Compression - An Approach for Energy Conservation in Wireless Networks

Sudhansh Sharma
Jaipuria Institute of Management Studies(JIMS)
Vasundhara,Ghaziabad,U.P., India
sudhansh74@rediffmail.com

Neetu Sharma
Gurukul - The School
Ghaziabad,U.P., India
neetu.sharma@gurukultheschool.in ,

Durgansh Sharma
Jaipuria Institute of Management,
Noida, U.P., India
durgansh@gmail.com

ABSTRACT

IEEE 802.11 has become more and more popular due to its low cost and easy deployment, it does not provide quality of service (QoS) support. QoS refers to the ability of network to provide some consistent services for data transmission. Thus, a lot of research works have been carried out to enhance the QoS support in IEEE 802.11 networks. Improvement in the QoS involves Data Transfer Rate(DTR) optimization as a prime factor for reliable data transfer in energy efficient manner. The performed work explores data compression as a technique to optimize the Data Transfer rate (DTR), because reduction in the effective size of the data to be transferred on the network leads to reduce the effective file transfer time and hence preserves the network energy.

KEYWORDS

IEEE 802.11 Wireless Networks
Energy Conservation
Huffman Compression
QoS

PREAMBLE

The development of Information Communication Technology (ICT) following Moore's Law has resulted in a situation where network users are able to make use of a wealth of versatile services. But, batteries desired to operate the electronics technology has not followed this development, which has resulted in a situation where network device user's battery can only enable a few hours of active use. Therefore, we need to focus increasingly on energy efficient wireless communication to reduce energy consumption, and also to cut down greenhouse emissions. It has been observed that there is significant energy consumption, while transmitting data over wireless networks. So, data compression techniques are one of the simplest way to trade the overhead of compression for less communication energy[i][ii]. Work performed here demonstrates the usages of data compression to reduce the energy consumption in modern devices; this involves implementation of Huffman coding, a data compression algorithm and using its outcome to secure the network energy significantly.

REVIEW OF LITERATURE

In a traditional LAN we are connecting computers to the network through cables. But the wireless local area network (WLAN) is a flexible data communications system that can use either infrared or radio frequency technology to transmit and receive information over the air. Here each computer has a radio Modem and Antenna with which it can communicate with other systems. One important advantage of WLAN is the simplicity of its installation. Installing a wireless LAN system is easy and can eliminate the needs to pull cable through walls and ceilings. WLANs allow greater flexibility and portability than do traditional wired local area networks (LAN). IEEE 802.11 was implemented as the first WLAN standard. It is based on radio technology operating in the 2.4 GHz frequency and has a maximum throughput of 1 to 2 Mbps.

A wireless Local area network (WLAN) is a flexible data communication system implemented as an extension to, or as an alternative for a wired LAN. As the name suggests a wireless LAN is one that makes use of wireless transmission medium, i.e. wireless LAN transmits and receives data over air, and minimizing the need for the wired connection. Thus wireless LAN combines data connectivity with user mobility. WLANs also allow greater flexibility and portability than traditional wired LAN which requires a wire to connect a user computer to the network. The initial cost for WLAN hardware can be higher than the cost of wired LAN hardware. But the overall installation expenses and lifecycle cost can be significantly lower. With WLAN users can access shared information without looking for a place to plug in, and network managers can setup or argument networks without installing or moving wires. There are many reasons people choose to deploy a wireless LAN, Increase the productivity due to increase mobility, Lower infrastructure cost compared to wired networks, Rapid deployment schedules etc.

The IEEE 802.11 Wireless Local Area Network (WLAN) is one of the most widely deployed wireless network technologies in the world today. Although IEEE 802.11 has become more and more popular due to its low cost and easy deployment, it does not provide quality of service (QoS) support. QoS refers to the ability of network to provide some consistent services for data transmission. Improvement in the QoS involves various factors like network energy conservation, network time utilization, Data Transfer Rate (DTR) optimization etc. as the prime factors for reliable data transfer in energy efficient manner.

To improve the QoS of the WLAN network, we explored data compression as a technique to optimize the data transfer rate in wireless networks. Reduction in the effective size of the data to be transferred on the network leads to reduce the effective file transfer time and hence preserves the network energy. The performed work exhibits the conservation of energy in both transfer of file over the network, by reducing the file transfer time through the compression and decompression mechanism performed at server and client side respectively, apart from this the energy of the battery consumed at both ends of the network is also saved[i][ii].

In the performed work, Huffman coding is implemented for file compression and its outcome is used in client server environment, to study the improvement in network DTR. Huffman codes are intact the Prefix codes, which is a type of code system (typically a variable-length code)

distinguished by its possession of the "prefix property"; which states that there is no valid code word in the system that is a prefix (start) of any other valid code word in the set.

A code for a message set is a mapping from each message to a bit string. Each bit string is called codeword, which are denoted using the syntax C={(S1,W1),(S2,W2),……..(SM,WM)} Typically in computer science we deal with fixed length codes, such as the ASCII code which maps every printable character and some control characters into 7 bits. For compression, however, we would like code words that can vary in length based on the probability of the message. Such variable length codes have the potential problem that if we are sending one codeword after the other it can be hard or impossible to tell where one codeword finishes and the next starts. For example- given the code {(a,1),(b,01),(c,101),(d,011)}, the bit-sequence 1011 could either be could either be decoded as aba, ca, or ad. To avoid this ambiguity we could add a special stop symbol to the end of each codeword (e.g., a 2 in a 3-valued alphabet), or send a length before each symbol. These solutions, however, require sending extra data.

A more efficient solution is to design codes in which we can always uniquely decipher a bit sequence into its code words. We will call such uniquely decodable code ,a prefix code which is a special kind of uniquely decodable code in which no bit-string is a prefix of another one, for example {(a,1),(b,01),(c,101),(d,011)} 1 01 000 001 . All prefix codes are uniquely decodable since once we get a match, there is no longer code that can also match[iv].

Huffman codes are optimal prefix codes generated from a set of probabilities by a particular algorithm, the Huffman Coding Algorithm. The algorithm is now probably the most prevalently used component of compression algorithms, used as the back end of GZIP, JPEG and many other utilities.

Huffman coding finds the optimal way to take advantage of varying character frequencies in a particular file. On average, using Huffman coding on standard files can shrink them anywhere from 10% to 30% depending to the character distribution. (The more skewed the distribution, the better Huffman coding will do.)

The idea behind the coding is to give less frequent characters and groups of characters longer codes. Also, the coding is constructed in such a way that no two constructed codes are prefixes of each other. This property about the code is crucial with respect to easily deciphering the code.

ALGORITHM

1. **The two free nodes with the lowest weights are located.**
2. **A parent node for these two nodes is created. It is assigned a weight equal to the sum of the two child nodes.**
3. **The parent node is added to the list of free nodes, and the two child nodes are removed from the list.**
4. **One of the child nodes is designated as the path taken from the parent node when decoding a 0 bit. The other is arbitrarily set to the 1 bit.**
5. **The previous steps are repeated until only one free node is left. This free node is designated the root of the tree.**

To reconstruct the data from the compressed file, we need to decode the file data. File decoding requires the encoded file and Huffman code tree. Before, reading bits from encoded file a pointer pointing on the root of Huffman code tree is defined. Then the encoded file is read in bitwise order ; if it's 0, then the left child of the current node is traversed; if it's 1, go to the right child of the node. The process of traversal is repeated till the leaf node is reached, arrival at the leaf node decodes a character. The decoded character is written to the decoded file, repoint the pointer on the Huffman code tree root and continue reading from encoded file until you reach the file end[iii].

RESEARCH OBJECTIVES

- To implement The Huffman Code for data compression and use it in client server environment, for the analysis of the improvement in network Data Transfer Rate DTR, achieved through Compression.
- To determine the effectiveness of implemented Compression Algorithm, by evaluating the compression ratio.

RESEARCH METHODOLOGY

The implementation involves compression and decompression of data at server and client side respectively, as a tool to conserve energy of the network. The Data is compressed at the server side and transferred from the server to the client side where the received data is decompressed to regain its original format. Due to the reduction of file size it takes lesser time and hence lesser network energy, for the transfer of data over the network. Which is of prime concern when the networks are wireless, because saving the time in a wireless network, for data transfer directly impacts on the saving of energy related to the network and devices attached to the network like laptops, mobiles etc, by saving their battery consumption. The Compression and Decompression of file is performed through Huffman coding. So, the project implementation as a whole involves following :

1. File Compression & Decompression[v]
2. Client Server communication & file transfer[vi]

FILE COMPRESSION & DECOMPRESSION

The file to be transferred in client server environment, is compressed at the server side and then passed on to the client, where it is received and then decompressed to its original format. The compression and decompression are performed by implementing the through Huffman algorithm.

Huffman compression is a lossless compression algorithm that is ideal for compressing text or program files. Huffman compression belongs into a family of algorithms with a variable codeword length. That means that individual symbols (characters in a text file, for instance) are replaced by bit sequences that have a distinct length. So, symbols that occur a lot in a file are given a short sequence while others that are used seldom get a longer bit sequence[v].

The file compression and decompression involves the development of following codes :

1. bool CompressHuffman(BYTE *pSrc, int nSrcLen, BYTE *&pDes, int &nDesLen);
2. bool DecompressHuffman(BYTE *pSrc, int nSrcLen, BYTE *&pDes, int &nDesLen);

COMPRESSION

1. The compression code starts by initializing 511 of Huffman nodes by its ASCII values:

```
CHuffmanNode nodes[511];
for(int nCount = 0; nCount < 256; nCount++)
nodes[nCount].byAscii = nCount;
```

2. Then, it calculates each ASCII frequency in the input buffer:

```
for(nCount = 0; nCount < nSrcLen; nCount++)
nodes[pSrc[nCount]].nFrequency++;
```

3. Then, it sorts ASCII characters depending on frequency:

```
qsort(nodes,256,sizeof(CHuffmanNode),
frequencyCompare);
```

4. Then, it constructs Huffman tree, to get each ASCII code bit that will be replaced in the output buffer:

```
int nNodeCount = GetHuffmanTree(nodes);
```

Constructing Huffman involves putting all nodes in a queue, and replacing the two lowest frequency nodes with one node that has the sum of their frequencies so that this new node will be the parent of these two nodes. And do this step till the queue just contains one node (tree root).

```
// parent node
pNode = &nodes[nParentNode++];
// pop first child
pNode->pLeft = PopNode(pNodes,
nBackNode--, false);
// pop second child
```

```
pNode->pRight = PopNode(pNodes, nBackNode--, true);
// adjust parent of the two poped nodes
pNode->pLeft->pParent = pNode->pRight->pParent = pNode;
// adjust parent frequency
pNode->nFrequency = pNode->pLeft->nFrequency + pNode->pRight->nFrequency;
```

5. Then, the final step in the compression is to write each ASCII code in the output buffer:

```
int nDesIndex = 0;
// loop to write codes
for(nCount = 0; nCount < nSrcLen; nCount++)
{
*(DWORD*)(pDesPtr+(nDesIndex>>3)) = nodes[pSrc[nCount]].dwCode << (nDesIndex&7);
nDesIndex += nodes[pSrc[nCount]].nCodeLength;
 }
```

- (nDesIndex>>3): >>3 to divide by 8 to reach the right byte to start with.
- (nDesIndex&7): &7 to get the remainder of dividing by 8, to get the start bit.

At the compressed buffer, we save Huffman tree nodes with its frequencies so we can construct Huffman tree again at the time of decompression (just the ASCIIs that have a frequency).

DECOMPRESSION

The decompression involves the constructed Huffman tree, then loop in the input buffer to replace each code with its ASCII. the input buffer, in this case, is a stream of bits that contain the codes of each ASCII. To replace the code with the ASCII, we need to iterate Huffman tree with the bit stream till we find a leaf [v]. Then, we can append its ASCII at the output buffer:

```
int nDesIndex = 0;
DWORD nCode;
while(nDesIndex < nDesLen)
{
 nCode = (*(DWORD*)(pSrc+(nSrcIndex>>3)))>>(nSrcIndex&7);
      pNode = pRoot;
      while(pNode->pLeft)
          {
              pNode = (nCode&1) ? pNode->pRight : pNode->pLeft;
              nCode >>= 1;
              nSrcIndex++;
           }
    pDes[nDesIndex++] = pNode->byAscii;
}
```

- (nSrcIndex>>3): >>3 to divide by 8 to reach the right byte to start with.
- (nSrcIndex&7): &7 to get the remainder of dividing by 8, to get the start bit.

CLIENT SERVER COMMUNICATION & FILE TRANSFER

The file compressed at the server side is passed to the client side where it is decompressed; however the client server communication involves the implementation of TCP/IP protocol and simple file transfer protocol, by using MFC CSocket class. The data to be communicated between client and server is a compressed file, compressed through Huffman algorithm. The work performed, related to server and client side coding is as follows :

SERVER CODE

There are three major functions performed by this code:

1. first listen for and establish a connection with the client,
2. next send the client the length of the file,
3. finally send the file to the client in chunks.

In the first part, the code creates a CSocket object named sockSrvr and configures it to listen on a pre-

designated port (which must be known to the server, #defined 8686).

Secondly, when a connection request is received on the port, a CSocket object named sockConnection is created to handle the connection. The connection is accepted by the sockConnectionobject in the call to CSocket::Accept(), which actually hands the connection off to a new port address, leaving sockSrvr free to listen for further connection requests on the originally-designated port.

Finally, after the acceptance of connection, the server imposes simple protocol for file transfer. Before actual transfer of file data, the server sends the total length of the file (in bytes). CFile::GetLength() retrieves the file's length in bytes, and the next function, htonl(), compensates for differences in machines that store integers in big-endian versus little-endian format i.e. htonl() is used to ensure platform independence of raw socket code, and to ease porting issues form one machine to another. Since we're using CSocket, there's no chance that the code will be used on anything but a Windows/Intel platform. The code involves a loop to send the length of the file to the client. In the loop, CSocket::Send() is called repeatedly until all bytes of the file's length are sent to the client[vi].

CLIENT CODE

There are many parallels between the client-side code and that of the server, as before, there are three main parts:

1. making a connection,
2. getting the file's length,
3. getting the file's data in chunks.

In the first part, a CSocket object named sockClient is created and attempts a connection to the server on the pre-designated port(8686). The address of the server is specified by theCString object named strIP which can store either a dotted IP address or a machine name.

Once the connection is made, the second part callsCSocket::Receive() in a loop to get the length of the file, which is converted by the ntohl() function into big-endian or little-endian format as appropriate.

In the third part, a BYTE buffer of size RECV_BUFFER_SIZE is allocated from the heap, and CSocket::Receive() is called in a loop until all bytes of the file are received. Note that RECV_BUFFER_SIZE can be different from SEND_BUFFER_SIZE, although in this code, they are the same (both are #defined to 4096).

The only tricky part of this code is the determination of the number of bytes to ask for in CSocket::Receive(). It is coded to get as many bytes as possible, up to the size of the buffer, except in the last call to CSocket::Receive(), in which we want only the remaining bytes in the file. The C++ ternary operation (i.e., (condition)?(cond=TRUE):(cond=FALSE)) is used for this purpose. The code works even if CSocket::Receive() does not retrieve the number of bytes requested. For example, suppose 4096 bytes are asked(i.e.,iiGet ==4096) but CSocket::Receive() only returns 2143 bytes. Then only write the number of bytes actually received, and update the remainder in the number of bytes left to receive, based only on the number actually received, not the number asked for[v].

ANALYSIS AND INTERPRETATION

The file compression mechanism implemented here, is Huffman Code Algorithm, which leads to lossless compression. This class of compression is used to preserve the file bits, because while transferring the files over the network if there is any loss of bits then the file is damaged, thus it will be of no use at the receiver end. Further, to conserve the network energy, reliable and efficient file compression and decompression mechanism is highly desired. In the performed work, it is demonstrated and analyzed that how the compression and decompression mechanism are implemented and are contributing to conserve the network energy. Here, it is analyzed by recording and comparing the file transfer time of uncompressed file with that of the compressed file.

The results are as follows:

1. Average Compression Ratio = 0.40
2. Average Percent Compression = 60%
3. Average Percent Data Transfer Time Saved = 47%
4. Average Percent improvement in DTR =

6.5%

The observed variation in compression ratio is graphically shown in figure 1.1

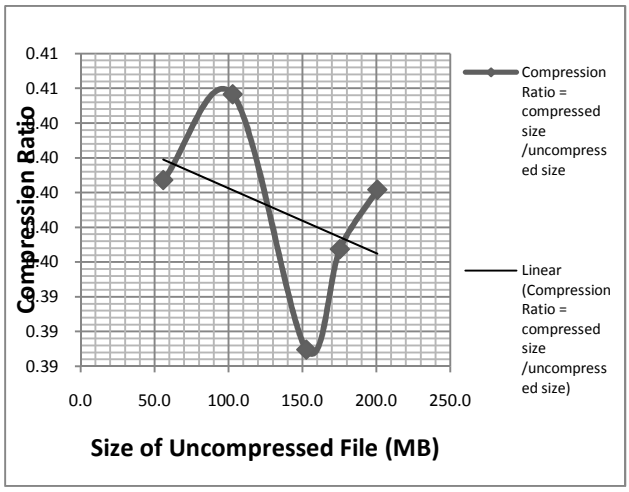

Figure 1.1: Compression Ratio Vs Size of Uncompressed File (MB)

The graph, shown in figure 1.1 above, appears to be sinusoidal, but the least square fitting of the curve, shows the linear variation in the compression ratio with the file size. However the observed variation in the compression ratio is the result of the content of the independent files, and as the implemented compression mechanism is Huffman coding which is a lossless compression mechanism by nature. Thus, some of the files which are rich in images or so, achieved lesser compression ratio. But, the files with more of the textual content and less of the images shows better compression ratio. The observed variation in compression ratio is from 0.39 to 0.41, the average compression ratio is 0.40 with a standard deviation of ~0.01. Further, the implemented compression mechanism works well and achieves 60% of average percent file compression, which is quite significant.

FINDING AND DISCUSSION

The level of file compression achieved here, has a positive impact over network energy consumption. The same is demonstrated by recording and comparing the file transfer time of the uncompressed file with that of the compressed file. It is observed that, due to compression, on an average 47% of the Data Transfer Time is saved. Further, it is observed that due to compression, average improvement in the Data Transfer Rate is 6.5% .

LIMITATION OF THE STUDY

The performed study has only evaluated the implementation of one compression algorithm i.e. Huffman coding, for evaluating the effect of compression over network energy conservation. The improvement in network DTR can be evaluated by considering various other compression algorithms.

CONCLUSION

Through the Performed work it is observed that compression of data, while transferring it over the network saves the network energy by reducing the file transfer time. If the compression and decompression codes are used at transmitter and receiver end (or Client and Server end) respectively, then the combination can be quite useful and energy conserving, in the sense, instead of sending a big file from server to client (or one node to other), which in fact will consume more network data transfer time. It will be beneficial to compress the file at the transmission end and then pass it over the network, and the receiver will decompress it in to its original format, hence less of the network transmission time will be consumed which in turn will save the energy of all the devices connected to the network.

The performed work is specifically useful for the wireless networks, because the wireless network devices are generally battery operated, and if these devices seek more time in sending and receiving the files from the network, the more the battery or energy is consumed. Thus file compression with suitably good compression ratio and decompression mechanism, will definitely saves energy of wireless network.

The interpretation of the results , leads to the conclusion that, “The higher the compression ratio is, the lesser is the file transfer time, hence lesser is the energy consumption of the network”

REFERENCES

i. Prof. Rathnakar Acharya, Dr. V. Vityanathan, Dr. Pethur Raj Chellaih "WLAN QoS Issues and IEEE 802.11e QoS Enhancement "International Journal of Computer Theory and Engineering, Vol. 2, No. 1793-8201,pg 143-149, 1 February, 2010

ii. Le Wang Manner, J., "Evaluation of data compression for energy-aware communication in mobile networks", Proceedings Cyber-Enabled Distributed Computing and Knowledge Discovery, 2009. Cyber C '09. International Conference on 10-11 Oct. 2009, p: 69 - 76

iii. I. F. Akyildiz et al, "Wireless sensor networks: a survey," Computer Networks, vol. 38, pp. 393-422, March 2002.

iv. Mark Nelson ,Jean Loup Gailly, "The Data Compression", Second Edition, Publisher: IDG Books Worldwide, Inc.,ISBN: 1558514341

v. "Simple & Fast Huffman Coding", http://www.codeproject.com/KB/recipes/Huffman_coding.aspx

vi. "Network Transfer Of Files Using MFC's CSocket Class",http://www.codeproject.com/KB/IP/SocketFileTransfer.aspx

vii. Arya V., Mittal A., Joshi R. C., "An Efficient Coding Method for Teleconferencing and Medical Image Sequences", Proceedings of Third International Conference on Intelligent Sensing and Information 2005, pp. 8-13, 14-17 December 2005.

viii. Wen-Jyi Hwang, Ching-Fung Chine, Kuo-Jung Li, "Scalable Medical Data Compression and Transmission using Wavelet Transform for Telemedicine Applications", IEEE Transactions on Information Technology in Biomedicine, Vol. 7, No. 1, pp. 54-63, March 2003.

ix. Benedettli A., Scarabottolo N., "Towards a Dedicated Compression Pipeline for Document Image Archicing", Proceedings of Workshop on Document Image Analysis 1997, pp. 40-43, 20th January 1997.

x. Yi Sun, Yi-Jin Yang, Phing Zhou, "Wavelet-Based Compression of Terrains", Proceedings of IEEE International Geo-science and Remote Sensing Symposium 2003, Vol. 3, pp. 2030-2032, 21-25 July 2003.

xi. Mello C. A¿ B.,"Synthesis of Images of Historical Documents for Web Visualization", Proceedings of 10th International Multimedia M0delling Conference 2004, pp. 220-226, January 2004.

xii. Capon J., "A Probabilistic Model for Run-Length Coding of Pictures", IRE Transactions on Information Theory, pp. 157-163, 1959.

xiii. Huffman D. A., "A Method for the Construction of Minimum Redundancy Codes", Proceedings of IRE, Vol. 40, No. 10, pp. 1098-1101, 1952.

xiv. Abramson N., "Information Theory and Coding", McGraw-Hill, New York, 1963.

xv. Welch T. A., "A Technique for High-Performance Data Compression", IEEE Computer, pp. 8-19, 1984.

xvi. Freeman A. (translator), Fourier J., "The Analytical Theory of Heat", Cambridge University Press, 1878.

xvii. Jayant N. S. and Noll P., "Digital Coding of Waveforms", Prentice Hall, 1984.

xviii. Antionini M., Barlaud M., Mathieu P. and Daubechies I., "Image Coding using Wavelet Transform", IEEE Trans. on Image Proc., Vol. 1, No. 2, pp. 205-220, April 1992.

xix. WAP Forum Ltd., "WAP WAE Specification - Version 1.1", May 1999.