Compressibility of WML and WMLScript byte code: Initial results

Eetu Ojanen and Jari Veijalainen

Department of Computer Science and Information Systems
University of Jyväskylä
P.O.Box 35 Jyväskylä
Finland
{esajanj,veijalai}@cs.jyu.fi

Abstract

Rapid technical development of wireless cellular networks and the strong proliferation of hand-held mobile terminals among users on one hand, and the development of the Internet technologies on the other hand, have made evident the need to combine these technologies. In 1997, an industry-driven organization called WAP-Forum was established to develop technical standards that bridge the gap between the mobile telephone networks and the WWW world. The first versions (1.0) of Wireless Application Protocol (WAP) standards have been accepted by the Forum in April 1998 and the current versions (1.1) in June 1999. Versions 1.2 are now (12/1999) developed.

Mobile terminals are small in size, have a relatively small memory and processor capacity and have small batteries. The wireless bandwidth in GSM and other similar second generation networks is also rather limited as compared to wire-line, networks, ranging from 9.6 kbps to ca. 170 kbps. In this paper we examine how much the byte code used in the WAP environment can be compressed and whether the reduced transmission time of the application data warrants the increased memory and processor overhead caused by the compression and decompression.

1. Introduction

Rapid technical development of wireless cellular networks and the strong proliferation of hand-held mobile terminals among users on one hand, and the development of the Internet technologies on the other, have made evident the need to make these technologies interoperable. In 1997, an industry-driven organization called Wireless Application Protocol Forum Ltd was established to develop technical standards that bridge the gap between the mobile telephone networks and the WWW world. The latest versions (1.1) of Wireless Application Protocol (WAP) standards were adopted in June 1999. The standards define a protocol stack (WAP suite) that will run in the terminal and in an entity called WAP gateway, an XML-based language called WML (Wireless Markup Language) and a scripting language called WMLScript. The latter are used to program applications that are loaded from a WAP-server over the network and run by the WML-microbrowser in a similar manner as JavaScript, are run by a WWW-browser. The first terminals supporting the WAP protocol stack and the above languages are available on the market place starting autumn 1999, among the first ones Nokia 7110.

Mobile terminals are small in size, have a rather limited memory and processor capacity and have batteries with small capacity. The wireless bandwidth in GSM and other similar networks is also rather limited as compared to wire-line networks, ranging from 9.6 kbps in basic GSM to 57.6 kbps in HSCSD, up to ca. 170 kbps in GPRS. In this paper we examine how much the byte code used in the WAP environment can be compressed and whether the reduced transmission time of the application data warrants the increased memory and processor overhead caused by the compression and decompression.

The paper is organized as follows. In Section 2 we give a short overview of the WAP environment including WML and WMLScript languages. In Section 3 we present empirical compression rates for a set of WML and WMLScript byte code files with sizes varying between 18 and ca. 1400 bytes. The algorithms used in the experiment are gzip (deflate), bzip2, ELS (Unary Logarithmic Scale), arithmetic encoding, Burrows-Wheeler Transform, Move-To-Front encoding, and Run-Length encoding. In Section 4 we investigate the space and time complexity of the above compression algorithms and establish principles that state when it would make sense to use compression. Important parameters in these considerations are the transmission speed between the terminal and the server and the decompression speed at the terminal. Section 5 concludes.
2. Wireless Application Protocol Environment

Wireless Application Protocol (WAP) is a new wireless technology developed by an industry consortium called WAP Forum [14]. The goals of the work are [15]:
- to bring Internet contents and advanced data services to digital cellular phones and other wireless terminals
- to create a wireless global protocol specification that will work across differing wireless network technologies
- to enable the creation of content and applications that scale across a very wide range of bearer networks and device types
- to embrace and extend existing standards and technologies wherever appropriate.

The consortium has defined a set of interrelated standards that specify the environment. The vertical architecture consists mainly of a protocol stack that is supposed to run at mobile terminals, at Internet gateways, at WAP-capable WWW servers, and other similar servers. The lowest layer in the protocol stack contains the bearers, such as GSM data, IS-136, CDMA, UDP/IP, etc. On top of this layer there is the actual WAP stack consisting of Transport Layer (WDP), Security Layer (WTLS), Transaction Layer (WTP), Session Layer (WSP), and Application Layer (WAE).

Horizontally, the architecture specifies a mobile client, a gateway, and the origin server. The latter can be a usual WWW server or a WAP enabled server servicing the requests coming from the client through the gateway. The gateway runs two protocol stacks (TCP/IP+HTTP and WAP stack) and translates the WAP requests to WWW requests conforming to HTTP specification. In the opposite direction it translates the results from HTML to WML (see below). It might also have functionality necessary to cope with the communication autonomy of the terminals (C-autonomy, see e.g. [13]). Because it represents mobile WAP clients towards HTTP servers as WWW clients, the gateway is also called WAP Proxy.

The languages defined for the environment are Wireless Mark-up Language (WML) and WMLScript. The former is an XML based, rather simple mark-up language to define WML card decks (corresponding to HTML pages). For diverse multimedia contents (e.g. images, phone book records, etc.) there are or will be contents formats. The WAP terminals (e.g. cellular phones with WAP capabilities) are assumed to be able to run a Micro-browser that is able to interpret and display WML contents. It is also assumed to run applications written with WMLScript - a scripting language similar to JavaScript™. The WML contents and WMLScript applications are compiled into WML byte code before interpretation. We only use the byte codes below. The reader is urged to consult [14] for more details on WAP.

3. Experimental Compression Results

Our main endeavor was to check, whether the byte code generated from WMLScript can be compressed to such an extent that further studies and possibly real compression mechanisms at terminals and servers are feasible. Therefore, we did not try to develop any special algorithms for the WML world, but we rather just took a few well-known algorithms and ran tests with them.

3.1 The test data

Our test data consisted of about 90 files. We took example WML and WMLScript files distributed with the Nokia WAP SDK 1.1 [10] and Ericsson WAPIDE 2.0B4 [7], but downloaded also some production files through [16]. While looking at the test results, one should keep in mind that the test data consisted of rather small byte code files having size between 18 and 1374 bytes, 416 bytes in average (biggest source was ca. 4 KB). The codes offer only simple application functionality. A rapid growth of real world WML code files can be expected, because only so, more functionality can be offered. Further, new versions of WML and WMLScript specification will most probably be more complicated than the current ones - a fact that also tends to make applications larger in size. The increase in size may change the efficiency rank order of the algorithms included in our experimental tests, but the main message of our experiments seems clear: the bigger the code files, the bigger savings can be achieved by compression. This can easily be verified also by grasping the principles applied in the compression algorithms.

3.2 The algorithms and results

Below, we only compress byte code files. Thus only lossless compression algorithms make sense in this context. There are general and well-known algorithms in this category. We chose to use the latest versions gzip and bzip2 and downloaded them from their respective official home pages [3,9]. The FLS encoder was compiled from the source code accompanying the original paper [6] and all the other algorithms and encoders were distributed with Mark Nelson's article on Burrows-Wheeler transform [4].
The deflate algorithm is probably one of the most famous lossless compression methods applicable to any type of data [5]. It is a combination of Lempel-Ziv 77 [11] algorithm with secondary compression from Huffman encoding and widely used in several compression utility programs, such as gzip [8]. The gzip starts to rapidly increase the result file size as the source file size drops below approx. 300 bytes (see Figure 1). As the average size of our test files was merely 416 bytes, the gzip test resulted in only 9% gain in average.

![Graph showing compression efficiency of gzip, bzip2, and ELS](image)

**Figure 1:** Comparison of compression efficiency of gzip, bzip2, alacritous ELS encoding and Burrows-Wheeler transform followed by Move-To-Front encoding and alacritous ELS encoding. CR = \frac{\text{original size}}{\text{compressed size}}, \text{Gain} = 1-(1/CR)

The characteristics of a WML and WMLScript byte code come pretty close to a text file that has been already processed with a compression algorithm applying dictionary, such as Lempel-Ziv 77 [11]. Therefore, the next compression tests were done with entropy coders, such as order-0 adaptive arithmetic encoding and ELS (Entropy Logarithmic Scale) algorithm [6]. The ELS encoding was done in two different modes: alacritous and inalacritous symbol probability updating. With arithmetic coding or ELS, we could also break the limit of one bit per symbol. While arithmetic coding assigns the symbols to codes with optimal length, its implementations often require multiplication and division operations that might be too slow or power consuming to be used in WAP terminals. The ELS approach is purely integer-based and thus better suited for environment with very limited CPU resources. Surprisingly, the ELS in alacritous mode reached 6% higher gain than adaptive arithmetic encoding with our test material.

The WML and WMLScript language tokens are mapped to byte codes, while the literal strings remain intact. This could explain why we could get only approximately 16% to 22% size reduction from the test material by using only entropy coding algorithms. For us, the next logical step was to add data modeling step before the entropy coding. Our choice was the Burrows-Wheeler transform (BWT), which is also used in a popular compression utility called bzip2. According to the official home page of bzip2 [3], it compresses up to 2 times and decompresses up to six times faster than gzip, that also uses the Huffman method for entropy coding. In order to improve the efficiency of the BWT, Move-To-Front (MTF) encoder and Run-Length Encoding (RLE) can be placed between the transform and entropy coding [4]. We ended up compressing the test material with the following sequences of these algorithms: BWT, BWT+MTF and BWT+MTF+RLE. Each of these sequences was tested with our choice of entropy coders (arithmetic coding and the two symbol probability updating modes of ELS). The combination of BWT+MTF encoded with ELS in alacritous mode resulted in average 24% size reduction per file, which turned out to be the highest gain in all the tests we performed. Finally, we tried the bzip2 compression utility, which was claimed to “typically compress files to within 10% to 12% of the best available techniques, whilst being around twice as fast at compression and six times faster at decompression” [3]. However, bzip2 resulted in average 2% gain as its performance with files smaller than 300 bytes was worse than gzip’s (see Figure 1). The deflate algorithm used by gzip tends to perform better than bzip2 with small files, as was observed by the authors already in [12]; there we compared the efficiency of gzip and bzip2 with data files compressed in different block sizes.

Common to all the tested algorithms is that they tend to compress better, the longer the source files get. The growth of the compression gain will slow down at the point where the original file size exceeds the Lempel-Ziv 77 or Burrows-Wheeler transform block size. The gzip's implementation of Lempel-Ziv 77 can refer up to 32KB backwards, while the bzip2 lets the user specify the Burrows-Wheeler transform block size up to 900KB in steps of 100KB. We used the 900 kilobytes block size with bzip2 and 200,000 bytes block size with BWT. Plain entropy encoders did not increase the result file size even with the smallest test files, but as the test file size grew...
above 300 bytes, was their gain smaller than that of algorithms with data modeling steps. Both gzip and bzip2 write file headers (with CRCs, platform information etc.) that always require a certain amount of space in the beginning of each compressed file. This is the explanation for their relatively poor performance on the smallest test files.

As Figure 1 shows, the three best methods tend to yield the same compression ratio for larger file sizes. Figure 1 also hints that compression method selection should be done based on the size of the code file. It remains to be seen, whether a special compression method designed for WML and WMLScript byte code would perform better than the generally applicable compression methods.

4. Practical relevance of compression of WML and WMLScript byte code

4.1. The effect of bearers on the feasibility of compression

Does compression improve performance or save resources? Let us first look at an example. Let us assume that the uncompressed byte code for an application is at most 160 bytes (which is the case for about twenty files in our sample). This means that it can be transferred from the server to the mobile client in one GSM short message. The actual transfer takes, say, at most one second. Because a short message has fixed length, 160 characters, sending the same byte code compressed for these small applications does not reduce response time. On the contrary, the terminal has to decompress the byte code after getting the whole short message, which it does not need to do with the uncompressed byte code. Thus, compression makes the overall turn-around time longer. How much? This depends on the decompression speed at the terminal. If the decompression happened at the same speed as the data transmission, then it would take at most a second. If it were ten times faster than the data transmission, decompression would take at most 0.1 seconds. In the former case the user might notice the difference, if the overall turn-around time was less than 3 seconds, otherwise he or she would probably neither notice nor care. Regarding transmission costs, the usually applied fixed tariffs for short messages would yield the same cost for compressed and uncompressed application code in this case.

Let us assume that the length of the byte code was uncompressed between 161 and 320 bytes. This changes the situation somewhat if we can compress the code so that it fits into one short message, instead of two. In our sample this happens in the best case for nine applications.

Now, the user would get the application code decompressed and ready to run in less than 2 seconds assuming the decompression speed is higher than the transmission speed and that the compression time at the server is negligible - or that the application code is stored compressed at the server. In the uncompressed case, with the above assumptions, the turn-around time is longer than in the compressed case. Thus, compression would shorten the time the user has to wait for the application to run. Additionally, compression makes the transaction cheaper for the customer, because only one short message needs to be sent instead of two from the server to the terminal. So, from customer's point of view, compression makes much sense in this case, both in terms of saving time and saving money.

The same conclusions can be drawn if we assume that the customer uses connection-based bearers, like CSD and HSCSD, or connection-less GPRS. In the former two cases, necessary connection time becomes shorter if the data is sent compressed and thus the costs come down. The same holds for the packet-switched GPRS, where billing will be based on the number of bytes transferred. If the size of the byte code is less than one kilobyte, then its transfer time over a 9.6-kbps wireless link (CSD) is less than a second. In this case even compression rate 2 does not much matter for the turn-around time, because the connection set-up time is typically several seconds. For HSCSD the transfer speed is up to 57.6 kbps and for GPRS up to 171.2 kbps - which make the gain in turn-around time insignificant for small applications (small meaning byte code smaller than, say, 1 kilobyte). For GPRS the connection set-up time is 0.5 -1 second, so that only for files bigger than 20 kbytes the compression begins to matter with respect to turn-around time. Still, the billing will be based on the number of bytes transferred, compression would decrease the costs customer would have to pay also in this case.

From the mobile operator's point of view the situation is more complicated. On the one hand, compression decreases the number of short messages (or connection time or number of bytes transferred by GPRS or other bearers) that can be billed in the context of a customer transaction. Thus, the compression tends to decrease revenues per customer transaction. On the other hand, the operator can allocate the saved bandwidth and other network resources to other purposes and thus earn the same revenues, but now from several customers operating in parallel. The operator also saves disk space if it stores the applications in a compressed form. Thus, the infrastructure investment per application is smaller if the applications are stored in compressed form, as compared with the uncompressed case. The space savings would be,
however, negligible if the application byte code sizes will be in the future as modest as in our samples (max 1.4 kB).

If the byte code files were stored uncompressed and compressed on the fly at the server, the operator would need more powerful processors than in the uncompressed case that are able to compress the data with at least the transmission speed. Thus, in this case the considerations the operator would have to make would be between the saved bandwidth and increased server capacity.

4.2. Turnaround time for sequential data transfer and decompression at the terminal

Let us now analyze, under which conditions the turnaround time would decrease assuming that the server sends the whole compressed application byte code to the client and the client decompresses it after having got it in its entirety. First, we assume that the interaction between the client and server follows the pure request-response pattern, i.e., the client requests an application from the server that sends it as a response. Whether the WML deck byte code requested is stored compressed or uncompressed does not have any effect on the time that is required to send the request to the server. We assume for the moment that the possible on-the-fly compression of the WML or byte code at the server is at least 10 times faster than transmission of the data from the server to the terminal (or the compression is done in parallel with the transmission, see below). In other words, the compression time is negligible and we can simply assume that the application data is stored compressed at the server.

The compression has effect on the transfer time of the WML deck (byte code) and we must also assume that it takes some time to decompress it at the terminal. Once the decompression has been performed at the terminal, loading and running the application do not differ in any way of the uncompressed case. Thus, the main issue, from the response time point of view is, under which conditions the time saved in compressing and transferring the compressed byte code is larger than the time spent in decompressing the byte code at the terminal.

Let $\text{Size}(AC)$ denote the size of the application AC in bytes, $\text{Speed}_{\text{tr}}$ denote the (average) transfer speed of the network from server to client in bytes/s, and $\text{Speed}_{\text{dec}}$ denote the (average) decompression speed of the compressed data at the terminal in bytes/s. Let further $\text{cr}$ denote the inverse of the compression ratio CR, i.e.

\begin{equation}
(1) \quad \text{cr} = \frac{\text{Size}(\text{Compr}(AC))}{\text{Size}(AC)} = 1/\text{CR}.
\end{equation}

Let us further assume that the decompression speed neither depends on the compression ratio, nor on the size of the compressed data, but only on the compression algorithm and on the properties of the processor running the algorithm. Further, let us assume that the compressed byte code is so large that it requires several (short) messages to be transferred (so that the anomalies in the example in 4.1. do not play a role). Remembering that $\text{cr} \cdot \text{Size}(AC) = \text{Size}(\text{Compr}(AC))$, we get the equation

\begin{equation}
(2) \quad \frac{\text{Size}(AC)}{\text{Speed}_{\text{tr}}} = \text{cr} \cdot \frac{\text{Size}(AC)}{\text{Speed}_{\text{tr}}} + \text{cr} \cdot \frac{\text{Size}(AC)}{\text{Speed}_{\text{dec}}}
\end{equation}

do to the point where both compressed and uncompressed case yield the same response time. If the right-hand side is larger than the left-hand side, compression makes the system slower than without compression and the vice versa. By solving the equation, we get

\begin{equation}
(3) \quad \frac{\text{cr}(1 - \text{cr})}{\text{Speed}_{\text{dec}}} = \frac{1}{\text{Speed}_{\text{tr}}}
\end{equation}

equation (3) relates the compression ratio and the diverse speeds. It shows that the trade-off does not depend on the size of the file. The left-hand side is a growing function of $\text{cr}$ that goes asymptotically to infinity when $\text{cr}$ (and CR) goes to 1. Fixing now two values, one can calculate the third one. For instance, for $\text{cr} = 0.5$ $\text{Speed}_{\text{tr}} = \text{Speed}_{\text{dec}}$; that is, if the data is compressed to 50% of its original size then the terminal should be able to decompress the data with the same speed as it is transferred in order not to slow down the system from uncompressed case. For $\text{cr} = 0.6$ the left-hand side yields 1.5 that is, the terminal should decompress the data 50% faster than it can be transferred by the network. For 0.7 the relation is 2.33, for 0.8 it is 4 and for 0.9 it is 9. That is, if we are able to compress only 10% the data then the decompression should be about ten times faster than the data transfer, will we still shorten the response time.

Reflecting this to the experimental results in section 3, where the average inverse of the compression ratio was about 0.8, the decompression should be about 4 times faster than data transmission, if we want that the response time should not grow from an uncompressed case. For 14.4 kbps this would mean about 57.600 bits/s or 7.2 kbps decompression speed. Assuming a rather realistic compression rate 1/0.7 for bigger applications one would only need 4.2 kbps decompression speed for the compressed data assuming 14.4 kbps transmission speed.

In [2, p. 266] some forms of Lempel-Ziv were tested on 1 Mips VAX 11/780. Decompression speed of 16 kbps was observed for LZB and 11 kbps for LZFG. Although not directly applicable here, the results hint that the
current processors used in WAP terminals could reach at least the same decompression speed as VAX 11/780 ten years ago. The actual decompression performance of the WAP terminals must still be further investigated.

4.3. Pipelining data transfer and decompression

The sequential case is the extreme in the sense that it does not make use of the parallelism inherent in decompression and data transfer. It is, however, simpler to implement at the terminal, than more complicated schemes. The results in 4.2. are actually valid also for cases, where the entire uncompressed code fits into one compression block. Lossless compression we are interested here in is namely always based on finite data blocks that are compressed and decompressed separately. As was discussed in Section 3, the block size is usually a parameter of the algorithm. Now, if the uncompressed code is larger than the block size then more than one block is separately compressed and decompressed. Thus, they can be transferred and decompressed in parallel. Let us assume that the terminal uses two buffers to transfer and decompress the data. They are both at most of the size of the compression block B (for the sake of example, assume for the moment 2 or 4 kB, see [12]).

There are two processes at the terminal. The transfer process transfers data to one buffer and the decompression process decompresses data from the other buffer simultaneously placing the decompressed data into the code memory. If the decompression process tries to run ahead of data transfer process, the former pauses, and vice versa. The transfer/decompression system is started by first filling one buffer by the transfer process, after which the processes run in parallel, and the systems halts after the decompression process has processed the last block. We assume that there are N blocks, where N > 1.

In this kind of arrangement the time to get the data into the code memory of the terminal from the server can be calculated in the following way:

\[ \text{Speed}_{\text{tr}} = \left( N-1 \right) \times \max\left( \text{cr} \times \text{Size}(B) / \text{Speed}_{\text{tr}} + \text{cr} \times \text{Size}(B) / \text{Speed}_{\text{dec}} + \right) \]

If the transfer speed is at most the decompression speed, then the two-buffer system is optimal, because the transfer process dominates the overall time and increasing the number of buffers at the terminal would not decrease the overall time. Assuming \[ \text{Speed}_{\text{tr}} \leq \text{Speed}_{\text{dec}} \] we get from (4) for the break-even point the equation:

\[ \text{Size}(N \times B) / \text{Speed}_{\text{tr}} = \left( N-1 \right) \times \max\left( \text{cr} \times \text{Size}(B) / \text{Speed}_{\text{tr}} + \text{cr} \times \text{Size}(B) / \text{Speed}_{\text{dec}} + \right) \]

Observing that \( N \times \text{Size}(B) = N \times \text{Size}(B) \) we get from (5):

\[ \text{cr} \times \text{Size}(B) / \text{Speed}_{\text{tr}} + \text{cr} \times \text{Size}(B) / \text{Speed}_{\text{dec}} \]

By dividing (6) by \( \text{Size}(B) / \text{Speed}_{\text{tr}} \) yields:

\[ \text{cr}(1 + \text{Speed}_{\text{tr}} / \text{Speed}_{\text{dec}}) = 1 \]

Dividing (7) further by cr and subtracting 1 from both sides finally yields:

\[ 1 / \text{cr} - 1 = \text{Speed}_{\text{tr}} / \text{Speed}_{\text{dec}} \]

Assuming for instance \( \text{cr} = 0.9 \) and \( \text{Speed}_{\text{dec}} = \text{Speed}_{\text{tr}} \) and solving equation (8) for \( N \) yields \( N = 9 \). That is, for the file sizes larger than nine compression blocks the response time is shorter than for an uncompressed case, assuming decompression is as fast as transmission. Correspondingly, for \( \text{cr} = 0.7 \) and the same transmission and decompression speed we get \( N = 3 \); i.e., the file needs to be only three or more compression blocks large in size in order to lead to saving in response time. If we assume that decompression speed is twice the transfer speed we get from (8) correspondingly \( N = 5 \) for the \( \text{cr} = 0.9 \) and \( N = 2 \) for \( \text{cr} = 0.7 \).

If the decompression of a block takes longer or the same time than transferring it, then the optimal system would use unlimited number of buffers. The transfer process could proceed unhalted increasing the number of buffers, and the decompression process would follow in the same order as the buffers were filled.

In this case, i.e. \( \text{Speed}_{\text{tr}} > \text{Speed}_{\text{dec}} \), the break-even equation is:

\[ \text{Size}(N \times B) / \text{Speed}_{\text{tr}} = \text{cr} \times \text{Size}(B) / \text{Speed}_{\text{tr}} + (N - 1) \times \text{Size}(B) / \text{Speed}_{\text{dec}} \]

Using the fact that \( N \times \text{Size}(B) = N \times \text{Size}(B) \) we get, through dividing (9) by \( \text{Size}(B) / \text{Speed}_{\text{tr}} \):

\[ \text{N} = \text{cr} \times (1 + \text{N} \times \text{Speed}_{\text{tr}} / \text{Speed}_{\text{dec}}) \]

Dividing (10) further by \( \text{cr} \times \text{N} \) and moving terms we get:

\[ 1 / \text{cr} - 1 = \text{Speed}_{\text{tr}} / \text{Speed}_{\text{dec}} \]

Letting the ratio of the two speeds in (11) approach one, the limit would be (8), i.e. \( 1 / \text{cr} - 1 = 1 / \text{N} \). This shows that there is no discontinuity between (8) and (11).
Assuming in (11) \( \text{Speed}_{\text{dec}} = 0.5 \times \text{Speed}_{\text{tr}} \), we get for \( r = 0.9 \) no solution. The same holds for all values greater than 0.5, for which the left-hand side of (11) would be at most zero and the right-hand side would asymptotically approach zero as \( N \) goes to infinity. For \( r = 0.4 \), i.e. compression ratio 2.5 or higher we would already get \( N = 2 \) from (8) under the same speed ratio as above.

This pipelining idea can be applied at the server side for files larger than the compression block. If the compression speed is at least \( \text{CR} \times \text{Speed}_{\text{tr}} \), the transmission will proceed without interruptions, and indeed the compression time is negligible, as we assumed above, for the response time. Compression would still require more processor and memory resources as compared to uncompressed case, so it does not come for free. It is for further study, how significant load it would represent at the server.

4.4. Time and space complexity and energy consumption

The time and space complexity of the decompression depends on the decompression algorithms used and the buffering regime. Very coarsely, the (RAM memory) space required by the decompression is typically the size of compressed block, plus the size of the space required by the same block in the decompressed format, plus the size of the data structures (dictionaries, fast look-up tables, etc.), plus the code used by the decompression algorithms. The latter two depend largely on the implementation. In the case where the transmission is slower than the decompression, we might add one block for the parallel transfer. In the opposite case the space requirement is in the worst case the whole compressed data in \( N \) buffers and the space needed to run the decompression algorithm.

The practical amount of memory with which the decompressor would run smoothly, might be around 10-20 kByte for the Lempel-Ziv variants. This space also includes the program code. The latter is possible to squeeze into 1-2 kByte. In [2, p. 266] it was observed that LZW variant only needed 8 kByte memory for decompression and reached 16 kByte decompression speed. If this estimate is correct then WAP terminals should be able to cope with the decompression (Nokia 7110 has almost 1 MB RAM for applications and data). The decompressor code should maybe be placed into a ROM in terminals.

Time complexity of the decompression algorithms is clearly at least linear with respect to the size of the uncompressed data, because the entire data must be generated by the decompressor. The time complexity of compressor can be higher than that of the decompressor, which should remain linear. Some versions of Lempel-Ziv algorithms have linear time complexity (best/average) both for compression and decompression [1, p. 12-19], so they are viable candidates for this environment. Our experiments hint the same. The complexity issues are for further study.

The most unclear but maybe the most relevant issue in this context is the energy consumption of the various decompression and other algorithms. It is of course rather directly linked with the time complexity; the more steps the (abstract) algorithm has to take in order to decompress a certain data block, the more processor cycles are needed to run the corresponding program. Thus, energy consumption grows with the time complexity. Thus, using decompression algorithms at the terminal is in general more energy consuming than not using them (i.e. just loading the data into memory versus first decompressing it and then loading). The compression helps, however, in saving energy on the radio path, because the receiving time of the data is shorter than in the uncompressed case. The trade-off between decreased energy consumption of the RF-parts and increased energy consumption of the processor cannot be easily assessed without more accurate parameter values of the terminals. These are also for further study.

5. Conclusions

In this paper we have investigated the compressibility of the WML and WMLScript byte code and analyzed under which conditions the response time would not increase due to the time used to decompress the data at the terminal. The equations we have deduced relate the compression ratio, the transmission speed of the network, and the decompression speed. The absolute file size is not relevant for the break-even point in sequential case, although it influences the absolute turn-around time. In parallel case the file size has influence on both.

We made experiments in order to check to which extent some sample files can be compressed with generally used algorithms. With the smallest files (size less than 160 bytes) there was no significant reduction in size; on the contrary, some compressed files were larger than the non-compressed ones. In average, the compression ratio in the sample was ca. 1.5 and the larger file had the higher compression ratio (en. 2.2 for the 1.4 kB file). This observation and the fact that applications tend to become more complicated in functionality and larger in size over time makes compression more attractive. It is unquestionable that if the compression ratio is larger than one in average, compression saves bandwidth and also
customer costs in networks where billing is based on connection time or number of bytes transferred. It might still cause increase in response time if the decompression speed at the terminal is slower than the transmission speed of the network. We discovered that for the smallest files tiny compression might save one or more short messages and thus, relatively taken, a considerable amount of customer money. On the other hand, there is evidence that for large XML documents (several tens of kilobytes in size) the compression ratio can be 10:1, when specialized algorithms are used; It remains to be seen whether so large documents are feasible in the environment and whether the same principles carry over to byte code files.

A remark should be made here. Our test material did not contain any images, since the compression methods we experimented can not be successfully applied to both byte code and image data. Regarding that a picture filling the screen of a current WAP terminal (Nokia 7110) with 65 rows of 96 pixels means that a picture is about 780 bytes in size. Thus, including even one full-screen image into the application causes its size to exceed a kilobyte, assuming the average size in our sample. This is already a case for compression. Lossless compression that can be applied to any type of data, would not be aware of the two (or more) dimensions of image data and therefore, would probably exhibit poor efficiency. It is foreseen that WBMP format for WAP images would be compressed with a special algorithm. It remains to be seen how such an algorithm would perform.

WAP Forum has suggested that compression should be done at the Transmission Layer. This approach would compress the payload data but also the headers of the PDUs of the higher layers. Our results have some relevance for this idea, too. It seems that the compression blocks should be large enough to yield significant gains. Are the PDUs at this level big enough and if not, what would be a suitable compression block? Second, even the general lossless compression algorithms would achieve considerable compression ratios if the compression block size would be in kilobytes.

All in all, these initial results suggest that compression of WML and the byte code should be studied further and specialized algorithms developed. The capabilities of the terminals must be studied further both in terms of the decompression speed achievable and with respect to the energy consumption characteristics of the decompression. These studies also will shed light into whether storing WML source or byte code compressed at the terminals would make sense.

6. References