaging layer below acts as a thin control layer that provides duplicate deletion and reliable delivery of messages based on a simple stop-and-wait mechanism for retransmissions.

Tailored to the requirements of constrained environments, CoAP adopts a compact binary message format instead of the HTTP text message, and CoAP runs over UDP rather than TCP. What's more, CoAP provides several features that goes beyond HTTP 1.1, so that it can fit the IoT scenarios better. Examples are the resource observation [9], blockwise transfer of messages, group communications, alternative transports, and so on. The resource observation means the client can subscribe to a resource by building an observe relationship with the server, and then receive stage-change notifications pushed by the server. Blockwise transfer allows messages to be fragmented when it exceeds the maximum transmission unit (MTU) of the physical environment. Group communications and other features are not the main concern of this paper and will not be covered here.

Above discussions lead to the insight that although many fields of CoAP and HTTP can be directly mapped to each other, the fundamental difference between them can not be neglected: CoAP naturally supports two-way communication between the client and the server, while HTTP can only be initiated by the client and the server responds passively. This raises one obvious limitation that "translating" proxy may not preserve some CoAP features when translating a CoAP message into a HTTP message. A typical scenario is CoAP resource observing that the CoAP server is enabled to push the real-time state changes to all registered clients. In the case of HTTP/CoAP proxy, however, when the CoAP server finishes pushing notifications to the H/C proxy, the H/C proxy will have trouble delivering it to the registered client since it communicates with the client using HTTP but HTTP server does not have the ability to initiate a session. A remedy is to keep the HTTP client polling [10, 11] the server whether there is a notification, but polling will incur much extra network traffic as Fig. 2 shows. In the following section we will prove that CoAP over WebSocket proxy can solve this problem.

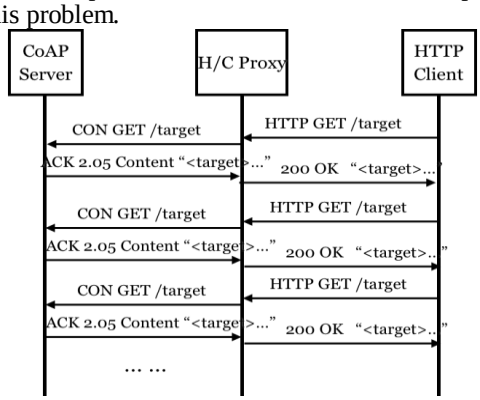

Figure 2. Resource observing (with a "translating" proxy).

B. *HTML5 WebSocket introduction and CoAP over WebSocket proxy*

In the stateless HTTP protocol, the server is unable to identify the client, neither can it take the initiative to send a message to the client. Consequently, the WebSocket [12] API was put forward in HTML5, aiming to make bidirectional communications in application layer possible. Since the limitation of client-initiate session is eliminated, high real-time communication also becomes available. WebSocket technology is now widely used in instant messaging, bullet-screen video, multiplayer online games, smart home and various other fields. Bidirectional communication and real-time communication are the key features of IoT, because the IoT services particularly need to keep the data updated dynamically, and the resource observations is also intended to get the subscribed clients informed whenever the resource is updated. HTML5 WebSocket is readily available on both the client side and the server side. Popular web browsers such as Chrome, Firefox, Opera all support HTML5 WebSocket protocol. On the server side, popular platforms like JavaEE, Node.JS and PHP also support it. In addition, WebSocket runs over the TCP protocol, so it can provide reliability while incurring only a very small overhead.

Given above factors, the CoAP over WebSocket proxy is implemented in this paper. As a result, the overhead of mutual translation between two protocols is avoided, and the CoAP features are well preserved. Fig. 3 Illustrates general process of a WebSocket Client accessing a CoAP server through a WebSocket proxy: Between the browser and proxy, it employs the WebSocket protocol to carry CoAP messages; and between the proxy and the server, it is exact CoAP protocol. Such designation calls for CoAP knowledge of the client (the browser). While CoAP has not been supported by most web browser, this paper develops a browser extension to address the problem.

III. COMPONENTS AND FUNCTIONALITIES

There have been more than 30 kinds of implementations of CoAP, developed in various programming languages. Yet research and implementations on CoAP proxy are still to be enhanced. This paper presents an implementation of CoAP over WebSocket proxy based on Californium [13], an open-source CoAP framework published on Eclipse. A reimplementation of Copper [14], a Firefox add-on, is also achieved. Copper aims to provide easy-to-use browsing experience of CoAP resources, and our work makes it possible to access CoAP resources with a WebSocket proxy, as illustrated in Fig. 3.

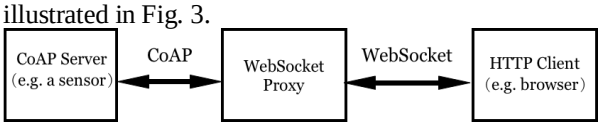

Figure 3. CoAP Over WebSocket proxy.

A. *Client: the Browser*

Our CoAP over WebSocket proxy is designed to communicate between Web browser and the proxy using

WebSocket protocol, and the WebSocket message message carries a payload of CoAP message. Thus the browser should be equipped with the abilities of both WebSocket communication and CoAP message construction. Copper has already implemented the CoAP protocol and is widely used by developers to test their CoAP applications' functionality. This paper re-implemented Copper, adding support for WebSocket communication and WebSocket URI.

The browser interface is shown in Fig. 4. In the address bar, the user types in a WebSocket URI like "ws://127.0.0.1:8887" and requests to establish a WebSocket connection with the proxy. When the connection is established, the client configures URI of the desired CoAP resource. Then it is ok to perform the Ping, Discover, GET, Observe and other operations. According to the operation, the browser constructs a corresponding CoAP message and sends it as the WebSocket payload to the proxy. The proxy is obliged to pick out the CoAP payload and transfer it to the CoAP server, and later transfer the CoAP response in return.

B. WebSocket Proxy

1) Resource Discovery

A node in a resource-constrained network can hold assortment of resources, and it may add or remove a resource at any time. CoAP allows the user to browse and retrieve resources available. It is called Resource Discovery [15]. The RFC 7252 declares that available resources on a CoAP server will be returned when the client sends requests for the well-known address "coap://[host:port]/.well-known/core". This is quite similar to the website navigation function in many Web browsers.

As to the implementation, when the proxy receives a WebSocket message from the browser, it acts as a WebSocket server, picking out the binary CoAP payload from the received message, reading out the URI of the requested CoAP resource, and sending the CoAP request to the CoAP server. The CoAP server receives the resource discovering request and generates a response message containing the information of all resources it holds, then responds to the proxy. The proxy finally delivers the response to the web browser. As a result, the user will see a resource directory as shown in the left-side area of Fig. 4.

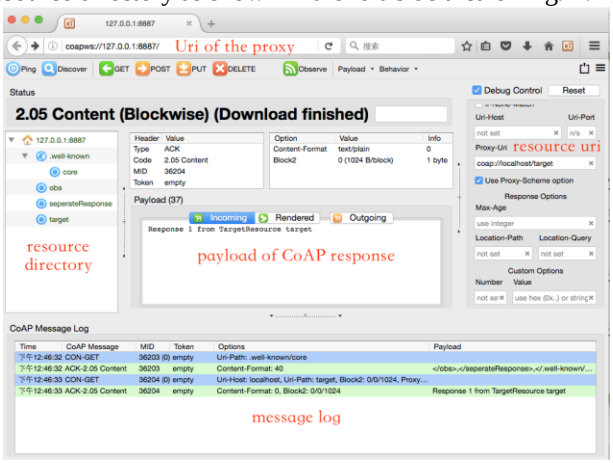

Figure 4. Accessing CoAP resources from the Web browser.

2) Common Requests/Responses

For a common request from the Web browser, for example, to GET a certain resource, the WebSocket proxy will directly transfer the CoAP server's response. The CoAP server's response to a confirmable request can either be sent piggy-backed with the ACK or in a separate confirmable response. In the former case, the ACK message piggy-backs the resource content; and in the latter case, the resource is not ready at present, and the CoAP server responds with an empty ACK as acknowledgement of the request. When the resource is ready, the server will send it to the client.

3) Resources Observation

As IoT has a high demand for real-time data, resource observation on the client side and resource publishing and updating on the server side are particularly important. CoAP protocol provides resources observation mechanism to allow users to subscribe to specific resources and receive CoAP notification messages upon a state change of the resource. In this paper, HTML5 WebSocket technology and the java-websocket open-source library [16] are used to implement the CoAP over WebSocket proxy. The java-websocket open-source library implements the full set of WebSocket protocol interfaces, and it supports sending both text messages and binary data. Developers can write their own processing logic according to actual needs when the messages are received.

As illustrated in Fig. 5, the WebSocket client (the browser) sends a resource observation request to the proxy and the proxy forwards the request to the specified CoAP Server. When the CoAP Server receives that request containing an Observe option, it will send notification messages to proxy once the observed resource is updated and the proxy then forwards notification messages to the WebSocket client.

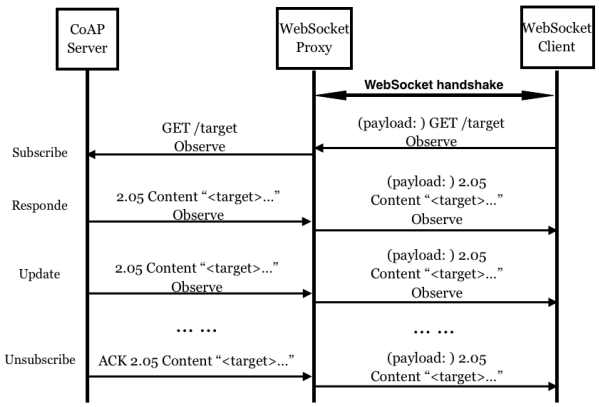

Figure 5. Resource observation (with the WebSocket proxy).

4) Blockwise Transfer

Typically, the amount of data CoAP a message is very small, just dozens of bytes, but sometimes data packets of bigger size need to be transmitted, such as when a client node taking a POST or PUT operation to upload the data. Therefore, CoAP designed the blockwise transfer mechanism [17]. Before a CoAP message is sent, the message sub-layer detects the packet length. If it exceeds a threshold value, the

message will be fragmented into several blocks and sent block by block. Each block of message has a “Block” option.

The process of a typical blockwise scenario is shown in Fig. 6: The CoAP response message to a GET request is divided into three blocks for separate transmission. The proxy sends the GET request to the CoAP Server, and receives a response containing the Block option. This is the first block, length 128 Bytes, and followed by other blocks. The proxy then sends another GET request and expects to receive another response, until the last block is received.

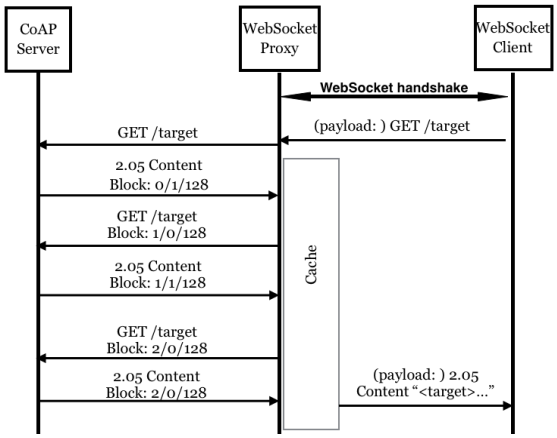

Figure 6. Blockwise transfer.

C. CoAP Server and the Resorce

In this paper, the CoAP server is deployed on a PC running Californium framework, and different kinds of resources in accordance with CoAP’s featured functionalities are provided to clients.

IV. EVALUATION

We test the WebSocket proxy and the HTTP/CoAP under the same condition. Both proxies run on PC with 8GB RAM and 1.6GHz, in a less constrained network. The network is set to be IEEE 802.11g Wi-Fi.

For the sake of higher accuracy, CoAP Server and the proxies are placed in the same network. In this way, the test result will be least affected by the network delay.

A. Testing the Responding Time

We test the responding time in three different cases: using no proxy, using a HTTP/CoAP proxy, and using a WebSocket proxy. The test program starts when the client sends a request, and ends when the client receives the response. We repeat the same test program for ten times and calculate the average value as the test result, as Table 1 shows.

TABLE I. RESPONDING TIME OF DIFFERENT CASES

The case	Responding time (millisecond)
Use no proxy	6.020
Use H/C proxy	10.020
Use WS proxy	7.062

From the table we see that there is no significant difference between the case of using no proxy and the case of using WebSocket proxy, while using WebSocket proxy takes much less time than using HTTP/CoAP proxy.

B. Stress and Performance Test

In this part, we test and compare the performance of both proxies at different concurrency levels. For each concurrency level, the test program creats the specified number of virtual clients, and each client keeps sending as many CoAP requests as possible in a certain amount of time, say 60 seconds. Every time the request is sent, the virtual client waits to receive the response before sending the next request. To avoid the case that the client fails to receive a response thus waits for too long before sending another request, a timeout can be set for the waiting period. When the test is finished, we count the total number of successful requests and divide it by the test time to get the throughput (request per second) of the proxy, as shown in Table 2.

TABLE II. THROUGHPUT COMPARISON OF TWO PROXIES

Concurrency level / throughput	10	20	30	50	100
H/C proxy	998	1686	2676	4915	10946
WS proxy	1416	2972	3729	7220	13840

(continued)

Concurrency level / throughput	200	300	500	600
H/C proxy	18579	28065	53579	55901
WS proxy	25980	34650	66950	67800

(contiued)

Concurrency level / throughput	700	800	900	1000
H/C proxy	66079	79132	83047	112150
WS proxy	89670	107380	107380	140304

On the line chart, Fig. 7, the performance differences between HTTP/CoAP proxy and CoAP over WebSocket proxy is depicted more vividly. The result suggests that CoAP over WebSocket proxy can serve no less than 1,000 concurrent users steadily.

V. CONCLUSION

Starting from the background of IoT applications and services, this paper aims to solve the problem of accessing CoAP resources held on constrained nodes from the Web browser. We focus on how to design an intermediate proxy that not only meets the need of the client but also preserves the features of CoAP to the most. Firstly, we analyze the reason why using the standard HTTP/CoAP proxy recommended by IETF CoRE working group is not the best solution. Two most important reasons are the overhead caused by "translation" between two protocols, and the loss of some additional features of CoAP protocol that are not supported by HTTP. Considering above reasons, we designed and implemented CoAP over WebSocket proxy on basis of Californium open-source framework. Finally, we conduct performance tests for the two kinds of proxies and compare the results. It turns out that the WebSocket proxy is advantageous to the HTTP/CoAP proxy in term of respond speed and the ability to deal with high concurrency requests.

In this paper, the CoAP Server is deployed in the Wi-Fi network environment. Wi-Fi is counted as a less constrained network, and it is not quite like the real constrained environment such as Wireless Sensor Networks (WSN). To obtain more exact test data, similar experiments need to be run in real constrained environments.

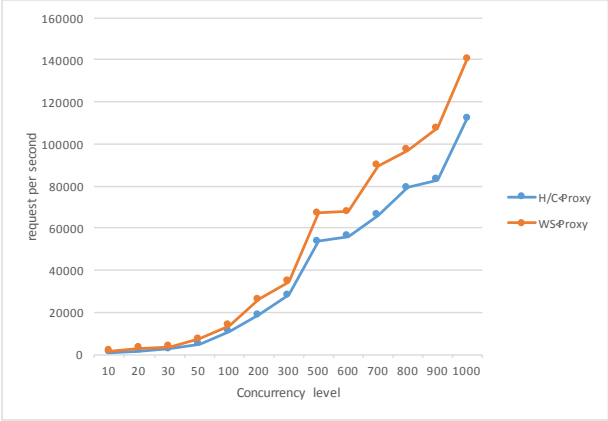

Figure 7. Stress performance test result.

REFERENCES

[1] Sundmaeker H, Guillemin P, Friess P. Vision and challenges for realizing the Internet of Things. Cluster of European Research Projects on the Internet of Things, European Commission, 2010.

[2] Fielding R T. Architectural styles and the design of network-based software architectures. University of California, Irvine, 2000.

[3] Shelby Z, Hartke K, Bormann C. The constrained application protocol (CoAP). RFC 7252. 2014

[4] Sun Qi-bo, Liu Jie, Li Shan. Internet of Things: Summarize on Concepts, Architecture and Key Technology problem. Journal of Beijing University of Posts and Telecommunications, 2010, 33(3), pp. 1-9.

[5] Fette I, Melnikov A. The websocket protocol. RFC 6455, 2011.

[6] Giang N K, Ha M, Kim D. SCoAP: An integration of CoAP protocol with web-based application. 2013 IEEE Global Communications Conference (GLOBECOM). IEEE, 2013, pp. 2648-2653.

[7] Shelby Z, Bormann C. 6LoWPAN: The wireless embedded Internet. John Wiley & Sons, 2011.

[8] Fielding R, Gettys J, Mogul J. Hypertext transfer protocol--HTTP/1.1. RFC 2616. 1999.

[9] Hartke K. Observing resources in the Constrained Application Protocol (CoAP). RFC 7641. 2015.

[10] Garrett J J. Ajax: A new approach to web applications. 2005.

[11] Loreto S, Saint-Andre P, Salsano S, et al. RFC 6202-Known issues and best practices for the use of long polling and streaming in bidirectional HTTP. 2011.

[12] Wang V, Salim F, Moskovits P. The definitive guide to HTML5 WebSocket. Berkeley, Calif, USA: Apress, 2013.

[13] Kovatsch M, Lanter M, Shelby Z. Californium: Scalable cloud services for the internet of things with coap[C]//Internet of Things (IOT), 2014 International Conference on the. IEEE, 2014, pp. 1-6.

[14] Kovatsch M. Demo abstract: human-CoAP interaction with copper. Distributed Computing in Sensor Systems and Workshops (DCOSS), 2011 International Conference on. IEEE, 2011, pp. 1-2.

[15] Krco S, Shelby Z, Bormann D C. Core resource directory. March. 2016. Digital Object Identifiers (DOIs): https://tools.ietf.org/html/draft-ietf-core-resource-directory-07

[16] Nathan R. Java WebSockets. August. 2015. Digital Object Identifiers (DOIs): https://github.com/TooTallNate/Java-WebSocket

[17] Shelby Z, Bormann C. Block-wise transfers in CoAP. 2016.