Delayed-Dictionary Compression for Packet Networks

Yossi Matias
School of Computer Science
Tel-Aviv University, Israel
matias@post.tau.ac.il

Raanan Refua
School of Computer Science
Tel-Aviv University, Israel
raananr@post.tau.ac.il

***Abstract*— This paper considers compression in packet networks. Since data packets may be dropped or arrive reordered, streaming compression algorithms result in a considerable decoding latency. On the other hand, standard stateless packet compression algorithms that compress each packet independently, give a relatively poor compression ratio. We introduce a novel compression algorithm for packet networks: *delayed-dictionary compression*. By allowing delay in the dictionary construction, the algorithm handles effectively the problems of packet drops and packet reordering, while resulting with a compression quality which is often substantially better than standard stateless packet compression and has a smaller decoding latency than that of streaming compression. We conducted extensive experiments to establish the potential improvement for packet compression techniques, using many data files including the Calgary corpus and the Canterbury corpus. Experimental results of the new delayed-dictionary compression show that its main advantage is in low to medium speed links.**

I. Introduction

We consider data compression in packet networks, in which data is transmitted by partitioning it into packets. Packet compression allows better bandwidth utilization of a communication line resulting in much smaller amounts of packet drops, more simultaneous sessions, and a smooth and fast behavior of applications (see, e.g., [1]).

Packet compression can be obtained by a combination of *header compression* and *payload compression*, which are complementary methods. In this work we focus only on payload compression. We are particularly interested in dictionary-based compression. Many dictionary compression algorithms were developed following the seminal papers of Lempel and Ziv (see, [2]–[4]).

In dictionary compression, an input sequence is encoded based on a dictionary that is constructed dynamically according to the given text. The compression is done in a streaming fashion, enabling to leverage on redundancy in the input sequence.

In many packet networks, including ATM, Frame Relay, Wireless, and others, packets are sent via different routes, and may arrive reordered, due to different network characteristics, or due to retransmissions in case of dropped packets. Since streaming compression assumes that the compressed sequence arrives at the decoder in the order in which it was sent by the encoder, the decoder must hold packets in a buffer until all preceding packets arrive. This causes *decoding latency*, which may be unacceptable in some applications.

To alleviate decoding latency, standard stateless packet compression algorithms are based on a packet-by-packet compression. For each packet, its payload is compressed using a dictionary compression algorithm, independently to other packets. While the decoding latency is addressed properly, this may often result in poor compression quality, since the inherent redundancy within a packet is significantly smaller than in the entire stream.

A. Contributions

We introduce a novel compression algorithm suitable for packet networks: the *delayed-dictionary compression* (DDC). The DDC is a general framework that applies to any dictionary algorithm. It considers the dictionary construction and the dictionary-based parsing of the input text as separate processes, and it imposes a delay Δ in the dictionary construction. As a result, when decoding a packet, the decoder does not depend on any of the Δ preceding packets, eliminating or diminishing the problems of out-of-order packets and packet drops compared to streaming compression, still with a good traffic compression ratio.

We focus on two alternative encoding methods for the DDC algorithm. The first method adapts to the network propagation delay and the probability for packet loss. The second method, called *confirmed-dictionary compression*, ensures zero decoding latency. The DDC ensures that the compression ratio will be at least as good as that of stateless compression, and quite close to that of the streaming compression, with decoding latency close or equal to that of stateless compression.

There are two main alternatives for the DDC algorithm. The first is called *DDC-min* in which the compressed length of a packet is the minimum between the original uncompressed length, the Stateless compressed length, and the *Basic-DDC* (BDDC) which is a sub-method of DDC. The second alternative is the *DDC-Union* in which the dictionary is the Union of the Stateless dictionary and the BDDC dictionary.

A full tradeoff between compression ratio and decoding latency can be obtained, bridging between the extreme alternative of streaming compression (best compression ratio and worst decoding latency) and that of confirmed dictionary

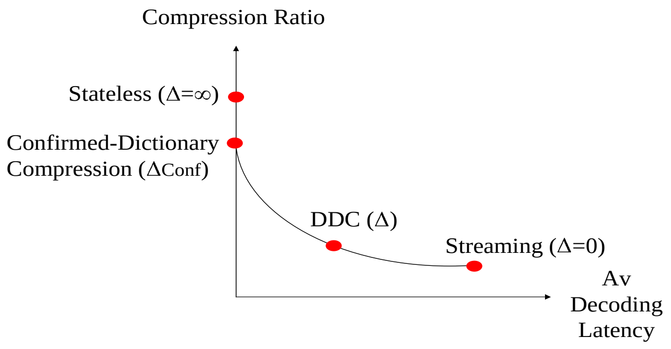

Fig. 1. Packet Compression Algorithms: A tradeoff exists between the compression ratio and the average decoding latency. Streaming has the best compression ratio and the worst decoding latency. DDC has compression ratio close to that of streaming compression, and also has average decoding latency which is close to that of stateless. The *confirmed-dictionary compression* algorithm ensures a zero decoding latency.

compression (same decoding latency as of stateless compression, yet better compression ratio). This tradeoff is depicted in Fig. 1. Thus, the DDC has the benefits of both stateless compression and streaming compression. With the right choices of the dictionary delay parameter, it can have a decoding latency which is close to that of stateless compression, and with a compression ratio which is close to that of streaming compression.

For example, for a concatenation of the Calgary corpus files, fragmented into packets with a payload of 125 bytes, in streaming the compression ratio is 0.52 with an average decoding latency of 62 packets, while in DDC-min with a large dictionary delay of 200 packets we obtain a compression ratio of 0.64 and only an average decoding latency of 14.3 packets[1].

These results can be generalized to any on-line compression algorithm that uses an on-line encoder, even if the algorithm does not use a dictionary. The DDC method is particularly good for low to medium speed communication links. Its advantage is most significant for applications in which the latency is important, and in which the order of decoded packets is not important.

To establish the potential benefit of DDC we conducted extensive experimentation. First, we compared various compression algorithms which are applicable to communication purposes, including the Deflate (gzip, winzip, based on LZ77 and Huffman), LZW (Unix compress), Predictor (Cisco), and FP-LZW (the flexible-parsing version of LZW).

We conducted extensive experimental studies to establish the gap in compression ratio when using the streaming version versus the stateless version of FP-LZW. The experiments were conducted for a large set of data files, including the Calgary corpus and the Canterbury corpus. For data files that were not already compressed, the gap is shown to be quite significant. The experiments were conducted for two packet sizes, demonstrating that the gap is more significant (as expected) for smaller packets. We show that the amount of potentially saved bandwidth may be more than a factor of 2 (e.g., for the file `rfc_index.txt` the factor is 2.23).

[1]We define the traffic compression ratio as the size of the compressed traffic divided by the size of the uncompressed traffic.

We study the dependency of compression quality and the imposed dictionary delay, showing that the improvement in compression over stateless compression could be significant even for a relatively large dictionary delay.

We also consider the effect of the dictionary delay on the performance in terms of the decoding latency. A sufficiently large dictionary delay will practically provide a zero decoding latency. We compare the decoding latencies of streaming compression vs. BDDC, showing that the latter is indeed considerably better. For streaming compression, the maximal decoding latency in seconds is $4RTT$ (e.g., for $RTT = 5000$msec and a payload size of 125 bytes the decoding latency is 963 packets) while in BDDC we can control it to be zero.

B. Related Work

Header compression was studied by Westphal [1], Lilley *et al* [5], and others.

There are several IETF (Internet Engineering Task Force) RFCs that deal with dictionary based stateless compression of packet payload, including those of Monsour *et al* [6], Friend *et al* [7], Pereira [8], and Rand [9].

The closest work to ours is that of Dorward and Quinlan [10], which introduces an approach called *acknowledged compression* for compressing payload of packets relying on acknowledgments. This method is somewhat similar to the DDC implementation of confirmed dictionary, when using the Deflate compression algorithm. In each transmitted packet an additional header is used to indicate which exact subset of packets was used as history for the compression of each packet. The implementation uses a history of up to 9 packets. In case of a large history this method has to transmit a large history information, making the method significantly less attractive. This work shows good improvement for large packets and much smaller improvement for small packets.

The DDC algorithm is a more general framework in several aspects. It allows usage of a history related to all transferred packets, resulting in better compression ratios. Using DDC, we obtain a significant improvement for small packets and large packets, even for very large dictionary delays. The decoding latency issue was addressed thoroughly in our work. In addition, we show that the method is good for any on-line compression algorithm.

A separation between dictionary construction and parsing was previously presented in [11], with a different motivation: improving the compression ratio for given data files by fixing the dictionary construction and modifying the parsing method. In contrast, in the DDC method, the parsing method is fixed (using any algorithm of choice), while we allow adaptation in the dictionary update method in terms of the imposed dictionary delay, with the objective of alleviating out-of-order phenomena.

Another work, addressing issues of the compression time factor is that of Jeannot *et al* [12].

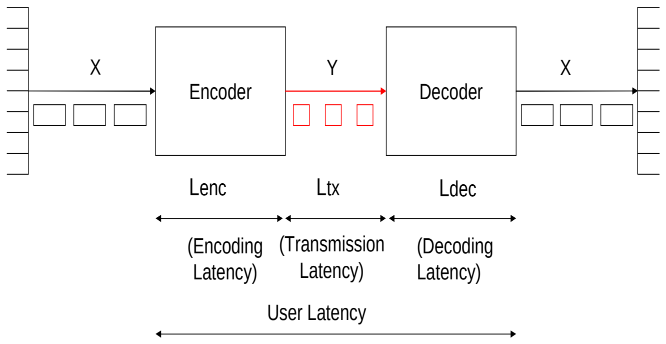

Fig. 2. Framework: Two compression enabled network processors are used, one on each side of the communication link. A LAN is connected to each side of the communication link. The total user latency is the total of the encoding latency, transmission latency, and decoding latency. X is the original uncompressed traffic and Y is the compressed traffic. Our interest is in the decoding latency and the traffic compression ratio.

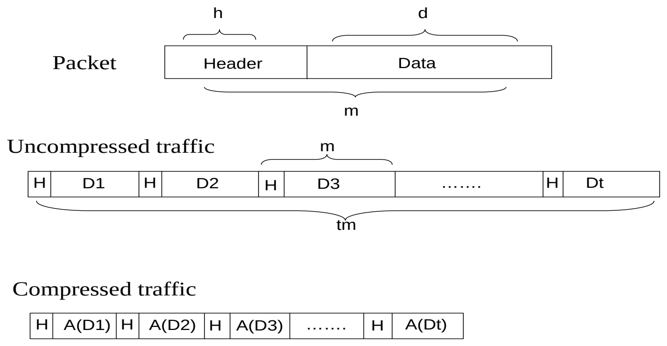
h
d
Packet
Header
Data
m
Uncompressed traffic
m
H D1 H D2 H D3 .... H Dt
tm
Compressed traffic
H A(D1) H A(D2) H A(D3) .... H A(Dt)

Fig. 3. The upper hand side of the figure is a structure of a packet which consists of a header and a payload. The middle part of the figure is the structure of uncompressed network traffic while the lower part is the structure of stateless compression traffic (each payload is compressed with algorithm A independently). The header has a *length* field indicating the length of the payload.

C. Outline

The rest of the paper is organized as follows: in Section II we present the framework, describe packet compression algorithms, and present the problem definition. In Section III we present the potential for improvement in packet compression. Section IV presents our new solution for packet compression. In Section V we present our experimental study for packet compression using a specific compression algorithm. In Section VI we present example applications. Finally, concluding remarks appear in Section VII.

II. Basics

In this section we present the framework, describe what a packet compression algorithm is, categorize the packet compression algorithms, and present the problem definition.

A. Framework

We examine the case of *end-to-end* compression over a communication link. We assume the existence of network processors, one on each side of the communication link. The original traffic that was previously transmitted to the communication link is now transmitted to the network processor and from there to the physical link. The network processors are compression enabled. This framework is depicted in Fig. 2. By compressing the traffic we achieve lower utilization[2] of a line or alternatively a larger bandwidth. This results in a smaller amount of packet drops, resulting in smaller amount of packet retransmissions. At this level we have packets carrying pieces of the original data which may be a stream of data, a file, or any other application information.

B. Packet Compression Algorithms

Packet compression algorithms are algorithms that compress packets by using standard lossless compression algorithms such as LZ77 [2], LZW [4], Deflate [13], Predictor [9], etc.

[2]The line utilization is defined as the ratio between the current transferred traffic and the line capacity for a given time period.

Packet compression algorithms use a dictionary for the compression of every packet. The decoder reconstructs the dictionary of the encoder by using phrases derived from the compressed packets.

There are basically three main types of packet compression algorithms: stateless, streaming, and offline compressions. In all the types, if a compressed packet length is greater than or equal to the original one, the original packet is transmitted.

Stateless compression (*Packet by Packet Compression*): Each packet is compressed independently, the history space is initialized after every packet is compressed or decompressed. Since each packet is independent, it can always be decompressed by the receiver, regardless of the order of arrival or of packet drops. In stateless compression, the decoding latency is minimal since packets are independent. The following IETF (Internet Engineering Task Force) RFCs are examples of dictionary based stateless compression algorithms: Monsour *et al* [6], Friend *et al* [7], Pereira [8], and Rand [9].

A structure of a single packet, the structure of the traffic without compression and the traffic in stateless compression are depicted in Fig. 3. The lossless compression algorithm A is executed on the payload of every packet separately. The required buffer size is small, since it only has a dictionary with phrases derived from a single packet.

Streaming compression (*Continuous Compression*): In streaming compression the history buffer is not initialized after every packet is encoded (resp. decoded). Each packet is encoded by using a history which is derived from all preceding packets and from the current packet. In this method packets are encoded (resp. decoded) in their consecutive order. When decoding a received packet, if some prior packets are missing, the decoder must store the current packet until all prior packets are received by *retransmissions*, resulting in large values of decoding latency. The required buffer size is larger than that of stateless compression, since it contains phrases derived from the entire encoded traffic. The buffer also contains the pending packets in which their amount is determined by the packet reordering probability. When the reliability of a link is poor, streaming compression is unattractive.

Offline compression (*Compress and Send*): First we compress the data offline, at the application layer, then we break the compressed data into packets, and finally send the packets to the receiver. This approach achieves a good compression ratio compared to the previous methods. Since compression is performed at the application layer and not inside the network processors, offline compression does not require any buffer at the network processors, nor do these cause any decoding latency at the network processor. The network processors see the packets as standard packets, since the data carried by the packets is compressed data. The packets are not compressed by the network processor, since it is not worth while to compress compressed data. The required buffer is the same buffer of the compression algorithm at the *application layer*. All known lossless compression algorithms are good for offline compression.

Some packet compression algorithms such as *thwack* (see [10]) may be combinations or variants of the three categories.

C. Problem Definition

Definition 1 *Let A be a lossless compression algorithm. Let B be a packet compression algorithm which uses A. Let X (resp. Y) be the uncompressed (resp. compressed) traffic. The* traffic compression ratio $r_{A,B}$ *is* $\frac{|Y|}{|X|}$.

The traffic compression ratio is also the ratio between the utilization of the line after compression, and the utilization of the line before compression. If the line is in full utilization, the compression ratio is identical to the line utilization. Note that the traffic compression ratio is not fixed since it depends on the currently transferred traffic, i.e. compressed data.

Let $B_{\text{stateless}}$ (resp. $B_{\text{streaming}}, B_{\text{offline}}$) be a stateless (resp. streaming, offline) compression algorithm which uses A. It is easy to prove that the following inequality holds:

$$r_{A,B_{\text{stateless}}} \geq r_{A,B_{\text{streaming}}} \geq r_{A,B_{\text{offline}}}$$

Definition 2 *Let P_i be a packet that was received by the decoder. The* decoding latency *of P_i, denoted by $L_{dec}(P_i)$, is the number of additional packets that P_i had to wait for until P_i was decodable. The average decoding latency is denoted by $\overline{L_{dec}}$.*

Our goal is to find an efficient packet compression algorithm that compresses packets along a communication line. This algorithm should have a small traffic compression ratio and also a small $\overline{L_{dec}}$.

Due to the large required delay, achieving $r_{A,B_{\text{offline}}}$ is not practical in a network processor. Therefore, $r_{A,B_{\text{streaming}}}$ would be regarded as our objective for compression ratio improvement.

III. Potential Improvement in Packet Compression

All the preliminary experiments were performed on the Calgary corpus files, Canterbury corpus files [14], and a set of various large files that consist of text files, bmp, documents, audio files, worksheets, presentations, pdf, images, mpegs, zip, etc.

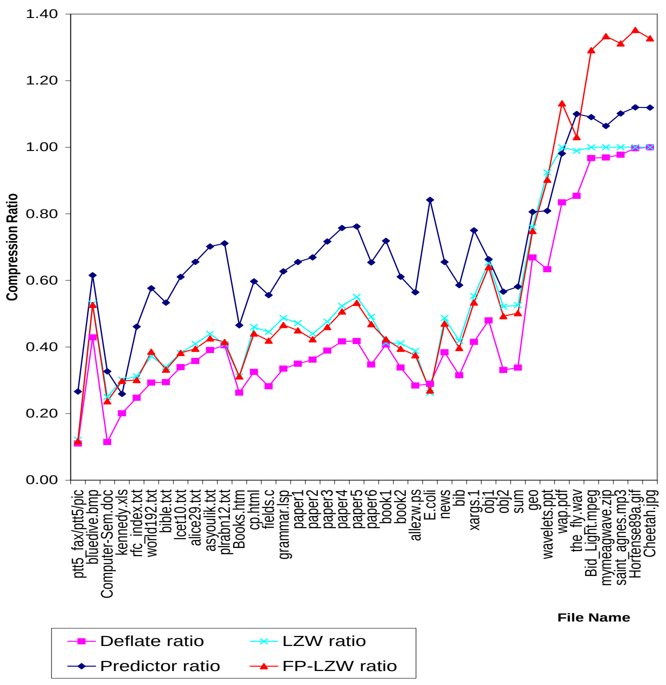

Fig. 4. Offline compression ratios of various files measured with various compression algorithms: Deflate (gzip), Predictor, LZW (compress), FP-LZW.

The purpose of the first experiment was to rank four different compression algorithms that are suitable for communication purposes with respect to their compression ratio. The algorithms are: LZW (the same algorithm used by Unix compress), FP-LZW [11], Deflate [13] (same algorithm used by gzip, for the implementation we used the zlib compression library [15]), and Predictor [9].

Note that our definition for compression ratio is *the ratio between the compressed data size and the uncompressed data size*.

The compression ratios of the various compression algorithms on the input files are depicted in Fig. 4. Deflate gives the best compression ratios, FP-LZW and LZW give close results (FP-LZW gives results at least as good as those of LZW), while Predictor gives the poorest compression ratio and is the fastest compression algorithm.

In order to measure the potential for improvement we used $r_{\text{A,Bstreaming}}$ as the goal for improvement. The goal of this experiment is to measure the actual differences between the compression ratios of stateless compression and streaming compression. A comparison of the stateless compression ratio and the streaming compression ratio when the packet size varies, with respect to the FP-LZW compression algorithm, is given in Fig. 5. We assume in this measurement a header size of 24 bytes. Two packet sizes were measured.

From observing the graphs we learn that the larger the payload the better the compression ratio, since the compression algorithm uses longer phrases.

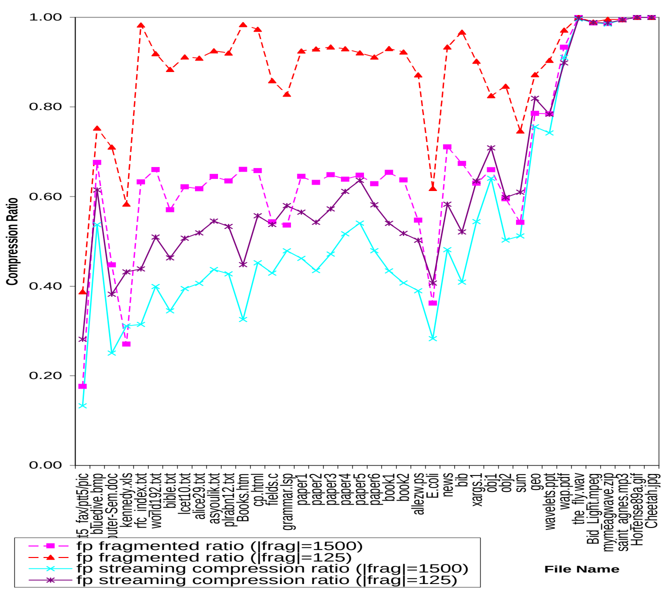

Fig. 5. The effect of the fragment size on the overall compression ratio, with respect to the FP-LZW compression algorithm: When the fragment size is very small, the quality of the compression is poor. When packet size rises, the compression ratio improves. The ratio between stateless compression ratio and streaming compression ratio for a given file is the potential improvement.

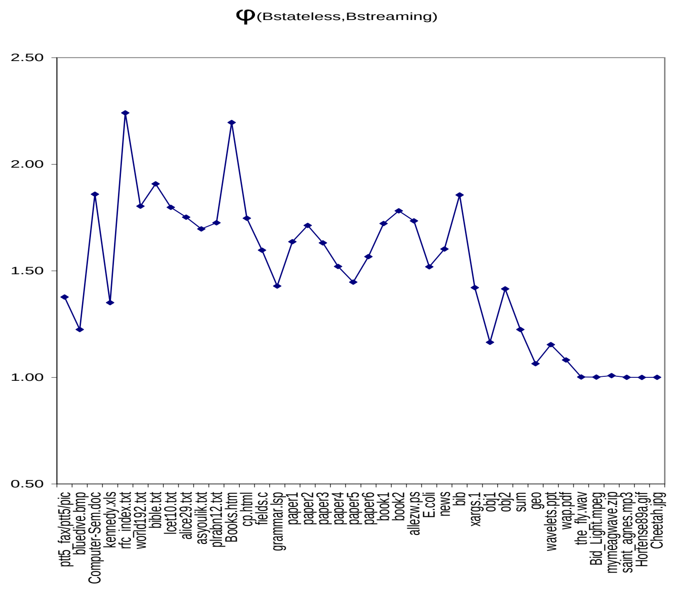

Fig. 6. The potential improvement for Calgary corpus files, Canterbury corpus files, and some files of our own. The payload size is 125 bytes.

Definition 3 *Let A be a compression algorithm. Let $B_{\text{stateless}}$ (resp. $B_{\text{streaming}}$) be a stateless (resp. streaming) compression algorithm which uses A. Let us define φ as the potential for compression ratio improvement, when using a specific packet size.*

$\varphi(B_{\text{stateless}}, B_{\text{streaming}}) = \frac{r_{A,B_{\text{stateless}}}}{r_{A,B_{\text{streaming}}}}$

The results of the experiment according to this definition are depicted in Fig. 6. These results are for small packets with a payload size of 125 bytes. If the value of the ratio is 1, we have nothing to improve. If the ratio is larger than 1, we have more room for improvements. For example, for the file `rfc_index.txt` it is $\frac{0.98}{0.44} = 2.23$. For other compressed files such as Cheetah.jpg the ratio is 1, since the file is already compressed.

IV. Delayed-Dictionary Compression

We present a generalization of an on-line dictionary compression algorithms model, the DDC method which relies on this model, and finally we describe when DDC should be used.

A. On-line Dictionary Compression Algorithms Model

Our work relies on a conceptual separation between the parsing process and the dictionary update process of dictionary based compression algorithms, as shown by Matias and Sahinalp in [11] and depicted in Fig. 7 and 8. This separation is true for any dictionary based compression algorithm.[3] This separation enables us to update the dictionary independtly to the parsing process, a fact that is used by DDC. The model

[3]The Flexible Parsing (FP) algorithm introduced in [11] explicitly enables this separation, while giving an optimal parsing method in the sense of minimizing the number of phrases produced as output for a given dictionary.

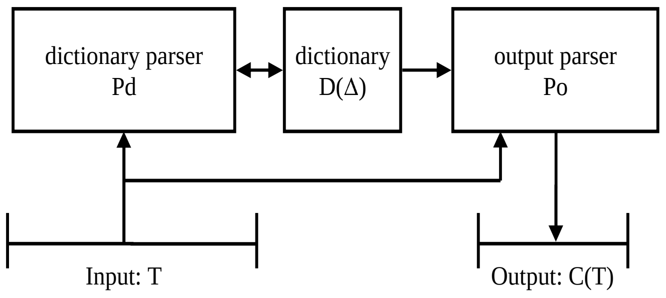

Fig. 7. Encoder: A model for incremental dictionary compression algorithm C. The model enables complete separation between the dictionary parser and the output parser.

M presented in [11] is suitable for all known dictionary compression algorithms. In this model there is a compression algorithm C and respectively a decompression algorithm $C^{\leftarrow}$. The compression algorithm C uses and possibly maintains a set of substrings D, denoted as a *dictionary*. The output of C is a sequence of codewords which is called the *compressed text* and denoted as $C(T)$. The decompression algorithm $C^{\leftarrow}$ takes as input a compressed text $C(T)$, and maintains the same dictionary D as the compression algorithm $C^{\leftarrow}$. The parsing process for constructing the dictionary is called *dictionary parser* and is denoted as Pd; the second process is the *output parser* and is denoted as Po. The encoding algorithm is depicted in Fig. 7. The input $C(T)$ to the decompression algorithm $C^{\leftarrow}$ is a sequence of codewords. After reading each codeword, $C^{\leftarrow}$ replaces it with its corresponding phrase, while building the exact same dictionary D that C builds for T. The decoding algorithm $C^{\leftarrow}$, with its different components, is given in Fig. 8.

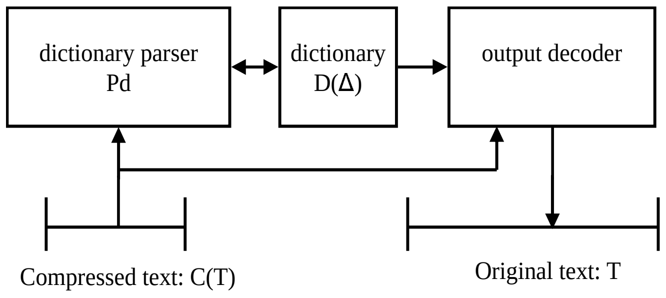

Fig. 8. Decoder: A model for incremental dictionary decompression $C^{\leftarrow}$. The model enables complete separation between the dictionary parser and the output decoder.

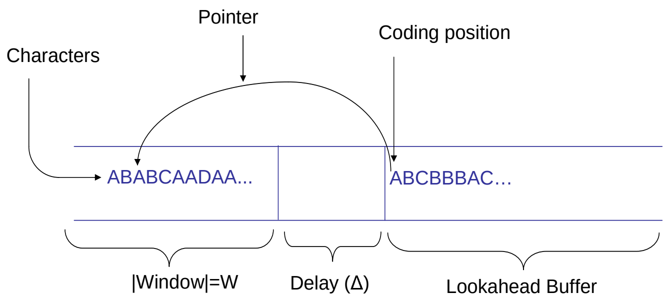

Fig. 9. BDDC-LZ77: The LZ77 compression algorithm encodes phrases as pairs consisting of a pointer and a match length to a phrase which is in a predetermined window. In BDDC-LZ77, the window is shifted Δ characters backward. Every match starts and ends in the limits of the window, and without considering the delay gap.

B. The Delayed-Dictionary Compression algorithm

We present a generalization of the model M by adding an additional parameter denoted by Δ which is a non-negative integer. The dictionary is updated in a delay of Δ units, which can be either characters or packets. In terms of M, D is a function of all the $n - \Delta - 1$ units read from the input, for $n \geq \Delta{+}1$. For all standard dictionary compression algorithms, Δ is 0 by definition. This approach is called *basic delayed-dictionary compression* (BDDC).

An example of this method for the LZ77 [2] is given in Fig. 9. The LZ77 searches the window for the longest match with the beginning of the lookahead buffer and outputs a pointer to that match. The LZ77 has a dictionary which is defined as all the strings within the window. The BDDC-LZ77 shifts the window Δ characters backward. In this case the encoded phrase depends only on the characters that preceded the last Δ characters before the lookahead buffer.

The model M can be enhanced at the encoder process. The encoding of a current character in T is a function of all the characters prior to the last Δ characters. In this case any on-line compression algorithm will do, providing its on-line encoder is a function of the text.

When considering the model M in terms of a network, we have to consider the data in a granularity of *packets*, as it appears in Fig. 10. The BDDC-LZ77 algorithm encodes the currently encoded packet by pointing to a phrase which is in packets that preceded the last Δ packets prior to the currently encoded packet.

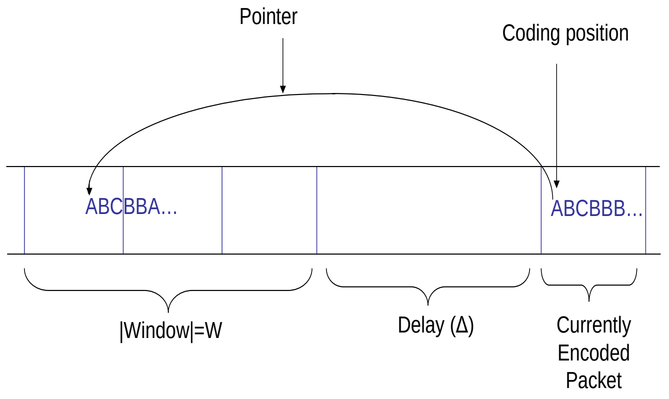

Fig. 10. Basic Delayed-Dictionary Compression with LZ77 in granularity level of packets: The LZ77 algorithm encodes the currently encoded packet by pointing to a phrase which is in packets that preceded the last Δ packets prior to the currently encoded packet.

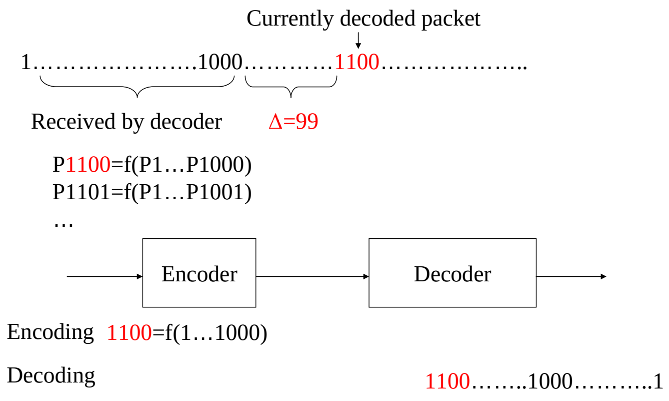

Fig. 11. Basic Delayed-Dictionary Compression: The upper part of the figure demonstrates an encoding packet with *serial number*=1100. The packet is encoded with a history based on packets 1...1000. The lower part of the figure demonstrates the decoding time - when packet 1100 is decoded, all the packets 1...1000 were already received by the decoder.

The phrases created from compressing a packet are inserted to the dictionary in a delay of Δ packets. Δ can be a constant, or adaptive according to any rule that we choose. An illustration for the case of $\Delta = 99$ is given in Fig. 11. The BDDC method compresses the network packets according to a dictionary which is updated with a delay Δ proportional to the network propagation delay.

Each packet is compressed by using a history based on all the packets that were originally received by the encoder, except for the last Δ packets that preceded the currently encoded packet. When this packet is decoded, our ability to decode it is of high probability, even without receiving all the packets that preceded it, since the original encoding is not using them. Even if some packets were dropped or reordered, we may still be able to decode the received packet, since due to the use of delay we do not depend on the last Δ packets (they are in the "delay gap"). In case of packet drops, a *retransmission scheme* will resend the missing packets. If the retransmission scheme is not part of the original transport protocol, it can be implemented by the encoder and the decoder pair. After the compression of a packet the encoder transmits the encoded packet. After a constant time period ($2RTT$), if no acknowledgment is received by the encoder, the encoder will retransmit the compressed packet.

The delay Δ has a strong effect on the performance of the BDDC, compression wise. If the delay is very large, the BDDC compresses the current traffic with a history based on very ancient traffic. In this case the compression ratio will be poor. When considering the probability of packet decoding success, the smaller the delay is, the smaller our chances to decode a packet, since there is a higher probability that the decoder did not receive all the packets that were required to decode the current packet.

Each header must contain a serial number of the current packet, as well as the serial number of the last packet that was used during the encoding process. The original header contains up to 2 bits that indicate the compression method, and can also indicate whether the packet is compressed or not. The ability to assign packets as uncompressed is good for traffic that was already compressed as well as for encrypted traffic. An adaptive algorithm layer using these special bits can be used when traffic compressibility changes along a session as described in [6].

This extra header information takes up to 4 bytes. If the *compressed length* $+$ $|$*additional header*$|$ $\geq$ *uncompressed length*, the encoder will transmit the original packet without compression (we can recover the required phrases for the dictionary synchronization by compressing the packet at the decoder side).

The Encoder:

For the implementation of the delay we use a FIFO queue as a data structure which uses real *packet* structures, or empty *dummy* packet structures. Each *packet* structure contains a list (implemented by another FIFO queue) which stores all the dictionary phrases which were created due to the compression of the packet.

We can assume that each dictionary based compression algorithm has a conceptual *InsertPhraseToDictionary* function. This function was replaced with another function that inserts a *packet* queue element to the dictionary. The main FIFO queue is initialized with *delay* dummy packets in order to immediately start the encoding with a delay. Each time we insert a *packet* structure to the queue we also extract a *packet* structure from the queue and insert all the phrases in it to the dictionary. The internal structure of the encoder (resp. decoder) appears in Fig. 12. The structure of the encoder in BDDC is identical to the structure of the encoder of streaming, except for the FIFO queue. In streaming compression the dictionary is a function of all prior packets and the current packet, while in BDDC it is a function of all the packets prior to the last preceding Δ packets. However, in stateless compression the dictionary is only a function of the current packet.

The Decoder:

The *output decoder*: When a packet is received by the decoder, if the serial number of the last packet that was inserted to the

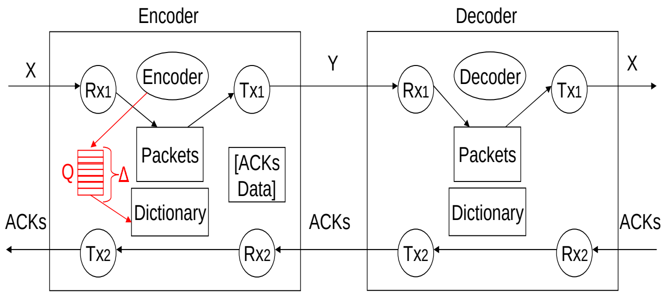

Fig. 12. Internal structure of the Encoder and the Decoder: The encoder task transfers phrases to the dictionary task by using a FIFO queue. The queue is initialized to Δ dummy packets.

dictionary $\geq$ the serial number of the last packet used for compression (taken from the header), then the decoder can certainly decode the received packet. If the dictionary does not have all the required packets (marked by the last packet used for compression), the packet is inserted to a data structure that contains all the packets that are waiting for decoding. When a new packet is received the decoder examines whether the packet is required for the decoding of the waiting packets and decodes accordingly.

The *dictionary parser*: If a received packet i has all the predecessors packets in the dictionary, then the decoder decodes the packet and updates the dictionary.

We present four encoding algorithms based on the Basic DDC approach:

Adaptive Delay Algorithm: This algorithm maintains a *delay* parameter which is changed dynamically according to the decoding success and the changes in the probability for packet drops.

If the decoder fails to constantly decode packets, it will send a message to the encoder asking it to increase the delay used for encoding (this is done by adding dummy packets to the FIFO queue). If the decoder succeeds in decoding packets for a large amount of packets, it will send a message to the encoder asking it to decrease the delay used.

When the decoder senses that the probability for packet loss increases, it signals the encoder to increase the *delay* parameter resulting in deterioration of the traffic compression ratio. However, when using a small delay there is a higher probability for not receiving all the packets required for the decoding of a specific packet. This will result in adding the specific packet to the list of packets that are pending decoding, and therefore larger L_{dec}. The decoder will signal the encoder to increase the *delay* parameter. This *feedback* method will ultimately converge to a specific *delay* value. See our measurements for the values of the compression ratios for each converged delay value. This approach can also be enhanced to a different method of *prediction* of the required delay by the encoder.

The BDDC method has an inherent conflict between $\overline{L_{dec}}$ and the traffic compression ratio: increasing the dictionary delay will cause a decrease in the $\overline{L_{dec}}$, and also a poor traffic

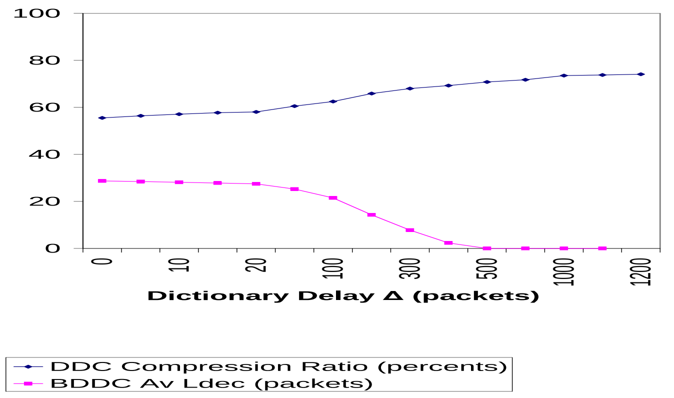

Fig. 13. BDDC Conflict: Increasing the dictionary delay will cause a decrease in $\overline{L_{dec}}$, and also a poor traffic compression ratio.

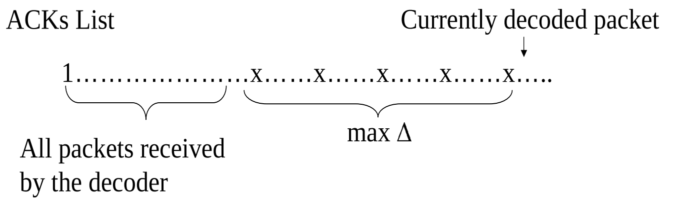

Fig. 14. Confirmed-Dictionary Compression: Each packet is encoded by choosing a maximal Δ value such that the packet is encoded with a history based on packets that have surely arrived to the decoder. The encoder uses the acknowledgments to maintain an acknowledgements data structure.

compression ratio. This conflict is depicted in Fig. 13.

Confirmed-Dictionary Compression Algorithm: This algorithm ensures immediate decoding. A similar approach was introduced in [10]. In this approach the encoder registers the acknowledgements transmitted from the decoder side. The encoder knows which packets were already received by the decoder, thus the encoder encodes the current packet with a history which is based on packets that have surely arrived to the decoder. The encoder maintains an *acknowledgments data structure* based on packets that were received by the decoder. This list appears in Fig. 14. Therefore, the encoder is guaranteed that the new encoded packets will be successfully decoded by the decoder. In this approach we set the dictionary delay in the encoder to a maximal value such that every packet received by the decoder will be decoded immediately, i.e. $L_{dec} = 0$. This algorithm does not require any buffer for storing packets waiting to be decoded. The tradeoff in this algorithm is the compression ratio.

The DDC Algorithms: The DDC algorithms are a combination of BDDC and stateless compression. The purpose of these algorithms is to ensure that the compression ratio of DDC does not become worse than stateless compression. This may happen in small files, since once the FIFO queue is large and the stream is short, the phrases in the queue do not have a chance to enter the dictionary. Therefore the basic DDC method is good for long streams.

There are two alternative algorithms for DDC: At the first alternative, each packet will be compressed twice - by the BDDC dictionary and the stateless dictionary. The result will be the minimum between the uncompressed length, the stateless compressed length, and the BDDC compressed length. This method is known as *DDC-min*. Since we use the stateless dictionary, the compression ratio will be at least as good as the stateless compression ratio. When the decoder receives such a packet, it will have the same DDC dictionary, and it will reconstruct the required phrases from the compressed packet. Note that one of the options of the minimum is *uncompressed length*. This is useful for implementation of an adaptive compression algorithm that identifies the compressibility of the traffic along a session (e.g. when traffic is encrypted or compressed).

At the second alternative, each packet is compressed with a dictionary which is the union of the current DDC dictionary and the stateless dictionary to be created from this packet during compression. This method is known as *DDC-Union*. This algorithm is part of our future work.

All the algorithms described in this section must address an inherent problem: many dictionaries re-initialize themselves when they reach their maximal capacity. This may cause decoding problems: the decoder may receive a packet, such that all the phrases that are needed for decoding are not in the dictionary of the decoder anymore, due to re-initialization of the decoder's dictionary.

To address this problem we can maintain two dictionaries in the decoder: *current* dictionary and *previous* dictionary. *Previous* is the dictionary before re-initialization and *current* is the dictionary after re-initialization. If a decoded packet requires phrases from the *previous* dictionary due to the network propagation delay, it will simply use the *previous* dictionary.

The required buffer size for the dictionary is twice than the required buffer size in streaming compression, due to the dictionary re-initialization problem. The buffer required for storing waiting packets at the decoder is smaller by a factor of Δ compared to streaming compression.

C. When to use DDC

DDC is useful in latency critical applications, especially when the compression is in the network processor, and not in the application. If the application is unknown we cannot assume anything about the latency requirements of the application, and therefore streaming is unacceptable. If the latency is not important (which means that we have information about the application) we may consider using streaming or even concatenate packets and use off-line compression.

DDC has an advantage over streaming when the order of the packets is not required.

If packet drops are allowed by the application, we can use DDC and add a retransmission mechanism for dictionary update purposes.

If the application requires order and packet drops are not allowed (retransmissions exist), there is no point in using DDC. Streaming is better in this case since it has a better compression ratio, and the packets will have to wait in any case for the application concatenation (e.g., TCP). However, this criteria requires knowledge about the application by the network processor.

Choosing a dictionary delay of $\Delta \geq 2RTT$ ensures an immediate decoding in practical terms. We assume that a dropped packet will reach its destination after a retransmission. We also assume that the timer in the retransmission mechanism expires after $2RTT$. This assumption means that the time from the point that a packet drop was detected by the decoder, until the time it was received at the decoder after a retransmission is exactly $2RTT$. Let us assume $\Delta \geq 2RTT$. Let P_i be a packet that has been received by the decoder. P_i was compressed by using a history which depends only on packets that were received by the decoder at least $2RTT$ ago. When Pi is received by the decoder all the packets that P_i depends on were already received by the decoder. If one of these packets was dropped, it had enough time for a retransmission according to the assumptions. Such packets have arrived to the decoder before P_i was received by the decoder.

Currently DDC uses one dictionary for all the connections between the two ends. In case of web traffic, it is worth while to keep one dictionary, and not create a different dictionary for every separate connection, since there is an inherent redundancy across several different connections, as shown by Spring and Wetherall in [16]. This is application dependent, therefore in some applications a different dictionary should be created for every connection.

By using the results of our measurements and a simple arithmetic calculation, it can easily be shown that DDC is good for low to medium speed links. When using high speed communication links there is not much point in data compression. In a fast line it is better to send the information as is, since the time required for compression, transmission, and decompression will be longer than just transmitting it, within a high probability.

V. Experiments

For all the experiments we used software written in C++, using the Linux OS. For the compression ratio experiments we used local Linux machines. Appropriate encoders were implemented for this purpose. For the decoding latency experiments we used hundreds of remote nodes over the internet of the Linux based Planet-Lab [17] testing environment, which is suitable for networking measurements.

A. Compression Ratio vs. Dictionary Delay

The algorithm we chose for the experiment is FP-LZW. As we mentioned earlier, there is a conceptual separation between the parsing process and the dictionary update process. The implementation of FP-LZW explicitly gives this separation, and is very convenient for modifications [18].

For the measurements we used a concatenation of the 18 Calgary corpus files [14]. In order to avoid a deviation in our measurements we assumed an incoming header size of 20 bytes, which is relatively large. The outgoing header size is 24 bytes (original header size + our additional 4 bytes).

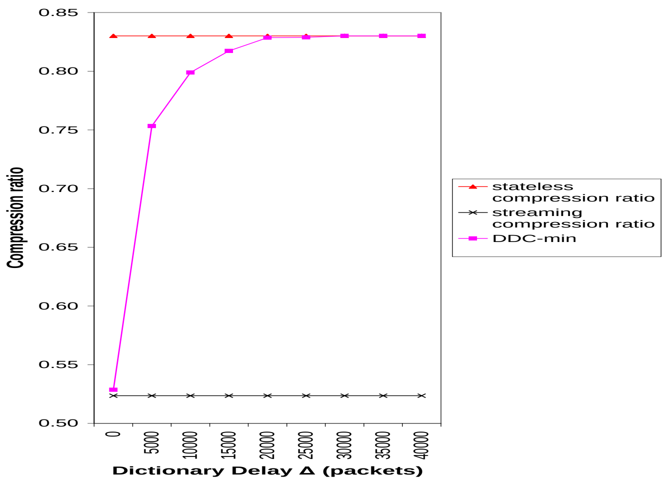

Fig. 15. Compression ratio of DDC-min as a function of the dictionary delay in packets, compared to stateless compression ratio and to streaming compression ratio. The ratio for streaming compression is very close to the DDC-min ratio with zero dictionary delay. The data file in use is the concatenation of 18 Calgary corpus files, $|Header| = 20, |Payload| = 125$. DDC-min obtains a good compression ratio even for large dictionary delays.

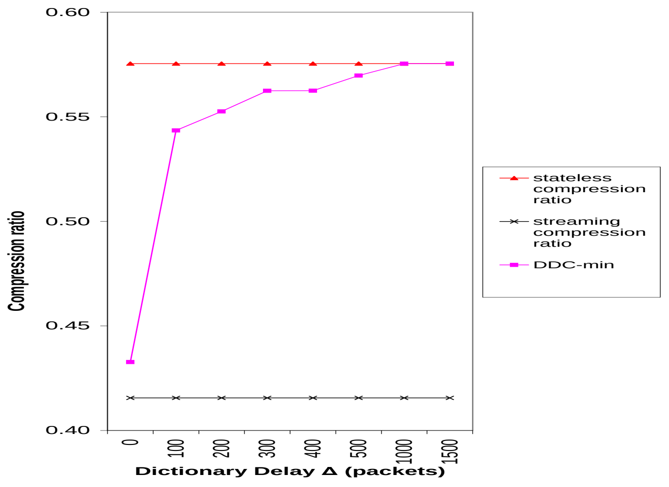

Fig. 16. Compression ratio of DDC-min as a function of the dictionary delay in packets, compared to stateless compression ratio and to streaming compression ratio. The data file in use is the concatenation of 18 Calgary corpus files, $|Header = 20|, |Payload| = 1500$. DDC-min is useful for Δ of up to 1500 packets.

A description of the traffic compression ratio for each delay value for the case of DDC-min is given in Fig. 15 and 16. Two main cases are presented: a payload of 125 bytes and 1500 bytes. Our results show that a significant improvement is achieved, even with a large dictionary delay, which is considerably better than the result of *thwack* shown in [10]. For example, let us consider the case of small packets where $\Delta = 15$. Let $Deflate_{\text{streaming}}$ (resp. $FP - LZW_{\text{streaming}}$) be the streaming version of Deflate (resp. FP-LZW). The ratio $\frac{r_{\text{Deflate,thwack}}}{r_{\text{Deflate,Deflate}_{\text{streaming}}}}$ is approximately 1.5, which means that there is still more room for improvement. However, in DDC the ratio $\frac{r_{\text{FP-LZW,DDC}}}{r_{\text{FP-LZW,FP-LZW}_{\text{streaming}}}}$ is only 1.04, i.e. DDC is very close to streaming. In addition, the measurements in [10] assume a zero length header which is not practical, and causes the compression ratio to become better.

For large packets, we also receive a good improvement. The DDC-min method is good for a dictionary delay of up to 1500 packets. As shown earlier in Fig. 5, for a larger payload, there is a better compression ratio.

B. Decoding Latency vs. Dictionary Delay

Real network experiments were conducted to measure the packet drop probability, packet reordering probability and the improvement of $\overline{L_{dec}}$ of BDDC over streaming compression. In all the experiments we used hundreds of nodes over the internet of the *Planet-Lab* environment [17] which is good for networking measurements. We used a concatenation of 18 Calgary corpus [14] files, which has a total size of 3.25M bytes. We broke the concatenated files to fragments of 125 bytes and transmitted them with UDP to 269 different Planet-Lab nodes in different geographical positions all over the world. In each transmission there were 26012 packets. The transmission rate was 6k bytes of data per second.

The first experiment measured the packet drop probability, which for all the measurements is 1.27%, or alternatively each packet will reach its destination in a probability of 98.73%. The variance is 0.002. These results suggest that in case the packet has been dropped, it will reach its destination after a single retransmission within a very high probability. Packet drops at the internet was also studied by Borella [19].

The packet reordering probability, without using retransmissions (due to change of paths, etc.), was negligible. It happened only in 137 packets out of a total of 7M packets. When retransmissions were used the packet reordering probability was increased to 0.008. The packet reordering probability was also studied by Bellardo [20].

The goal of the second experiment was to compare $\overline{L_{dec}}$ of streaming compression vs. $\overline{L_{dec}}$ of BDDC. The measurements were for various round trip times of 1000…5000msec. The total average L_{dec} of streaming compression and BDDC with $\Delta = 100$ as a function of the round trip time is depicted in Fig. 17. This is a total average of all the averages derived from all the 269 transmissions. The total average L_{dec} of BDDC is smaller than that of streaming.

Let us examine the case where $RTT = 5000$msec. In Fig. 18 we see the $\overline{L_{dec}}$ of streaming compression for every transmission out of the 269 transmissions. The variance in this case is very high. In Fig. 19 we compare the total average L_{dec} of streaming compression for the case of $RTT = 5000$msec with the total average L_{dec} of BDDC for various dictionary delays. We can see that for all values of Δ the total average L_{dec} of BDDC in terms of packets is smaller than that of streaming compression. When increasing the dictionary delay of BDDC, the total average L_{dec} becomes smaller. In particular for Δ=500 each packet in streaming is waiting on average for 62 packets, while in BDDC no packets are waiting at all for decoding.

The distribution of streaming compression for $RTT = 5000$msec vs. BDDC with $\Delta = 300$ is given in Fig. 20.

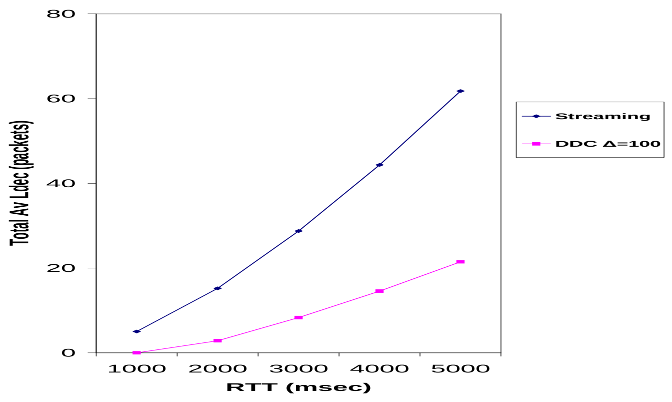

Fig. 17. The total average L_{dec} of streaming compression and BDDC with $\Delta = 100$, over all the 269 transmissions for various round trip times of 1000…5000msec. The total average L_{dec} of BDDC is smaller than that of streaming.

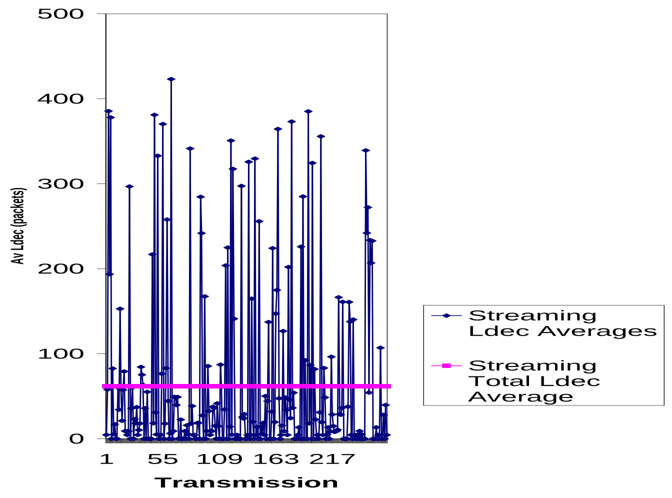

Fig. 18. $\overline{L_{dec}}$ of streaming for every transmission out of the 269 transmissions. The variance in terms of packets is very high. The total average is represented by the horizontal line.

In streaming, the percentage of pending packets, i.e, packets with $L_{dec} \neq 0$, is 18.8 % with maximal L_{dec} value of 963 which represents $4RTT$[4]. In the case of BDDC, the percentage of pending packets is reduced to 17.1% in BDDC. The maximal L_{dec} for BDDC is 383 packets which represents $2RTT - \Delta \geq 0$.

VI. Example Application

In this section we will describe an example for an application: a PC to mobile chat over SMS (Short Message Service) messages. Such applications exist in PC based instant messenger clients. The PC based messenger client enables sending of SMS messages to mobile phones through its PC clients. The SMS messages are transferred to the SMSC (Short Message Service Center) by using special SMS protocols. Then the SMS messages are transferred to the mobile phone

[4]In case of a massive packet drop that comes in a large burst which is equal to $2RTT$ in terms of packets, the first non-dropped packet after the burst may wait up to $4RTT$ since the retransmission timer is set to $2RTT$.

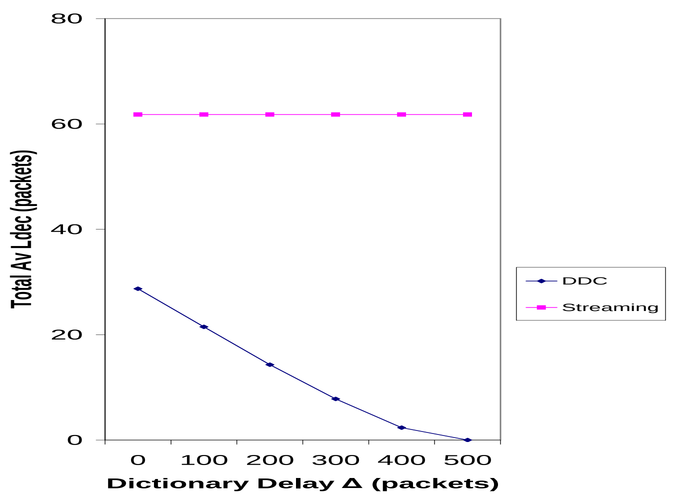

Fig. 19. The Effect of the Dictionary Delay on the Decoding Latency: Increasing the dictionary delay will cause a decrease of $\overline{L_{dec}}$ at the decoder. The graph shows that on average every packet in streaming waits more than a packet in BDDC. In particular when $\Delta = 500$ packets in BDDC do not wait at all while packets in streaming wait on average for 62 packets. $RTT = 5000$msec.

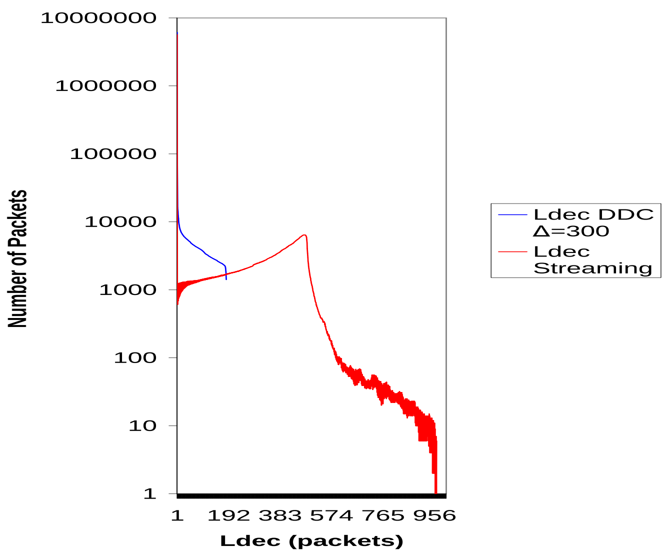

Fig. 20. Comparison between the Decoding Latency Distribution of BDDC and Streaming: $RTT = 5000$msec is assumed. The maximal L_{dec} value for streaming is $4RTT$ which is 963 in terms of packets. The other graph is for BDDC with $\Delta = 300$. The maximal L_{dec} value for DDC is $2RTT - \Delta$ which is 383 packets. BDDC achieves smaller L_{dec} values compared to streaming.

through dedicated low speed SS7 (Signaling System 7) control links. The speed of the SS7 control links is 56kbps or 64kbps, therefore they are considered as a bottle neck. The system overview of this application appears in Fig. 21. ICQ[5] is a commercial example of such a PC client.

The amounts of SMS messages are known to increase from day to day due to the high demand for this service. This causes a major problem in the SS7 links. Our method enables compressing traffic, while maintaining on a reasonable latency in user terms. The compression enables more SMS messages to be sent along the given bandwidth, with good L_{dec}. Other examples would be *instant messaging clients*: Yahoo Messenger[6], MSN Messenger[7], ICQ send short textual messages. There are organizations that are physically divided into remote geographical places. Such organizations may use the instant messaging clients with a compression feature, such as DDC, thus saving bandwidth in the network paths that connect parts of the organization.

DDC can be used in *network management applications*. When checking if nodes are functioning, we do not care about the order of the check. We only want to know whether failure exists. Many compressed management commands can go through the same channel to various nodes.

Sensor Networks constantly send data, which can be transmitted via a bus using compressed packets, thus saving transmission energy.

VII. Conclusions

We show a method for compressing the payload of network packets named *delayed-dictionary compression* (DDC). The method deals with the problems of packet drops, packet reordering and the dictionary synchronization problems that ensue. The DDC method has an advantage over the traditional streaming compression, since when decoding a packet, the decoder does not have to wait for all its predecessor packets. It also has an advantage over stateless compression in that it can use dictionaries that are based on a large number of packets, resulting in a better compression ratio. The method has particular advantage for slow to medium speed communication links. While the current paper focused on dictionary-compression algorithm, the method may be helpful for various types of online compression algorithms. Experimental study establishes the potential for compression improvement in packet networks, as well as the actual improvements obtained by the DDC algorithm presented here.

The research presented here focuses on the issues of com-

[5]http://www.icq.com
[6]http://messenger.yahoo.com
[7]http://messenger.microsoft.com

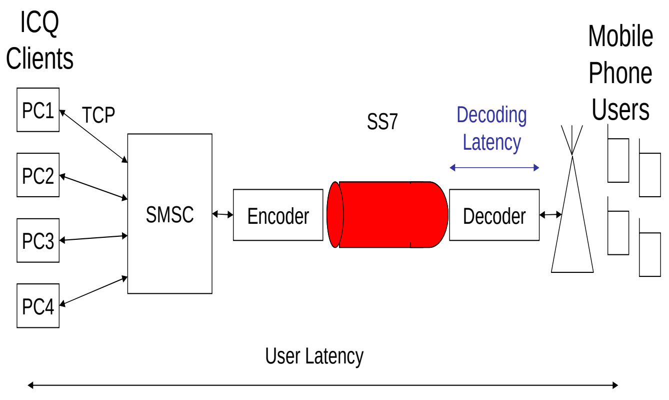

Fig. 21. SMS Chat from PC to Mobile: SMS messages are sent from Instant Messenger PC clients to mobile phones by using TCP at first, and SS7 links after wards. The bottle neck is the low speed SS7 links.

pression quality versus latency at the decoding end. Another issue of significance in the context of packet networks is the time in which encoding and decoding take place. When a single session exists, if the time spent on compression and transmission is longer than the time required to transmit the information without compression, there is no point in using data compression. However, even if the compression method increases the overall latency while allowing more sessions to be transferred on the communication link, compression is indeed useful. Jeannot *et al* [12] present an algorithm that allows overlapping of communications with compression and to automatically adapting the compression effort to currently available network and processor resources. Addressing the issue of time for DDC is the subject of research in progress.

Future work will deal with the implementation of the *DDC-Union* which ensures a compression ratio at least as good as stateless compression. We are studying other methods of DDC, including a decoding method called *eager decoding*. The *eager-decoding* is expected to give better decoding latency, by attempting to decode every received packet without relating to the required history, since all required phrases for decoding may exist in the dictionary. We are also studying the *confirmed-dictionary compression* algorithm.

ACKNOWLEDGEMENTS

Research supported in part by the Israel Science Foundation. The Planet-Lab installation at Tel Aviv University was supported by an Intel Research Grant.

The authors wish to thank Amir Averbuch, Hanoch Levy, and Uri Yechiali for helpful discussions in the course of this work.

VIII. APPENDIX

The files that were used in the experiments are the Calgary corpus files, Canterbury files, and some of our own files. The list of the file names and their size is given in Table. I.

TABLE I
FILES USED FOR THE MEASUREMENTS

Name	Size
ptt5_fax/ptt5/pic	513,216
bluedive.bmp	2,359,434
Computer-Sem.doc	383,488
kennedy.xls	1,029,744
rfc_index.txt	497,008
world192.txt	2,473,400
bible.txt	4,047,392
lcet10.txt	426,754
alice29.txt	152,089
asyoulik.txt	125,179
plrabn12.txt	481,861
Books.htm	3,498,064
cp.html	24,603
fields.c	11,150
grammar.lsp	3,721
paper1	53,161
paper2	82,199
paper3	46,526
paper4	13,286
paper5	11,954
paper6	38,105
book1	768,771
book2	610,856
allezw.ps	344,346
E.coli	4,638,690
news	377,109
bib	111,261
xargs.1	4,227
obj1	21,504
obj2	246,814
sum	38,240
geo	102,400
wavelets.ppt	437,2487
wap.pdf	1,231,123
the_fly.wav	921,992
Bid_Light.mpeg	1,159,172
mymeagwave.zip	634,320
saint_agnes.mp3	1,957,430
Hortense89a.gif	1,577,504
Cheetah.jpg	277,978

REFERENCES

[1] C. Westphal, “A user-based frequency-dependent IP header compression architecture,” *IEEE, Globecom*, 2002.

[2] J. Ziv and A. Lempel, “A universal algorithm for sequential data compression,” *IEEE Transactions on Information Theory*, vol. 23, no. 3, pp. 337–343, 1977.

[3] ——, “Compression of individual sequences via variable-rate coding,” *IEEE Transactions on Information Theory*, vol. 24, no. 5, pp. 530–536, 1978.

[4] T. Welch, “A technique for high performance data compression,” *IEEE Computer*, vol. 17, no. 6, pp. 8–19, 1984.

[5] J. Lilley, J. Yang, H. Balakrishnan, and S. Seshan, “A unified header compression framework for low-bandwidth links,” *ACM/IEEE International Conference on Mobile Computing and Networking*, pp. 131–142, 2000.

[6] A. Shacham, R. Monsour, R. Pereira, and M. Thomas, “IP payload compression protocol (IPComp),” *IETF, RFC 2393*, 1998, http://www.ietf.org/rfc/rfc2393.txt.

[7] R. Friend and R. Monsour, “IP payload compression using LZS,” *IETF, RFC 2395*, 1998, http://www.ietf.org/rfc/rfc2395.txt.

[8] R. Pereira, “IP payload compression using DEFLATE,” *IETF, RFC 2394*, 1998, http://www.ietf.org/rfc/rfc2394.txt.

[9] D. Rand, “PPP Predictor compression protocol,” *IETF, RFC 1978*, 1996, http://www.ietf.org/rfc/rfc1978.txt.

[10] S. Dorward and S. Quinlan, “Robust data compression of network packets,” *Bell Labs*, 2000, unpublished manuscript.

[11] Y. Matias and S. C. Sahinalp, “On the optimality of parsing in dynamic dictionary based data compression,” *SODA: ACM-SIAM Symposium on Discrete Algorithms*, 1999.

[12] E. Jeannot, B. Knutsson, and M. Bjorkman, “Adaptive online data compression,” *HPDC*, 2002.

[13] P. Deutsch, “Deflate compressed data format specification version 1.3,” *IETF, RFC 1951*, 1996, http://www.ietf.org/rfc/rfc1951.txt.

[14] T. C. Bell, J. G. Cleary, and I. H. Witten, *Text Compression*. Prentice Hall Advanced Reference Series, 1990.

[15] J. Gailly and M. Adler, “Zlib - a massively spiffy yet delicately unobtrusive compression library,” 1996, http://www.zlib.org.

[16] N. Spring and D. Wetherall, “A protocol independent technique for eliminating redundant network traffic,” *ACM SIGCOMM*, 2000.

[17] “Planet-lab,” http://www.planet-lab.org.

[18] Y. Matias, N. Rajpoot, and S. C. Sahinalp, “The effect of flexible parsing for dynamic dictionary based data compression,” *Data Compression Conference*, pp. 238–246, 1999.

[19] M. Borella, “Measurement and interpretation of internet packet loss,” *Communication and Networks*, vol. 2, no. 2, pp. 93–102, 2000.

[20] J. Bellardo and S. Savage, “Measuring packet reordering,” *ACM SIGCOMM*, 2002.