PERIODICA POLYTECHNICA SER. EL. ENG. VOL. 47, NO. 1–2, PP. 37–56 (2003)

SIP COMPRESSION

Márta FIDRICH*, Vilmos BILICKI**, Zoltán SÓGOR*** and Gábor SEY***

*Research Group on Artificial Intelligence, Hungarian Academy of Sciences,
H–6720 Szeged, Aradi vértanuk tere 1, Hungary
e-mail: fidrich@sol.cc.u-szeged.hu
**Department of Fundations of Computer Science
H–6720 Szeged, Árpád tér 2, Hungary
e-mail: bilickiv@inf.u-szeged.hu,
*** University of Szeged Department of Informatics
H–6720 Szeged, Árpád tér 2, Hungary
e-mail: weth@nokia5.inf.u-szeged.hu,Sey.Gabor.Lajos@stud.u-szeged.hu

Received: May 20, 2003

Abstract

The wired line network has been well studied and widely used for a long time. Most of its protocols are so successful that passed the test of time. There are many similar tasks in mobile and wired line environment, and we would like to achieve compatible, inter-working solutions. So it is a plausible idea to use the protocols of the wired line network in mobile environment too. However, the mobile and wired line environment differ significantly; mainly the bandwidth is different in the two networks. Although the difference is going to be smaller with the help of new generation of mobile networks, it will still remain significant. An acceptable solution is to compress these protocols. We have not found such a solution in the literature so our opinion is that this article is the first dealing with SIP compression.

We have created a demonstration system, which connects two SIP user agents to each other and ensures the compression and decompression of the messages between them. In this article we show our development about adapting various compressing algorithms for SIP compression, and we evaluate them.

Keywords: SigComp, UDVM, compression, bytecode, state, dictionary

1. Motivation

Wired line networks have been used for 20 years. These networks grew up from their child disease and now they are in a productive, well tested state. We have a knowledge about wired network design, we have a large number of well tested and relatively cheap wired network elements (switches, routers), and the most important is that: these techniques are widely used. So it is not surprising that the 3rd Generation Partnership Project (3GPP) community chose the IP protocol as the backbone for future Universal Mobile Telecommunication System (UMTS). The UMTS will be an all-IP solution. That is why the mobile core system developers tend to switch from their individual protocols to commonly used Internet protocols. In 3G telephony, this progress gets up and new protocols appear. It is worth because the different manufacturers' units become compatible with each other and several universities and companies can participate in the development of these protocols.

In addition to the similarities, there are several differences between wired and mobile network. The most important one is the bandwidth, which is the bottleneck of mobile core systems. The number and speed of processors, the memory, and the capacity of backing storages are easily increasable, but the communication speed between the units is limited.

Mobile system providers invested a huge amount of money into their systems. To get this money back they have to provide acceptable services for customers. The customers would like to use these services independently of their location and hardware framework. There is a need for a communication protocol to check the available and demanded services and their parameters. The Session Initiation Protocol (SIP) [7] has been chosen by 3GPP for this purpose. It can be seen by now, that SIP is one of the most important Internet protocols in 3G mobile core systems. However, it is highly redundant, because it is an extensible ASCII-based communication protocol. Recently, alerts have been raised in 3GPP that session setups in the all-IP network were too lengthy due to excessive signaling over the radio link. The delays were estimated to be as long as 10 seconds [8]. To overcome this problem, they agreed to use some kind of signaling compression along with UDP/IP header compression to shorten the transmitted messages. 3GPP escalated the issue to IETF. After evaluating several proposals, the Robust Header Compressing Group (ROHC) [13] has come up with the idea of the universal decompressor, and defined a communication layer, called Signaling Compression (SigComp) for this functionality [19, 20, 21]. IETF has left many issues open or implementation specific, such as the negotiation of algorithms and parameters (such as buffer sizes). It is also unknown yet how to integrate SigComp with the SIP protocol, which is the main subject to compression.

The SigComp layer consists of two main interworking entities: the compressor and the Universal Decompressor Virtual Machine (UDVM) [22]. There are several compression algorithms, and for the freedom of choosing any kind of them, SigComp is designed in such a way that it contains a universal decompressor. Thus not only the compressed messages can be sent, but also their compression algorithms, if needed. The UDVM has its own language for implementing and executing decompression algorithms; the only task is to upload the appropriate decompressing code to UDVM.

The protocol header compression is not a new idea. In 1990 Van Jacobson proposed a TCP/IP specific compression algorithm [15], sending between 3-5 bytes of the 40 bytes header. Another interesting approach is suggested in article [16]; it describes a universal framework which includes a simple platform-independent header description language that protocol implementers can use to describe high-level header properties, and a platform-specific code generation tool that produces kernel source code automatically from this header specification. In our case, these approaches are not appropriate, because they are based on inter-packet redundancy. We have to deal with stand-alone packets because it is possible that we have only a one-way communication channel with error prone link, so we do not know which packets arrived on the far side of communication channel. The classic compression algorithms are not applicable without modifications because of the short message lengths. We did not find acceptable solution in the literature for our special criterion. So we think that our article is the first approach to study the compressibility of the SIP protocol, and in broad manner, the asymmetric[1] protocols to develop applicable compression methods.

In this article we describe our research and results about SIP compression. Our goal was to implement the SigComp layer and to study the SIP protocol to find the best compressing algorithm for it.

This article is organized as follows: We give an overview on SIP in Section 2, in the next section we briefly present the theoretical background of compression and some well-known algorithms. After that, our development on compression algorithms is explained. Finally, we compare and evaluate our test results and give a conclusion.

2. Session Initiation Protocol (SIP)

2.1. *Brief Description*

There are many applications of the Internet that require the creation and management of a session, where a session is considered an exchange of data between an association of participants. The implementation of these applications is complicated by the practices of participants: users may move between endpoints, they may be addressable by multiple names, and they may communicate in several different media – sometimes simultaneously. Numerous protocols have been developed that carry various forms of real-time multimedia session data such as voice, video, or text messages. The Session Initiation Protocol (SIP) [6, 7] works with these protocols by enabling Internet endpoints (called user agents) to discover one another, and to agree on a characterization of a session they would like to share.

SIP is an application-layer control protocol that can establish, modify, and terminate multimedia sessions (conferences) such as Internet telephony calls. SIP can also invite participants to already existing sessions, such as multicast conferences. Media can be added to (and removed from) an existing session. SIP transparently supports name mapping and redirection services, which contributes to personal mobility – users can maintain a single externally visible identifier regardless of their network location. For locating prospective session participants, and for other functions, SIP enables the creation of an infrastructure of network hosts (called proxy servers) to which user agents can send registrations, invitations to sessions, and other requests. SIP works independently of underlying transport protocols and without dependency on the type of session that is being established.

Changeover of IPv4 [18] and IPv6 [4] protocols is studied and extensively analyzed recently, and agreed to be a necessary but not easy task [12, 23]. The 3G

[1]Protocols without full handshaking

networks use SIP as their call control protocol and IPv6 as network layer protocol for wireless communication, while currently IPv4 is used as network protocol for the Internet. Thus a need is emerged to connect a mobile SIP user agent based on IPv6 and another SIP user agent based on IPv4. In a previous work [24] we have created and tested a demonstration system, where the SIP communication between the IPv6 and the IPv4 networks is established between two SIP proxies (IPv6 and IPv4) via a special NAPT-PT. This work has presented a new technique to ensure communication between 3G mobile networks and Internet phones. As a continuation we would like to examine how to integrate the SIP protocol – which is the main subject to compression – into the SigComp layer.

Instead of describing the whole functionality of SIP [3, 25], we present here a sample message, and a typical message flow.

```
INVITE sip:bob@biloxi.com SIP/2.0
Via:  SIP/2.0/UDP pc33.atlanta.com
  ;branch=z9hG4bK776asdhds
Max-Forwards:  70
To:  Bob <sip:bob@biloxi.com>
From:  Alice <sip:alice@atlanta.com>;tag=1928301774
Call-ID: a84b4c76e66710@pc33.atlanta.com
CSeq:  314159 INVITE
Contact:  <sip:alice@pc33.atlanta.com>
Content-Type:  application/sdp
Content-Length:  142


SIP/2.0 200 OK
Via:  SIP/2.0/UDP server10.biloxi.com
  ;branch=z9hG4bKnashds8;received=192.0.2.3
Via:  SIP/2.0/UDP bigbox3.site3.atlanta.com
  ;branch=z9hG4bK77ef4c2312983.1;received=192.0.2.2
Via:  SIP/2.0/UDP pc33.atlanta.com
  ;branch=z9hG4bK776asdhds ;received=192.0.2.1
To:  Bob <sip:bob@biloxi.com>;tag=a6c85cf
From:  Alice <sip:alice@atlanta.com>;tag=1928301774
Call-ID: a84b4c76e66710@pc33.atlanta.com
CSeq:  314159 INVITE
Contact:  <sip:bob@192.0.2.4>
Content-Type:  application/sdp
Content-Length:  131
```

Fig. 1 shows a typical example of a SIP message exchange between two users.

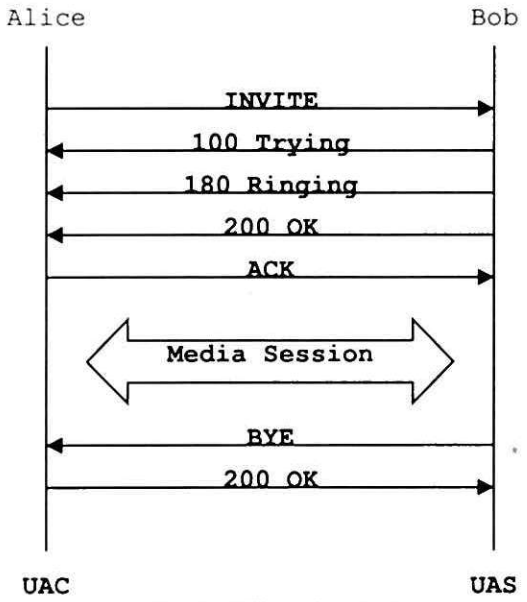

Fig. 1. SIP session setup

3. Overview on Compression

3.1. Theoretical Background

Data can be compressed whenever some symbols are more likely to occur than others. There are several theories for describing the information content of a data set. Entropy, as defined by Shannon, is the uncertainty regarding which symbols are chosen from a set of symbols with given apriori probabilities. If there is more disorder, or entropy, then more information is required to reconstruct the correct set of symbols [1]. The Shannon's entropy is for a set of possible symbols S

$$H = \sum_{s \in S} p(s) \log_2 \frac{1}{p(s)}, \tag{1}$$

where $p(s)$ is the probability of symbol s. If we consider symbols $s \in S$, Shannon defined the notion of self-information of a symbol as:

$$i(s) = \log_2 \frac{1}{p(s)}. \tag{2}$$

This self-information represents the number of bits of information contained in it and, roughly speaking, the number of bits we should use to send that symbol. The entropy is simply a weighted average of the information of each symbol, and therefore the average number of bits of information in the set of symbols. This entropy (more precisely *The First order Entropy*) gives us an upper limit for data compressibility when we do not know anything about partial transition probabilities between symbols, and we are coding only characters and only one message. Using (1) we have found that the entropy of on typical SIP message is about 6.7. The information can be coded with H bits/symbols (this can be achieved only with arithmetic coding). With this entropy we can calculate the achievable compression rate:

$$cr = \frac{\text{compressed}}{\text{original}} = \frac{6.7\ \text{bit/char}}{8\ \text{bit/char}} = 0.83. \tag{3}$$

How can we achieve a better compression ratio than Shannon's first order entropy? With second, third, ... order entropy. If we have some knowledge about dependence between symbol probability and its context, then we can calculate with the help of conditional probabilities. The conditional entropy for a set of symbols S and context set C :

$$H(S \mid C) = \sum_{c \in C} p(c) \sum_{s \in S} p(s \mid c) \log_2 \frac{1}{p(s \mid c)} \tag{4}$$

When the conditional probability distribution of S is dependent of context C, then $H(S \mid C) < H(S)$; otherwise $H(S \mid C) = H(S)$.

As we have seen, we can calculate upper limits for data compressibility, if we know the symbols probability (conditional probability) distributions. But we would like to make effective compressing algorithms in practice. We have to deal with the following parts:

- data modeling – If we have some knowledge about data, we can provide an accurate probability model for each symbol (or group of symbols). This is mostly a dictionary, sorted by symbol (or group of symbol) probability. (Probability in most cases is the relative frequency of a symbol in a message).
- symbol coding – With the help of symbol coding we can link the symbols (or group of symbols) to an approriate codeword (whose lengths depend on symbol probability, the greater the probability the shorter the assigned codeword).

We have to mark the borders between codewords because they have different lengths. Symbol coding can be byte or bit based. In the case of byte-based coding, we can use special symbols, in the other case we can use prefix free codewords to mark the borders. A codeword is prefix free, if none of the codeword is a prefix of an another codeword.

3.2. Summary of Some Well-known Algorithms

LZ77 is a byte-based run-length encoding algorithm. It scans the message and tries to find the largest equal parts with the dictionary. If it finds one, then it replaces it with the following pair of numbers: the starting address of the hit and the length of replaced character group. (Sometimes the distance of the hit from the beginning and the length is used.)

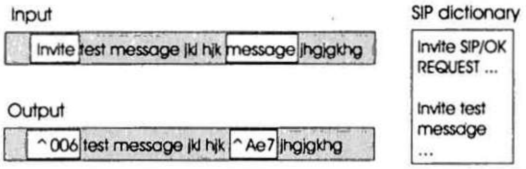

Fig. 2. Illustration how LZ77 algorithm operates

The following four algorithms are based on prefix-free encoding:

The Golomb-Rice encoder [9] is a prefix-free coding that assigns codes to the numbers 0,1,2,3,... according to the following description: first a power of 2 is chosen, denote it by 2^k. If we want to encode the number n, first $\lfloor n/2^k \rfloor$ 1 digits are written, a 0 after them and last the number (n mod 2^k) k-length long even if it can be written shorter, which corresponds to the following formula:

$$\text{code} = \frac{n}{2^k} \text{ unary code} + 0 + n \bmod 2^k \tag{5}$$

SubExponenetial encoder [11] is also prefix-free, which assigns codes to numbers. The main difference compared to the previous Rice encoder is that the lengths of codes are growing logarithmically depending on the numbers to be encoded.
First a power of 2 is chosen, denote it 2^k. Let n be the number we want to code. Then the code of n is:

- If $n < 2^k$, then the code of n is a 0 and n in exactly k-bit long.
- If $n \geq 2^k$, then the code of n starts $\lfloor \log(n) \rfloor - k + 1$ bits 1, a 0 and last the lowest $\lfloor \log(n) \rfloor$ bits of n.

the corresponding formula:

$$\text{code} = \begin{cases} 0 + k \text{ unary code} & \text{if } n < 2^k; \\ \log(n - k + 1) \text{ unary code} + 0 + n \bmod \log n & \text{if } n > 2^k \end{cases} \tag{6}$$

Synth With the help of Arithmetic compression [10] we can approach the best compression ratio, the Shannon entropy. The main idea of arithmetic coding is to represent each possible sequence of n messages by a separate interval on the number line between 0 and 1. For a sequence of symbols with probabilities $p_1, p_2, \ldots, p_n$, the algorithm will assign the sequence interval size $\prod_{i=1}^{n} p(i)$ starting with an interval of size 1. We have simplified this algorithm with fixed size intervals (fixed probabilities). The size of intervals is a power of 2.
The number n, thus, can be coded by the following rules:

- If $0 \leq n \leq 15$, then the code of n is a 0 (only 1 bit) and after that n is represented in 4 bits.
- If $16 \leq n \leq 31$, then the code of n is 10 (2 bits) and $(n - 16)$ is represented in 4 bits.
- If $32 \leq n \leq 63$, then the code of n is 110 (3 bits) and $(n - 32)$ is represented in 5 bits.
- If $64 \leq n \leq 127$, then the code of n is 1110 (4 bits) and $(n - 64)$ is represented in 6 bits.
- If $128 \leq n \leq 255$, then the code of n is 11110 (5 bits) and $(n - 128)$ is represented in 7 bits.
- If $256 \leq n \leq 767$, then the code of n is 11111 (5 bits) and $(n - 256)$ is represented in 9 bits.

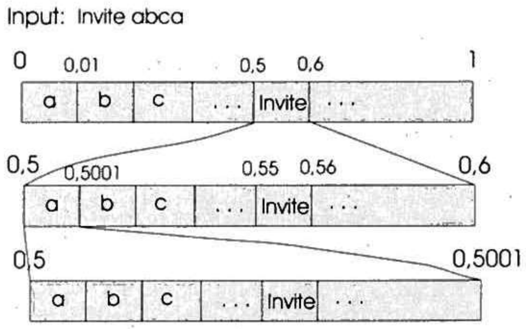

Fig. 3. Illustration how arithmetic compression works

Huffman encoding is an optimal prefix-free coding. The algorithm is based on the well-known Huffman tree. The Huffman-tree is built on the frequency of the keywords in the message, where every leaf contains a keyword. The messages are then compressed using this tree.

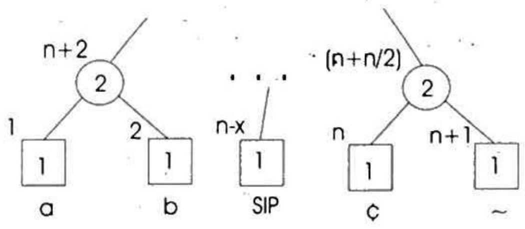

Fig. 4. A sample Huffman tree

Deflate algorithm [5] is a combination of two different compression algorithms. First the message is compressed by the algorithm LZ77 (or its variant) and after that the encoded message is compressed by the Huffman algorithm.

4. Our Results

4.1. SIP Compressibility

We present three approaches that are based on each other indicating the level of SIP specificity.

- In our first attempts we handled the messages as they were simple text-based messages, that was why we could not write efficient compression algorithms. Our experiments are summarized in Subsection 4.2.
- After that we used dictionary-based algorithms, because we had preliminary information about the message that can be incorporated into the compression algorithms as probabilities. We have found a dictionary with SIP instructions ordered in decreasing probabilities [6]. We know that numbers and normal characters are more probable than special characters, and groups of symbols may appear more than once (run-length encoding). Some details about our dictionary are presented in Subsection 4.3.
  We constructed several compression algorithms (mainly differing in symbol coding) starting from the well-known algorithms summarized in Section 3.2. These algorithms were modified so that they could use our dictionary efficiently. Moreover, we made all of them adaptive. We used a kind of *Move to Front* algorithm to change the symbol probabilities. When a symbol occurred, we decreased its position in the dictionary. We used the dictionary both at encoder and decoder side, improving the efficiency of the compression. The messages could be compressed efficiently with this solution; subsections 4.3 - 4.6 describe what we developed specifically for the various algorithms.
- The entropy of a message flow is better than the entropy of a message, so we can achieve better compression ratios when compressing message flows. Further improving of the compression could be reached by using previous messages (dynamic compression) because the similarity of the messages could be high (e.g.: the address of sender and the receiver is the same) and there are algorithms which can profit from this property. As a future work we would like to study this approach.

4.2. Experiments of First Attempts

First we tested the algorithm LZ77. Since the dictionary contained only few keywords as well as all the 256 ASCII characters, the compression ratio was over 100%.

At the second attempt, the Huffman algorithm was used. The tree was built based on the frequency of occurrence of the characters in the message and then the message was compressed character by character. Since the tree was built dynamically, we had to attach information about the tree that is needed to decompress the message. This solution did not work because the compressed message was longer than the original one. The compression ratio was over 150%, but for shorter messages, the ratio was even more than 250%. This terrible (un)compression ratio clearly shows that the well-known compression algorithms cannot be used 'as they are' in the case of data transfer with the SigComp layer; instead they have to be adapted or even re-designed.

Before the next compression methods, we developed a good dictionary containing all the SIP keywords and the 256 ASCII characters. This dictionary consists of more than 600 keywords.

4.3. Creating a Dictionary

The SIP messages are about 300-600 bytes long. A message can contain all of the ASCII characters. The content can be divided into user data such as e-mail, host name ... etc, and SIP instructions. We studied several SIP message flows which can be found on Internet [8, 25], and used an article [6] where a static SIP dictionary was presented. From this dataset we constructed a dictionary, which consisted of 600 elements and could be divided into 5 parts:

1. most probable special symbols ('=',' ', ...) (our elements)
2. numbers (ten elements)
3. alphanumeric characters (26 x 2 elements)
4. SIP instructions (~500 elements)
5. special characters (~140 elements)

We note that our algorithms use the same dictionary (the one presented here), they only differ in the symbol coding algorithm.

4.4. Modification of LZ77

The first effectively working (i.e. compressing) encoder was an improvement of LZ77. The dictionary has high importance in the case of the LZ77 algorithm, thus we used the revised dictionary mentioned above. It resulted some additional enhancements that the length was represented only on 1 byte, while at the first attempt, the length was 2 bytes. However, the real improvement was that we did not encode the short matches (1, 2, or 3 long) by LZ77, but these were forwarded un-coded. We introduced a special sign to distinguish the encoded and the un-coded parts. The efficiency of the modified algorithm was still not as good as the one of the Huffman or SubExponential compressors, but it was more efficient than the methods implemented as first attempts. The compression ratio was between 55% and 75%. Recurring parts in the message to be compressed further increase the efficiency, and a SIP message is one of this kind of messages.

4.5. Prefix-Free Encoding

During prefix-free encoding we encoded numbers, so first the message had to be transformed to numbers. The message was splitted to the keywords of the dictionary (only exact matchings were allowed), after that the keywords were changed to the numbers assigned in advance to the keywords. So we could use the following encodings.

The Rice encoding was the first we reached good results with. During the encoding we encoded the numbers transformed from the message as described above. The parameter of the Rice-encoding (only one number) was chosen dynamically according to the message, and the numbers assigned to keywords were permutated permanently using a kind of 'Move to Front' algorithm, thus improving the efficiency of encoding. This permutation can be used easily at decompression too. Only some (1 or 2) parameters needed to be attached to the compressed message so it did not decay the efficiency. The compression ratio was 40-60% with this method, which means significant improving compared to the previous methods.

The next attempt was the SubExponential algorithm. It corresponds to the Rice encoding, but the length of codes was growing logarithmically depending on the numbers to be encoded, while in the previous case the growing was linear. The parameter (like at the Rice-encoding) was chosen depending on the message, and the permutation was also used, moreover, it was improved. The enhancement of this algorithm compared to the Rice encoding could be as high as 5% function of messages, but there were cases when it was negligible.

We developed the algorithm Synth as the simplification of the Arithmetic encoder. Although its philosophy is rather different from the one of SubExponential encoder, the test results are very similar, even more, the codes of keywords are almost the same.

Finally, we returned to the Huffman algorithm. Because of the large size of the dictionary, it was impossible to attach the tree to the message, so we had to choose a solution that used a predefined tree. This tree was independent of the message to be compressed, so it could be stored both at the sending and receiving endpoint in advance, thus, we did not need to send the tree. Since the tree was not built based on the message, it was not optimal, but it approached the optimum quite well. The rate of approach naturally depends how well we could estimate and define the tree, thus, building of the definite tree was preceded by a lot of thorough tests. After having the tree, the compressing was the same as usual. We wanted to improve the compression ratio, so first an adaptive version was implemented, but there was no significant improvement. Instead, we used a similar permutation method as in the case of Rice or SubExponential encoder.

4.6. *Deflate and its Modification*

The Deflate compression means the execution of two subsequent encodings. In the first step the message is compressed by the LZ77 algorithm, while in the second step the message is compressed again with the Huffman encoder.

We were proceeding in the same way, but we used our modified LZ77 instead of the original. After that we did not compress the message immediately with Huffman. We separated the un-coded characters (including the special characters) and these were compressed with the SubExponential algorithm. The lengths were compressed with the SubExponential algorithm too, but independently of the type of characters. The MSBs of the positions (2 bytes) were compressed with the Huffman algorithm, while the LSBs were not compressed at all. We chose the SubExponential algorithm, because its efficiency was the same as the one of Huffman, but it was faster.

5. Evaluation

To understand the importance of the compression ratio and time for compression/decompression, we have to analyze the mobile call setup. In the GSM system a typical call setup time is about 3.6 seconds [17]. The setup time of a SIP call is approximately 7.9 seconds [8]. We can calculate with 140 ms RTT (Round Trip Time) and with a LinkSpeed of 9.6 kbps. In a typical SIP call acording to the arcticle [8], there are about 11 messages with an aggregated length of about 4,200 bytes. The SIP call setup time can be calculated from previous values with the help of the following formula [8]:

$$\text{OneWayDelay} = \frac{\text{MessageSize [bits]}}{\text{LinkSpeed [bits/sec]}} + \frac{\text{RTT [sec]}}{2}. \tag{7}$$

We have obtained the following results:

TimeForTransmission	≈ 3500 ms
TotalRTT	≈ 630 ms
BearerSetup	≈ 3000 ms
TotalDelay	≈ 7130 ms.

The BearerSetup and the TotalRTT could not be shortened without expensive technical investment. The only solution is to decrease the TimeForTransmission portion with the help of compression. It is difficult to see how efficient the algorithms are because it depends on the message very much. We used the SIP message samples from articles [8, 25] as the test set for our compressing algorithms. In the following part we describe the advantages and disadvantages of the algorithms and mutually compare them.

5.1. Efficiency of Compression

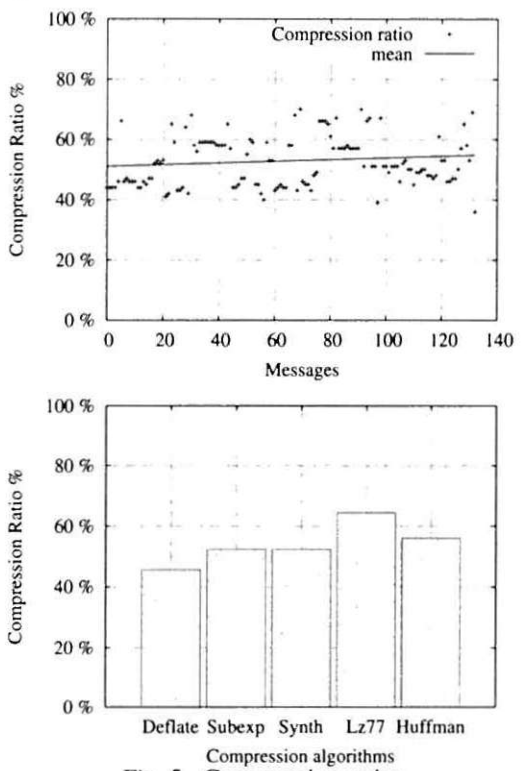

Fig. 5. Compression ratios

For the compressing ratios please consult *Fig. 5.* First of all, we can note that the compression ratio highly depends on messages (left figure). On the right side we can conclude that the LZ77 algorithm is still 10-20% worse than the others (Deflate, Huffman, ...) despite the fact that our modification improved it substantially. As already mentioned, it is difficult to choose the best of the three prefix-free encodings, therefore we compare them to our Deflate development. We can see that the Deflate provides the best compression ratios. In the second place there are the prefix free encoding algorithms. On the third place there is the Huffman tree based compression algorithm.

Our experience is that the Deflate seems to be better because it is more efficient in the bigger part of the tests and – more importantly – has two good features. One of them is that its ratio compression is not 'too bad' even in the case of extreme messages, while in that case, the prefix-free encoders perform very bad. The other feature is that if a message "can be compressed well", the Deflate can indeed compress it much better than the others. This means that in some cases every encoder can reach a better ratio than in average cases. However, the efficiency of prefix-free encoders increases only by 5%, while the efficiency of Deflate encoder increases by 10–12%.

5.2. *Measuring Time*

We implemented both compressor and decompressor side algorithms but the measured running time can not be used for the estimation of the compression/decompression time because of the multitasking operating environment and different architecture. To estimate the previous mentioned values, we have to use a theoretical approach. Both algorithms can be used on a mobile device and on a proxy server too. First of all, we need some information about central processing unit (CPU) of the mobile phone. According to the info sheet [14], CPUs in the todays mobile phones have about 100 MHz clock rate. In the case of the decompression there is the interpreted program execution (our byte code runs on a Universal Decompressor Virtual Machine) we need to approximate the real clock rate with dividing the original clock rate with 10. We get the formulas (8) and (9). We did not consider the possibility of multiple instruction execution, nor the possibility of complex instruction and hardware implementation of the virtual machine, so we would like to calculate the worst case. Now we need only the required CPU cycles of the different algorithms.

$$\text{TimeOfTheDecompression} \approx 10 \times \frac{\text{NumberOfNeededCPUCycles [cycles]}}{100\ \text{MHz}} \tag{8}$$

$$\text{TimeOfTheCompression} \approx \frac{\text{NumberOfNeededCPUCycles [cycles]}}{100\ \text{MHz}} \tag{9}$$

5.3. *Time of Compression*

The time of compression and time of decompression can be more important than the compression ratio. Our goal is not to achieve the best compression ratio, but to achieve the minimal transmission time. It is evident that the time of compression depends on the (length of the) message, therefore, we do not emphasize it separately later.

To estimate the time of compression for two compression algorithm groups (Deflat, LZ77, and prefix-free) we need different assumptions. First let us calculate the time of compression for prefix-free algorithms. There are two main parts:

- the tokenization — the algorithm divides the message into the largest possible tokens based on the dictionary. This part is the most calculation intensive. We can determine the mean of the average cut/byte with help of the *Fig. 7*. The computer cycles needed for the cut operations are between 1 and 10. As a worst case we use 10 CPU cycles for every cut operation. For dictionary maintance we need another 10 CPU cycles (move to front algorithm).

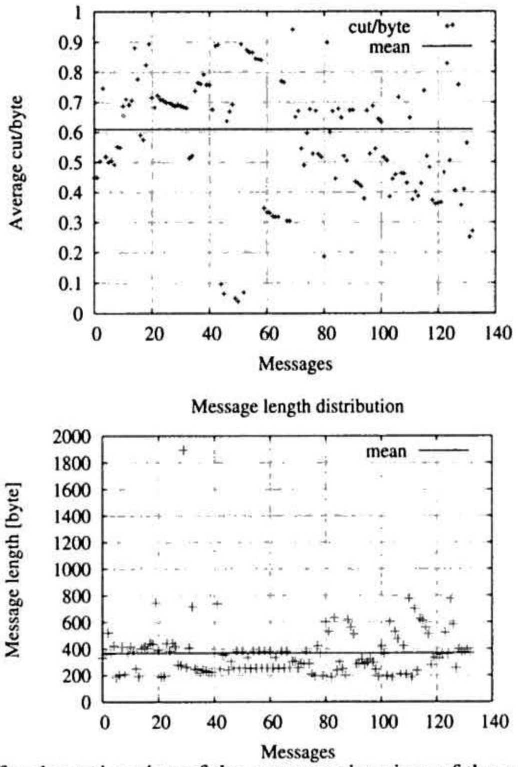

Fig. 6. Parameters for the estimation of the compression time of the prefix-free algorithms

- the token encoding — this part is faster than previous. Let we assume 2 CPU cycles for every byte encoding.

With the help of previous assumptions we get the following results:

$$\text{TimeOfTheCompression} \approx \frac{\text{Cut/Bytes} \times \text{MLength} \times \text{CutCost}}{100 \text{ Mhz}} = 453\ \mu s \quad (10)$$

The time of compression of the LZ77 and Deflate algorithms highly depends on the size of the dictionary. In *Fig. 7* we can see that the number of dictionary scans is about 136. To achieve anadaptive dictionary we concatenate the message to the end of the dictionary. With the help of previous assumptions we get the following results:

$$\text{TimeOfTheCompression} \approx \frac{\text{Scans/Bytes} \times \text{MLength} \times (\text{Dict.Lenght} + \text{MLength})}{100 \text{ Mhz}} = 1290\ \mu s. \quad (11)$$

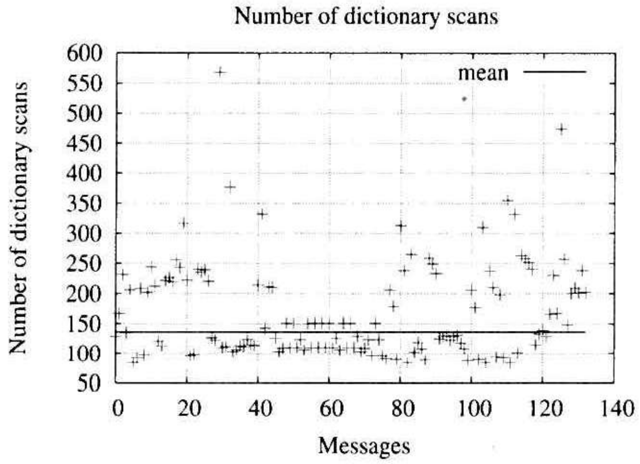

Fig. 7. Parameters for the estimation of the compression time of the LZ77 and the Deflate algorithms

We can conclude that the prefix-free encoders are much faster than the LZ77 encoder or the Deflate on the compressor side.

5.4. *Time of Decompression*

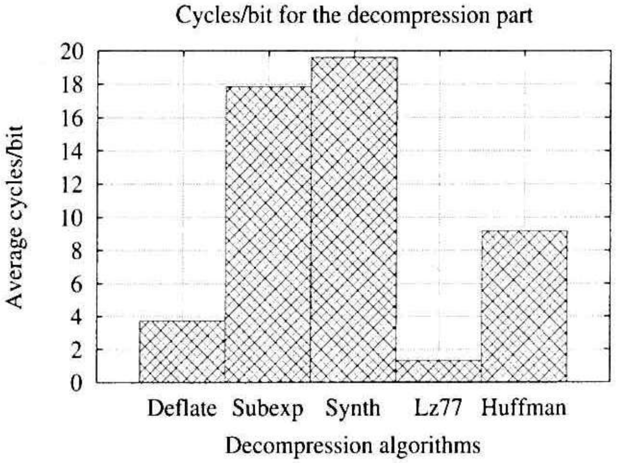

Fig. 8. Cycles/bit for the decompression part

Since the decompression algorithm is executed by the UDVM, it is very important to examine how fast our one algorithm is. The application determines a parameter that limits the cycles to be used to decompress the message (more precisely, the application limits the usable cycles/bit).

The LZ77 encoder is the fastest. Only 1 or 2 cycles/bit are needed for the decompression, therefore it can be used always. The Deflate encoder is not much worse because in its second step the prefix-free algorithm is used with less elements and without permutating them. The SubExponential encoder uses 18-19 cycles/bit, the Synth encoder uses 19-20 cycles/bit, while the Huffman encoder uses 8-9 cycles/bit. Nevertheless, these values are not too high, because at least 16 can be used for the decompression (the application parameter is at least 16 cycles/bit).

5.5. *Memory*

On the compression side, there is no big difference in the memory usage, because all the encoders use the dictionary and with the exception of the LZ77 they use other data structures as well. Thus, there is no encoder which must be mentioned here because of its large memory usage.

On the decompression side, it is highly important how much memory is used by the algorithms. Indeed, the UDVM has a total memory of 64 Kbytes, but it could happen that less memory is available. In the following we summarize only the needed memory of the algorithms; but we should add the length of the uncompressed message to it because it is stored also in the memory of UDVM.

The LZ77 algorithm uses the least memory (3.5 Kbytes), although the Deflate does not use much more (4.2 Kbytes). The Synth and the SubExponential decoders use 6.7–6.8 Kbytes memory, while the Huffman decoder uses 12 Kbytes memory. The latter is not surprising because it uses the Huffman tree to decompress the message.

Table 1. Conclusion

Algorithm	Comp. Ratio	Comp. T.	Decomp. T.	Transmission part of the Setup Time (without compression $\approx$ 3.5 s)
LZ77	64 %	1.55 ms	0.38 ms	2.2 s
Deflate	45.63 %	1,29 ms	1,09 ms	1.62 s
Synth	52.35 %	0.45 ms	5,78 ms	1.90 s
SubExp	52.38 %	0.45 ms	5.26 ms	1.89 s

6. Conclusion

As mentioned previously, we did not find any articles in the literature which dealt with SIP compression. So our task was first to analyze the SIP protocol and the compression theory to find feasible approaches for the compression. Our present study is about the analysis, synthesis and comparison of several compressing algorithms in the SigComp layer. We have developed a demonstration system, which connects two SIP user agents to each other and ensures the compression and decompression of the messages between them. The main parts of the system are the compressor/decompressor dispatcher, state handler, compressor containing various encoder algorithms, decompressor containing UDVM and a mnemonic-to-bytecode compiler. Several tests were executed to find out the performance of the algorithms (speed and memory usage) as well compressing ratio; and additionally, the conformance and robustness of our implementation.

As we have seen, there is no optimal solution. Each compression algorithm can be good or bad depending on the criteria specified by the application. We emphasize that although the compression ratio is usually considered to be mainly the most important; in the case of data transfer using the SigComp layer, special requirements exist and as a consequence, the speed and the memory usage are more important than the accessible compression ratio.

The modified Deflate (run length encoding + context modeling) gives us the best SIP call setup transfer time (1.62 s). However, this is the slowest solution on the compressor side and in the case of many concurrent sessions (SIP proxy) this can be a bottleneck. The modified LZ77 is the fastest on decompressor side. The context modelling (i.e. prefix-free encoders) is the fastest on the compressor side. We remark that the algorithms are dictionary-based which is optimal for messages with no special symbols. With the help of adaptive methods we can expand this optimality to messages with special symbols too.

We achieved compressing ratios below 50% and decreased the total delay of SIP call setup by 1.5 seconds, with dynamic compression (compressing message flows), thus the compression ratio could be under 30% and the transmission delay about 1 s. Our experiements, thus, showed that SIP messages could be efficiently compressed. In a future work we would like to study dynamic compression and how efficiently SIP can be integrated into the SigComp layer.

We think that our tests justified the philosophy of the protocol architecture, that is, let's allow the communicating parties select the best method for them according to their requirements.

Acknowledgement

This article presents the result of a research initiated and financed by NOKIA Hungary as a joint project between the University of Szeged and NOKIA Hungary.

We would like to thank to Tamás Fehér and Róbert Sugár for drawing our attention to this topic and also for their help in some technical questions. Thanks also for István Siket, Péter Siket, Tamás Szilágyi, Zsolt Galambos, and Zsolt Márton for their participation in the development of the SigComp layer.

References

[1] BLELLOCH, G. E., *Introduction to Data Compression*, www-2.cs.cmu.edu/afs/cs/project/pscico-guyb/realworld/www/compression.pdf.

[2] COOKSON, M. D. – SMITH, D. G., 3G Service Control, *British Telecom Technology Journal*, **19** (2001), www.sipcenter.com/files/3G_Service_Ctl.pdf.

[3] Ubiquity Software Corporation, *SIP Enhanced Mobile Network*, www.sipcenter.com/aboutsip/sipmobil.htm.

[4] DEERING, S. – HINDEN, R., *Internet Protocol, Version 6 (IPv6) Specification [RFC 2460]*, www.ietf.org/rfc/rfc2460.txt, December, 1998.

[5] DEUTCH, P., *DEFLATE Compressed Data Format Specification version 1.3 [RFC 1951]*, http://www.ietf.org/rfc/rfc1951.txt, May, 1996.

[6] GRACIA, M. – BROMANN, C. – OTT, J. – PRICE, R. – ROACH, A., *The Session Initiation Protocol (SIP) and Session Description Protocol (SDP) static dictionary for Signaling Compression (SigComp) [Internet Draft]*, www.ietf.org/internet-drafts/draft-ietf-sipping-sigcomp-sip-dictionary-03.txt, May, 2002.

[7] HANDLEY, M. – SCHULZRINNE, H. – SCHOOLER, E. – ROSENBERG, J., *SIP: Session Initiation Protocol [RFC 2543]*, www.ietf.org/rfc/rfc2543.txt, March, 1999.

[8] HANNU, H., *Signaling Compression Requirements & Assumptions*, http:www.ietf.org/internet-drafts/draft-ietf-rohc-signaling-req-assump-06.txt, June, 2002.

[9] HOWARD, P. G., *The Design and Analysis of Efficient Lossless Data Compression Systems*, Brown University, 1993, citeseer.nj.nec.com/howard93design.html.

[10] HOWARD, P. G. – VITTER, J. S., Practical Implementations of Arithmetic Coding, *Image and Text Compression*, 1992, pp. 85–112.

[11] HOWARD, P. G. – VITTER, J. S., Fast Progressive Lossless Image Compression, 1994.

[12] HUITEMA, C., *The new Internet Protocol* Prentice Hall, 1996.

[13] IETF, *ROHC group*, http:www.ietf.org/html.charters/rohc-charter.html.

[14] Intel, *Intel PXA800F Cellular Processor Developer Manual*, http://www.intel.com/design/pca/prodbref/252336.htm?iid=devnav_btn1+hw_proc_wap_pxa800f\&.

[15] JACOBSON, V., *Compressing TCP/IP Headers for Low-Speed Serial links [RFC 1144]*, http://www.ietf.org/rfc/rfc1144.txt, February, 1990.

[16] LILLEY, J. – YANG, J. – BALAKRISHAN, H. – SESHAN, S., *A Unified Header Compression Framework for Low Bandwith Links*, MIT Laboratory for Computer Science.

[17] NETWORKS, N., *A Comparison Between GERAN Packet-Switched Call Setup Using SIP and GSM Circuit-Switched Call Setup Using RIL3-CC, RIL3-MM, RIL3-RR, and DTAPRev. 0.3*, October, 2000.

[18] POSTEL, J., *Internet Protocol [RFC 0791]*, www.ietf.org/rfc/rfc0791.txt, September, 1981.

[19] PRICE, R. – HANNU, H. – BORMANN, C. – CHRISTOFFERSSON, J. – LIU, Z. – ROSENBERG, J., *Signaling Compression [Internet Draft]*, www.ietf.org/internet-drafts/draft-ietf-rohc-sigcomp-04.txt, February, 2002.

[20] PRICE, R. – HANNU, H. – BORMANN, C. – CHRISTOFFERSSON, J. – LIU, Z. – ROSENBERG, J., *Signaling Compression Reference [Internet Draft]*, www.ietf.org/internet-drafts/draft-ietf-rohc-sigcomp-05.txt, March, 2002.

[21] PRICE, R. – HANNU, H. – BORMANN, C. – CHRISTOFFERSSON, J. – LIU, Z. – ROSENBERG, J., *Signaling Compression [Internet Draft]*, www.ietf.org/internet-drafts/draft-ietf-rohc-sigcomp-07.txt, June, 2002.

[22] PRICE, R. – ROSENBERG, J. – BORMANN, C. – HANNU, H. – LIU, Z., *Universal Decompressor Virtual Machine (UDVM) [Internet Draft]*, www.ietf.org/internet-drafts/draft-ietf-rohc-sigcomp-udvm-00.txt. January, 2002.

[23] SÓGOR, L. – FIDRICH, M. – MARTONOSSY, L. – HENDLEIN, P. – SOMLAI, G., Comparison of Inter-operation Mechanisms between IPv4 and IPv6, *9th IFIP Working Conference on Performance Modelling and Evaluation of ATM & IP networks*, June (2001), pp. 402–413.

[24] SÓGOR, L. – HENDLEIN, P. – NOTAISZ, K. – FIDRICH, M. – FÓRIS, G. – KISS, G., Development of a Communication Environment between IPv6 and IPv4, *7th Int. Symposium on Programming Languages and Software Tools (SPLST)*, June (2001), pp. 231–243.

[25] VEMURI, A. – PETERSON, J., *SIP for Telephones (SIP-T): Context and Architectures [Internet Draft]*, www.softarmor.com/sipping/drafts/rfced/draft-ietf-sipping-sipt-04.txt, June, 2002.