Analysis of Wireless Video Sensor Network Platforms

over AJAX, CGI and WebRTC

by

Sri Harsha Rentala

A Thesis Presented in Partial Fulfillment
of the Requirements for the Degree
Master of Science

Approved February 2016 by the
Graduate Supervisory Committee:

Martin Reisslein, Chair
Jennifer Kitchen,
Michael McGarry

ARIZONA STATE UNIVERSITY

May 2016

ABSTRACT

Since the inception of Internet of Things (IoT) framework, the amount of interaction between electronic devices has tremendously increased and the ease of implementing software between such devices has bettered. Such data exchange between devices, whether between Node to Server or Node to Node, has paved way for creating new business models. Wireless Video Sensor Network Platforms are being used to monitor and understand the surroundings better. Both hardware and software supporting such devices have become much smaller and yet stronger to enable these. Specifically, the invention of better software that enable Wireless data transfer have become more simpler and lightweight technologies such as HTML5 for video rendering, Common Gateway Interface(CGI) scripts enabling interactions between client and server and WebRTC from Google for peer to peer interactions. The role of web browsers in enabling these has been vastly increasing.

Although HTTP is the most reliable and consistent data transfer protocol for such interactions, the most important underlying challenge with such platforms is the performance based on power consumption and latency in data transfer.

In the scope of this thesis, two applications using CGI and WebRTC for data transfer over HTTP will be presented and the power consumption by the peripherals in transmitting the data and the possible implications for those will be discussed.

TABLE OF CONTENTS

Page

LIST OF TABLES ……....iii

LIST OF FIGURES ……..iv

CHAPTER

1. INTRODUCTION…….. 1
2. INTERNET OF THINGS……..... 5
3. WEB APPLICATIONS FOR WIRELESS SENSOR NETWORKS……..……10

    AJAX Role in Fetching Data onto WVSNP Player...……12

4. INTEGRATED WVSNP DASHBOARD...……...16
5. WebRTC: P2P Communications……...23

    Serverless-WebRTC Application…….26

6. Test Bed Explanation…….29

    Comparison between CGI and WebRTC……..29

    Test-Bed for ZigBee Data Transfer.……..34

7. Results and Recommendations……. 37

    Comparison of File Transfer between CGI and WebRTC..……..38

    Comparison of File Transfer over ZigBee and Wi-Fi using CGI.……44

8. Future Work……...53
9. References……. 55

LIST OF TABLES

Table Page

1. Power Comparison between CGI & WebRTC……...50

2. Power Comparison between Wi-Fi & ZigBee……....50

LIST OF FIGURES

Figure Page

1. WVSNP-DASH Player built on HTML5……..13
2. Flow Chart showing Client Interaction with Server using XHR..……13
3. Node Functions on the WVSNP Dashboard……..19
4. FFmpeg Capture Function on the Dashboard……....21
5. NAT issues over Networks……...…….24
6. WebRTC: Serverless –WebRTC application…….....27
7. Measurement of system vitals in Powerstat……...30
8. Block Diagram to represent Test Bed for CGI..……32
9. Block Diagram to represent Test Bed for WebRTC...……...33
10. Block Diagram to represent Test Bed for ZigBee Data Transfer..……35
11. CGI vs WebRTC transmitting a 50 MB file……..38
12. CGI vs WebRTC transmitting a 100 MB file……39
13. CGI vs WebRTC transmitting a 125 MB file……40
14. CGI vs WebRTC transmitting a 300 MB file……41
15. CGI performance for different File Sizes……..43
16. WebRTC performance for different File Sizes……..44
17. ZigBee vs Wi-Fi for 10KB File……45
18. ZigBee vs Wi-Fi for 50KB File……46
19. ZigBee vs Wi-Fi for 200KB File……...46
20. ZigBee vs Wi-Fi for 250KB File……...47
21. ZigBee Power comparison over Wi-Fi.....……..…….....48
22. Wi-Fi Power comparison over CGI File……....48
23. AJAX Power comparison for different segment sizes……...49

Figure Page

24. AJAX Power comparison for buffering segments of the same file……...49

Chapter 1

INTRODUCTION

In the past few decades technology has evolved to a very large extent giving the end user a better opportunity every day to be more creative and make mundane activities easier, simpler and more importantly, faster. The power of hardware resources has increased exponentially. Although the basic components of a computer more or less remain the same since its inception, the betterment of each component has driven to faster computation speeds and data access rates. For example, hard drives have now been replaced by Solid State devices and Flash drives [1, 2]. This has led to faster access of data within the devices. With the rise of smart handheld devices, the sizes of the hardware have further gotten smaller and more powerful [43, 44]. To support such devices, the software on these platforms have unarguably become lightweight and efficient in performing basic activities like capturing images, recording video and playing them back on compatible platforms or transfer data between modules and yet have better performance and less power consumption [4, 5]. Similarly, the potential of wireless networking to handle huge data transfer from one device to another has reached new heights [6]. Also, research and advancements in the field in Optical fibers to enable high speed data communications with bigger bandwidths and their interactions with wireless protocols [69] has paved way for better connectivity. Passive Optical Networks (PON) and gigabit Ethernet class WLANs have come into existence. All these developments have expanded the horizon for a user to think and develop new simpler and yet more

powerful platforms to solve some basic everyday problems. A simple example is when a user reaches within the radius of his Personal Area Network (PAN), such as within his router accessible area, he/she could have a small embedded system to automatically perform some task such as opening the garage door [45] and switch the lights on inside the house or turn on the thermostat inside his residence to a particular temperature. Another example is to have sensors in his residence to send an alert [46] to the user instantly if there is any suspicious activity such as a break in. Enhancements in cloud platforms and virtualization have also seen a tremendous growth in trying to solve problems [7, 8].

The developments in lightweight software technologies such as HTML5 [18, 19, 20], Asynchronous JavaScript and XML (AJAX) and newer Application Programming Indexes (API) such as WebRTC [11, 12, 15] have enabled data sharing easier than ever. With such hardware and software developments, the next most important challenge has been connectivity between such devices using wireless networking technologies and this has led to a new framework called the Internet of Things (IoT) [24,25,26,27]. The Internet of Things is basically a collection of sensors gathering data, working in tandem with hardware and software to provide an analysis of that data to solve problems. It uses a centrally configured system, which could be a server or a cloud platform to understand data and provide solutions or making decisions. Simple examples include creating weather maps using temperature sensors [47], gauging variables in locations not easily accessible, capturing images or controlling devices based on proximity and acting accordingly.

In this document, one such application on how video data is captured using a camera and it's transmission using Wi-Fi to a customized centrally managed Mongoose server to render the live video captured on a custom built Wireless Video Sensor Network Platform player (henceforth called WVSNP player) is discussed. The WVSNP-DASH player [34, 35] uses AJAX [61] to render the data onto the player built on HTML5. The focus of this document is on how the data is transmitted from the client to the server host using CGI scripts [31] and from one peer to another using WebRTC [16] and the performance in terms of power consumed and latency in transmitting the data.

CGI is a standard used to create executable binary scripts on a server that respond to a call from a client side script that understands and generates web pages dynamically on a web browser. The client request could be either from a user accessing a HTML page or an internal JavaScript call to the CGI executable on the server.

WebRTC is an acronym for Web Real Time Communications which is a multimedia platform introduced by Google in 2011 for browser to browser communications. This platform enables multimedia and file transfer between peers [48]. This document discusses how an open source application built using WebRTC works and how it can be used to transfer data from one peer to another without the involvement of an external server platform and compare between the performances of gathering data using CGI and WebRTC.

In Chapter 2, the role of Internet of Things in communications and how interactions between modules keep users informed and how versatile IoT can be, is discussed. Also, importance of lightweight technologies in embedded systems, specifically the

Application layer in the OSI architecture is detailed. Chapter 3 discusses how Web applications for Wireless Video Sensor Network platforms work and how video data is rendered onto the custom built WVSNP player along with how the existing CGI scripts work to invoke sensors to take snapshots or capture video data and save it into the Mongoose server. Chapter 4 elaborates the integrated WVSNP dashboard and explains other functionalities enabled Chapter 5 describes about WebRTC, the new age browser platform for enabling data transfer between peers and its advantages. Chapter 6 discusses the testbed using Powerstat, an open source tool to read the power consumption and latency differences between fetching files using CGI and WebRTC. Also, it shows the test bed for using ZigBee modules for file transfer. Chapter 7 evaluates and compares the performances between each mode of data transmission and based on them the recommendations are outlined.

Chapter 2

INTERNET of THINGS

Today's technology hugely relies on networking and vice versa. The interconnection of physical sensors that are embedded with small electronic devices to interact with software and provide data for analysis is the prime motto of this IoT framework. This framework is now gaining a lot of momentum due to the fact that devices have now gotten much smarter than before and are integrated with better sensors and software applications that can invoke and read data from those sensors and provide an analysis or better decision making using such data.

Interaction between devices using such data has paved ways for new business models and creating better economic scenarios [28]. For example, integrating a camera at a high security location for video surveillance is a scenario existing from long, but having cameras secure residential areas whenever the resident is leaving the house (invoking a camera once the mobile device used by the house owner is not in his residences' Wi-Fi range) is a novice, which is now cost effective and easier to handle [49]. Tracking our physical stress levels by using different types of sensors like accelerometers have reached new heights [50]. The accuracy of sensors, smaller sizes of physical hardware systems and easier connectivity of these devices to the internet using Bluetooth, Wi-Fi or ZigBee [56] modules have enabled such low cost and efficient mechanisms to sustain and grow the standard of living. These recent developments have paved way to the big technology organizations such as Intel Corp., Dell and Freescale semiconductors (now NXP) [51] to

take interest and invest in Healthcare solutions [29] and create specific products to solve problems on the IoT platform.

The growth of data analysis has been very crucial to such developments. Big data solutions using data analysis technology platforms such as Hadoop, R, and Matlab are playing a major role in enhancing the creativity in providing better and faster solutions.

Data storage, cloud computing and virtualization [7, 8] have enabled technicians working on such platforms in being more free in exploring new solutions. Organizations providing cloud storage spaces and solutions have enabled people to focus on the technical aspects of solving a problem while the hardware and networking logistics are taken care of very easily by these organizations. One such example is Amazon Web Services. Businesses that run on handheld devices such as social networking applications, messaging services and other innovative startup ideas have gained immense acceptance that the platforms that support them need more data storage spaces. Services that offer such storage space solutions simply increase the number of hardware devices that support the networking speeds to cater to the increasing number of users [62].

The most important factor for better performance in terms of speed and power consumption largely depends on the type and size of the software going into these IoT peripherals [52]. Linux is an open source operating system which gives a developer the freedom to customize it according to the needs of the user [36]. For example, the WVSNP dashboard (discussed in detail in the further sections) has the capability to receive data wirelessly from small embedded peripherals. These peripherals simply carry a small Linux kernel over which the necessary libraries to perform certain set of specific tasks are embedded such as capturing video data or taking snapshots using open source

multimedia platforms such as Gstreamer [37] and FFmpeg [38] and save them to the mongoose server [39]. For the case discussed here, Yocto kernel distros [53] are used. The size of the distro (Linux distribution) is under 200 MB which is comparatively a very small size.

The WVSNP dashboard is a browser agnostic integrated application that runs on very lightweight technologies such as HTML5 and JavaScript (JS) on the client's side. Every browser that exists today is JavaScript enabled to understand the client requests. Earlier versions of HTML were not capable of handling audio or video using simple tags. To create a media element, JavaScript was used to render it on a player or an external plugin such as flash was to be installed which meant the overhead of the code to do so was much larger that it is now. With the introduction of HTML5, the handling of these have become much easier.

A simple example for audio and video tags:

Video tag:

```
<video width="300" height="200" controls>
    <source src="home/harsha/movie.mp4" type="video/mp4">
    <source src="home/harsha/movie.ogg" type="video/ogg">
    Your browser does not support the video tag.
</video>
```

Audio tag:

```
<audio src="home/harsha//test/audio.ogg">
<p>Your browser does not support the <code>audio</code> element.</p>
</audio>
```

The above code snippet simply makes sure that a video element and an audio element is rendered onto the html page with the source from the local filesystem. These tags were introduced very recently in HTML5 and replace the earlier <object> tag that was typically used to render multimedia on the browsers. Also, a HTML5 <canvas> tag is introduced to modify graphics on the html page. This tag is simply a graphics container that renders changes via a scripting language which is generally JavaScript since the browser understands it easily.

With HTML5 becoming an industry standard to enable a more interactive browser developments, there have been drastic visible changes within applications to incorporate the handling of multimedia elements using this technology. For example, YouTube, which earlier required flash plugin to be manually installed on the browser to enable video and audio streaming, has almost completely moved to HTML5 [40], which simply means that the browser now does less amount of work to render the video on the player. The mobile game application industry vastly uses HTML5 to draw images on the applications. It also provides developers with important tools such as GeoLocation API, Canvas drawing and supporting CSS3 on browsers. Offline storage of data in the browser received is another advantage with HTML5 which gives the option to store files and playback on the video player to support streaming data while the connectivity to the internet is down.

Such developments have led to the rise of JavaScript enhancements and opening new corridors for innovation in media data transmission. WebRTC (discussed in detail in further sections) is one such platform that enables peer to peer communication without

the presence of an intermediate service provider. Google hangouts is an application used for data transfer or video conferencing which supports WebRTC. Since this is an open source project maintained by Google developers any user can use it for their application to communicate with their peers.

With these developments, Internet of Things framework has proven its ability to become the next important platform for development. With rise of new technology, there are obviously new issues faced, of which security is the one the primary ones. Data such as Video and Image transfer over the internet needs to be encrypted at all times to maintain high security standards. Sensors that capture images and video needs transformation and encryption at all times to make sure that they are safe and are reaching the intended recipient without them being manipulated. Research in this direction [71, 72, 73, 74] have given a better understanding of security that could be implemented in multiple layers in the OSI architecture scheme. WebRTC is a platform that answers this issue. Further sections will discuss this in detail.

Chapter 3

WEB APPLICATIONS FOR WIRELESS SENSOR NETWORKS

There are multiple wireless protocols to enable data transmission between devices. Based on the type of environment and the necessity, these standards are used accordingly. As the speeds of the wireless standard increases, the power consumed by the device also increases. Examples are Wi-Fi, Bluetooth and ZigBee. With the help of the application layer in the Open System Interconnection (OSI) architecture, the performance of the system can be bettered using technologies that work with it. HTTP is one such protocol.

The WVSNP player is based on this protocol which is contemporary to the existing video rendering standards available in the industry, namely, HTTP Live Streaming (HLS) [41] and MPEG-Dynamic Adaptive Streaming over HTTP (DASH) [42]. Previous studies on the scalability of the video traffic over the internet and their quality characterization of the video segments using MPEG file formats have given a new perspective in providing heterogeneous and scalable libraries for long scale video traces [70]. The basic difference between the aforementioned protocols and the WVSNP-DASH [34, 35] is the way the file is named i.e. all the information of the files being recorded or stored is encoded into the file rather than having a different name to it and decode the file name to fetch it for playback. The data that is being transmitted contains the entire information of the file that is being rendered and the number of chunks it has been segmented into. In contrast, the HLS and WVSNP-DASH players use the method of renaming files with a protocol

specific naming convention which later is decoded while fetching the data back on the players using these protocols. All of this takes place once the data is being recorded or when is being fetched into the respective players. It has been determined that the performance by the WVSNP player is better than player implementing the HLS protocol and almost equaling the MPEG-DASH protocol [4]. The following is the naming convention used for this player:

<filename>-<maxpresentation>-<presentation>-<mode>-<maxsegment>-

<startindex>.< ext >

Filename represents the initial name given to the file before data is recorded. It could either be a URL to a file or a location in the filesystem.

Maxpresentation is an integer value that depicts the maximum segment that is buffered into the player.

Presentation is another integer that depicts the current segment being rendered onto the player at that instant.

Mode tells the user whether the video rendered is live (LIVE) or video on demand (VOD).

Maxsegment is an integer value that depicts the maximum number of segments that will be recorded or available for the corresponding video rendered on the player.

StartIndex gives the current index being rendered onto the player at that instant. Based on the segment size this index increments.

Ext is the media source file type. If the browser or the player is incompatible with the media source extension then it would return an error when rendered.

3.1 AJAX role in fetching data onto the WVSNP Player

The representation of the WVSNP-DASH files in the previous section is the resultant of implementation software using different high level front end technologies. These technologies put together can be simply called AJAX (Asynchronous JavaScript and XML). This web application is browser agnostic and as the name suggests it is asynchronous in the sense that the video player has the capability to fetch and render any segment from the server that has been captured already. In general, AJAX is a term for a set of technologies that are used to fetch data from the server using XML HTTP Requests (XHR) to load the new data into the HTML page without having to refresh the entire page. AJAX is not a standardized term. For a developer, it is called an XMLHttpRequest object, which is basically a HTML code working with JavaScript to send a request to a specific server, where the request carries some data as an XML (Extended Markup Language) or a JSON (JavaScript Object Notation) that the server understands based on which it responds to this request by sending some data. The server side has scripts to understand the request received and usually fetches data as a response to the request.

The server used for this application is called Mongoose which is an open source lightweight platform by an organization Cesanta [39] that is generally recommended for embedded applications that involve web Graphical User Interface (GUI) on embedded devices. The Mongoose web server runs on the embedded peripheral. The web server's binary points to the location in the filesystem based on the path mentioned in its configuration file mongoose.conf. When the server is up and running, the client enters the IP address of the server and accesses the WVSNP player through port 8080 and uses the wvsnpdash.shtml to invoke the player. Below is the face image of the WVNSP player.

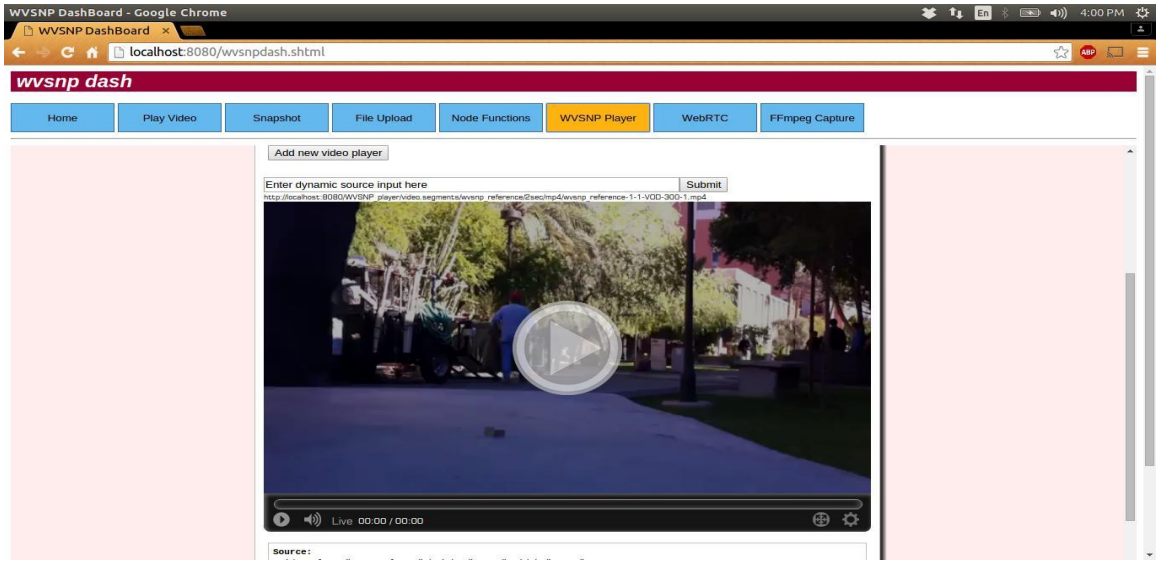


Figure 1: WVSNP-DASH player built on HTML5

The server runs the player on the client's machine and the most important point to be noted here is that the browser on which the player is being invoked should be JavaScript enabled, which generally is the case, unless the user intends to disable it.

In the image above, a default video is rendered which is already present in the server and to which the player currently points to. When the user clicks on the play link, an XMLHttpRequest (XHR) is made to the server as shown below.

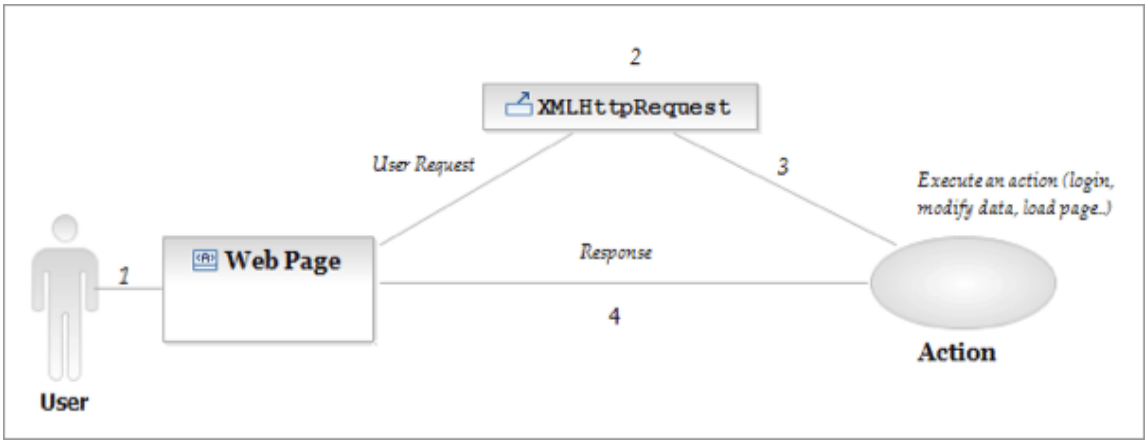


Figure 2: Flow chart showing client interaction with server

As shown in the figure above [63], the XHR carries the request from the HTML page using the JavaScript action defined for the request (as a form on HTML or an internal JavaScript call using Get or POST methods). In the WVSNP player, since HTML5 is used for media source interactions, the <video> and <canvas> tags are used for easy rendering on the player as discussed in the earlier sections. Once the play button on the WVSNP player is invoked, an XHR request is sent to the Mongoose server requesting for the data to be sent to the player from the location that is mentioned in the <video> tag. Below is the source that is referenced in the source code.

<source class="wvsnp_src"

src="video.segments/wvsnp_reference/2sec/mp4/wvsnp_reference-1-1-VOD-300-1.mp4" />

A ten minute long ASU video is used to display on the player as VOD and it is segmented into two second video segments which gives 300 segments from the single file. As discussed earlier about the video file format in this player, the segment number one which is two seconds long is fetched into the player as a response to the XHR and is rendered on the <video> tag, which then is played on the player. While this is being played, the video player is designed in a way to store the upcoming segments in the browser's cache using the file system API [64]. The sandboxing of data into the browser's cache is an important feature for the WVSNP player as this API enables fetching of upcoming segments (forwarding) or already played segments (rewinding) into the player. So basically, while the player is initiated, there are two tasks being performed simultaneously which are, fetching data using the Filesystem API and segment rendering

on the WVSNP player. The performance results for the player due to this have been presented by Lukas [4] and Tejas [5] in their respective documentation. Unfortunately, at this point, only chrome browser and the Google community is supporting the FileSystem API and is not being considered by any other major browsers and is not being considered as a World Wide Web Consortium (W3C) standard. Other browsers have shown very less interest and have not provided support over their browsers. At the time of developing, the player was enabled on all major browsers, but at this point, only Chrome supports it.

The documentation by Tejas and Lukas show that the dynamic adaptive streaming using this WVSNP player is better than the existing HLS (Apple) and MPEG-DASH based video players in terms of the CPU load and power profiling for various video segment sizes.

Chapter 4

INTEGRATED WVSNP-DASHBOARD

This section explains the various features available in the WVSNP-DASHBOARD (dashboard henceforth) other than the home screen, which shows the WVSNP logo. Before going further, it is essential to discuss how Common Gateway Interfaces (CGI henceforth) works as most of the features discussed in this section are based on CGI scripts residing in the mongoose server. CGI [31] is a W3C standardized scripting language to interface web servers with client side web applications to generate dynamic web pages. These programs reside in the web server and are simply called CGI scripts [30, 32, 33]. A CGI script can be written in any programming language. In the WVSNP player, all the scripts have been coded in the C language. Considering C as the basis, a CGI script is a binary program using CGIC header file, which is the C library for generating them. For a C programmer, it is a preprocessor directive that has the capability of creating World Wide Web applications. A CGI script has the capability to parse 'form' data from a HTML page, accept client generated data that has been internally sent to the script via a ‘GET’ or a ‘POST’ method using AJAX or jQuery and upload files that are instantly generated (such as capturing an image from a web cam) or from the clients file-system itself, among many functionalities.

The basic premise of creating such a binary is as simple as compiling a simple c file on a command terminal.

The following is a generic command that is used in the command line prompt to create

such a binary.

gcc cgic.h cgic.c inputCFile.c -o outputCGIBinary.cgi -lm

In the command above, a file named inputCFile.c is compiled to create a binary called outputCGIBinary.cgi. As the gnome compiler 'gcc' does not understand the specific CGI tasks performed in the C file, it is mandatory to use cgic.h and cgic.c which are the pertinent files. This creates the binary script in the filesystem as specified. This is synonymous to compiling a normal c file but the only difference is the fact that the extension to the binary is .cgi. Some of the basic pre written CGI header functions [33] that have been used to create these functionalities are discussed below.

- **cgiMain**: This is synonymous to the main function in every c file and is a must in every cgi script generated.
- **cgiContentHeaderType**: This function is used to output the header type of the file to the user. **Ex**: text/html, image/jpeg.
- **cgiOut**: This is a file pointer to the CGI output which is similar to the pointer given in a C file to write/append text to a file.
- **cgiFormSubmitClicked**: This is a function used to check if a particular submit button has been clicked on the user interface page by the client. It is similar to the submit button in JavaScript
- **cgiFormSuccess**: This is a pre-defined result code reference that shows whether a particular action is successful or otherwise.
- **cgiHtmlEscape**: This function removes any unwanted characters that are used in the HTML tags such as ‘&’, ‘ >’, ‘<’ symbols to make sure that the content is

mark-up free.

The various functionalities of the dashboard are discussed below.

Snapshot

Snapshot is a functionality based on CGI that enables the client space to capture an image using a hardware that is integrated to the device. In the dashboard, when the user opens the snapshot tab, he/she is provided with a text field to enter a name to the image to be captured after which he/she will click on the snapshot button which is basically a submit button. On click, the CGI invokes a shell script in which is a Gstreamer command to capture an image. The image is captured from the integrated hardware camera (webcam or CSI). The CGI script invokes a bash script 'snapshot.sh' which launches a gstreamer command to capture the image. The command is given below.

gst-launch-0.10 v4l2src ! ffmpegcolorspace ! pngenc ! filesink location=$1$2.png

By default, all the images have a .png extension which can be changed in the shell command. The images are saved in a ‘Snapshots’ folder in the server filesystem.

File upload

File upload on the dashboard is used to upload a file to the mongoose server. The basic HTML content provided to the user are two buttons namely 'Choose file' and 'Upload'. ‘Choose file’ invokes the functionality to select a file from the user's file system, while ‘Upload’ saves the file in a folder 'uploaded' in the server filesystem.

A specific function called ***cgiFormFileRead****(cgiFilePtr cfp, char *buffer, int bufferSize, int *gotP)* is used for this function which reads the size of the file that was previously opened and stores it in the *gotP variable. After upload, the CGI script returns the file

name, file type and the location where the file is stored to the client side as a HTML.

Node Functions

This functionality requires the use of two ZigBee modules [54, 55], one as a transmitter and the other receiver. ZigBee modules are small hardware peripherals that are used to transmit data within a range of 10 to 100 meters depending on the geography. ZigBee is an IEEE 802.15.4 wireless protocol which is defined for establishing simpler and economical Wireless Personal Area Networks (WPAN). The advantage with ZigBee modules is that they can from a mesh network and transmit data between intermediate devices to reach longer distances. Also data transmitted using these peripherals are encrypted which gives more security and are also defined to have longer battery life, with a data rate between peripherals in the order of 20 to 250 kb/ps. The modules are configured to either transmit or receive the data within the network.

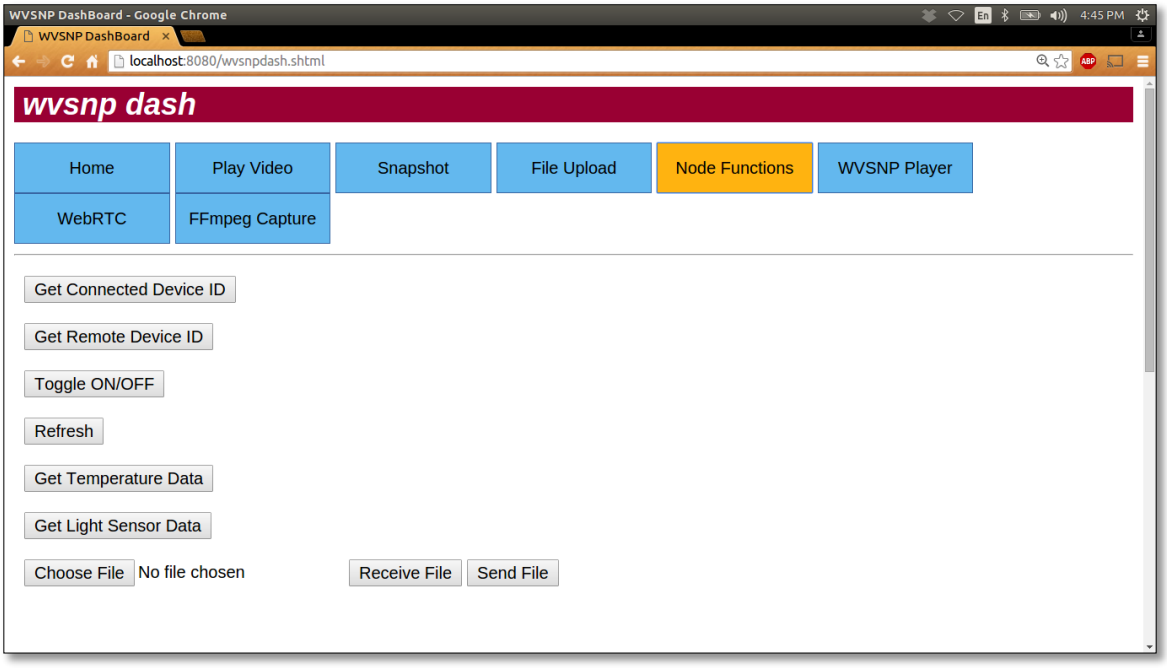


Figure 3: Node Functions on the WVSNP dashboard

In this application, the CGI script renders HTML content with different options. Based on the connected ZigBee modules, the user can request the ID for the connected peripheral by clicking ‘Get Connected Device ID’ and the remote peripheral by clicking ‘Get emote Device ID’. The ‘Toggle on/off’ is a submit button which on click, would send data to the remote ID to toggle between the high and low (based on the kind of sensor used). Example in this case is toggling between high and low on an LED connected to one of the ZigBee modules. Using sensor data to represent to gather light and temperature information is available using the integrated ZigBee module to the sensor. The most important feature pertinent to this functionality in the node functions is file transfer. A file can be chosen to be sent from the transmitter to the receiver. ZigBee data transmissions between modules is enabled by the preprocessor directive xbee.h. While compiling the c file the xBee functionalities included in xbee.h are called to create a CGI binary for the corresponding functionalities.

The entire code of the Node Functions is defined in a file buttonfunctions.c and is compiled using the below command to receive obtain the binary script for this file.

gcc buttonfunctions.c -o buttonfunctions.cgi cgic.c -lxbee -lm -L libxbee.so

The -lxbee is the xBee or ZigBee library interface for C required to compile any functionality involving xBee written in ANSI C. The usage of this code is basically dependent on the xBee hardware connected to the embedded peripheral and should also have the right transmitter or receiver MAC addresses in the corresponding code to identify each of them. A user who expects to receive the file clicks on ‘Receive File’ which tells the connected ZigBee to keep waiting for the file. The user who sends the file, clicks ‘Choose File’ and then clicks ‘Send File’. The ZigBee transmits the data and then

the receiver receives it. Once the transmission is complete, this page shows the name, size and the number of chunks the data has been transmitted as.

FFmpeg_capture

This feature is another most important in the dashboard alongside the WVSNP player. This functionality is used to capture video data from the corresponding integrated hardware camera. Below is the image showing the fields in this CGI rendered HTML page.

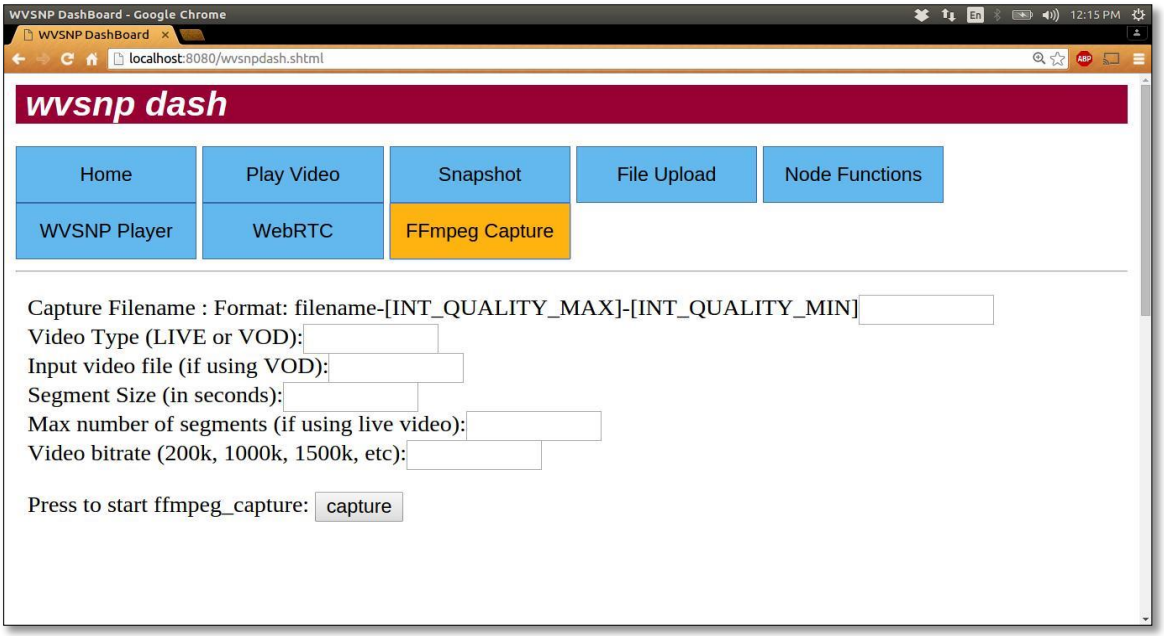


Figure 4: Ffmpeg capture function on the dashboard

This page requests the user to enter the fields depending on the type of video he/she wants. The filename is expected to be in the same format that the WVSNP protocol uses which has been explained earlier in this document. Example: WVSNP-1-1 where

WVSNP is the file name. If the video to be recorded is from the camera, then LIVE is to be entered and if it is video on demand, VOD is to be entered in the video type text field. Apart from the other fields which have been explained earlier, the new field that is seen here is the Video bitrate which is the quality of the video that the user wishes which can be one of the three options namely 200k, 1000k and 1500k. On clicking capture, live video is recorded which is enabled by FFmpeg. This basically is a multimedia framework that enables encoding, decoding, streaming and filtering any kind of multimedia data that is understood by man and machine. The FFmpeg command that is used in this functionality for capturing video data and storing it in the server is given below.

ffmpeg -f v4l2 -i /dev/video0 -f segment -segment_time $1 -reset_timestamps 1 -s 640x360 -c:v libx264 -r 24 -g 24 -map 0 -b:v $2 -pix_fmt yuv420p /www/WVSNP_player/video.segments/live/$3-$4-$5-%0d.mp4

For this command to run successfully, the client peripheral on which the camera is integrated need to have the FFmpeg plugins installed in the kernel to capture and format the video accordingly. The x264 [56] library used in the command above is free software library for encoding video streams in MPEG-4 [57] advanced video coding format.

Once the capture of the video data is started, the frames captured are saved based on the indexes and every segment size is same as the other and all the files are saved in server filesystem URL **'/WVSNP_player/video.segments/live/'** folder. The rendering of these video segments on the player can be done by simply entering this URL in the text field of the WVSNP player and the player starts fetching the files from the filesystem one segment after the other as explained in the earlier sections of this document.

Chapter 5

WEBRTC: P2P COMMUNICATIONS

WebRTC (Web Real Time Communication) is a tool using JavaScript based application programming indexes (API) that enable browser to browser communication for live video, audio and file data sharing without having to install external sources, applications or plugins [9, 10, 11, 12]. The invention of this platform occurred in 2011 by Google developers and is an open source platform to utilize data sharing between peers [15, 16]. Currently the API is being reviewed by W3C to make it an industry standard while it is being extensively supported by Chrome and Firefox browser communities.

In the world of technology, with interactions between any two entities such as peer to peer, one class to another, one forum to another, one industry to another or an entire society coming on a common platform such as social media, today the amount of data exchange between them is the highest ever. The avenues that provide such services have also increased. But at every point in time, the usage of external service providers have never been replaced. For example, Skype is a proprietary product of Microsoft, Facetime is a product owned by Apple. Although such products are very compact and work very well, they do not give the end user the freedom to create media communications or p2p file transfers at will without having these third party vendors. WebRTC [13, 17] is a tool that gives the end user this opportunity to create applications that use any type of data transfers with very minimum overhead and the most important feature is that it does not

require any intermediate server. Since this is a free platform, many major organizations such as WhatsApp, Facebook messenger and Google Hangouts support and use this API. A lot of online multiplayer platforms have enabled their games to support WebRTC to have players interact with each other while the game is being played. There are three basic APIs that WebRTC uses which are MediaStream (getUserMedia), RTCPeerConnection, RTCDataChannel. The API that is of interest here is the RTCPeerConection as this is the API required to enable p2p file transfer.

RTCPeerConnection uses the STUN, TURN and ICE protocols to establish connection with a peer. These servers are Internet Engineering Task Force (IETF) standards to tackle the Network Address Translation (NAT) issues [65] across the internet when a peer is trying to connect to another. The basic explanation of these standards are given below.

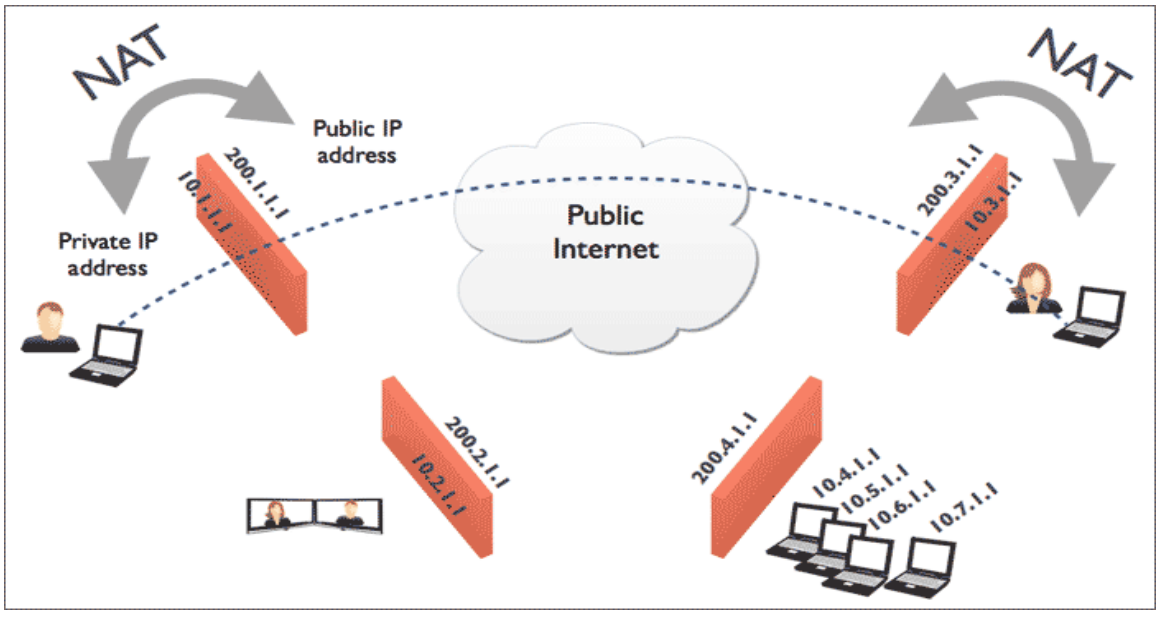


Figure 5: NAT issues over networks.

STUN:

This is an acronym for Session Traversal Utilities for NAT. When a packet is transmitted from node A to node B, the packet might have to traverse across multiple types of networks to reach its final destination. But there are occasions where the data in the packet is incompatible to read by some routers or access points in which case, the use of STUN server would resolute the address for the network to understand where the data packet needs to be sent.

TURN:

This is an acronym for Traversal Using Relays around NAT. This protocol is used to traverse a packet through the Internet discovering unknown paths to reach the destination. This protocol is only used when the data is unable to reach a destination due to networking issues. Because this is an additional step to contact a nearby server resource to identify the destination, it is considered an overhead in terms of power and time. It is only used during absolute necessity

ICE:

This is an acronym for Interactive Connectivity Establishment. This protocol uses the aforementioned two protocols to identify networks between peers to transfer data reliably. This is similar to identifying the network topology by switches to send the data.

5.1 Serverless-WebRTC application:

The application of interest here is an open source one [58] which establishes connection between peers to transfer data between them. The applications uses technologies such as jQuery and JavaScript in doing so. The most important feature of this application is it does not require any server to run it. The html file pertinent to it runs the application on the fly, which makes it easy to use the application.

To simply run the application and establish connection, the user opens serverless-WebRTC.html. This invokes the JavaScript within the application that prompts the user to create a data channel to establish connection with a new user or join an existing one. If the user wants to create a new connection, the 'Create' button is clicked. This generates a session variable which is a JavaScript Object Notification (JSON) data with the content that describes the MAC address, the IP and other details of the network he/she is in. This data needs to be sent to the user who wishes to join the session established by using other modes of data transmission (messaging). The other user, who then uses the 'Join' button, enters this JSON data into the modal pane to establish connection and sends back the JSON generated by him to the primary user to establish connection.

This basic handshaking between two users is to establish the connection and is termed 'signaling' in WebRTC jargon. Signaling is a very important event in WebRTC platform. This is left to the user to design a signaling event according to their convenience. This application uses creation of JSON content. Other applications such as online multimedia players are required to join specific ‘rooms’ to take part in the activity. Other applications create an ID that the user who wish to join that interaction needs to use. The signaling is like a passkey used to allow user acceptance into the session. Once the user uses this

session key, he/she belongs to that group. Once the user exits the session for any reason, the session needs to be re-established between them and the other users to re-initiate any kind of interactions. This protocol is JavaScript Session Establishment Protocol (JSEP) [59]. Below is the figure that shows how signaling is performed in this application.

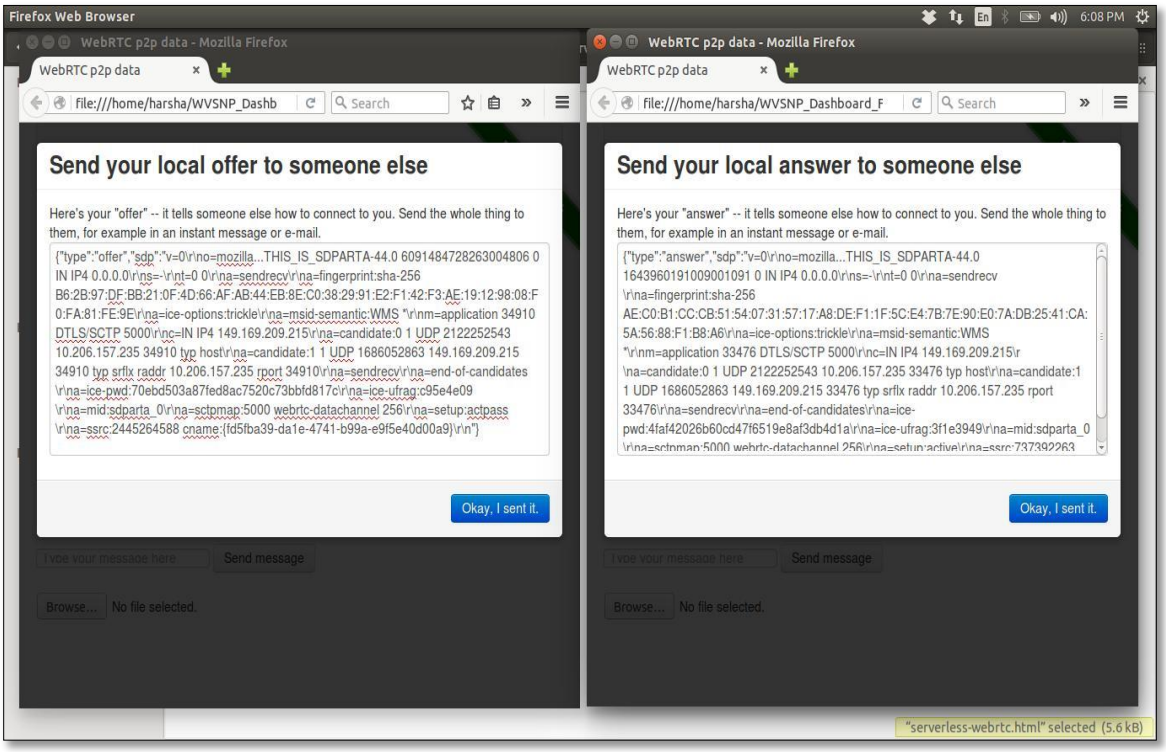


Figure 6: WebRTC: Left is the host & right window is the client

In the above figure, the left window is the host and the right window is the receiver. Both of them have the JSON content created in their respective windows which is exchanged between them called 'signaling' and hence the connection is established.

The current efforts in this application are going on to create the JSON session variable by each user trying to connect to the other and automatically save the data into the

mongoose server so that the handshake signaling is automated.

Signaling is not defined by WebRTC to avoid redundant data while using the APIs and also to create versatility in using varied technological standards. Different applications might want to use different protocols in establishing signaling and hence, it is left to the application owner to do so, which is a standard named as JavaScript Session Establishment Protocol (JSEP) [59].

Chapter 6

TEST BED EXPLANATION

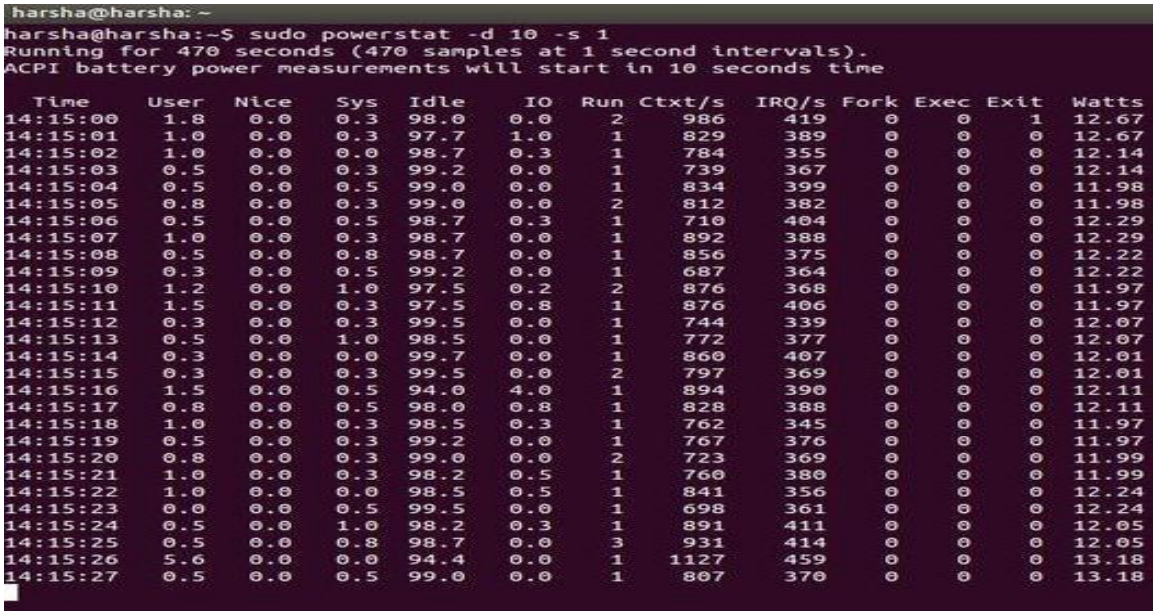
In this section, the power performance and latency in transmitting the file from one peripheral to another is described. There are two types of measurements described in this section. The first one is describing the comparisons in power consumed and latency of data transfer between CGI and WebRTC over Wi-Fi. The second comparison is between file transfer over Wi-Fi and ZigBee for the same variables.

6.1 Comparison between CGI and WebRTC:

The different video file sizes used for comparison are of sizes 50MB, 100MB, 125MB and 300 MB. The comparisons are made between the CGI based application and WebRTC based application. Since Wi-Fi is used for comparisons in this case and both CGI and WebRTC measurements are taken using the same parameters i.e. within the same network topology and keeping the transmitter and receiver within the same distance, it is safe to assume here that when the data is transmitted, the packet size in both the cases will remain the same.

Powerstat is the basic tool used to record power consumption and latency in data transfer. It is an open source Linux platform to show the power consumed by an embedded platform hosting a Linux kernel. It is generally used to measure the change in power consumption before and after a system upgrade or while a specific application is up and running to see the system's performance during that time. Below are some of the

important features shown while recording data on Powerstat.



```
harsha@harsha: ~
harsha@harsha:~$ sudo powerstat -d 10 -s 1
Running for 470 seconds (470 samples at 1 second intervals).
ACPI battery power measurements will start in 10 seconds time

  Time     User   Nice    Sys   Idle      IO  Run Ctxt/s   IRQ/s Fork Exec Exit   Watts
14:15:00    1.8    0.0    0.3   98.0     0.0    2    986     419    0    0    1   12.67
14:15:01    1.0    0.0    0.3   97.7     1.0    1    829     389    0    0    0   12.67
14:15:02    1.0    0.0    0.0   98.7     0.3    1    784     355    0    0    0   12.14
14:15:03    0.5    0.0    0.3   99.2     0.0    1    739     367    0    0    0   12.14
14:15:04    0.5    0.0    0.5   99.0     0.0    1    834     399    0    0    0   11.98
14:15:05    0.8    0.0    0.3   99.0     0.0    2    812     382    0    0    0   11.98
14:15:06    0.5    0.0    0.5   98.7     0.3    1    710     404    0    0    0   12.29
14:15:07    1.0    0.0    0.3   98.7     0.0    1    892     388    0    0    0   12.29
14:15:08    0.5    0.0    0.8   98.7     0.0    1    856     375    0    0    0   12.22
14:15:09    0.3    0.0    0.5   99.2     0.0    1    687     364    0    0    0   12.22
14:15:10    1.2    0.0    1.0   97.5     0.2    2    876     368    0    0    0   11.97
14:15:11    1.5    0.0    0.3   97.5     0.8    1    876     406    0    0    0   11.97
14:15:12    0.3    0.0    0.3   99.5     0.0    1    744     339    0    0    0   12.07
14:15:13    0.5    0.0    1.0   98.5     0.0    1    772     377    0    0    0   12.07
14:15:14    0.3    0.0    0.0   99.7     0.0    1    860     407    0    0    0   12.01
14:15:15    0.3    0.0    0.3   99.5     0.0    2    797     369    0    0    0   12.01
14:15:16    1.5    0.0    0.5   94.0     4.0    1    894     390    0    0    0   12.11
14:15:17    0.8    0.0    0.5   98.0     0.8    1    828     388    0    0    0   12.11
14:15:18    1.0    0.0    0.3   98.5     0.3    1    762     345    0    0    0   11.97
14:15:19    0.5    0.0    0.3   99.2     0.0    1    767     376    0    0    0   11.97
14:15:20    0.8    0.0    0.3   99.0     0.0    2    723     369    0    0    0   11.99
14:15:21    1.0    0.0    0.3   98.2     0.5    1    760     380    0    0    0   11.99
14:15:22    1.0    0.0    0.0   98.5     0.5    1    841     356    0    0    0   12.24
14:15:23    0.0    0.0    0.5   99.5     0.0    1    698     361    0    0    0   12.24
14:15:24    0.5    0.0    1.0   98.2     0.3    1    891     411    0    0    0   12.05
14:15:25    0.5    0.0    0.8   98.7     0.0    3    931     414    0    0    0   12.05
14:15:26    5.6    0.0    0.0   94.4     0.0    1   1127     459    0    0    0   13.18
14:15:27    0.5    0.0    0.5   99.0     0.0    1    807     370    0    0    0   13.18
```

Figure 7: show the measurement of system vitals in Powerstat.

Time: The start time at which the power is being captured for the device.

User: This column gives information about the CPU usage by the current process initiated by the user.

Idle: The percentage of CPU that is idle at that point in time.

Nice: This is a variable that shows the priority of the current process given by the CPU. Based on this value, the CPU gives less or more time to the current process.

Sys: This is the amount of work done by the CPU or simply put, this is the CPU time used by the system kernel residing on the computer on which Powerstat is running.

Idle: This variable depicts the amount of time the system has stayed idle, or the amount of time the CPU has no processes to execute.

IO: I/O indicates Input Output which is basically the interaction between the hardware of the system and the kernel. This refers to I/O wait, which means that the system would interact with the hardware on the system, but until the kernel receives a response from the hardware it stays idle, or executes some other process threads.

Run: Run shows the number of running processes in the system at any given point of time.

Ctxt/S: This is a variable that depicts the number of context switches that occur in one second. Context switching is basically switching on to executing another process that has higher priority than the currently being executed process.

IRQ/s: IRQ is a hardware line that sends requests to the CPU to interrupt a particular task to execute a task with higher priority. This variable shows the number of such requests per second.

Fork: This is a process of executing a thread to create a copy of the process that is currently running.

Watts: The power usage by the device at that point in time given in Watts.

In this case, all the measurements are recorded on a Dell Inspiron-14Z laptop with Intel® Core™ i3-2350M CPU running at 2.30GHz with 4GB RAM and 64 bit processor on which Linux Ubuntu 14.04 kernel is used, and all the measurements are taken while the client is running on DC power. All the applications running on this system were on Mozilla Firefox version 44.0 for Ubuntu to maintain consistency with the data recorded.

The following is the command used to record data:

harsha@harsha:~$ ***sudo powerstat -d 10 -s 1***

The command simply starts the Powerstat tool after a delay of 10 seconds and starts recording the system vitals' measurements after every 1 second. The following is the block diagram to show the test bed used for CGI data transfer.

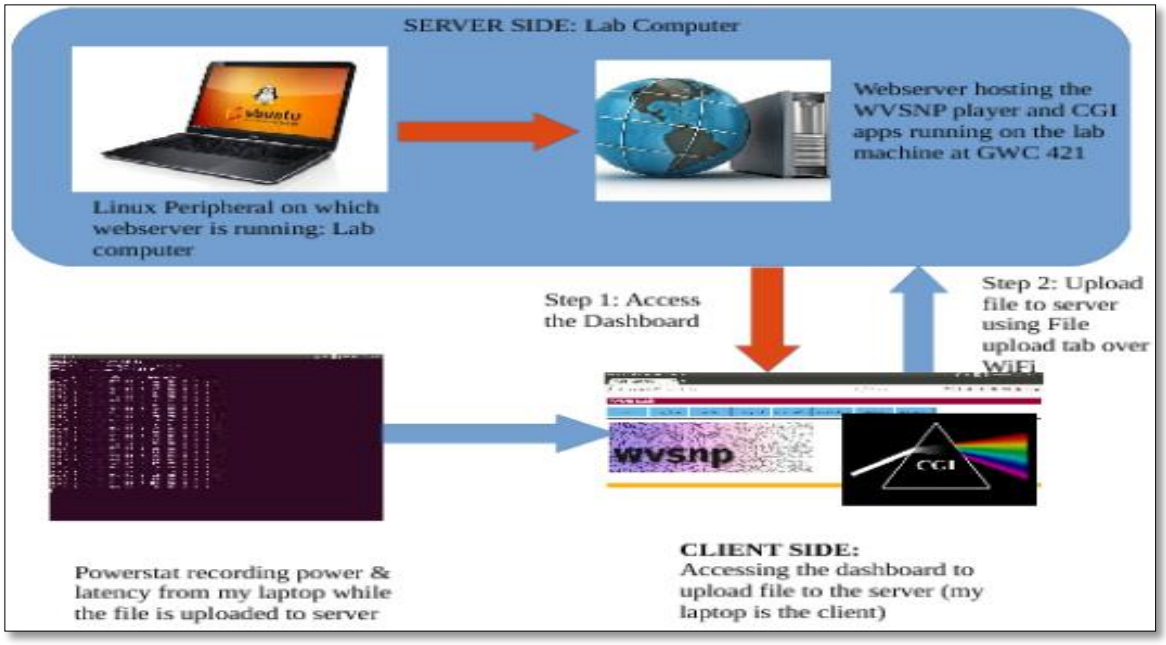


Figure 8: Block diagram to represent test bed for CGI

Another software tool that can be used to measure power consumed by a system is Powertop[68]. It is similar to what Powerstat does, but this tool requires comparatively more time to show the wattage of the system. Powertop also provides information per core, in the sense that the user interface for powerstat shows collective information which is the not the case in powertop. Also, this software system is believed to report more power savings than what the system actually does. On comparing with Powertop, Powerstat gives more accurate results with the wattage consumption and hence the reason to choose it over the former.

Data Collection for CGI:

The mongoose webserver is run on a Dell desktop on which Linux Ubuntu 14.04 kernel

is present. It is connected to the internet via Ethernet while the laptop which intends to send the data is connected to internet via Wi-Fi. The user accesses the dashboard on the server via the **IP address:8080** of the machine on which the server is running. The port on which the communication between the host and the client takes place is 8080. The user then accesses the 'File Upload' functionality from the dashboard to upload a file and send it to the server. Before sending the file, Powerstat is invoked with the command explained above to have it ready with recording the measurements once the data is uploaded. Once the file is uploaded and received, a few more measurements are recorded from Powerstat to evidently show the distinction between the power consumed while data is being transferred and when it is not. The data is then saved into a text file.

Data Collection for WebRTC:

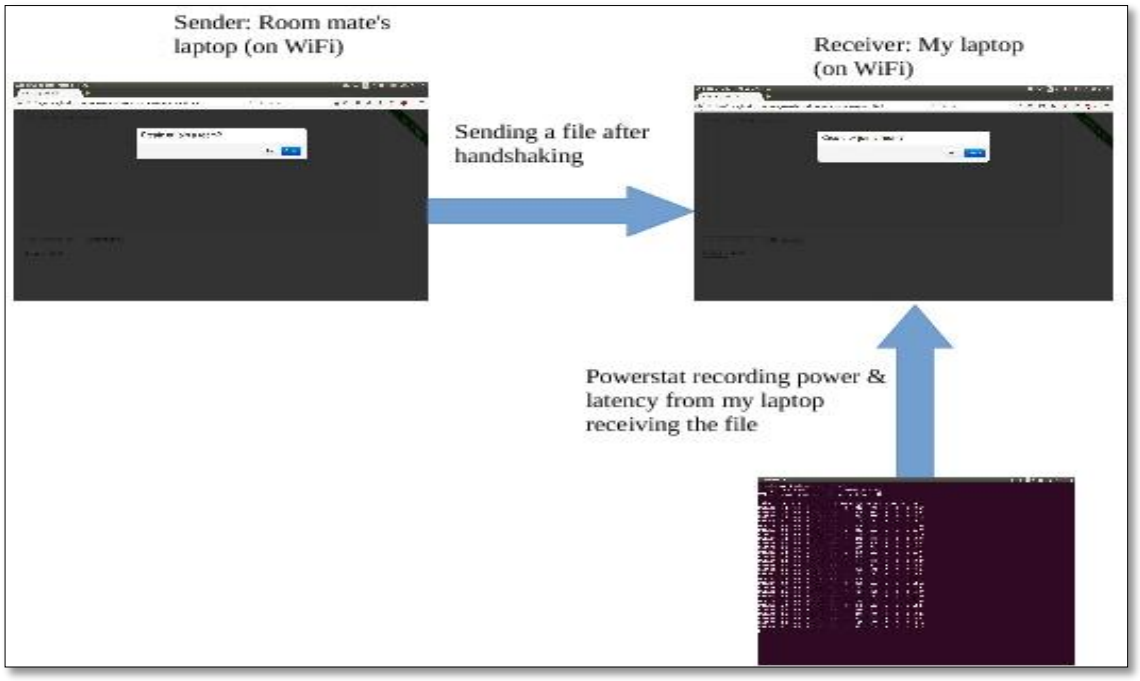


Figure 9: Block diagram to represent test bed for WebRTC

Similar to CGI, WebRTC test setup works between transferring files from one host to another. The first step is handshaking between the peers which is explained earlier in this document. For this test bed, we do not consider handshaking for the power measurements because it is a manual process to transfer the JSON variable to join a session. After the initial handshaking, the host transmits the file to the client on which Powerstat is running. Both the devices are running on Wi-Fi and similar to CGI, Powerstat starts recording the measurements before the data is uploaded for transmission and ends after recording a few more samples after completing the data transfer.

6.2 Test bed for ZigBee data transfer:

In the second mode of data measurement, ZigBee modules are used for data transfer. There are four different file sizes used for data transmission with the ZigBee namely 10KB, 50KB, 200KB and 250KB. To compare the performance over Wi-Fi, the 'File Upload' functionality explained earlier in this section is used with for the same file sizes being used for ZigBee. In simple terms, the comparison is between two different Wireless protocols namely ZigBee protocol (IEEE 802.15.4) and Wi-Fi (IEEE 802.11) using file transfers when using CGI scripts.

There are two modules that form a Personal Area Network (PAN) in which one is the transmitter and the other is the receiver. In a more general scenario, there are more than two ZigBee modules forming a bigger mesh network in which there is a central device, the coordinator, which has the information about the entire network and this module has to be awake at all times. Other modules which relay data from one peripheral to another are called Routers which do not sleep like the coordinator but are intermediate devices

and their functionality is to relay data to other peripherals. The third type of device is the end device, which is connected to the sensor and is responsible for sending the data to the router nodes. These devices can sleep and have the minimum information about the network to function according to the needs. In this network, the modules dynamically decide which particular module to send data to. If a particular device is down, the coordinator will re-route accordingly.

Below is the block diagram to show the test bed for data transmission using ZigBee.

Figure 10: Block diagram to represent test bed for WebRTC

In this case, since there are only two devices, one is the transmitter and the other is the receiver. Both these modules are programmed according to their MAC addresses and

hardcoded to be a transmitter and receiver respectively. The test bed for this case is more of a scenario that mimics an end device on which there is a sensor connected. The sensor basically records some data. The connected ZigBee understands that it needs to transmit this data to another peripheral, which could be a router or the coordinator itself. The receiver receives the data and stores it in the server. Here, the transmitter is connected to the peripheral (Dell Inspiron 14Z laptop) on which the readings are recorded, to gauge the power consumed. Before the data begins to transmit, Powerstat starts recording data. Measurements are taken until the data is completely sent and beyond to show the power after transmission.

CHAPTER 7

RESULTS & RECOMMENDATIONS

This section shows the graphical representation of the results gathered and the analysis of what it means in terms of system analysis. The basis of the comparisons are the power consumed by the peripheral transmitting the data. The X- axis shows time samples for each case which is one sample per second. The Y-axis shows the power consumed by the device transmitting the file before, during and after the process of file transmission based on the graphs.

In each case, the area under the curve is calculated to compare the overall power consumption in that case, which gives more information about the performance. To provide the worst case scenario, the least amount of power consumed by the peripheral within the duration of the entire measurement is subtracted from the entire list of values to get the total amount of power consumed. Following is the mathematical representation for it.

Step 1: *Array[Power measured] = Array[Power measured] – Least value in the array*

Step 2: *Total Power consumed = Sum of Elements**(** Array[Power measured])*

7.1 Comparison of file transfer between CGI and WebRTC

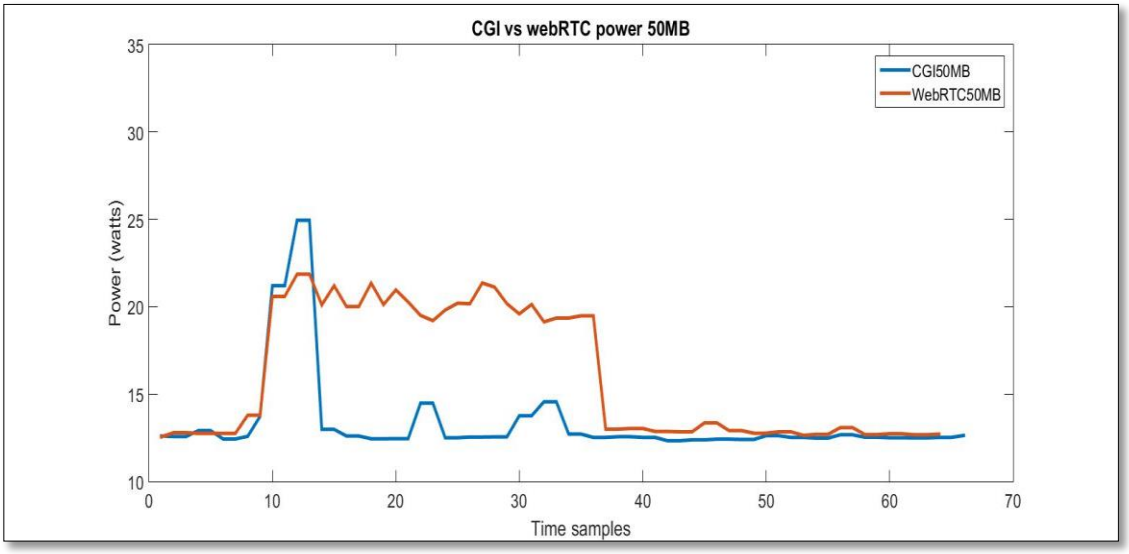


Figure 11: CGI vs WebRTC transmitting a 50 MB file

The above figure represents data transmission using Wi-Fi for CGI and WebRTC for a 50 MB file. From the graph, it can be inferred that the time taken to transmit the data using CGI is much lower than the time taken by WebRTC. But the power spike that CGI produces is higher than that of WebRTC. This is probably because CGI is more memory intensive. The amount of time consumed by CGI is way lesser than WebRTC is probably because the CGI application here does not offer security in terms of data encryption and hence the lesser time, but it brings in a new perspective of security concerns. It has to be understood that there are mechanisms to encrypt the data before sending it over HTTP. Today most of the websites are secure by sending data over HTTPS. In the case of WebRTC the amount of power consumed is lesser but takes more time in sending data as the platform comes with an inbuilt data encryption system. Hence the power consumed

for encryption at the sender's end and decryption at the receivers end before the data is delivered to the client is comparatively much more. On calculating the area covered under the power graph by the method explained earlier in this section, CGI consumes an area of 67.93 watts under the curve while WebRTC consumes 220.68 watts.

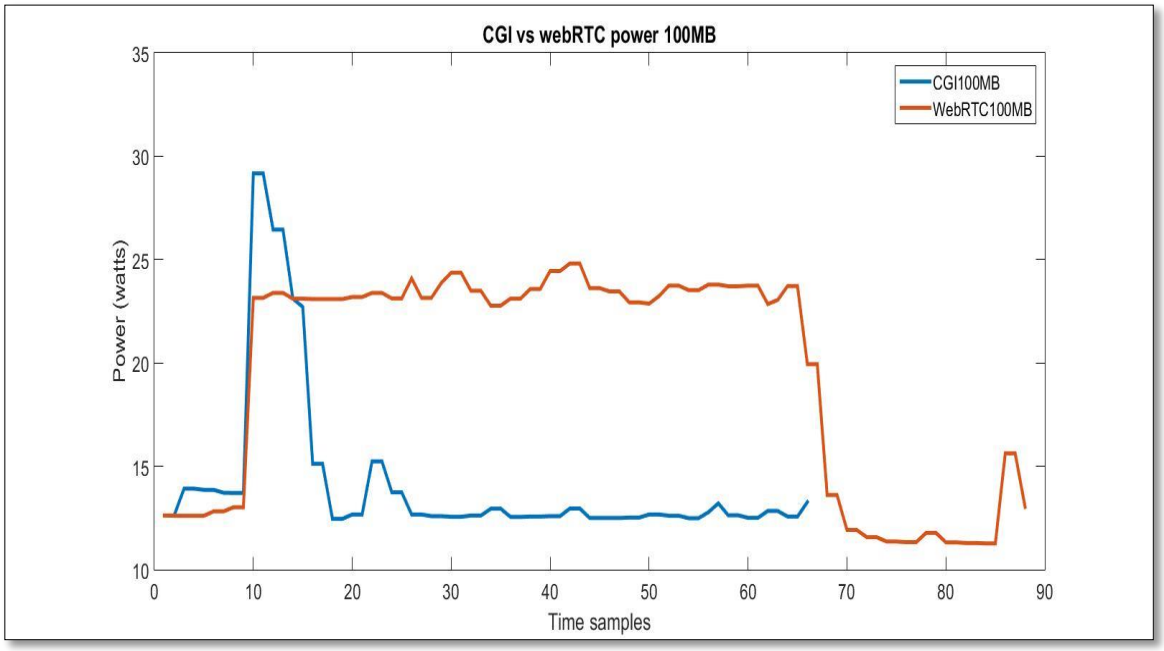


Figure 12: CGI vs WebRTC transmitting a 100 MB file

In the figure above for the power and latency analysis for a 100 MB file, the maximum level of power consumed by CGI again is much higher than WebRTC but the time taken to transmit the file is much lower. In the case of CGI, the technology used to produce the binary scripts is ANSI C which is a low level language and does not require an external compiler as every kernel today comes with the default C compiler. So, here the only work being done is to transfer the file over Wi-Fi without any hindrance. Hence the high speeds of data transfer and inversely, hence the high power spikes. In WebRTC, all the

work is being done by the browser specific code. Browsers are optimized to perform better in terms of power as generally users open more than one tab in a browser, which cumulatively adds a lot to power consumption and CPU load.

In every case of graphical representation here, it is to be noted that the file transfer starts after every 9 seconds to maintain consistency in understanding the latency in file transfer. The area covered in terms of power consumption for WebRTC in this case is 733 watts while for CGI is 114.16 watts using the methodology discussed earlier.

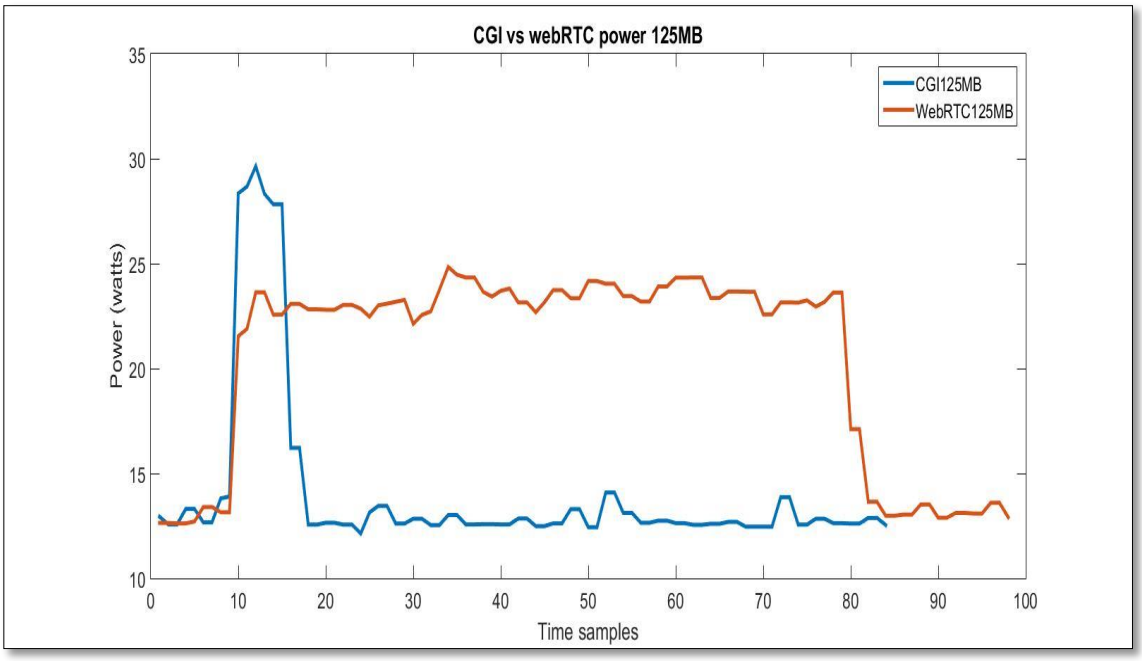


Figure 13: (Above) CGI vs WebRTC transmitting a 125 MB file

In the above shown graph, the CGI peak covers an area of 156.97 watts while the WebRTC graph covers an area of 773.16 watts.

Below is the graph representing the comparison between WebRTC and CGI for a 300MB file.

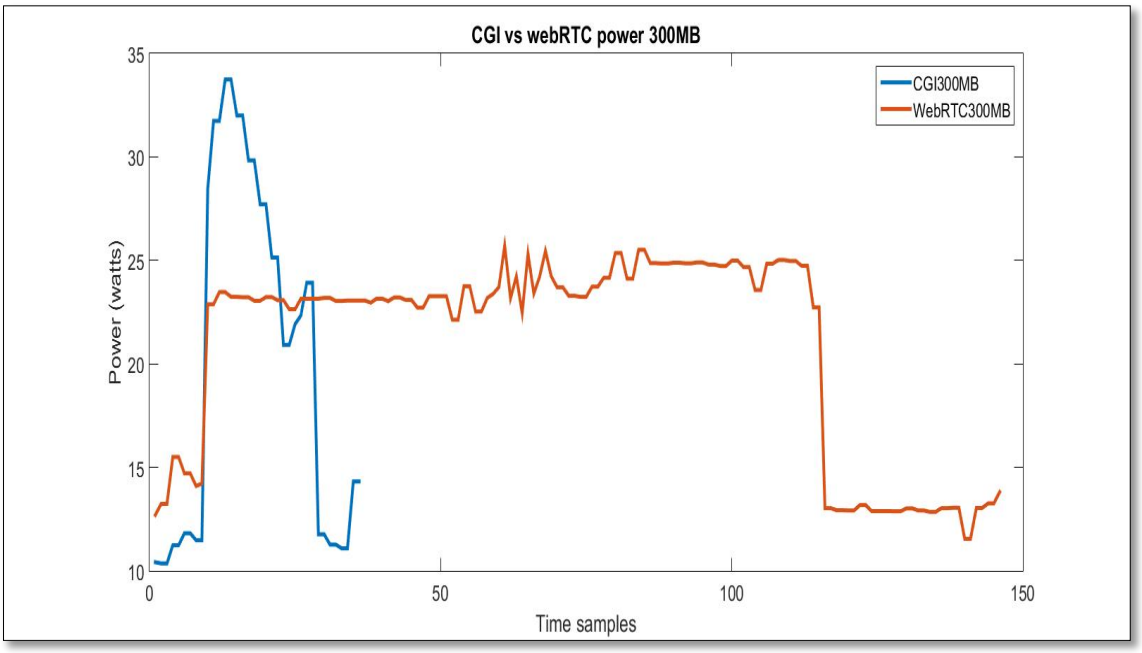


Figure 14: CGI vs WebRTC transmitting a 300 MB file

In the graph above, the CGI peak covers an area of 409.09 watts while the WebRTC graph covers an area of 1363.02 watts.

From the graphs shown above, it can be seen that the peak power consumed by the CGI application is always higher than the WebRTC application but the resources being transmitted to the server is always faster with a very considerable margin. The major reason for the speeds is the fact that CGI program does not directly interact with the user or interacts directly with the browser or a graphical interface. It always resides on the server side and responds to requests from the user based on the kind of application. Also,

CGI can be defined and written in any specific programming language to create a binary. Once an executable file is created from the code, all the machine needs to do is to make sure that the requests from the client are processed since it is already in a machine readable format. In this application, all the CGI needs to understand is HTML, which is known for its speed and compatibility with every browser and rendering data onto the browser. But the disadvantage is that CGI scripts are highly memory intensive programs, which means that whenever a CGI file is invoked the amount of power consumed to execute it is high. Also, this application is uploading a file which would require more memory space on the client end to store the data in the memory hardware and transmit it accordingly. But the area of power consumed by CGI is always lower than the overall power consumed by WebRTC for the same file size.

In WebRTC's case, the basic premise is that the handshaking is already completed and the devices are ready to transmit the data from one host to another. All the data that is transmitted using WebRTC API is secure in the sense that all the files are encrypted by the platform before transmitting. So when large files are being transmitted, the API automatically takes a much longer time to cut down the data into chunks, encrypt it and then transmit the data. This is based on the fact that WebRTC uses the Datagram Transport Layer Protocol (DTLP) which is a derivative of SSL, which means that any data that is being transmitted is secure. The reason why it consumes lesser power here is probably because the WebRTC platform is a small layer in the browser. The JavaScript code required by WebRTC need not be changed or encoded, this is taken care of by the browser itself. The rest of the content is done by the browser which is, data encryption and transmission over TCP/IP. Another major reason why WebRTC consumes longer

time could be to resolve NATs as discussed earlier. If an IP address on the network is not established, the API contacts servers using STUN, TURN and ICE to resolve a peer to peer connection. But this case is not relevant here as all the devices were on Arizona State University's network at the time of data measurements.

Below are the power and latency for WebRTC and CGI based on different file sizes.

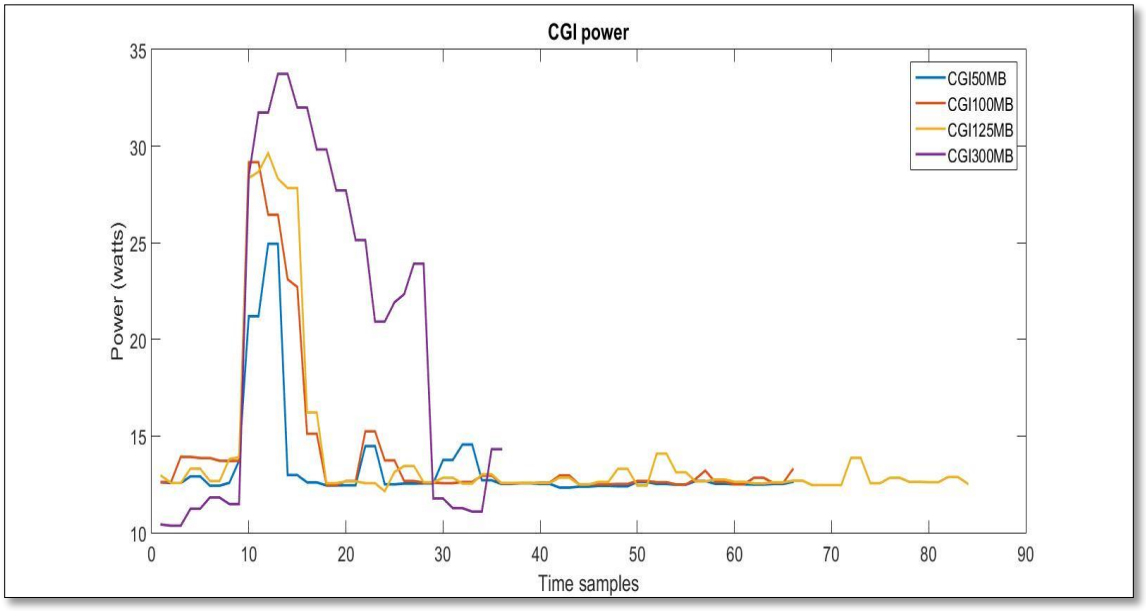


Figure 15: CGI performance for different file sizes

From the figure above, it can be concluded that the amount of power consumed and the time taken to transmit a file by CGI applications increases as the file sizes increases and hence the amount of area covered also increases.

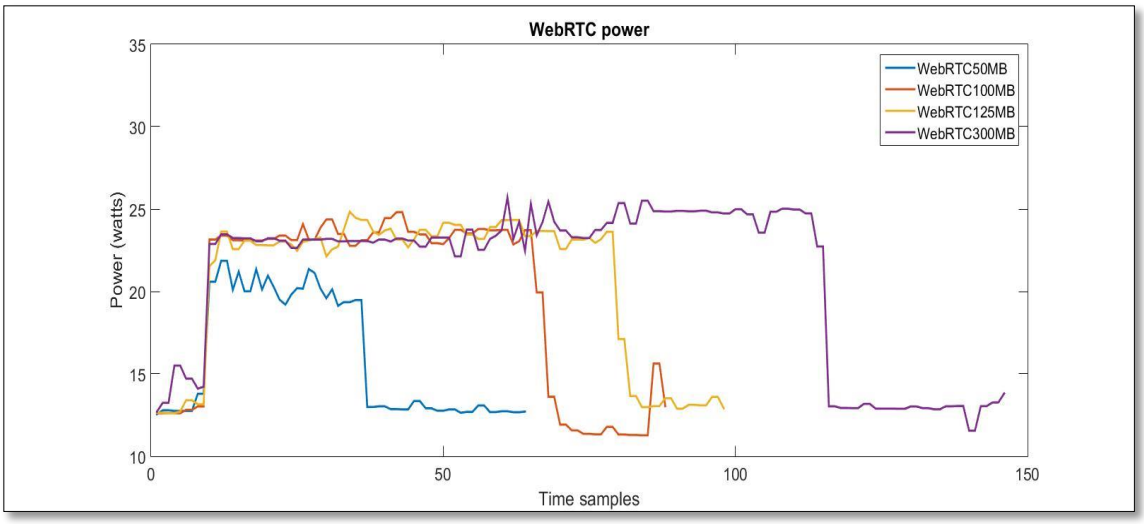


Figure 16: WebRTC performance for different file sizes

From the above figure, it can be concluded that the spike in power consumed by WebRTC application during file transfer for different file sizes only increases by a small amount, but the time taken to transmit a file and hence the total area of power consumption over multiple time samples increases.

7.2 Comparison of file transfer over ZigBee and Wi-Fi using CGI

As described earlier, two ZigBee modules are used to transfer data for multiple file sizes namely, 10KB, 50KB, 200KB and 250KB from the WVSNP dashboard functionality 'Node Functions'. Powerstat records the data as mentioned in the test bed section. This data is compared to the Wi-Fi performance using the 'File Upload' functionality in the dashboard. The following is the graphical representation for the data collected.

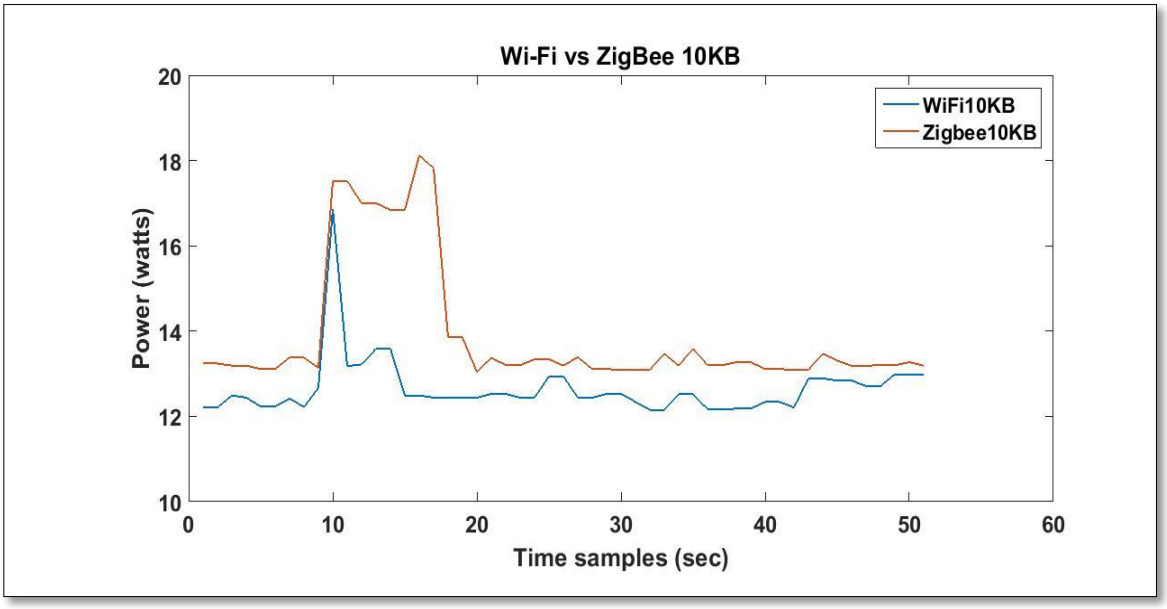


Figure 17: ZigBee vs WiFi for 10KB file

The above graph shows the comparison in power between ZigBee and WiFi in transferring a 10KB file. This gives an inference that ZigBee is not only consuming longer time to transmit data but also in the process consuming more power in doing so. The area covered in terms of power by ZigBee for this transmission is 61.75 watts while for Wi-Fi it is 36.30 watts.

The basic reason for this is because ZigBee has got much lesser data speeds over the air when compared to Wi-Fi. Wi-Fi can put in speeds up to 600 Mbps but the average could be anywhere between 50 to 200 Mbps while ZigBee has the capability to reach up to 250 Kbps at most.

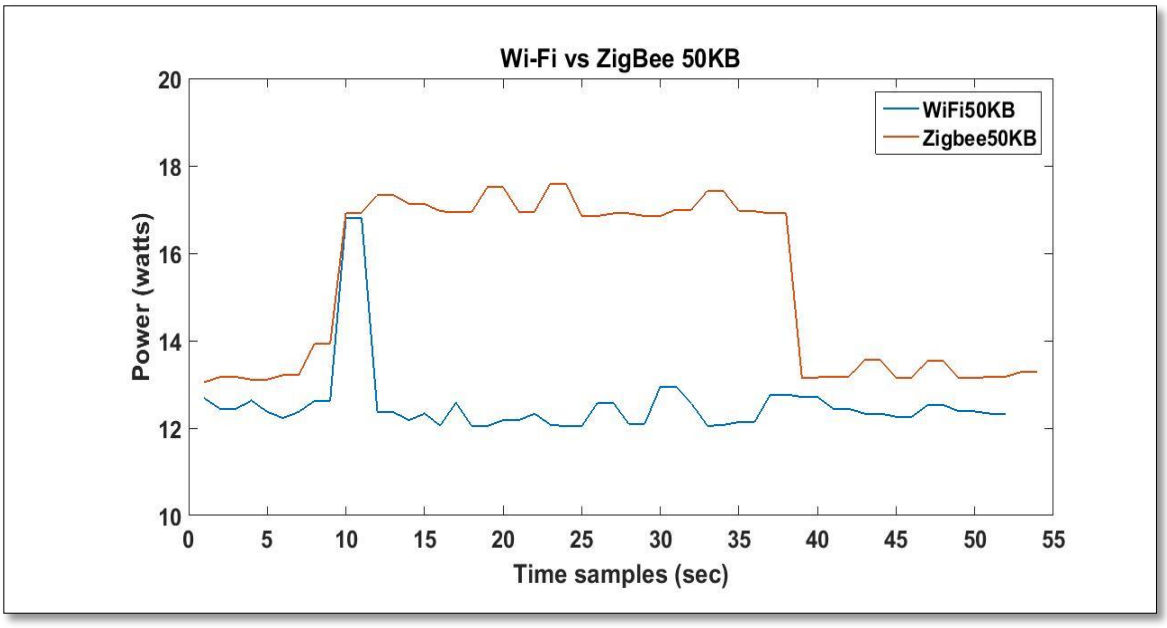


Figure 18: ZigBee vs Wi-Fi over CGI for 50KB file

Power consumed by ZigBee is more here than over Wi-Fi. Power area covered for ZigBee is 220.61 watts while for Wi-Fi is 30.22 watts.

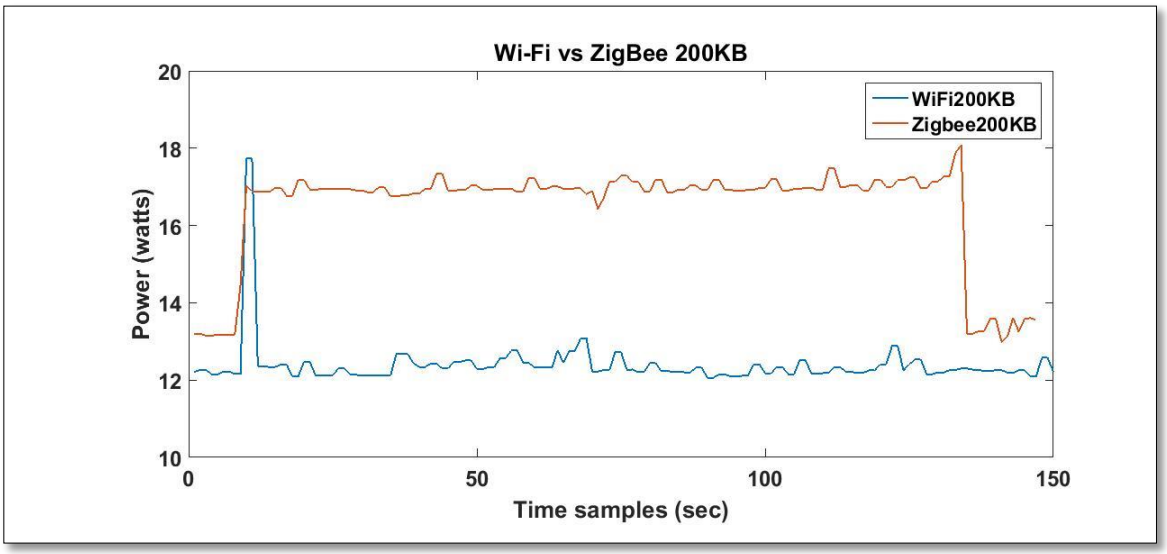


Figure 19: ZigBee vs Wi-Fi over CGI for 200KB file

In the above graph, power consumed by ZigBee is 703.04 watts while under Wi-Fi is 60.69 watts.

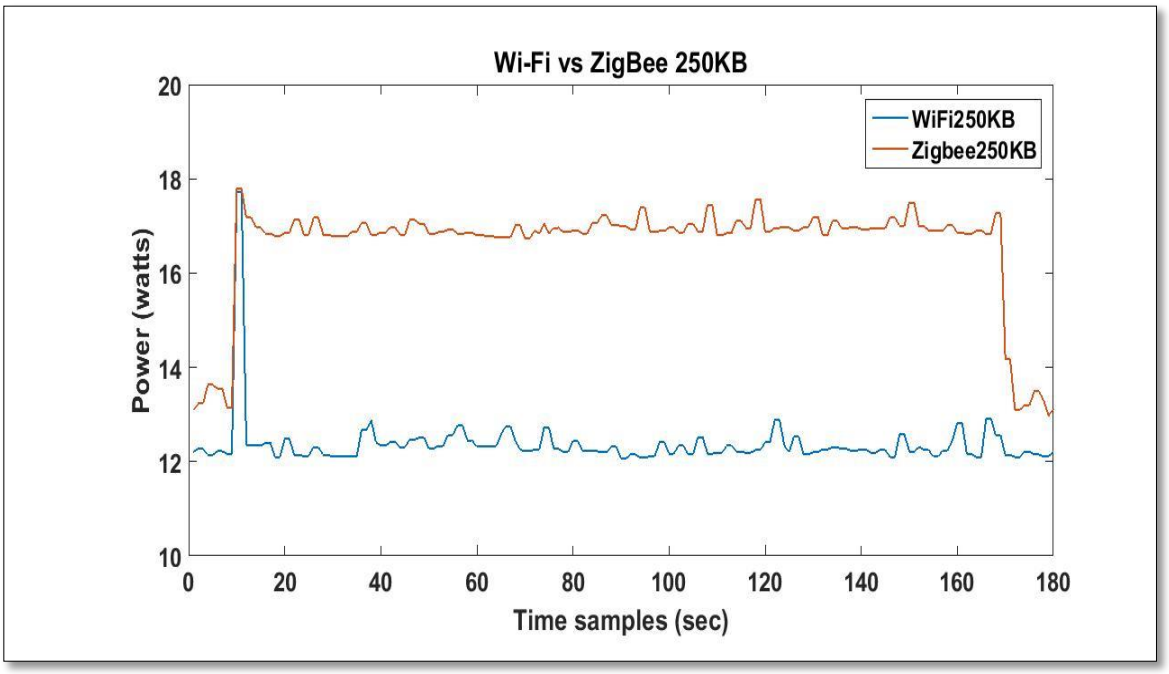


Figure 20: ZigBee vs Wi-Fi over CGI for 250KB file

In the above graph, power consumed by ZigBee is 902.87 watts while under Wi-Fi is 63.02 watts.

The above comparisons for file transfer between ZigBee and Wi-Fi modules using the CGI based applications show enough evidence that ZigBee modules are consuming more power while Wi-Fi does not use as much power. The Wi-Fi speeds are high and the file size is not big to use too much power or higher latency times. This is complementing the earlier results shown for comparing power and latency between WebRTC and CGI.

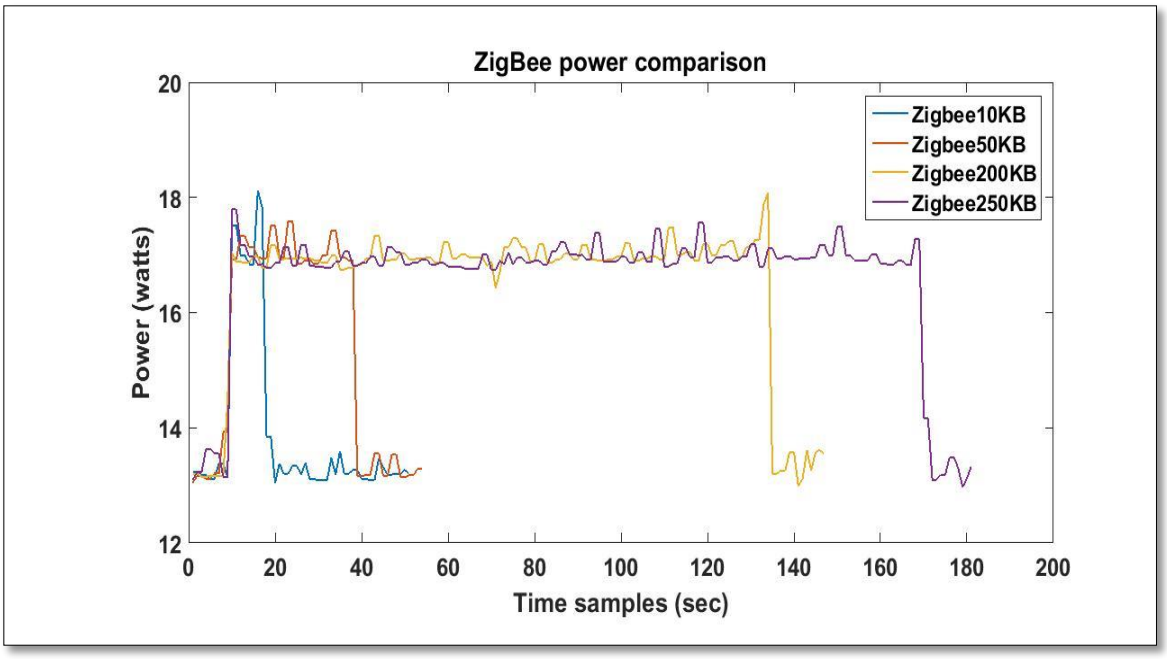


Figure 21: ZigBee Power comparison

The above graph compares the ZigBee performances with increase in file sizes.

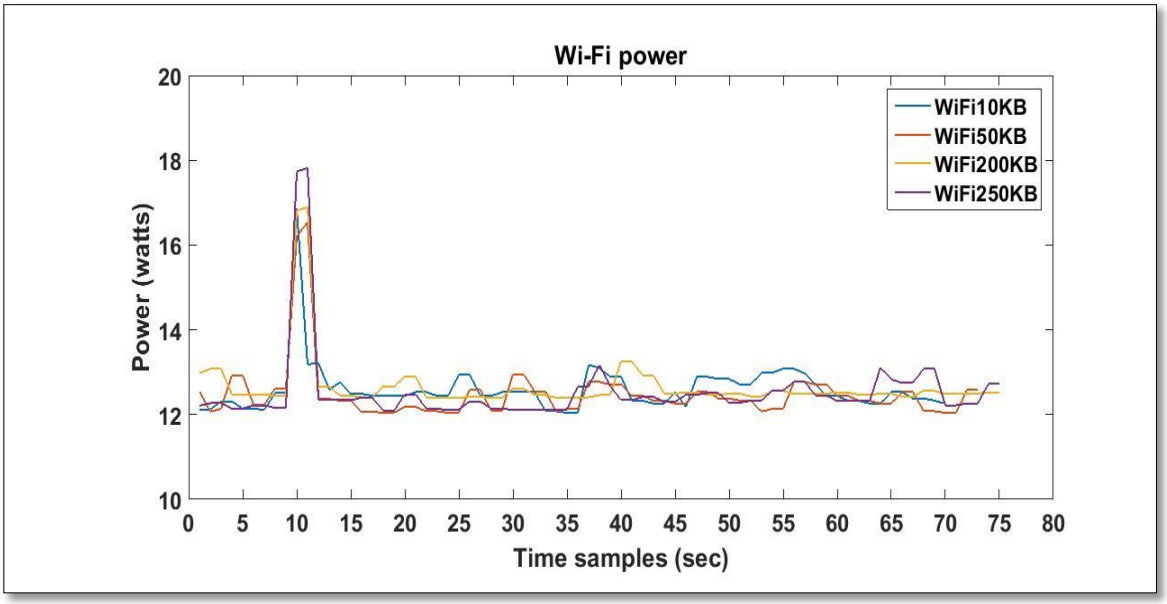


Figure 22: Wi-Fi power comparison

The above graph compares the Wi-Fi performances with increase in file sizes over CGI.

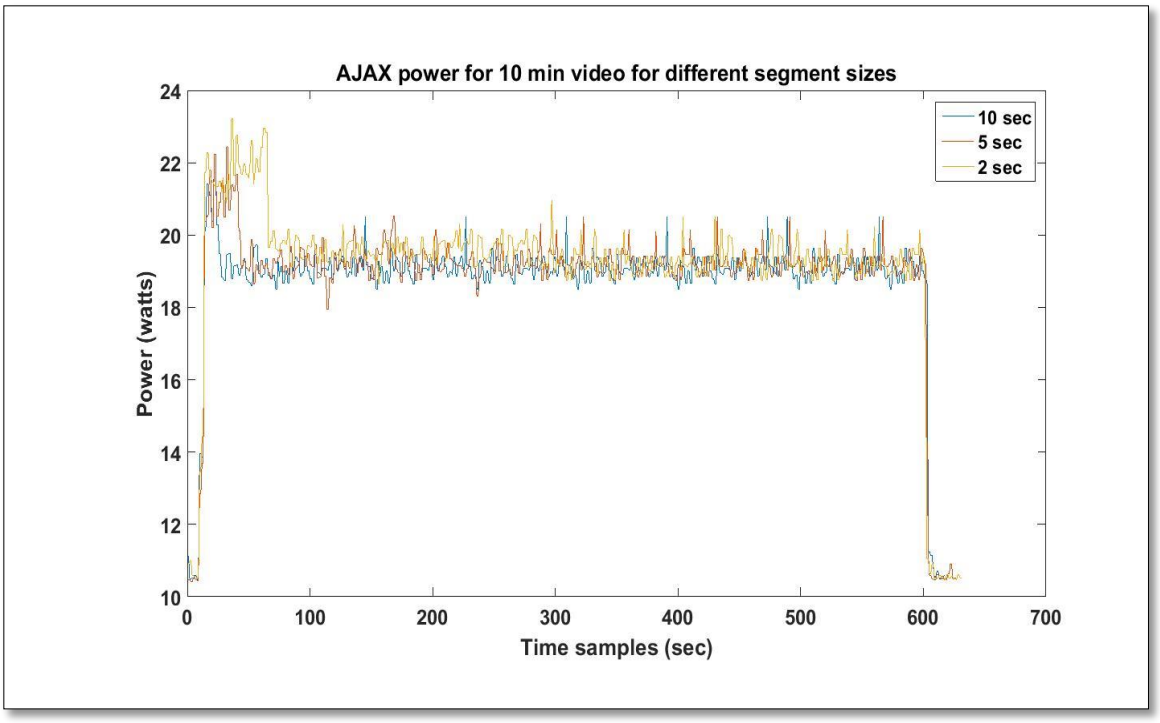


Figure 23: AJAX power comparison for different segment sizes

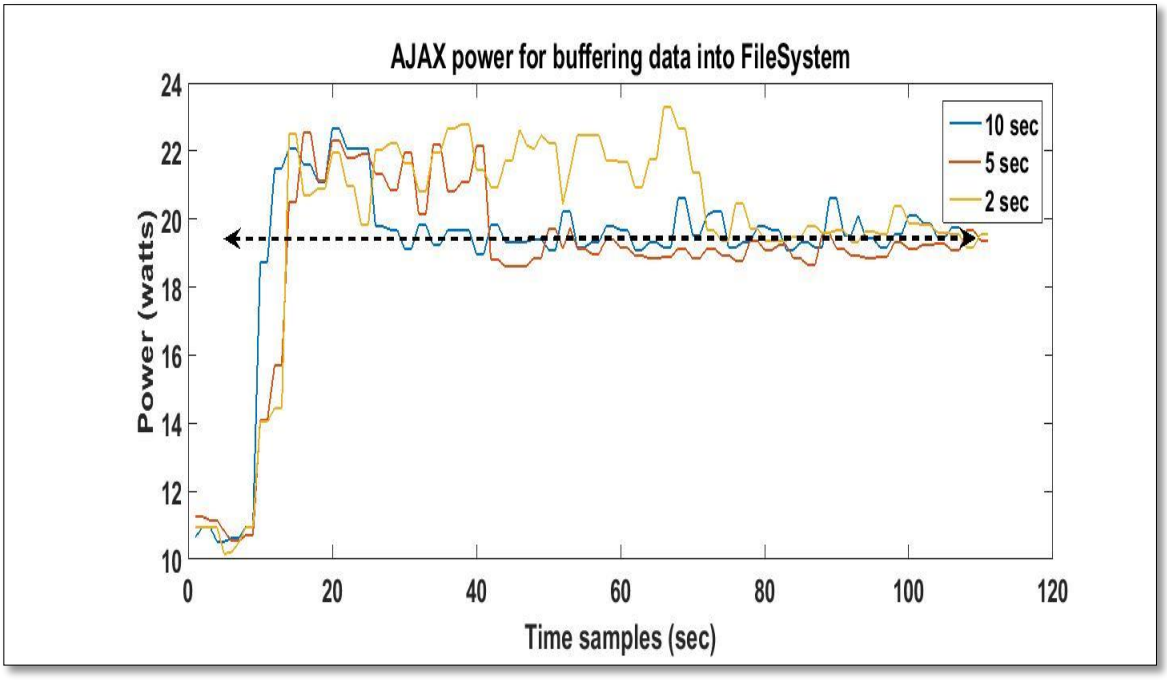


Figure 24: AJAX power comparison for buffering multiple segment of the same file

File Size	Power consumed by CGI in Watts	Power consumed by WebRTC in Watts
50MB	67.93	220.68
100MB	114.16	733
125MB	156.97	773.16
300MB	409.09	1363.02

Table 1: Power comparisons between CGI & WebRTC for different file sizes.

File Size	Average Power consumed by Wi-Fi in Watts	Average Power consumed by ZigBee in Watts
10KB	12.6502	13.8925
50KB	12.5490	15.3302
200KB	12.3939	16.4598
250KB	12.3672	16.5532

Table 2: Power comparisons between CGI & WebRTC for different file sizes.

RECOMMENDATIONS

Based on the experiments performed and the results achieved, the following are the recommendations for technologies involved with wireless sensor network platforms.

- For the same amount of data transmitted, WebRTC drains the battery faster than CGI as it consumes more amount of power from the battery as calculated from the area of power consumed from the graphs above. So using CGI is definitely more power efficient. But another important point to draw from the graphs is that the for the same amount of data being transmitted, the maximum amount of power used to transmit at a particular instant for CGI is more than that of WebRTC. So if the client's hardware is old or substandard, then it could cause issues in terms of the life span of the device.
- The maximum level of power consumed by WebRTC is lesser than CGI for the same size. If a large enough file is being transmitted consistently, in CGI's case, the power consumed will be regularly high which means there need to be cooling mechanisms (like in a laptop or huge server) to the existing platform which could be an overhead. But if smaller files are being sent every time, the best way to transmit is through Wi-Fi using CGI scripts.
- It is always recommended to enforce security while transmitting data over the internet. Although the performance here for CGI is good, it does not have security which definitely makes the data vulnerable for hacking and third party polling for the data being transmitted.
- ZigBee modules are very reliable in the sense that they have 128 bit inbuilt

hardware AES encryption that can be enabled based on the security byte in the MAC frame for the packet being transmitted and also are simple to maintain with a high battery life. They form really good beacons for implementation on IoT platforms within a personal network.

For wireless video sensor network platforms, as the name suggests, more often than not, the primary concern is power. WebRTC could be a good choice for data transfer. But in these cases, the peripherals transmitting the data might not have browsers, as installing a browser on an embedded peripheral, if it is not manually operated, would mean a lot of overhead for installing and maintaining it. This would simply mean that the device would have to forego a lot of power consumption.

These results and recommendations show that CGI consumes more power but less time and vice versa for WebRTC. But the implementations of these technologies will depend on the kind of environment and power profiling the developer or the consumer are concerned with [67]. The throughput varies depending on the hardware being used and the type of protocols that it is implemented on.

FUTURE WORK

Going forward, the following are the changes that can be incorporated into the WVSNP dashboard and the WebRTC application.

- In the ‘File upload’ and the ‘Node functions’ tab in the dashboard, automating segment wise data transfer from the client to the server by just uploading the first segment file will give a better perspective in power consumed and time taken for that data transfer.
- In the WebRTC application, the signaling can be automated by saving the session variable directly into the server so that the other user can access it by just contacting the server and not the other client. This is currently the ongoing work. Also, this application can be enabled to transfer multiple segments by just uploading the first segment to gauge how it performs in terms of power and latency.
- The CGI ‘File upload’ application on the dashboard used for file transfer is not secure in sending the data. Future work can go into encrypting the data before sending it from the client device to the server. Also, more analysis can be done on gathering power consumption at the server end to understand its performance.
- As discussed earlier in the document, the WVSNP player is only compatible with Chrome browser as the filesystem API is not supported by any other browser at this point. But Media Source Extensions (MSE) [66] is a specification that is supported by many browsers that allows JavaScript to send streams of data to send to the media codes within the web browser that supports HTML5 video tag.

So going forward, the WVSNP player can use this specification to enable the video playback on any type of browser and on any type of operating system. This would make the WVSNP dashboard a standalone application to run on any compatible device.

REFERENCES

[1] Andrew Baxter. SSD VS HDD http://www.storagereview.com/ssd_vs_hdd, 2015.

[2] JOEL SANTO DOMINGO, SSD VS HDD WHATS THE DIFFERENCE http://www.pcmag.com/article2/0,2817,2404258,00.asp, February 17, 2015.

[3] Bradley M. Kuhn, The state of free software in mobile devices, URL http://timreview.ca/article/336 March 2010.

[4] Lukas, A Schwoebel, Browser and Codec Agnostic HTML5 Video-streaming in the wireless video sensor network platform, July 4, 2014.

[5] Tejas Shah, A cross layer power analysis and Profiling of Wireless Video Sensor Node Platform Applications. May 2014.

[6] Sebastian Anthony, Increasing wireless networking speed by 1000%, URL http://www.extremetech.com/computing/138424-increasing-wireless-network-speed-by-1000-by-replacing-packets-with-algebra, October 23, 2012

[7] Durairaj. M, Kannan.P A Study On Virtualization Techniques And Challenges In Cloud Computing, INTERNATIONAL JOURNAL OF SCIENTIFIC & TECHNOLOGY RESEARCH VOLUME 3, ISSUE 11, NOVEMBER 2014.

[8] Awodele Oludele, Emmanuel C. Ogu, Kuyoro 'Shade, Umezuruike Chinecherem. On the evolution of Virtualization and Cloud Computing. Journal of Computer Sciences and Applications, 2014 2 (3), pp 40-43. December 25, 2014.

[9] Cola, C. , Valean, H. On multi user web conference using WebRTC. System Theory, Control and Computing (ICSTCC), 2014 18th International Conference, October 2014.

[10] Elleuch, W. Models for Multimedia conference between browsers based on WebRTC. Wireless and Mobile Computing, Networking and Communications (WiMob), 2013 IEEE 9th International Conference, October 2013.

[11] Segec, P., Paluch, P. ; Papan, J. ; Kubina, M. The integration of WebRTC and SIP: Way of enhancing real-time, interactive multimedia communication.

[12] Flávio Ribeiro Nogueira Barbosa , Luiz Fernando Gomes Soares. Towards the Application of WebRTC Peer-to-Peer to Scale Live Video Streaming over the Internet, 2013.

[13] Berkvist, A., Burnett, D., Jennings, C. Narayanan, A. (2011) “WebRTC 1.0: Real-Time Communication Between Browsers”. Working Draft.

[14] Cho, S., Cho, J., Shin, S. (2010) “Playback Latency Reduction for Internet Live Video Services in CDN-P2P Hybrid Architecture”. 2013 IEEE International Conference on Communications.

[15] Li, B., Xie, S., Qu, Y., Keung, G.Y. (2008) “Inside the New CoolStreaming: Principles, Measurements and Performance Implications”. IEEE 27th Conference on Computer Communications.

[16] Naylor, A. (2013) “WebRTC is almost here, and it will change the web”. http://goo.gl/IgxF33, accessed in March 2014.

[17] A.Johnston, J.Yoakum and K.Singh, "Taking on WebRTC in an enterprise", IEEE Communications Magazine, Vol. 51, No.4, April 2013, doi:10.1109/MCOM.2013.6495760

[18] Xing Yan ; Digital Media Technol., Commun. Univ. of China, Beijing, China ; Lei Yang ; Shanzhen Lan ; Xiaolong Tong. Application of HTML5 Multimedia. Computer Science and Information Processing (CSIP), 2012 International Conference. August 2012.

[19] François Daoust , Philipp Hoschka , Charalampos Z. Patrikakis , Rui S. Cruz , Mário S. Nunes , David Salama Osborne. Towards Video on the Web with HTML5. URL https://www.w3.org/2010/Talks/1014-html5-video-fd/video-html5.pdf, 2010.

[20] W3Schools website for HTML5. URL http://www.w3schools.com/html/html5_intro.asp. Accessed since January 2014.

[21] Philippe De Ryck, Lieven Desmet, Frank Piessens and Wouter Joosen. A Security Analysis of Emerging Web Standards HTML5 and Friends, from Specification to Implementation. URL https://lirias.kuleuven.be/bitstream/123456789/353864/1/SECRYPT_2012_64_CR.pdf, 2012.

[22] S. Panagiotakis1, K. Kapetanakis2,*, A. G. Malamos. Architecture for Real Time communiactions over Web. International Journal of Web Engineering 2013, 2(1): 1-8 DOI: 10.5923/j.web.20130201.01, 2013.

[23] HTML5rocks website. URL http://www.html5rocks.com/en/, Accessed in 2014-15.

[24] Jayavardhana Gubbi , Rajkumar Buyya, Slaven Marusic , Marimuthu Palaniswami. Internet of Things (IoT): A vision, architectural elements, and future directions, Future Generation Computer Systems. 2013.

[25] H. Sundmaeker, P. Guillemin, P. Friess, S. Woelfflé, Vision and challenges for realising the Internet of Things, Cluster of European Research Projects on the Internet of Things—CERP IoT, 2010.

[26] Li Li; Hu Xiaoguang ; Chen Ke ; He Ketai. The applications of WiFi based Wireless Sensor Network in Internet of Things and smart Grid. Industrial Electronics and Applications (ICIEA), 2011 6th IEEE Conference, 2011.

[27] Qian Zhu; Ruicong Wang ; Qi Chen ; Yan Liu. IOT Gateway: Bridging Wireless Sensor Networks into Internet of Things. Embedded and Ubiquitous Computing (EUC), 2010 IEEE/IFIP 8th International Conference, 2010.

[28] IoT Wiki website: Discussing IoT. URL http://internetofthingswiki.com, Accessed on February 15, 2016.

[29] L. Atzori, A. Iera, G. Morabito. The Internet of Things: a survey. Computer Networks, 54 (2010), pp. 2787–2805

[30] Venkitachalam, G, Tzi-cker Chiueh. High performance Common Gateway Interface invocation. Internet Applications, 1999. IEEE Workshop, August 1999.

[31] CGI W3C standard. URL: https://www.w3.org/CGI. Accessed on February 15, 2016.

[32] Shishir Gundavaram. CGI Programming on the World Wide Web. URL http://www.oreilly.com/openbook/cgi/ch01_01.html. 1st Edition March 1996. Accessed on February 15, 2016.

[33] Thomas Boutell. cgic: an ANSI C library for CGI Programming. URL http://www.boutell.com/cgic. Accessed on February 15, 2016.

[34] Adolph Seema. WVSNP-DASH: Wireless Video Sensor Node Platform's Dynamic Adaptive Streaming via HTTP: Integrating video sensor nodes into the internet of things. Arizona State University, 2011.

[35] Adolph Seema, Lukas Schwoebel, Tejas Shah, Jeffrey Morgan, Martin Reisslein. WVSNP-DASH: Name-based segmented video streaming. IEEE Trans. Broadcasting, Volume:61, Number:3, Pages:346-355, 2015.

[36] Adolph Seema and Martin Reisslein. Towards efficient wireless video sensor networks: A survey of existing node architectures and proposal for a flexi-wvsnp design. Communications Surveys & Tutorials, IEEE, 13(3):462–486,2011.

[37] Yocto Project. URL https://www.yoctoproject.org/. Accessed on February 15, 2016.

[38] Gstreamer, Multimedia open source framework. URL http://gstreamer.freedesktop.org/. Accessed on February 15, 2016.

[39] FFMpeg, A complete, cross platform solution for audio, video manipulation. URL https://www.ffmpeg.org/. Accessed on February 15, 2016.

[40] Mongoose-Embedded Webserver by Cesanta, URL https://github.com/cesanta/mongoose. Accessed on February 15, 2016.

[41] Youtube Engineering and Developers Blog. URL http://youtube-eng.blogspot.jp/2015/01/youtube-now-defaults-to-html5_27.html. January 2015.

[42] HTTP Live Streaming by Apple. URL https://developer.apple.com/streaming/. Accessed on February 15, 2016.

[43] MPEG-DASH, An overview. URL http://www.encoding.com/mpeg-dash/. Accessed on February 15, 2016.

[44] Gordon C. Bruner, Anand Kumar. Explaining consumer acceptance of handheld Internet devices. Journal of Business Research 58 (2005) 553 – 558.

[45] J. Roschelle. Keynote paper: Unlocking the learning value of wireless mobile devices. Journal of Computer Assisted Learning (2003) 19, 260-272

[46] Garageio: Controlling your garage door from your smartphone. https://garageio.com/. Accessed on February 15, 2016.

[47] Chalei. Linkit One IoT: Sending alert email. URL http://www.instructables.com/id/Linkit-One-IoT-sending-alert-email/. Accessed on February 15, 2016.

[48] Bruce Hartley. The Internet of Things - Weather Monitoring tool. Meteorological Service of New Zealand Ltd. New Zealand. 2012.

[49] WebRTC-1.0, Real time communication between browsers. URL http://w3c.github.io/webrtc-pc/ Accessed on February 15, 2016.

[50] Econais: Internet of things. Think WiSmart™. Audio/Video Applications. URL http://www.econais.com/applications/audio-video-applications/. Accessed on February 15, 2016.

[51] Castillejo, P.; Martinez, J.-F.; Rodriguez-Molina, J.; Cuerva, A. Integration of wearable devices in a wireless sensor network for an E-health application Wireless Communications, IEEE, Issue Date: August 2013

[52] Dr. Jose Fernandez Villasenor. How Connected Healthcare Today Will Keep the Doctor Away. The Impact That Wearables and the IoT Will Have on Patient Care. URL https://cache.freescale.com/files/corporate/doc/white_paper/CNHLTHTKPDCAWWP.pdf. Accessed in October, 2015.

[53] Aaron Carroll, Gernot Heiser. An Analysis of Power Consumption in a Smartphone. URL https://www.usenix.org/legacy/event/atc10/tech/full_papers/Carroll.pdf. Accessed on February 15, 2016.

[54] Darren Dart. Yocto Project Linux Kernel Development Manual. URL http://www.yoctoproject.org/docs/1.6.1/kernel-dev/kernel-dev.html. Accessed on February 15, 2016.

[55] Zigbee Alliance:What is Zigbee?. URL http://www.zigbee.org/what-is-zigbee/. Accessed on February 15, 2016.

[56] Digi XBee ZigBee. URL http://www.digi.com/products/xbee-rf-solutions/modules/xbee-zigbee. Accessed on February 15, 2016.

[57] Video Library x264. URL http://www.videolan.org/developers/x264.html. Accessed on February 15, 2016.

[58] MPEG-4: Multimedia Framework. URL https://www1.ethz.ch/replay/docs/whitepaper_mpegif.pdf. Accessed on February 15, 2016.

[59] Chris Ball, Serverless-WebRTC. URL https://github.com/cjb/serverless-webrtc. . Accessed on February 15, 2016.

[60] JSEP: JavaScript Session Establishment Protocol. URL https://tools.ietf.org/html/draft-ietf-rtcweb-jsep-03. Accessed on February 15, 2016.

[61] Greg Murray. Asynchronous JavaScript Technology and XML (Ajax). URL http://www.oracle.com/technetwork/articles/java/ajax-135201.html. Accessed on February 18, 2016.

[62] Martin Lucas. Storage Spaces: Understanding Storage Pool Expansion. URL https://blogs.technet.microsoft.com/askpfeplat/2013/09/24/storage-spaces-understanding-storage-pool-expansion/. Accessed on February 18, 2016.

[63] How AJAX works. JavaScript Web Development. URL http://www.webdesignerdepot.com/2008/11/how-ajax-works/. November 11, 2008.

[64] Eric Bidelman. Exploring the FileSystem APIs. URL http://www.html5rocks.com/en/tutorials/file/filesystem/. July 30th, 2013.

[65] STUN, TURN & ICE for NAT traversal. Eyeball Networks. URL http://www.eyeball.com/standards/stun-turn-ice/. Visited on February 18, 2016.

[66] W3C Media Source Extensions. URL https://w3c.github.io/media-source/. Visited on February 15th, 2016.

[67] JeongGil Ko, Kevin Klues, Christian Richter, Wanja Hofer, Branislav Kusy, Michael Bruenig, Thomas Schmid, Qiang Wang, Prabal Dutta, and Andreas Terzis. Low Power or High Performance? A Tradeoff Whose Time Has Come (and Nearly Gone). John Hopkins University. 2012.

[68] Powertop. URL https://wiki.archlinux.org/index.php/powertop. Visited on 02/27/2016.

[69] Aurzada, F., Levesque, M., Maier, M., & Reisslein, M. FiWi access networks based on next-generation PON and gigabit-class WLAN technologies: A capacity and delay analysis. *IEEE/ACM Transactions on Networking*, *22*(4), 1176-1189. [6557101]. 2014.

[70] Martin Reisslein, Jeremy Lassetter, Sampath Ratnam, Osama Lotfallah, Frank H.P. Fitzek, Sethuraman Panchanathan. Traffic and Quality Characterization of Scalable Encoded Video: A Large-Scale Trace-Based Study Part 1: Overview and Definitions. Arizona State Univ., Telecommunications Research Center, Tech. Rep. 2002.

[71] Rein, Stephan and Reisslein, Martin. Low-Memory Wavelet Transforms for Wireless Sensor Networks: A Tutorial. IEEE Communications Surveys & Tutorials. Volume 13. 2011.

[72] Rein, S., & Reisslein, M. Performance evaluation of the fractional wavelet filter: A low-memory image wavelet transform for multimedia sensor networks. *Ad Hoc Networks*, *9*(4), 482-496. 2011.

[73] M. Tausif, N.R. Kidwai, E. Khan, and M. Reisslein, "FrWF-Based LMBTC: Memory-Efficient Image Coding for Visual Sensors," IEEE Sensors Journal, vol. 15, no. 11, pp. 6218-6228, November 2015.

[74] Kidwai N. R.**,** Khan E., Reisslein M., "ZM-SPECK: A Fast and Memory-less Image Coder for Multimedia Sensor Networks.", IEEE Sensors Journal (Impact factor 1.762), (accepted for publication) D.O.I 10.1109/JSEN.2016.2519600. 2016