Wearable Application Battery-Saving Compression Techniques

Michael Harootoonyan
michael.harootoonyan.535@my.csun.edu

ABSTRACT

This paper investigates the relationship between data compression and battery performance when passing data from wearable devices and handheld devices. Two systems were developed, with two applications each. System I passes data from a handheld device to a wearable device and System II passes data from a wearable device to a handheld device. Application A uses data compression to compress and send that compressed data to be expanded by the receiving device. Application B does not use compression at all to complete the same task. The battery sync over a given time was monitored for both applications and compared to see which application was more battery efficient.

1. INTRODUCTION

Battery life is, arguably, the biggest pain point in wearables right now. You could certainly make a case for slow apps, or less-than-compelling software, or just plain unattractiveness being the biggest drawback. But those at least have the potential to change. Battery life is an altogether different problem. Consider System II, a wearable application that collects heart rate data for a diabetic patient, or maybe a developer testing out an application to collect sensor data in order to properly calibrate it. The wearable device would then pass this data over to a handheld device to be conveniently viewed by the user in the form of graphs, pie charts, tables, etc. The data is collected by writing into a text file on the sending device and then passes that data over to a receiving device after a certain amount of sensor data is collected. Imagine that the app remains powered on all day and multiple transfers occur throughout the day. These data transfers between the sending device and receiving device will then contribute to one of the main battery syncs in such a wearable application. Now consider System I, a handheld application that compresses images or videos to send over to a wearable application. Would a lossless data compression technique save battery? Also for smoother images, may using a lossy data compression technique prolong the life of the wearable devices battery upon data transfers. My experiment involves an application A, which aims to decrease the amount of time spent transferring data. Application A does this by first compressing the data before passing to a receiving device. It is then decoded by the receiving device for a user to be able to read. The cost of energy required to use the CPU to encode and decode the string is evaluated for both devices in application A. Application B has the advantage of not needing to expel any additional energy by having to compress the data before sending, and the receiving device does not need to expel any additional energy decoding the data.

2. RELATED WORK

Battery life can be dramatically improved by changing the way your apps handles data transfers between devices. Users want their data now, the trick is balancing user experience with transferring data and minimizing power usage. [2] “The cell radio is one of the biggest battery drains on a phone. Every time you send data, no matter how small, the radio is powered on for up for 20-30 seconds. Every decision you make should be based on minimizing the number of times the radio powers up”. Using the huffman encoding scheme, we can minimize the time spent passing data over to the receiving device. [1] “Standard encoding schemes, such as the ASCII and Unicode systems, use fixed-length binary strings to encode characters, with 7 bits in the ASCII system and 16 in the Unicode system.” So, for example, an English document whose length is 100 million characters would require at least 7 megabits to represent in ASCII and 16 megabits to represent in Unicode. This is a waste of bits because there are some characters, like the letters “e” and “t,”that are used so often that it is a waste of space to use the same number of bits for them.

3. SYSTEM ARCHITECTURE

In both systems, applications A and B use the MessageApi to send and receive data. The Message Api enables device components to send messages to other nodes. The way that the MessageApi works is by passing messages to connected network nodes or in our case a handheld device via bluetooth. [3]

System I, application A passes data from Handheld to Wearable and the process compresses data on the handheld device before passing the compressed data over to the wearable device, as shown in **figure 1** below. System I, application B uses no compression at all, as shown in **figure 2**.

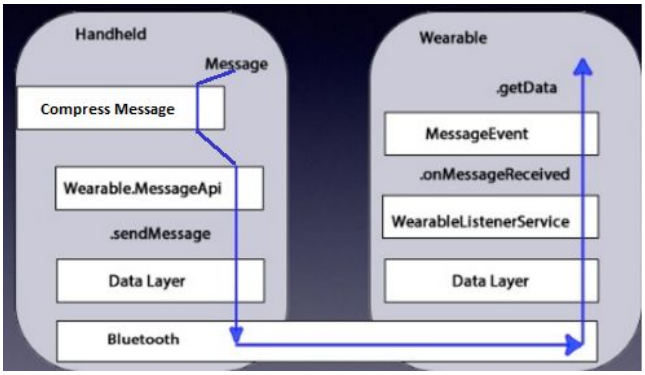

Figure 1: System I - Handheld to Wearable - application A Involves compression and sends out the compressed message.

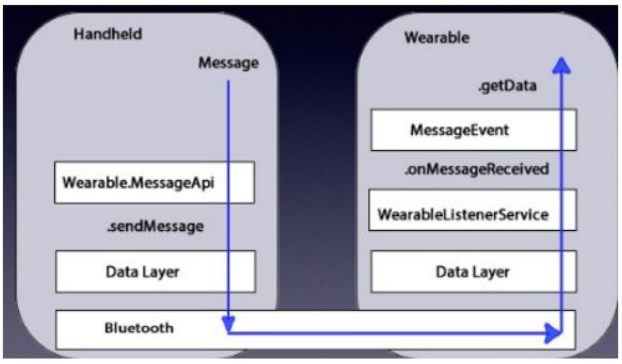

Figure 2: System I - Handheld to Wearable - application B Does not use compression and sends out the original uncompressed message.

A greedy approach to solving the text compression problem called Huffman coding was used in Application A to compress data from a text file before transferring data over to the wearable. The algorithm in **figure 3** was used in Application A to build the coding tree T, given a Set C of characters and the frequency of times they appear.

Algorithm Huffman(C):
Input: A set, C, of d characters, each with a given weight, f (c)
Output: A coding tree, T , for C, with minimum total path weight

Initialize a priority queue Q.

for each character c in C **do**
  Create a single-node binary tree T storing c.
  Insert T into Q with key f (c).

while Q.size() > 1 **do** {
  f 1 ← Q.minKey()
  T 1 ← Q.removeMin()
  f 2 ← Q.minKey()
  T 2 ← Q.removeMin()
  Create new binary tree T with left subtree T 1 and right subtree T.
  Insert T into Q with key f 1 + f 2.
}
return tree Q.removeMin()

Figure 3: Huffman Lossless Compression Algorithm

System II, application A passes data from a wearable device to a handheld device. The process compresses data on the wearable device before passing the compressed data over to the handheld device, as shown in **figure 4** below. System II, application B uses no compression at all, as shown in **figure 5**.

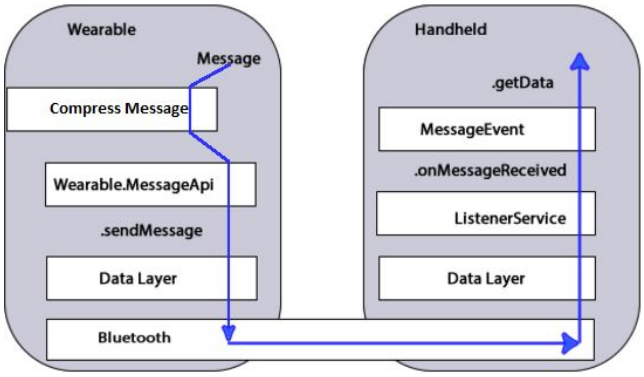

Figure 4: System II - Wearable to Handheld - application A involves compression and sends out the compressed message.

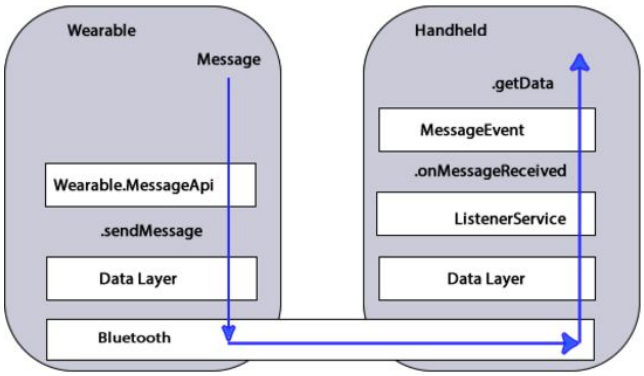

Figure 5: System II - Wearable to Handheld - application B Does not use compression and sends out the original uncompressed message.

4. APPROACH

I was forced to take a theoretical approach. Due to the limited time, I could not develop an automated system for System II's benchmark. I also did not have the time to explore other compression techniques for images and video processing. However, the charts for System I and System II reveal that sending out the compressed data did make less work for the kernel. This is because less data had to be transferred over and in turn the kernel had less work to do. The evidence gathered from System I and System II further inspires a new and improved benchmark to be developed. An ideal benchmark for System II in

the future involves an automated system. System II will be implemented in such a way where it can do a few hundred or thousand of these data transfers in a few hours or days. Overtime an implementation that uses the compression algorithm has the potential to add up to possibly save us battery life in the long run. Our ideal benchmark for System II would involve a long term investigation on the battery sync in order to see when the application of a compression algorithm may be beneficial for a wearable device. It was mentioned in the introduction section of this paper that the ideal benchmark for System I would involve implementing a system where we could evaluate if compression of images and videos would prolong battery life. The reason for this is because it is a more typical type of task that you would see commonly in handheld to wearable applications. To measure long term battery consumption of System II, I found two great tools, the Batterystats and Battery Historian. These tools are used together to analyze battery consumption using Android "bug report" files. Measuring the long term battery use which would have been a more suitable benchmark for this experiment for System II. Also rather than using the MessageApi for both system, the DataApi is more appropriate to use for these sort of transfer rather than the MessageApi. "The DataApi should be used for messages to nodes which are not currently connected (to be delivered on connection). Messages should generally contain ephemeral, small payloads. Use {@linkplain Asset assets} with the DataApi to store more persistent or larger data efficiently. A message is private to the application that created it and accessible only by that application on other nodes."[3]

One of the biggest challenges was debugging the Huffman encoding and decoding. I had trouble understanding why I could not expand the compressed data correctly. I had debugged for a few days until I found out I wasn't correctly specifying the right encoding scheme to write the compressed data to the text file that was to be sent out from the handheld device to the wearable device. I was originally using UTF-8 encoding which is a variable-length encoding. I kept getting the **?** character symbol which means that the text itself is encoded in some form of single byte encoding but interpreted in one of the unicode encoding. What I needed to use was Latin-1 (ISO-8859-1) which is a single-byte fixed length encoding. Latin-1 encodes just the first 256 code points of the Unicode character set, whereas UTF-8 can be used to encode all code points. At physical encoding level, only codepoints 0 - 127 get encoded identically; codepoints 128 - 255 differ by becoming 2-byte sequence with UTF-8 whereas they are single bytes with Latin-1.

After losing a few days of progress from debugging the Huffman encoding and decoding, I had decided to take a shortcut on my original idea and did not build an automated process. The shortcut I took for the experiment involved compressing a large text file containing content taken from random math class syllabus. I copied and pasted the syllabus a few times until there were exactly 31,808 characters in a string and did not make any additional modifications. This string was used to replicate a typical payload that may be collected from an automated process and then passed out using the MessageAPI.

5. TESTING AND VERIFICATION

In both systems Application A uses compression to compress and pass data and Application B use no compression at all and passes the data uncompressed. For the CPU section, the darker shading in the figures indicate the kernels use of the CPU and the lighter shading in the figures indicate the users use of the CPU. For the memory section, the darker shading in the figures indicate the allocated memory and the lighter shading in the figures indicate the free memory that is available. In both systems, application A and B, shown in **figure 6** and **figure 8**, we examine the darker shading and find that the kernel is doing more work in application B than application A. You can clearly see it even takes longer to send that data over in application B. This is because the the application B is sending a twice the amount of data.

We can see that using compression does in fact save power. Evaluating the monitor in Android Studio we can see that not using compression takes a lot longer to pass the message in both systems. In System I, application B it takes approximately 30 seconds, , shown in **figure 8**, where as in System II it takes approximately 45 seconds to complete, shown in **figure 12**. This is what was expected prior to the experiment since the handheld device is more powerful than the wearable device. In System I, application A it takes approximately 10 seconds to transfer that data over from the handheld device to the wearable device. The trend lines show roughly the same use of CPU In all the charts in figures 6 through 13, however, we can see from the charts that we find the running time of application A is significantly shorter than application B in both systems. My tests were consistent because I tested with both devices being charged at all time and made sure no additional programs, or updates were running to affect the results. I also ran three separate trials to make sure I was getting consistent results.

Clearly in both systems on the receiving device, application B uses more power. This is due to the expansion of the compressed data, and this is seen in **figure 7** and **figure 13**. The first peak of **figure 7** and **figure 13** is caused by the receiving of the data that was passed over from the sending device and the second peak is the expansion of the compressed data.

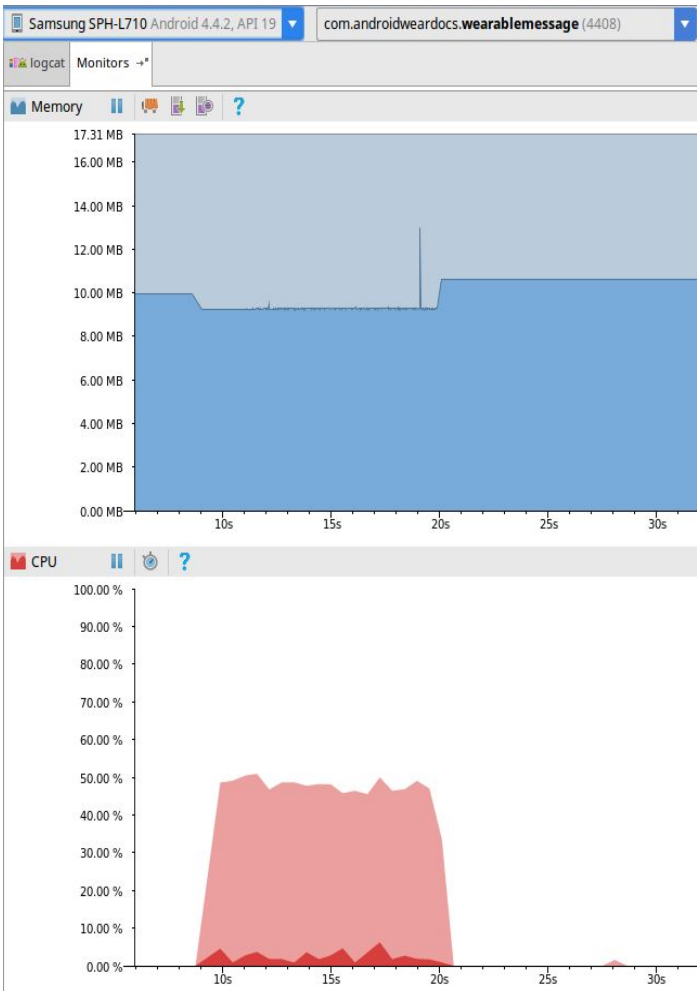

Figure 6 System I - Handheld to Wearable - Application A (handheld-monitor): With compression, exactly 31,808 characters compressed and sent by handheld device to wearable to decode.

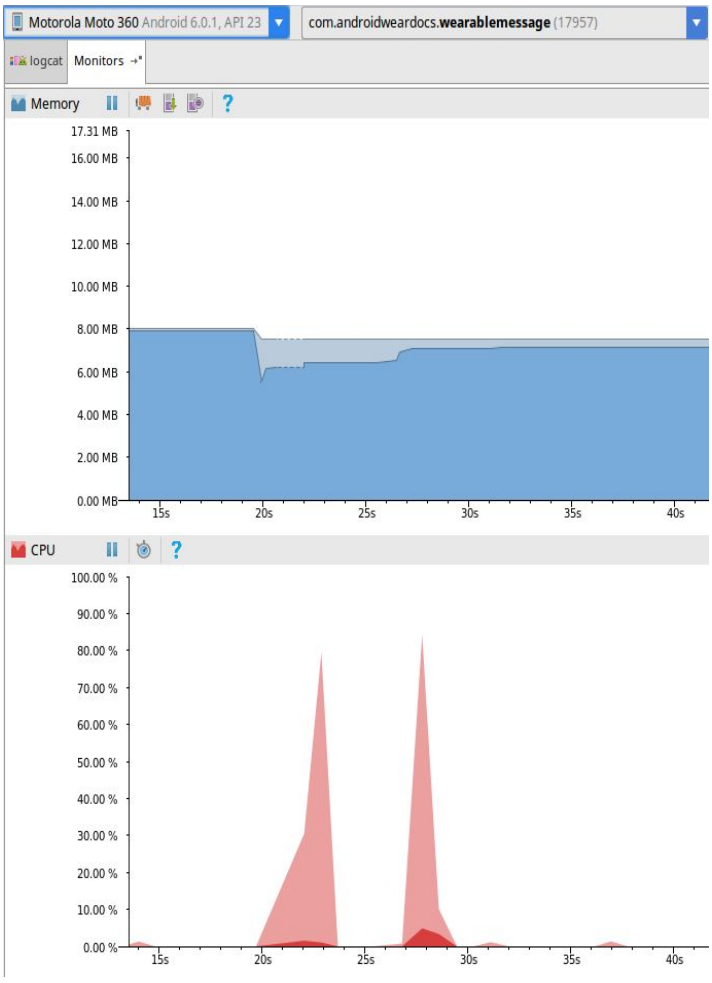

Figure 7 System I - Handheld to Wearable - Application A (wearable-monitor): With compression, exactly 31,808 characters compressed and sent by handheld device to wearable to decode. Second peak is expansion of the compressed text.

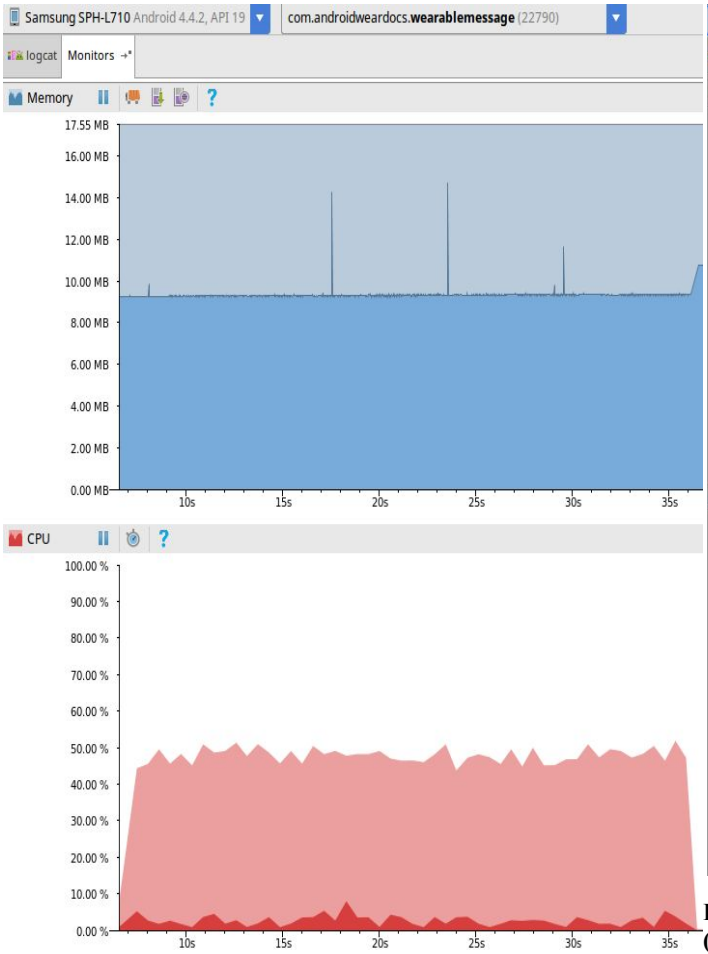

Figure 8 System I - Handheld to Wearable - Application B (handheld-monitor): Without compression, exactly 31,808 characters sent by wearable device to handheld device.

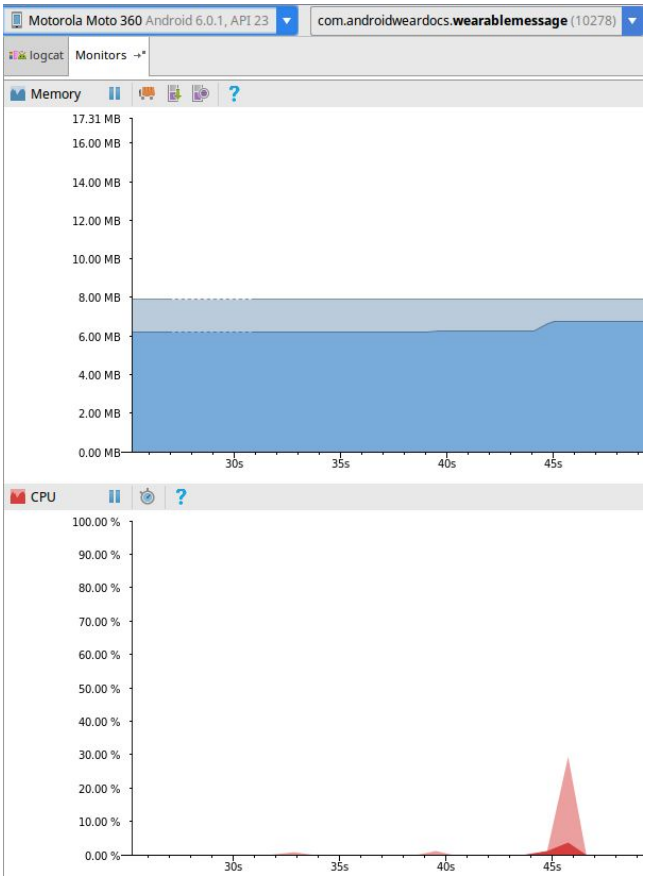

Figure 9 System I - Handheld to Wearable - Application B (wearable-monitor): Without compression, exactly 31,808 characters sent by handheld device to wearable.

Figure 10 System II - Wearable to Handheld - Application A (wearable-monitor): With compression, exactly 31,808 characters compressed and sent by wearable device to handheld device to decode.

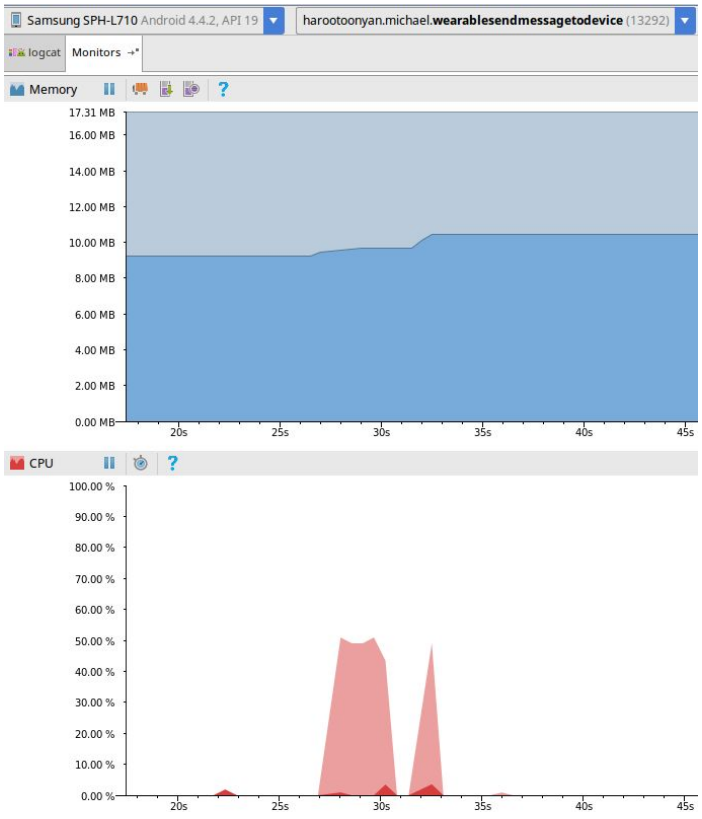

Figure 11 System II - Wearable to Handheld - Application A (handheld-monitor): With compression, exactly 31,808 characters compressed and sent by wearable device to handheld device to decode. Second peak is expansion of the compressed text.

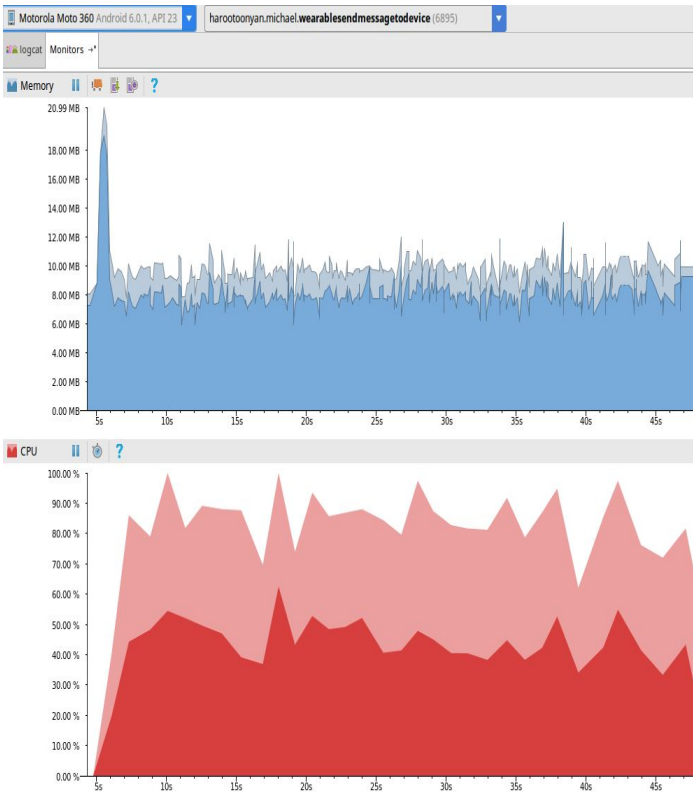

Figure 12 System II - Wearable to Handheld - Application B (wearable-monitor): Without compression, exactly 31,808 characters sent by wearable device to handheld device.

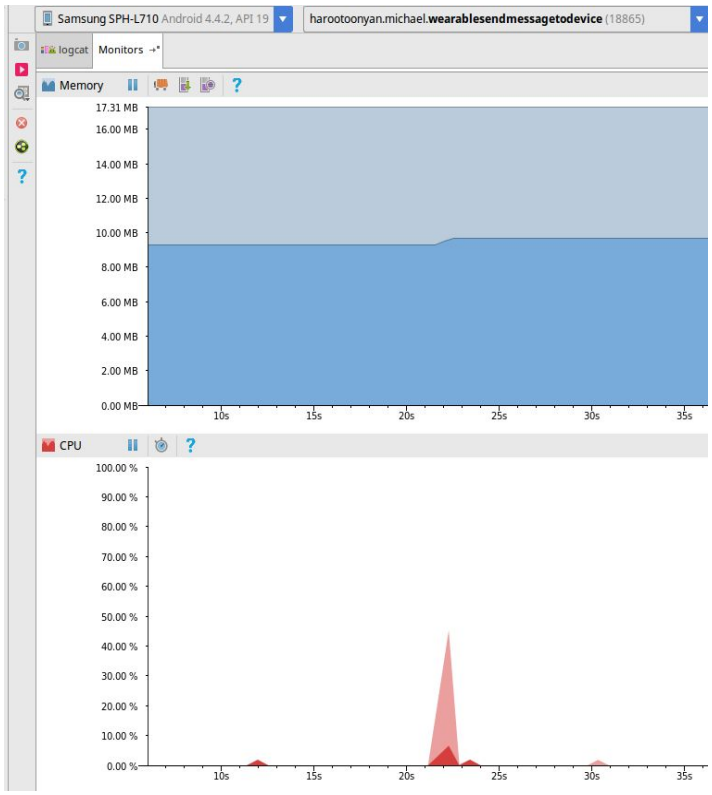

Figure 13 System II - Wearable to Handheld - Application B (handheld-monitor): Without compression, exactly 31,808 characters sent by wearable device to handheld device.

Figure 14 Source code Implementation of sendData function of System II - Wearable to Handheld - Application B

```
public void sendData() throws IOException {
  String message = ""; // about 65k chars limit

  //in order to run application B switch to false to use non
compression version
  boolean isCompressionVersion = false;

  // Writing the uncompressed text file from scratch or over an
existing one.
  PrintWriter writer = new
PrintWriter(dir+"uncompressed.txt", "UTF-8");
  writer.println(mStringData.STRING_DATA);
  writer.close();

  int fileLength = -1;
  InputStreamReader inputStreamReader = null;
  if(isCompressionVersion) {
    // Setting up the input and output file for Huffman
compression and using ISO-8859-1 because
    // we have written codepoint chars 128-256, as one BYTE,
instead of 2 BYTES like in UTF-8.
    System.setIn(new FileInputStream(dir +
"uncompressed.txt"));
    System.setOut(new PrintStream(dir + "compressed.txt",
"ISO-8859-1"));

    // Performing Huffman compression.
    Huffman.compress();


    // taking the compressed file creating a new FIS and
storing it in System.in
    System.setIn(new FileInputStream(dir +
"compressed.txt"));

    // need the file length to read all bytes of our compressed
file
    fileLength = (int) new File(dir +
"compressed.txt").length();

    // isr that reads System.in in ISO-8859-1 format
    inputStreamReader = new InputStreamReader(System.in,
"ISO-8859-1");
  } else {
    // taking the uncompressed file creating a new FIS and
storing it in System.in
    System.setIn(new
FileInputStream(dir+"uncompressed.txt"));
    fileLength = (int) new
File(dir+"uncompressed.txt").length();
    inputStreamReader = new InputStreamReader(System.in,
"UTF-8");
  }
  // fill our bytes array with the compressed data
  char[] bytes = new char[fileLength];
  inputStreamReader.read(bytes, 0, fileLength);

  // convert each byte to its char representation and append to
String message
  for (int i = 0; i < bytes.length; i++) {
    int k = (int) bytes[i];
    char p = (char) k;
    message += p;
  }


  //Requires a new thread to avoid blocking the UI
  new SendToDataLayerThread("/message_path",
message).start();
}
```

System I's sendData() implementation is identical to System II's. You can see by switching the statement over to true, runs application A, and switching the statement over to false, runs application B. The tests conclude that using compression in wearable application's does in fact save battery.

6. REFERENCES

[1] Algorithm Design and Applications [Michael T. Goodrich, Roberto Tamassia] .

[2] Cell Radio Optimization http://highscalability.com/blog/2013/9/18/if-youre-programming-a-cell-phone-like-a-server-youre-doing.html. Accessed April 13, 2017.

[3] Accessing Google APIs https://developers.google.com/android/reference/com/google/android/gms/wearable/MessageApi. Accessed April 30, 2017