IoT WebSocket Connection Management Algorithm for Early Warning Earthquake Alert Applications

Ajinkya Mulay
Indian Institute of Technology, Hyderabad
ee14btech11040@iith.ac.in

Hideya Ochiai
The University of Tokyo
ochiai@elab.ic.i.u-tokyo.ac.jp

Hiroshi Esaki
The University of Tokyo
hiroshi@wide.ad.jp

ABSTRACT

IoT devices are increasingly being used in various applications and their field is diversifying owing to their small size. Most of the IoT architecture takes client and server model – IoT devices are connected to a server on the Cloud. Real time communication, especially from the server to client, is necessary when we consider the earthquake alert applications. WebSocket protocol, which has been successfully used in browsers for bi-directional communication, can be applied to such communication. However, we have to carefully manage the loss of connection between the client and server. In this paper, we propose a WebSocket connection management algorithm for IoT, called reconnection with dynamic ping-pong algorithm (RDPPA), focusing on improving and adapting the WebSocket protocol for connecting IoT devices. We implemented the algorithm and carried out experiments for evaluating message delivery latency and the amount of traffic overhead.

KEYWORDS

WebSocket, Internet of Things, Connection Management, Earthquake

1 INTRODUCTION

With the increasing role of IoT in our life, the need for real-time and bidirectional communication between IoT device and IoT server is increasing. In the current Internet and Cloud - based architecture multiple applications are running on the Cloud. IoT devices are generally deployed behind a NAT or a Firewall which makes communication from the Cloud to the devices challenging. For solving this problem, it is considered that WebSocket will provide real-time and mutual communication[2][8].

WebSocket[10] takes a client-server model as an upgraded version of HTTP, and supports bidirectional communication. WebSocket itself is a browser based protocol originally, but because of the bidirectional communication capability, it is also adopted in IoT field. However, WebSocket communication is based on TCP, and sometimes the connection is lost because of the existence of middle boxes. Such middle boxes are network address translators (NAT), firewalls, Web proxy servers, and any other security management boxes but not IP routers.

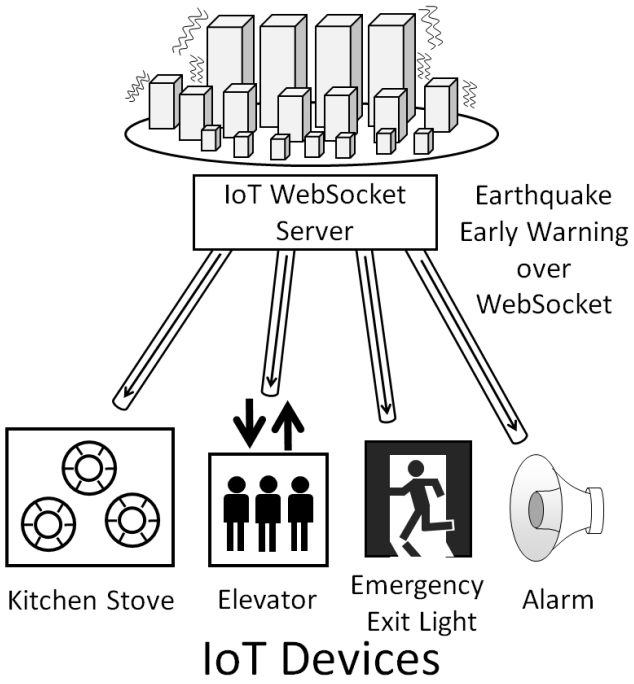

Figure 1: Earthquake early warning notification over WebSocket for (1) turning kitchen stove off, (2) stopping elevators, (3) showing emergency exits, and (4) alarming people before the S-wave, i.e., the main earthquake wave, arrives.

We propose a WebSocket connection management algorithm for IoT, called reconnection with dynamic ping-pong algorithm (RDPPA), assuming to build a real-time natural disaster emergency alert system. A server will be notified from the meteorological agency[14] and then if any parameter is above the disaster threshold, a message containing data about the possible disaster will be sent to the IoT device over WebSocket connection. The IoT device, will in turn take actions, e.g., (1) stopping its kitchen stove, (2) stopping its elevator, (3) lighting emergency exit light, and (4) sounding an alarm. Since all of this should occur in real-time within a second, it is very important to deliver alert messages to the IoT devices with no loss and delay.

IoT devices normally uses narrowband, i.e., bandwidth-limited, cellular communication to save the communication fee [1]. The communication cost depends on the total volume of traffic consumed by the IoT devices. So, it is very important to reduce traffic

UCC Companion'17, December 5–8, 2017, Austin, Texas, USA
 ISBN 978-1-4503-5149-2/2/17/12…$15.00
DOI: https://doi.org/10.1145/3147234.3148094

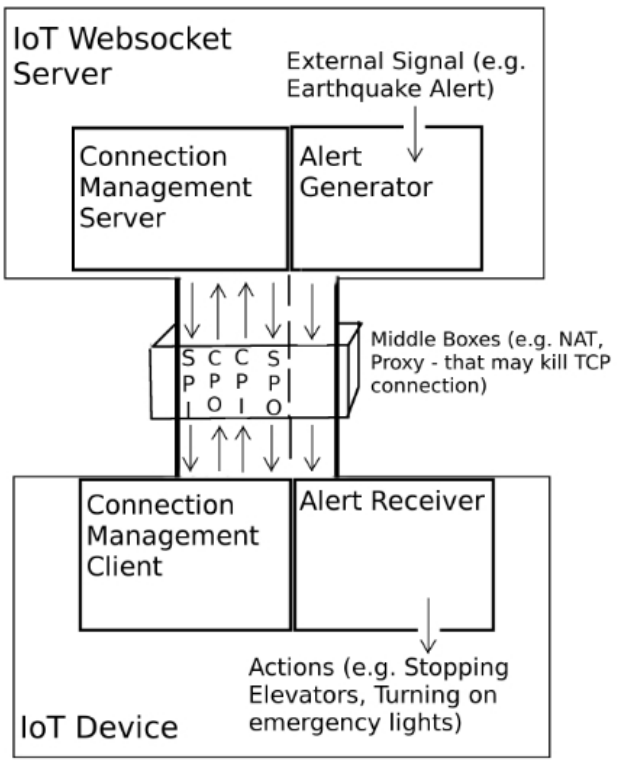

Figure 2: Architecture for IoT WebSocket connection management

as much as possible, while keeping connection to the WebSocket server.

For an emergency alert system its crucial that the delay should be really small while having the highest possible reachability. Our algorithm ensures to keep connection between IoT device and the server and provide immediate alert notification from the server. We developed a prototype system and evaluated message delivery latency and total amount of traffic.

The paper ahead is divided into the following sections. Section 2 talks about the related work. Section 3 explains the proposed system model and the algorithm. Section 4 performs the evaluation. Section 5 discusses the contribution of the algorithm and further possibilities. Section 6 gives the conclusion of this paper.

2 RELATED WORK

Historically there have been many protocols which enable bidirectional communication and also keeps system alive for a long time. Some of these include AJAX Long Polling and REST API. There are also a few IoT specific protocols like CoAP, MQTT, MQTT-SN besides WebSocket. But as concluded in [2], WebSocket is a much more superior communication protocol in terms of energy consumption, memory usage and performance. Since all three of these parameters are vital for IoT devices, we have decided to use it for the emergency alert system. By [3] we can see that WebSocket outperforms REST API in terms of energy consumption and latency.

In e.g., [4–6], they discuss the use of WebSocket for remote monitoring applications. But, they consider to use WebSocket for the communication between a central server and a web browser, i.e., user-interface.

[7] discusses the importance of bidirectional communication between an IoT server and a IoT device. [8] developed a tunneling of IEEE 1888 messages over WebSocket connection in order to provide bi-directional communication. [9] discuss the use of WebSocket for controlling servomotor from the server side, but it is in an educational program. All of these works do not discuss the connection maintenance of WebSocket, which we target in this paper.

3 IOT WEBSOCKET CONNECTION MANAGEMENT

3.1 Architecture

Fig 2 shows the architecture for IoT WebSocket connection management. It consists of IoT WebSocket server and IoT device. These two components are connected over a WebSocket channel, which communication is initiated from the IoT device. There potentially exists middle boxes (e.g., NAT, firewall, proxy) that may kill the TCP connection intentionally without notifying to the server and the device. The server and device exchange special messages, labelled as SPI, CPO, CPI and SPO in the figure, to check and keep the status of connection. The alert generator of the server receives external signal (e.g., the beginning of earthquake signal) and sends an alert signal to the affected IoT devices. The IoT devices can take actions to make our life safe, such as stopping elevators before the primary huge earthquake waves arrive.

For managing the WebSocket connection, we propose a reconnection with dynamic ping-pong algorithm (RDPPA). In the algorithm, server and device exchange, server ping (SPI), client pong (CPO) in response to SPI, client ping (CPI), and server pong (SPO) in response to CPI. Pings from client and server are issued with dynamic changing interval discovering the existence of the middle boxes. We here describe RDPPA server side algorithm and client side algorithm.

3.2 RDPPA Server Side

We will be referring to the **Algorithm 1** in this section. On the server side, the inputs to the RDPPA Algorithm are Port Number ($Server_{port}$) for webserver to start the connection on and the Server Path ($Server_{path}$) for one client. The server starts a webserver on $Server_{port}$ and then handles the incoming connections with the **Connection Handle Function** on its current IP address with the path specified by $Server_{path}$. The other inputs are $time_{init}$, $time_{fixed}$ and $delay_{fixed}$. Note that whenever a new connection joins the HTTP connection is upgraded to a WebSocket connection by an Upgrade function. After that the server algorithm proceeds as shown in **Algorithm 1**.

Whenever a new client initiates connection with the server, **Connection Handle Function** handles them. This function has 3 threads - (1) Receiving Thread, (2) Data sending Thread and (3) Ping-Pong Handling Thread.

3.2.1 Receiving Thread. When the client starts it sends introductory data which includes the Device Type, ID and Periodic Time Interval (PTI). The device type and ID helps the server differentiate

Algorithm 1 RDPPA Server Side

Inputs: $Server_{port}$, $Server_{path}$, $time_{init}$, $time_{fixed}$, $delay_{fixed}$
Connection Handle Function:
Initialisation:
currentTime, previousTime, sendTime, strikes = 0
waitingTime = $delay_{fixed}$
PTI = $time_{init}$
Receiving Thread:
while true **do**
  if Unique Introductory Data received from client **then**
    Store client device type and ID
    Update PTI to client PTI
  else if Repeated Introductory Data **then**
    Block device and disconnect
  else if CPI received from client **then**
    Send SPO back to client
  else if CPO received from client **then**
    PTI+=t_{fixed}
    strikes = 0
  end if
end while
Data Sending Thread:
Send any alert/data to client if available
Ping-Pong Handling Thread:
while true **do**
  currentTime = time.now()
  if (currentTime - sendTime) ≥ waitingTime **AND** (SPI sent) **AND** (CPO **NOT** received) **then**
    Quick Recovery Mode:
    if strikes ≤ 3 **then**
      strikes++
    else
      Close connection to client
      PTI/=2
    end if
  else if (currentTime - previousTime) ≥ PTI **OR** (strikes ≥ 1) **then**
    Send SPI to client
    sendTime = time.now()
  end if
end while
Main:
Listen and run server on $Server_{port}$ at $Server_{path}$ with handle(**Connection Handle Function**)

the incoming clients and avoid repetitions. The PTI is used for sending pings. The Receiving Thread handles incoming CPI and CPO and sends back SPO in response to the CPI. All these terms and their usage will be explained in the Ping-Pong Thread section.

3.2.2 Data Sending Thread. Whenever the server receives a command to send an Alert or some kind of data to the client, this thread sends it.

3.2.3 Ping-Pong Thread. As the main aim of this algorithm is to make sure the TCP connection under the WebSocket is not killed by intermediate boxes we have implemented a Ping-Pong system. To make sure the client-server connection is alive both the server and the client periodically check the connection at intervals of PTI, by sending pings and waiting for a pong to return back in under the waiting time of $delay_{fixed}$. The PTI is initialised with the amount of $time_{init}$. Now since these ping pong will add traffic overhead to the data messages we have implemented a dynamic ping-pong system. Whenever a ping sent gets the pong response withing the $delay_{fixed}$ time interval, we update the PTI by adding $time_{fixed}$ to the current PTI. This increases the delay between consecutive successful pings which reduces the traffic overhead over time.

If on the other hand, the pong response to the ping fails to reach within the $delay_{fixed}$ time interval we resort to **Quick Recovery Mode**. In this mode we send a pings until the device responds with a pong under the $delay_{fixed}$ time interval or 3 pings receive no pong response. If a response is received the connection is assumed to be alive and continues operating fine while if no response is received the server closes the connection.

3.3 RDPPA Client Side

Here we are going to refer to **Algorithm 2**. Initially the server is assumed to powered on and waiting for new client connections. The client then is powered on and the inputs to it are the Server IP Address ($Server_{addr}$), Server Port ($Server_{port}$), Webserver Host (WS_{host}), Webserver Path (WS_{path}), TCP client object (TCP_{client}), Web Socket Client Object (WS_{client}) and knows own MAC Address. The other inputs are $delay_{fixed}$, $time_{init}$ and $time_{fixed}$. The client initially turns on its Serial Monitor and then obtains IP address by using its own MAC address and DHCP on the network its connected to. After that, the client algorithm proceeds as described in **Algorithm 2**. Here there are 2 parts in the Client Algorithm - (1) Handshake Setup, (2) Main Loop.

3.3.1 Handshake Setup. Initially the **Handshake Setup** Function is executed. The TCP_{client} object sends a connection request on the port and path which the server is listening on via the network (Wi-Fi, Ethernet or cellular) the client is accessing. If the connection is not successful, then the client continues to try connecting periodically with a delay of 2 seconds until successful. Once successful the client sets the WebSocket host and the WebSocket Path and proceeds to initiate a WebSocket handshake. Again if the handshake is unsuccessful, then the client continues to try the handshake with a delay of 2 seconds until connected. Once the handshake is completed the client sends introductory data to server and breaks out of the **Handshake Setup** Loop and proceeds to the **Main Loop**.

3.3.2 Main Loop. On every loop the TCP_{client} object checks whether the client is connected. If it detects it is not connected then it closes the current connection and proceeds to retry handshake with the current PTI. Note that this method is unable to detect the TCP kills done by the intermediate boxes and works only when the underlying TCP connection is alive. If the TCP_{client} object identifies the connection is alive then the **Main Loop** is able to receive Alerts sent by the server as well as SPI and SPO. In response to SPI, the **Main Loop** sends back CPO. If SPO is received under the waiting time of $delay_{fixed}$ time interval then the PTI is updated by

Algorithm 2 RDPPA Client Side

Inputs: TCP_{client}, WS_{client}, $Server_{addr}$, $Server_{port}$, WS_{host}, WS_{path}, $delay_{fixed}$, $time_{init}$, $time_{fixed}$
Handshake Setup Function: (PTI)
while true **do**
  currentTime, previousTime, sendTime, strikes = 0
  waitingTime = $delay_{fixed}$
  if TCP_{client}.connect($Server_{addr}$, $Server_{port}$) is successful **then**
    Set WS_{client} host (WS_{host}) and path (WS_{path}) parameters
    if WS_{client} handshake is successful **then**
      Send PTI to server and **BREAK** from loop
    else
      Retry handshake
      Delay(2 seconds)
    end if
  else if TCP_{client}.connect($Server_{addr}$, $Server_{port}$) is unsuccessful **then**
    Retry connection
    Delay(2 seconds)
  end if
end while
Main:
PTI = $time_{init}$
Call **Handshake Setup** (PTI)
while true **do**
  if TCP_{client} is connected **then**
    currentTime = time.now()
    if Alert Received from server **then**
      Print out data and trigger hardware
    else if SPI received from server **then**
      Send CPO to server
    else if SPO received from server **then**
      PTI+=$time_{fixed}$
      strikes = 0
    end if
    if (currentTime - sendTime) ≥ waitingTime **AND** (CPI sent) **AND** (CPO **NOT** Received) **then**
      Quick Recovery Mode:
      if strikes ≤ 3 **then**
        strikes++
      else if strikes > 3 **then**
        PTI/=2
        Stop TCP_{client} and Re-initiate **Handshake Setup** (PTI)
      end if
    else if (currentTime - previousTime) ≥ PTI **OR** (strikes ≥ 1) **then**
      Send CPI
      sendTime = time.now()
    end if
  else if TCP_{client} **then**
    PTI/=2
    Stop TCP_{client} and Re-initiate **Handshake Setup** (PTI)
  end if
end while

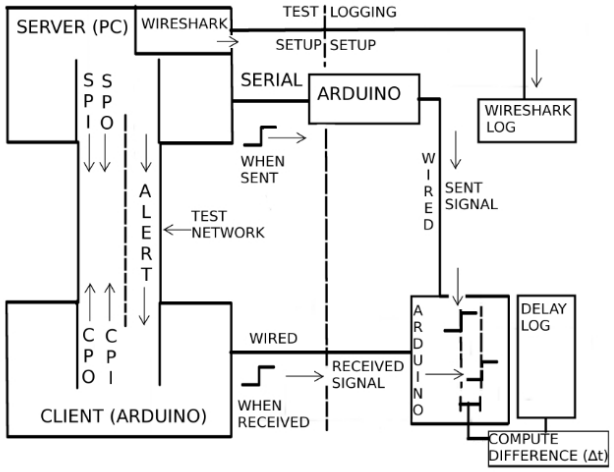

Figure 3: Configurations for measuring message delivery latency and WebSocket traffic

adding $time_{fixed}$ to the current PTI. The PTI has been initialised with the value of $time_{init}$. If however the SPO is not received under the $delay_{fixed}$ time interval then the Client goes into the **Quick Recovery Mode** just like the Server. It then just like the server proceeds to send CPI to server until it receives an SPO or it has completed 3 tries.

If the SPO is received within this time, then the connection is assumed to be alive and everything continues to function normally. On the other hand if the SPO is not received then the client closes the connection and proceeds to re-initiate **Handshake Setup** with the current PTI.

4 EVALUATION

We implemented RDPPA onto our prototype system and evaluated the latency of alert message delivery and the amount of total traffic. As a reference, we compared RDPPA with a simple re-connection algorithm (SRA). SRA does not exchange ping-pong messages but simply re-initiates a WebSocket connection from a client side when the client detects the loss of the TCP session.

4.1 Experiment Setting

Fig. 3 shows our experiment setting. We implemented IoT WebSocket server onto a laptop computer (DELL Inspiron 14, 5000) with Golang Gorilla WebSocket library[11] in golang. We implemented IoT devices onto an Arduino Mega compatible board with Arduino-WebSocket library[12]. We have prepared other two Arduinos for measuring latencies of the alert message from the server to the client. For Golang to Arduino Serial Communication Tarm Serial library was used. [13] As for the network, we prepared three types of networks as follows:

- **Case 1 (Ethernet):** Direct connection between client and server over Ethernet, i.e., on an IP network segment. There were no middle boxes.
- **Case 2 (Wi-Fi):** Connection over our university campus network. There was a NAT between client and server, but

Table 1: Average message delivery latency. 3G case has achieved almost same performance as Wi-Fi case. As for SRA for 3G, data was not available because of the failure of message delivery.

Test Network (applied Algorithm)	Average Delay [ms]
Ethernet (SRA)	0.010
Ethernet (RDPPA)	0.011
Wi-Fi (SRA)	6.05
Wi-Fi (RDPPA)	7.04
3G (SRA)	N/A
3G (RDPPA)	7.89

this NAT is configured not to kill TCP connection for 7 days.

- **Case 3 (3G):** Connection over a carrier network. There are supposed to be several black-boxed middle boxes between client and server.

In each of these networks we tested the websocket connection between the Arduino, i.e., IoT devices, and the IoT WebSocket server on SRA and RDPPA.

4.2 Message Delivery Latency

According to the Block diagram in Fig. 2, a common server is sending the exact same message to two Arduinos. One Arduino is connected to the server via one of the 3 MAC Layers (either Wi-Fi, Ethernet or 3G Cellular) over a websocket and the other is directly connected to the server via a wired serial link. Since the wired serial link has the least possible delay, we are comparing it with the websocket link to compare the timings of both.

Now to synchronise the timings for two independent Arduinos, we need to have a common clock. So, we have placed a Synchronising Arduino which is wired to the output ports of both the Arduinos. Whenever either of the Arduino receives any server message, that Arduino changes the state of its output PIN as high for a small time duration of 10 milliseconds. The synchronising Arduino is constantly monitoring the states of the output pins of the 2 Arduino through the wired link and every time the state is changed, the local clock timing is noted by the Synchronising Arduino. When the state of the other Arduino changes as well, the time difference is calculated and printed to the Serial console of the synchronising Arduino. The delay calculated has a precision of 4 microseconds, since that is the maximum precision offered by Arduino.

4.3 Traffic Overhead

While designing a reliable algorithm, a certain overhead is incurred in the form of data exchanged which leads to latency degradation. Considering that IoT devices might not have high speed lines dedicated for transmission especially in cellular communication, it becomes vital that the overhead occurred is affordable and useful in decreasing net delay while increasing the message accuracy.

In the basic algorithm of just re-connection we do not have the ping pong messages while in the advanced algorithm we have those messages too. This difference in the traffic overhead is what we want to calculate along. This is done by using 'tcpdump' command and then analysing the obtained file using inbuilt filters in Wireshark. The graphs for each of the networks have been presented below.

Table 2: Traffic volume measured by the experiment. RDPPA generated much larger traffic volume than SRA did. In 3G case, the WebSocket connection were kept available with RDPPA, but it failed with SRA.

Test Network (applied Algorithm)	Messages Sent	Messages Dropped	Packets (Total)	Bytes (Total)
Ethernet (SRA)	50	0	51	2923
Ethernet (RDPPA)	50	0	358	25023
Wi-Fi (SRA)	50	0	51	3570
Wi-Fi (RDPPA)	50	0	358	25023
3G (SRA)	50	49	51	130
3G (RDPPA)	50	0	730	61306

To take this into account we are sending the necessary periodic connection stay awake messages with a additive increase / multiplicative decrease algorithm.

4.4 Application of Earthquake Alarm

The architecture for deploying a real-time earthquake alert system is very similar to the one described in Fig. 2. Our prototype for demonstration setup is described in Fig. 5. As seen in the Fig. 5, the server is supposed to receive a notification of an imminent earthquake via a government agency/website like Japan Meteorological Agency. The moment this alert is received by the server, the server passes on this alert to the Arduino clients connected to it over cellular network or Wi-Fi via WebSocket protocol. In the alert sent from the server to the Arduino clients, the server specifies what hardware functions it wants the client to perform. The Arduino which is connected to devices like LEDs, speakers or even high powered machines can then turn on LEDs or sound an alarm or even cut off power to the high powered machines. Such a system is very much crucial to avoid accidents in case of an earthquake.

An Earthquake Early Warning System (EEW) have already been implemented in Japan [14]. Such a system, tracks the P-wave which are the preliminary tremors of the earthquake. Once these are detected it estimates the incoming seismic intensities and the arrival time of the S-wave or the principal motion of the earthquake. This intermediate time is very crucial to provide time for evacuation, stopping elevators and even the high-speed bullet trains. All these activities can be carried out by the system suggested in Fig. 2 using the low-latency WebSocket IoT Network. For practical demo, we carried out the blinking of an LED and stopping a motor using RDPPA over 3G network.

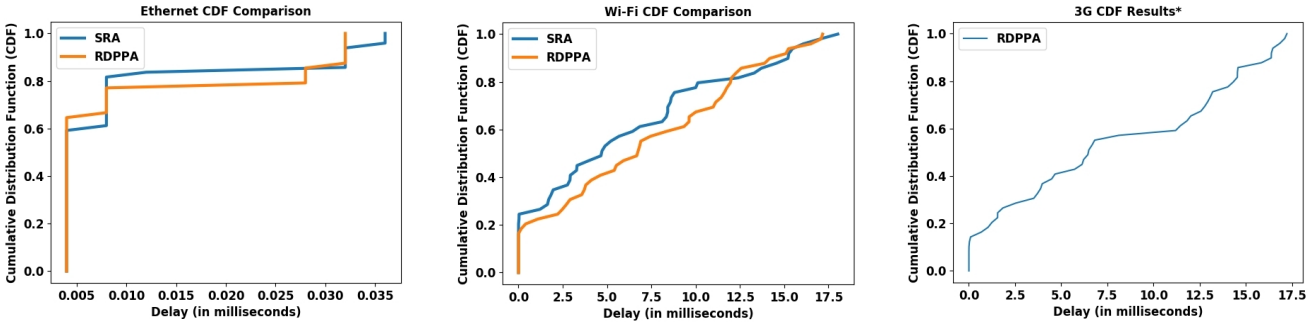

Figure 4: Cumulative distribution function (CDF) of message delivery latency for (1) Ethernet, (2) Wi-Fi and (3) 3G. (* - For 3G, there is no CDF for SRA, because of the failure of alert message delivery.)

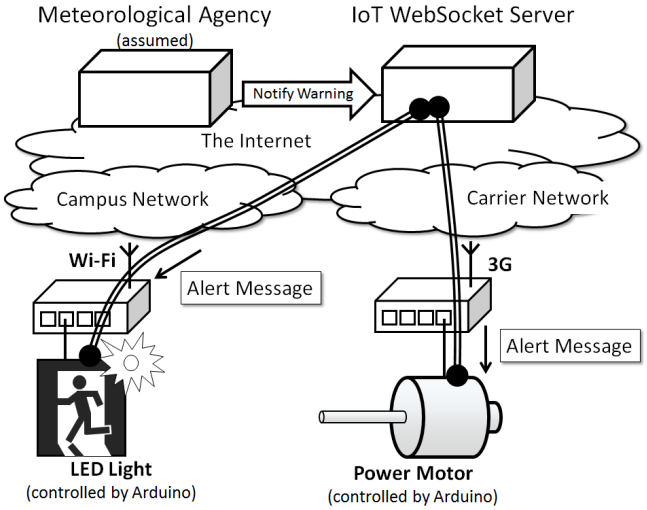

Figure 5: Earthquake alarm system implementation. Alert messages are delivered over the IoT WebSocket connections. A LED light and a power motor are controlled.

5 DISCUSSION

In the current system we have not used the secure version of WebSocket Protocol. We have used 'ws' protocol which is similar to HTTP and not 'wss' which is more similar to HTTPS. If used with the secure version the delay measurement might change due to security overhead. This is because we are focusing only on the communication parameters of the network and not the security.

Currently in our earthquake alert system, we are broadcasting the same messages to all the clients from the server since they are in a similar geographical area. But, if we consider a realistic scenario for earthquake, the alert level will be different in different regions. This can be taken care of by transmitting the latitude and longitude from the IoT devices. Since, these IoT devices will be stationary when deployed, we can set the location inside the client and it can be requested at any time. Or else, it may be acquired by triangulation which will take up some more processing power. Either ways, the location will be sent to the server on request and the server according to its data and region classification will send the appropriate alerts to the IoT devices.

6 CONCLUSION

In this paper, we proposed a WebSocket connection management algorithm for IoT, called reconnection with dynamic ping-pong algorithm (RDPPA), assuming to build a real-time natural disaster emergency alert system.

The results of our experiments indicate that RDPPA performs efficiently on several network cases regarding to the amount of total traffic required and message delivery latency with keeping connections alive. We demonstrated that WebSocket is useful for critical applications such as early-warning earthquake alert applications.

REFERENCES

[1] *Narrowband IoT (NB-IoT)*, https://www.u-blox.com/en/narrowband-iot-nb-iot
[2] Dae-Hyeok Mun, Minh Le Dinh, and Young-Woo Kwon. *An Assessment of Internet of Things Protocols for Resource-Constrained Applications.* In IEEE COMPSAC, 2016.
[3] Volker Herwig, Ren Fischer and Peter Braun. *Assessment of REST and Websocket in regards to their Energy Consumption for mobile Applications* In IEEE IDAACS, 2015.
[4] A. Hashibuan, M. Musrdi, E. Y. Syamsuddin, and M. A. Rosidi. *Design and Implementation of Modular Home Automation Based on Wireless Network, REST API, and WebSocket.* In IEEE ISPACS, 2015.
[5] L. Zhang and Xiaoxiao Shen. *Research and Development of Real-time Monitoring System Based on WebSocket Technology.* In IEEE MEC, 2013.
[6] Z. B. Babovic, J. Protic, and V. Milutinovic. *Web Performance Evaluation for Internet of Things Applications.* IEEE Access, vol. 4, pages 6974 - 6992, 2016.
[7] C. Doukas, L. Capra, F. Antonelli, E. Jaupaj, A. Tamilin, and I. Carreras. *Providing Generic Support for IoT and M2M for Mobile Devices.* In IEEE RIVF, 2015.
[8] Y. Tarutani, S. Murata, K. Matsuda, and M. Matsuoka. *IEEE1888 over WebSocket for communicating across a network boundary.* In IEEE COMPSAC, 2016.
[9] G. C. Fernandez, E. S. Ruiz, M. C. Gil, and F. M. Perez. *From RGB led laboratory to servomoter control with websockets and IoT as educational tool.* In IEEE REV, 2015.
[10] *The WebSocket Protocol.*, https://tools.ietf.org/html/rfc6455
[11] *Gorilla Websocket Library - A Go implementation of the Websocket protocol.* Retrieved from https://github.com/gorilla/websocket
[12] *Brandenhall Arduino-Websocket Library - Simple Library that implements WebSocket client and server running on Arduino.* Retrieved from https://github.com/brandenhall/Arduino-Websocket
[13] *Tarm Serial Library - Go package to allow you to read and write from the serial port as a stream of bytes.* Retrieved from https://github.com/tarm/serial
[14] *Earthquake Early Warning System*, http://www.jma.go.jp/jma/en/Activities/eew.html