2001

Propagation of updates to replicas using error-correcting codes

Karthik Palaniappan
West Virginia University

Propagation of Updates to Replicas using Error Correcting Codes

Karthik Palaniappan

Thesis submitted to the
College of Engineering and Mineral Resources
at West Virginia University
in partial fulfillment of the requirements
for the degree of

Master of Science
In
Computer Science

Dr. James Mooney, Ph.D., Chair
Dr. V. Jaganathan, Ph.D.
Dr. Mathew Valenti, Ph.D.
Dr. Sumitra Reddy, Ph.D.

Morgantown, West Virginia
2001

Keywords: Reed-Solomon Codes, Erasure, Error, Data Consistency, Replication, RSYNC, Orlitsky

Abstract

Propagation of Updates to Replicas using Error Correcting Codes

Karthik Palaniappan

With the increase in percentage of replicas of data in the Internet, reducing the amount of bandwidth needed for propagation of updates across the replicas has become a major issue. Objective of our investigation is to design an update propagation mechanism focused on reducing the amount of bandwidth needed to propagate the change across multiple distinct versions of the replicas in a distributed system. We obtain the estimated amount of bytes changed from the user and generate parity information needed to correct these bytes using Error Correcting Codes. Transferring the parity information propagates the update. The updated data can be constructed using the parity information and the outdated data. Our investigation proved that the approach would be bandwidth efficient but computation intensive. We conclude our investigation with an update propagation mechanism that we believe would be less computationally intensive and also reduced bandwidth requirements.

Acknowledgments

I am very thankful to my advisor, Dr. James Mooney for his support, motivation and guidance and providing me this opportunity to learn to be a good researcher.

I am also thankful to my committee members, Dr. Mathew Valenti for his interest, encouragement, technical guidance and pointing out my mistakes throughout the project, Dr. Sumitra Reddy and Dr.V. Jagannathan for their encouragement, support, active participation, guidance and co-operation throughout the project. The availability of my chair and committee members throughout the project has been the key for the successful completion of this project.

I would like to take this opportunity to thank Mr. Todd Montgomery for refining my initial proposed solution, for his technical guidance, support and availability throughout the project and Dr.Bojan Cukic for being my advisor during the first semester.

I am thankful to all my family and friends, especially Sentil, Dr. Sandhu, Rajesh , Shen and Baskar for their technical and moral support. I am also thankful to those professors and fellow researchers for making their code available to me and responding to my emails. I would like to thank all my professors for nourishing me with the knowledge required to pursue the project.

I would like to extend my gratitude to Dr. Michael Henry, Dr. Wils Cooley and the CSEE department for providing me financial aid and the system staff members Jennifer Stathakis, Mr. Jarod King and Marc Seery for their availability and support.

Most of all, I thank my parents for their support, encouragement and providing me with this opportunity.

Dedicated to my Mom and Dad for giving me this opportunity

Table of Contents

1 Introduction

With the use of replicated servers and the increase in demand for remote data access, the number of replicas of the information on the Internet has increased proportionally. As reported by ShivaKumar and Garcia-Molina [ShivaKumar98] in 1998, about 46% of all the text documents on the web have at least one “near-duplicate” and 5% of them have between 10 and 100 replicas. Although all of these replicas are not intentional and require being consistent, many of the items do need or prefer to be consistent. With the increasing demand for bandwidth, propagation of an update has become an important issue. There have been many instances where we wanted to update our file already in the ftp server. Dynamic web pages need to be fetched over again and again.

In all these places the simplest solution is to transfer the entire data from the source to the destination. However, the high demand for bandwidth and the availability of computation power at a lower cost obviates the need for a bargain. Many solutions have come through in the past few decades on establishing consistency among replicated data. Transmission of difference between outdated and updated data is common to these solutions. Notification servers [Patterson96] for distributed collaborating systems, Anti-Entropy protocols for propagation of application specific modification, RSYNC[Tridgelle] update protocol that is non-application specific and the Diff algorithm based data consistency mechanisms are all common.

Often we need to transfer large objects between computers. The object could be a large array, any user-defined data type, a file etc. The object could be modified by one or more applications. There are situations where these objects are subject to frequent minor modification. We could classify mechanisms involved in establishing consistency of such replicated objects as follows:

- Producer Initiated
- Consumer Initiated

In a consumer-initiated algorithm, consistency is established only if requested and not when a change occurs. In such cases we need to send the whole object, as it is typically not efficient to keep

track of the number, location and type of changes that have occurred since the last request. The situation becomes very complex if more than two versions are involved in establishing consistency.

By sending the whole object we are not exploiting the fact that the sender already knows a part of the information that we are going to send. All the algorithms stated earlier handle the issues differently and are subject to their own limitations. We investigated the approach of propagating changes across replicated copies using error-correcting codes.

1.1 Intuition behind the Approach

Logically, the outdated data is equivalent to data received with errors. The changes that occur in data are logically equivalent to that of errors that occur during transmission. The amount of information that is required to correct such errors is sufficient to update the copy. Hence we generate and send the parity information instead of the whole object. By transmitting just the parity information, we could reduce the amount of required bandwidth significantly.

A superficial look on the approach would lead to the conclusion that the amount of parity information is significantly more than the difference between the updated and the outdated file. The approach requires more bandwidth than the diff based update propagation algorithms, as we chose to transmit erasure codes instead of transmitting the data that differ between the outdated and updated file. While this is true mostly, the ability to correct multiple versions of the document and the one-way communication makes the approach worth investigation. With the increasing use of distributed computing, it is very likely that there could be more than two versions of the object of interest. While transmitting the difference between the outdated and updated file is bandwidth efficient on a per-transfer basis, the information transferred would be more specific to those versions of the file involved in finding the difference and hence cannot be reused from an intermediate node when either of the versions is different. By using the erasure codes to propagate the difference, the information transferred would be more generic and could correct many versions of the file that has the same amount of modification. The computational complexity will be higher than the diff-based and anti-entropy based propagation algorithm. The computational costs will vary with the error correction codes used.

Our approach makes sense as long as the amount of parity information that we send to reproduce the object is significantly less than the size of the object. The amount of parity information

that we send also depends on the error correction code that we adapt to propagate the change. While we leave unanswered the question of which error correcting code fits best, we have realized through our investigation that computation time (encoding time and decoding time) and the parity information required to propagate the update would be among the major deciding factors. Also for greater symbol sizes, we need a smaller interleaving[1] depth and hence less parity information, since the errors are distributed evenly. However we chose Reed Solomon codes for further analysis of our approach, as they are byte error correcting codes that require just twice as much parity information as the change in data. Also the parity information and the source information can be separated from the encoded data without any computational cost. Any statistical data that follows in this document in describing the throughput are specific to RS encoding.

1.2 Assumed Environment

We assume no knowledge of the application(s) or patterns, if any, that may be involved in the modification. The object could be of variable size. Modifications of the object could fall into any of the primitive operations viz. insertions, deletions and substitution. We assume no knowledge of the difference between the outdated and updated file. We also assume that the outdated and updated object are never available at the same location at the same instance. In our implementation, we used a file as an object. However, throughout the investigation the generality of the object is preserved and no specific advantage has been derived. In our implementation the size of the updated object needs to be known prior to construction of updated object from the outdated object. We also expect an estimate of the amount of modifications from the user. This estimate would play a major role in the amount of bandwidth required for propagating the change. The assumption of getting an estimate of the amount of change from the user has been made to remain focused in the issue of reducing the bandwidth. However the problem of finding the right estimates without user intervention is an interesting problem and would be a good candidate to work in the future as its often unlikely that the user can estimate the amount of change in bytes. We will be discussing more about the problem of finding the right estimate or alleviating the need for the right estimate in the chapter titled “future work”.

Lalit Bahl and Frederick Jelinek [Bahl75] have mentioned the approach of propagating updates across replicas using error-correcting codes. The focus of their work has been on Speech and Character

[1] Interleaving is a mechanism used to scatter the changes that occur in bursts across multiple packets.

recognition and is suitable for other applications where the set of possible modifications should be limited in location and symbol space to a few bits/bytes.

Alon Orlitsky [Orlitsky91] has proved that if Px knows only X and Py knows only Y then X can be communicated to Py using only negligibly more bits than the number needed if Px knew Y in advance. The protocols achieving this near minimum number of bits is interactive and requires Px and Py to exchange three messages. For one-way protocols it has been proved that they require twice as many bits as needed if Px knows Y. Orlitskty has also provided an algorithm for a restricted edit problem. We save our discussion until the chapter on relevant and future work.

The rest of the document is organized as follows

Chapter 2 provides the technical background information about the search algorithm, message digests and Reed-Solomon codes. The chapter also explains our choice of the message digest and search algorithm used in the implementation.

Chapter 3 describes the versions of our algorithm and those problems that enforced the revision. The chapter walks the reader through examples of interest to provide better understanding.

Chapter 4 on simulation describes the input model used and provides experimental results that illustrate the bandwidth efficiency for various categories of input.

Chapter 5 deals with the relevant work that has been completed and is of interest to us and the future work and improvements that needs to be done.

Chapter 6 provides the conclusion of our investigation.

2 Background

In this chapter we will discuss the search algorithm, the message digests and the Reed-Solomon code used in our algorithm and implementation. The extent of explanation provided for the above topics is limited to the understanding that is required for the purpose of our discussion. The reader is encouraged to skim through the chapter and get back to it when necessary.

2.1 Search Algorithms

We use pattern matching in the process of finding a speculated block in the outdated file that would match the blocks in updated file. In our implementation we have used Boyer-Moore pattern [Boyer77] matching algorithm. The need for a faster search algorithm and our justification for using the Boyer-Moore algorithm are described in this section.

Considering the fact that the size of the object is very large in our case a significant amount of time and computation cost is spent in searching for the patterns. A deletion for instance may induce many searches in the process of a weak signature match. Search algorithms with run-time of $O(n^2)$ and higher may deteriorate performance badly. Hence, regardless of their simplicity we chose not to use a linear pattern-matching algorithm. We needed a fast text pattern-matching algorithm.

Only pattern search algorithms that produce exact matches are of interest to us. Although search algorithms with non-exact matches would serve the purpose, the computation time involved in the calculating CRC increases with every wrong result obtained from the non-exact pattern matching algorithms.

We need to search for the patterns throughout the whole file in blocks. Hence algorithms that improve performance by pre-computation are a good choice. Also in our case, the alphabets in the super-string are vast and the average number of alphabets in the sub-string varies but typically is not expected to be very less. Boyer-Moore pattern search algorithm is considered very efficient in such cases and has been used in a variety of applications. For a search string of size m and a text of length n the worst case computational time complexity is O(nm) and the best case computational complexity is O(n/m) . However if worst-case complexity is of importance to a specific application, the Apostolico-Giancarlo algorithm, a variation of Boyer-Moore algorithm, may be a good choice.

2.1.1 Boyer-Moore Pattern search Algorithm

The source of information for the following discussion on Boyer-Moore search algorithm is from the Web site http://www-igm.univ-mlv.fr/~lecroq/string/node14.html#SECTION00140 on 2/12/01.

Boyer-Moore algorithm [Boyer77] is used to search for byte patterns in a large volume of data. In this section, we will be briefly looking at how this algorithm works.

The algorithm searches from the right most symbol. When a match occurs the character at the left of the right most character is matched. The process is repeated until all the characters match. When a mismatch occurs we need to align the characters with the next possible match. The location of next possible match is the maximum of two pre-computed functions viz. good-suffix shift and bad-character shift.

Pre-computed functions Good-suffix shift and bad-character shift are calculated as follows:

Scenario:

Let X be the pattern we search and Y be the data in which we search for X. Let X consist of m bytes and let Y consist of m+n bytes. Let us consider the scenario where bytes at location j through m of X match the bytes at location i+j through i+m of Y but the character at location j-1 of X does not match the character at location i+j-1 of Y. Let us refer to the matched byte sequence at location j through m of X as U_{jm}.

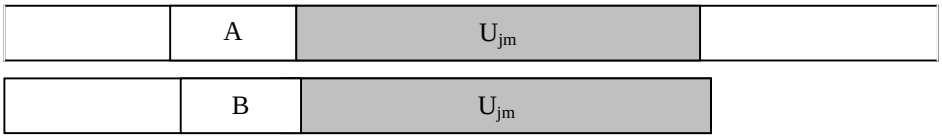


Figure 2-1 Initial Scenario

Good-suffix shift:

Good-suffix shift consist of aligning the right most occurrence of U_{jm} in location 1 through j-1 of X with characters at location i+j though i+m of Y.

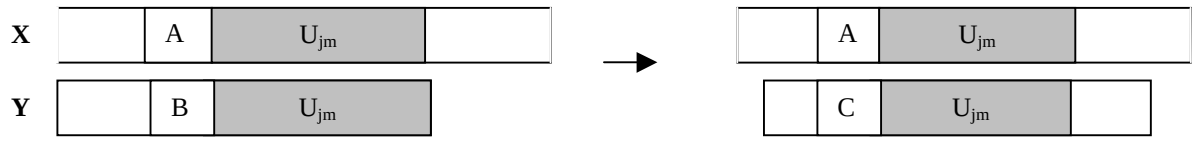


Figure 2-2 Good-suffix shift, *u* reappears in X

However if U_{jm} does not occur again in X, good-suffix shift consist of aligning the longest pattern V in characters at location 1 through j-1 of X with a left most matching pattern in characters at location i+j through i+m of Y.

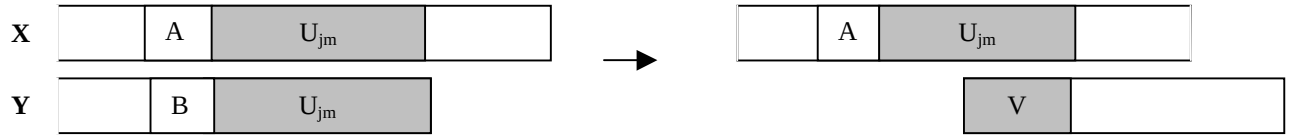


Figure 2-3 Good-suffix shift, only a prefix of *u* reappears in *x*

Bad-character shift:

Bad-character shift consist of aligning the character in Y that mismatched (character at location i+j-1 in our scenario) with its right most occurrence in characters at location 1 through j-1 of X.

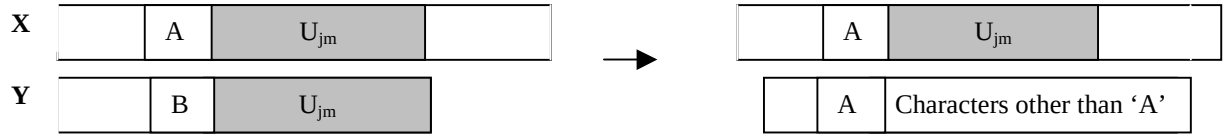


Figure 2-4 Bad-character shift, *a* appears in *x*

However if no match is found, the leftmost character at X is aligned with the character after the mismatch(character at location i+j+1 in our scenario).

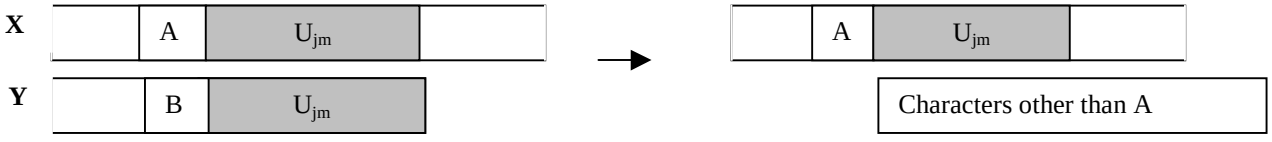


Figure 2-5 Bad-character shift, *a* does not appear in *x*

2.2 Reed-Solomon Codes

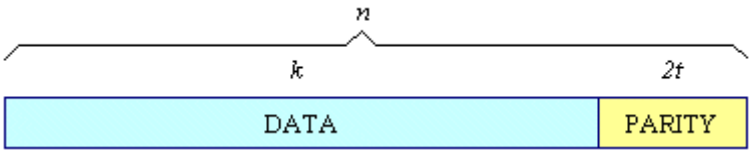
Reed-Solomon codes are block error correcting codes used in a wide range of applications in digital communications and storage devices. Reed Solomon codes can also correct erasures. Reed-Solomon codes are used in the following applications.

- Storage devices (tape, Compact Disk, DVD, barcodes, etc)
- Wireless or mobile communications (cellular data services, microwave links, etc)
- Satellite communications and communications with deep-space probes(Voyager, Galileo)
- High-speed modems such as ADSL, xDSL, etc.
- Reliable Multicast Protocols and Fault tolerance in RAID-like systems

2.2.1 Properties of Reed-Solomon codes

Reed Solomon codes are linear block codes that are a generalization of binary BCH codes. A Reed-Solomon code specified as RS(n,k) will encode k data symbols to produce n-k symbols of parity information.

The value of n depends on the symbol size. For a symbol size on n bits, $n=2^s-1$. A symbol is erroneous even if one bit in the symbol is modified. An RS(n,k) code can correct T erroneous symbols and 2*T erasures where 2*T=n-k. The following diagram shows a typical Reed-Solomon codeword (this is known as a Systematic code because the data is left unchanged and the parity symbols are appended):



For application like ours that require byte error correcting capability, symbol size is a multiple of 8. Throughout our discussion in the rest of the document we will be using 8 and 16 bit symbols only. A symbol size of 8 bits is more common compared to a symbol size of 16-bits due to the simplicity in design and low computational cost. Reed-Solomon can correct errors and erasures. The need for a symbol size of 16-bits will be illustrated in the chapter titled quantitative performance analysis.

2.2.2 Encoding

A Reed Solomon encoder takes k data symbols of data and produces a code word of n symbols. The codeword is constructed using:

$$c(x) = g(x).i(x)$$

where g(x) is the generator polynomial, i(x) is the information block, c(x) is a valid codeword and α is referred to as a primitive element of the field. As you may have realized generator polynomials are of degree n-k. The generator polynomial is of the form as below.

$$g(x) = (x - \alpha^{i})(x - \alpha^{i+1})\dots(x - \alpha^{i+2t})$$

Throughout our implementation the value of i has been 1.

Example: Generator for RS(255,249)

$$g(x) = (x - \alpha^{0})(x - \alpha^{1})(x - \alpha^{2})(x - \alpha^{3})(x - \alpha^{4})(x - \alpha^{5})$$

The code word consists of k symbols of data and n-k symbols of parity information. Reed Solomon codes are systematic codes and hence the parity information and data symbols can be segregated from one another in the code word. This property of Reed-Solomon codes makes the code adaptable to our application.

2.2.3 Decoding

The decoder accepts the received codeword r(x) and generates the codeword c(x). Received codeword r(x) is the original (transmitted) codeword c(x) plus errors e(x).

$$r(x) = c(x) + e(x)$$

In our case the errors are the changes made by the user.

The error correcting process consists of the following two steps.

1. Detecting the location of error
2. Correcting the error

Detecting the location of errors involve the following steps:

1. Calculate the syndromes.
2. Find the error locator polynomial
3. Finding roots of the Error locator polynomial

Syndrome Calculation

A Reed-Solomon codeword has 2t syndromes. The syndromes can be calculated by substituting the 2t roots of the generator polynomial g(x) into r(x).

Find an error locator polynomial

This can be done using the Berlekamp-Massey algorithm [Berleykamp65] or Euclid's algorithm [Imai90]. Euclid's algorithm is easier to implement and hence is widely used. Since we are not dealing with the issue of computational time taken by our algorithm, we used Euclid's algorithm. However, the Berlekamp-Massey algorithm tends to lead to more efficient hardware and software implementations.

Find the roots of this polynomial

The roots of polynomials are determined using Chien search algorithm [Chien64].

Finding the Symbol Error Values

Simultaneous equations with t unknowns are solved to determine the error symbol values. A widely used fast algorithm is the Forney algorithm [Forney65].

The source of above information on Reed Solomon codes is the Web page at http://www.4i2i.com/reed_solomon_codes.htm on 02/06/2001.

2.3 MD4 Algorithm

We use MD4 [Rivest] message digest algorithm to ensure that the file obtained as the output of our process is identical to the updated file. We chose MD4 as it is faster, reliable to our needs and implementation is available in the public domain. We would like to emphasize that we are not interested in deliberate attack causing a collision. Our interest as in RSYNC is on occurrences of random collision. The collision factor of MD4 is $1/2^{128}$. In applications that require highly reliable update where failure due to collision of MD4 is not tolerable, SHA-1 may prove to be more appropriate. SHA-1 is slower than MD4. SHA-1 takes a message of less than 2^{64} bits in length and produces a 160-bit message digest. The larger message digest makes it more reliable. Considering the wide use of RSYNC update algorithm successfully on large volumes of data in reality we do not expect the use of SHA-1. However MD5 is not of interest to us as both use a 128-bit fingerprint and

hence have a collision factor of $1/2^{128}$. We prefer MD4 to MD5 as MD5 takes about 1.3 times the taken by MD4 [Balenson] and the collision is not an intended collision as opposed to the cryptographic world.

The MD4 algorithm takes an input message of arbitrary length and produces a 128-bit signature of the input. The MD4 algorithm is intended for digital signature applications, where a large file must be "compressed" in a secure manner before being encrypted with a private (secret) key under a public-key cryptosystem such as RSA. The MD4 algorithm is designed to be quite fast on 32-bit machines and does not require any large substitution tables. For a description or implementation of the algorithm we suggest the reader to refer to RFC1320.

2.4 Cyclic Redundancy Check

Cyclic Redundancy check(CRC) is a fingerprint derived from a block of data in order to detect the similarity. When used as a message digest, CRC differs from MD4 by its low computation cost and less reliability. We use CRC to detect the location of block erasures in one of our two proposed algorithms. We use a 16 bit CRC.

2.4.1 A brief description of how CRC works

CRCs treat blocks of input bits as coefficient-sets for polynomials. E.g., binary 10100000 implies the polynomial:

$$1*x^7 + 0*x^6 + 1*x^5 + 0*x^4 + 0*x^3 + 0*x^2 + 0*x^1 + 0*x^0$$

This is the *message polynomial*. A second polynomial, with constant coefficients, is called the *generator polynomial* is chosen. This is divided into the message polynomial and coefficients of the remainder form the bits of the CRC. So, an order-n generator polynomial is necessary to generate a 32-bit CRC. The bit-set used for the generator polynomial will naturally affect the CRC that is computed. Hence the sender and receiver in our case must agree on the same generator polynomial.

2.4.2 Standard CRC Polynomials

CRC-8: 100000111

CRC-10: 11000110011

CRC-12: 1100000001111

CRC-16: 11000000000000101

CRC-CCITT: 10001000000100001

CRC-32: 100000100110000010001110110110111

Polynomials used vary based on the application's expected reliability. Ethernets use a CRC-32 polynomial.

The collision factor for CRC is $(m+n)/2^{n-1}$ where m is the block size and n is the number of CRC bits [Balenson].

In this chapter we discussed the background information required for understanding the design and implementation details. The topics we discussed include Boyer-Moore Pattern Search algorithm, Reed Solomon codes, MD4 algorithm and Cyclic Redundancy check. We also rationalize our design choice of using Reed Solomon codes for error correction and rationalize the use of message digest MD4 over MD5 for detecting inability of decoder to detect error.

3 Algorithm

3.1 Algorithm Description

3.1.1 Outline of Version 1

The following simple algorithm could be used for a fixed sized object where the only possible modification is substitution.

SENDER:

1. Obtain the estimated amount of bytes changed from the user. Determine the value of T from estimated amount of change in bytes.[2]
2. Split the updated data into blocks of fixed size(interleaving depth) such that (255-2*T)*block_size=object_size.
3. Interleave the blocks with a matrix interleaver of any fixed size[3].
4. Encode data block obtained from interleaving
5. Separate parity information from encoded data
6. Transmit parity information to the receiver

RECEIVER:

1. Obtain the value of T and parity information from the SENDER
2. Split outdated data into blocks of fixed size(interleaving depth) such that (255-2*T)*block_size=object_size.
3. Interleave the blocks with a matrix interleaver
4. Append the parity information obtained from the sender with the corresponding interleaved block.
5. Decode the data
6. De-interleave the decoded data and construct the data with the update.

As Error Correcting codes typically are designed for substitution, Insertions and deletions will result in shifted data and hence result in too many errors. Hence the simplified version needs further revision.

[2] Given the amount of bytes changed we obtain the value of T using our knowledge on simulation results. A few methods are proposed in the future work section as well.
[3] We will discuss about the optimal interleaving depth in the later sections of the chapter.

3.1.2 Outline of Version 2

The data is split into small blocks of fixed size(M) and the first n characters of the block, referred in the rest of the document as Clues are used to identify the block[4]. The clue forms the weak signature of the block. For every n character (clue) that has been received, a block of size M is written to a file. The block written will be the block beginning from the matched location of the clue in the outdated data in case of a match. If no match is found an arbitrary block is written to the file.

SENDER

1. Convert insertions and deletions to substitutions.
   1.1. From the updated file, transmit the first n characters (referred as weak signature throughout the document to be consistent with the terminology of RSYNC update algorithm) of each block of size M, to be searched for in the outdated file.
2. Do steps in version 1 with the outdated file as the file constructed in step 1.

RECEIVER

1. Convert insertions and deletions to substitutions.
   1.1. For every weak signature received, check whether a match for the weak signature could be found at the expected offset. If not, search for the weak signature.
   1.1. If a match is found check for a match of the weak signature of the preceding block if any and succeeding block if any at the expected offset from the weak signature of current block. Else copy an arbitrary block of size M into the temporary file. Go to step 1.3.
   1.2. If all the three weak signatures match copy the block of size M starting from location of match into a temporary file. Else, copy an arbitrary block of size M into the temporary file.
   1.3. Redo steps 1.1 and 1.2 for next weak signature if exists.

2. Do steps in version 1 with the outdated file as the file constructed in step 1

The above version of the algorithm would require large volume of data to be transferred for structured files, as the weak signature should be large enough to avoid any mismatch caused by patterns in structured files. The solution is to transfer the CRC of the weak signature instead of the

[4] An alternative is to generate random numbers and pick n characters from each of these random locations. The same seed must be used to generate random numbers. The approach may prove better than fixed offset approach we use in selecting the clues.

weak signature. However finding a matching CRC is considerably more computation intensive, as insertions and deletions would shift the weak signature from the expected offset requiring a CRC calculation of all possible consecutive n byte sequence within the location of file where the weak signature is expected to exist. The solution is to send a part of the text used in CRC calculation along with the CRC. This part of the text referred as clue, would reduce the computation time, as the CRC calculation needs to be done only when the clue matches. Our weak signature thus, will have two components, the CRC and the clue.

3.1.3 Outline of Version 3

SENDER:

1. Convert insertions and deletions to substitutions.

    1.1. From the updated file, transmit the first n characters (referred as clue) of each block of size M, to be searched for in the outdated file.

    1.2. From the updated file, compute the CRC for n+m characters of each block of size M beginning from the location of clue. The character sequence involved in the CRC computation is referred in the rest of the document as CRC block.

2. Do steps in version 1 with the outdated file as the file constructed in step 1.

RECEIVER:

1. Convert insertions and deletions to substitutions.

    1.1. For every weak signature received, check whether a match for the weak signature could be found.

        1.1.1. Check if the clue at the location matches. If not search for the clue.

        1.1.2. Determine the CRC of n+m characters starting from the location of match.

        1.1.3. If the CRC does not match, redo step 1.1 with the starting location of the search being the byte next to the location of previous match.

    1.2. If a matching weak signature is found check for the weak signature of the preceding block if any and succeeding block if any at the expected offset from the weak signature of current block. Else copy an arbitrary block of size M into the temporary file.

    1.3. If all the three weak signatures match copy the block of size M starting from location of match into a temporary file. Else, copy an arbitrary block of size M into the temporary file.

1.4. Redo steps 1.1, 1.2 and 1.3 for next weak signature if exists.

2. Do steps in version 1 with the outdated file as the file constructed in step 1

The smaller the clue, the smaller is the size of weak signature and hence less data is sent. However, smaller clues induce more false matches, increasing the number of CRC computations and hence the computational cost. We used a 16-bit CRC for the weak signature. However, any other hash value could be used.

In case of not finding a weak signature, our approach of copying an arbitrary block could be improvised to make a best guess by analyzing the cause of a weak signature not existing at the expected offset. The cause for not finding a matching weak signature is that the weak signature that we chose at the sender could be the data that is part of the update.

In such cases, if a substitution has occurred the best guess would be a block with the beginning location as the expected location. The best option would be to back track by block size M bytes from the location of next successful weak signature match to obtain the block of best guess. Fig 3.1 illustrates the best guess approach taken, in order to obtain a matching block if a weak signature could not be found.

Consider a substitution of block 2 and part of block 3.

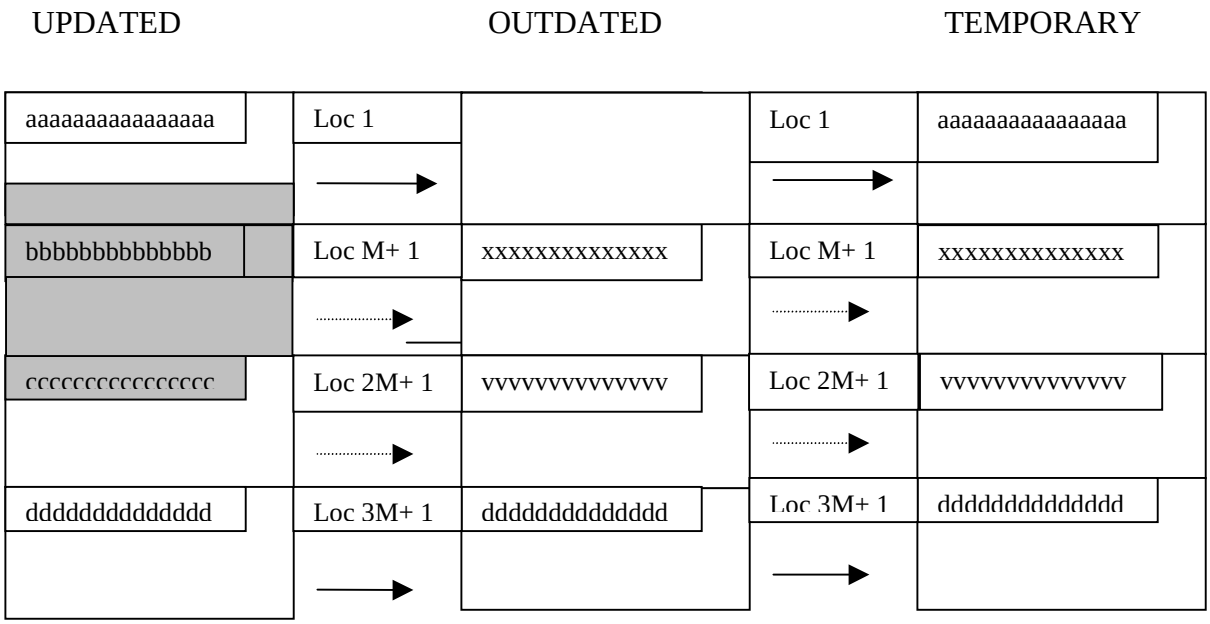


Figure 3-1 Best Guess on Location of Weak Signature for Substitutions

The weak signature at location M+1 and 2M+1 do not match. A match occurs for the next signature at location 3M+1.

Now consider a insertion at the block boundary of block 1 and 2.

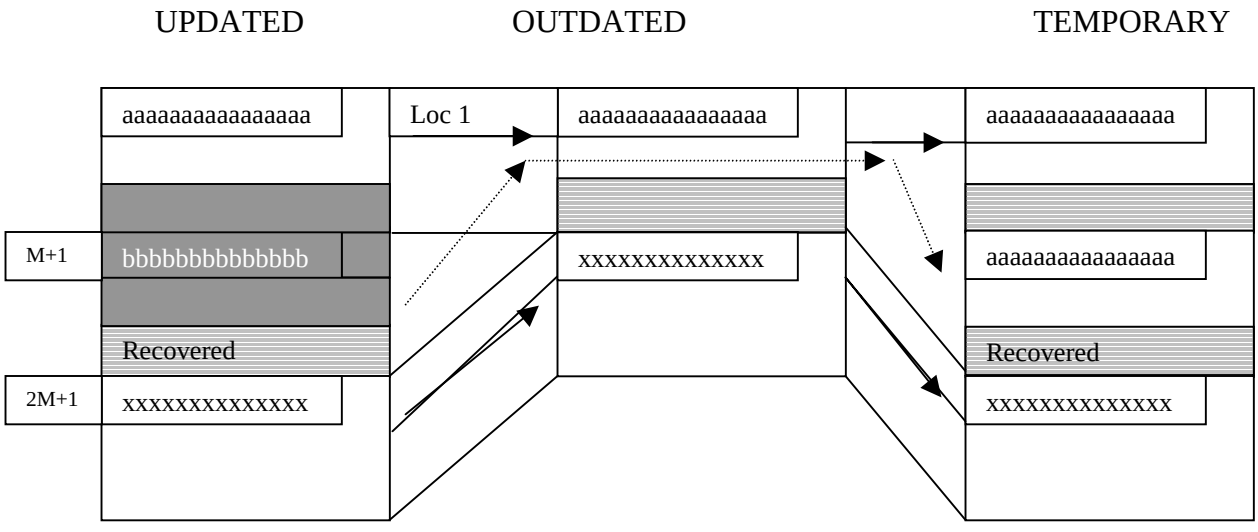


Figure 3-2 Best guess on Location of Weak Signature for Insertions

The weak signature at location M+1 does not match. A match occurs for the next signature at location 2M+1.

In both the cases the best guess is at M(block size) bytes before the location of the successful match.

3.2 Errors induced during conversion of insertions and deletions to substitution

3.2.1 Pseudo Errors

Errors introduced due to the inability of our system to identify the correct match for every block, in the process of converting insertions and deletions to substitution will be referred to as pseudo errors. Major causes, for the inability to recognize a block follow

- The weak signature by itself could be a new data that has been inserted, deleted or substituted.
- Due to repetitive patterns in the file, the weak signature could be wrongly matched.

We refer to errors introduced due to the above-specified causes as block mismatch Pseudo errors as not even a single character within the block could be matched correctly. Block mismatch pseudo errors occur in substitutions, insertions and deletions as well.

Often due to *insertions* we are unable to recognize the match for a block **completely** as the characters that follow the insertion are shifted. We refer to such errors as trailing pseudo errors. Figure 3.3 describes the cause for such errors. Upon identification of a matching weak signature for a received weak signature, we conclude that the blocks are likely to be the same, if the weak signature of the preceding block is found at the expected offset. Consider the case of the updated block corresponding to the matched block containing newly inserted data of size smaller than the block size. It would result in a shift of part of the block that follows, hence introducing pseudo errors. Figure 3.3 illustrates the problem.

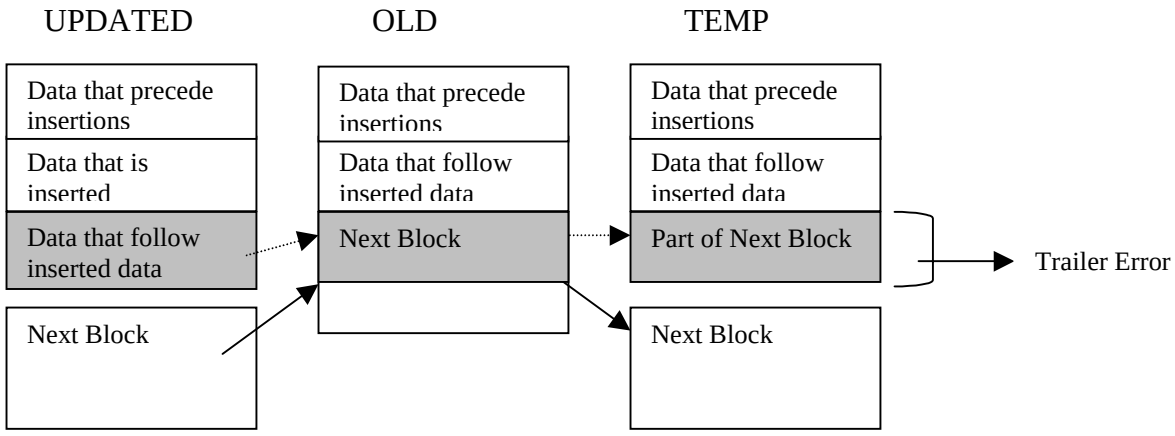


Figure 3-3 Trailer errors

In figure 3-3, as you may have noted part of next block is in place of data that follow inserted block. Although the data following the newly inserted data is available in the outdated data, we are unable to recognize it. Such errors are introduced only in case of insertions.

The best guess in such cases would be to construct a matching block with the first half of the block beginning at the end of previous block and the other half beginning at M/2+1 bytes prior to beginning of location of next block.

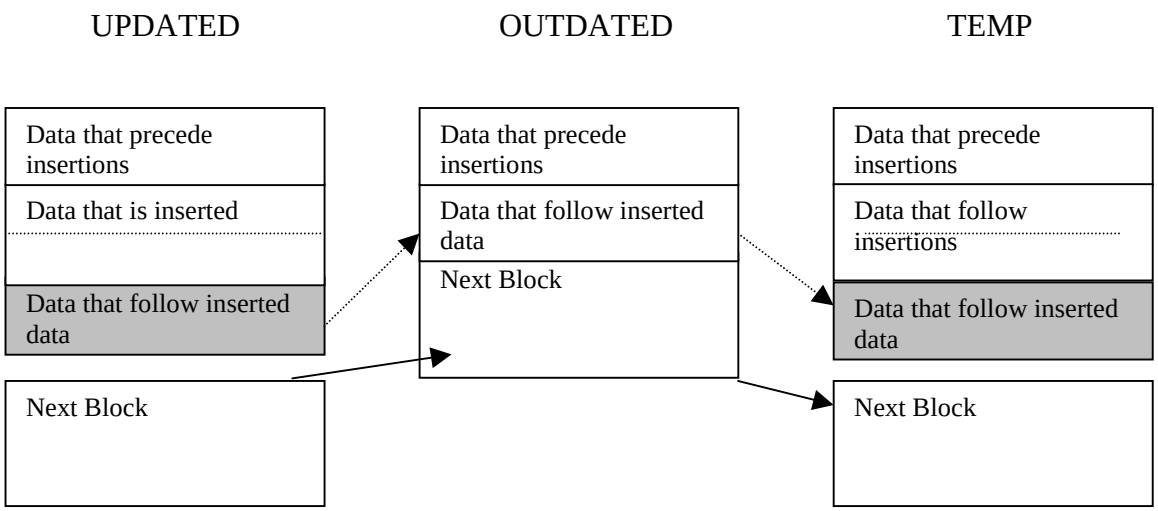


Figure 3-4 Adapting to Trailer Errors

3.2.2 Examples

The following examples illustrate the issues of interest involved in the process of converting insertions and deletions to substitutions. The examples are meant to provide an understanding of the algorithm and not intended to discuss how our algorithm reacts to all possible changes. We will skip step 1.2 of version 1on the receiver side for simplicity in some examples. In the following examples let us consider the blocks to be of size 5 and the weak signature to be of size 1. Fr simplicity we ignore the CRC component of the weak signature.

Consider a simple substitution of replacing X of outdated data by d in updated data.

UPDATED	**OUTDATED**	**TEMPORARY**
A b c **d** e	A b c **X** e	A b c **X** e
F g h i j	F g h i j	F g h i j
K l m n o	K l m n o	K l m n o

Illustration: Weak signatures sent are AFK. Starting with A, all the weak signatures are found at the expected locations. Hence the Temporary blocks are the same as outdated blocks.

Consider a simple substitution of replacing **A** of outdated data by **$** in updated data to see how the algorithm works if a weak signature is lost.

UPDATED	**OUTDATED**	**TEMPORARY**
$ b c d e	**A** b c x e	**A** b c x e
F g h i j	F g h i j	F g h i j
K l m n o	K l m n o	K l m n o

Illustration: Weak signature $ not found at expected location. We search through the blocks to find $. As we do not have a match, we proceed to the next weak signature F. We search for F. Once we find F with K existing at the expected offset we copy the blocks beginning with F and K as they are into the temporary file. *We make a guess as discussed earlier on the location($) as location(F) – block_size*.

Consider a simple substitution of replacing c of outdated data by A in updated data and A of outdated data by $ in updated data to see how the algorithm ignores a wrong match of weak signature.

UPDATED	OUTDATED	TEMPORARY
$ b **A** d e	**A** b **c** d e	**A** b **c** d e
F g h i j	F g h i j	F g h i j
K l m n o	K l m n o	K l m n o

Illustration: Weak signature $ not found at expected location. We search through the blocks to find $. Though we find a match, F is not found at the expected location viz. location(A)+5. Hence we proceed to the next weak signature F. We search for F. Once we find F with K existing at the expected offset we copy the blocks beginning with F and K as they are into the temporary file. *We make a guess as discussed earlier on the location($) as location(F) – block_size*.

Consider a simple insertion of char X in front of b.

UPDATED	**OUTDATED**	**TEMPORARY**
A b c d e	A b c d e	A b c d e
F g h i j	F g h i j	F g h i j
K **X** l m n	K l m n o	K **l m** m n
o p q r s	p q r s	o p q r s

Illustration: Weak signature o is not found at expected location. We search through the blocks to find o. We recover the block beginning with o completely. Our best guess on the block beginning with K would be half the block from the end of previous block and half the block prior to the beginning of previous block. We reduce the trailer errors by this approach but not completely. As you would have noticed the character l could not be recovered.

Now that we have converted insertions and deletions to substitutions we need to propagate the changes (substitution) using Reed-Solomon codes. This could be done as said earlier in version 1 of the algorithm. The steps are repeated below for convenience.

OWNER OF UPDATED FILE		*OWNER OF OUDATED FILE*
1: Interleave updated file		Interleave Outdated File
2: Encode interleaved blocks		Do nothing
3: Send parity Info	⟶	Get parity Info
4:		Decode Blocks
5:		De-interleave Blocks
6:		Write to file

Before proceeding further we would like to emphasize that our objective is bandwidth efficient propagation of updates *without compromising the generality* if the cost paid in terms of bandwidth is less. By generality we refer to the ability of the information transferred to propagate the change across multiple versions of the outdated data. We would also avoid back and forth communication to make the design of network protocols simple and efficient.

3.3 Need for interleaving

In order to understand the need for interleaving, lets first consider a solution without interleaving Consider a block of 30 characters split into blocks of 10 characters as shown.

```
A B C D E F G H I J
a b c d e f g h I j
1 2 3 4 5 6 7 8 9 0
```

Lets say for instance, that character “F” in block 1 is replaced by character “X”.
The updated block looks like

```
A B C D E X G H I J
a b c d e f g h I j
1 2 3 4 5 6 7 8 9 0
```

All we need to do is send the parity information required for correcting the one-byte error for the first block of 10 characters alone.

However there could be many versions of the file with a one-byte change but the location may differ.

Example

```
A B C D E F G H I J
A b c d e x g h I j
1 2 3 4 5 6 7 8 9 0
```

The parity information that was sent earlier can only correct changes in block one. Thus in order to preserve the generality of parity information, we send parity information for correcting each of these blocks.

Now consider a change of 6 characters occurring at a stretch within block 1.

```
A B C ! * # $ % & J
A b c d e f g h I j
1 2 3 4 5 6 7 8 9 0
```

As a cost for generality of parity information, we need to send parity information required for correcting 6 byte errors for each of the block i.e. 36 bytes (more than the size of block for a change of 5 bytes!!!).

The solution to this problem is interleaving. We would refer to the blocks obtained after interleaving as interleaved block.

Interleaving refers to the process of permuting the symbols to distribute the change. We use Matrix interleaver, a block interleaver, to permute the data. Each ten-byte block forms a row. Each column forms the interleaved block. Encoding will be done on the interleaved data.

Consider the following block with five errors as earlier.

```
A B C ! * # $ % & J
a b c d e f g h I j
1 2 3 4 5 6 7 8 9 0
```

The interleaved blocks would be

{Aa1, Bb2, Cc3, **!**d4, *****e5, #f6, **$**g7, **%**h8, **&**I9, Jj0 }

Some of the interleaved blocks have an error of 1 byte, while the others will be error free. Hence we need to send the parity information required to correct one byte, for each of the interleaved block. That is, 20 bytes. We would be transferring 20 bytes in place of 12 bytes to achieve the generality.

3.4 Imbalance in Interleaving

The term could be best explained by an example. Consider the following block to be in the sender as well as the receiver. Lets say that the interleaving offset is 5.

```
A B C D E
M N O P Q
V W X Y Z
```

Upon interleaving we get the following interleaved packets.

AMV, BNW, COX, DPY, EQZ

Let us consider two simple substitutions, one substitution with characters V, W and X being replaced by v, w and x(note that they differ by case)and other substitution with characters O,P and Q being replaced by o, p and q.

The resultant block would be

A B C D E
M N o p q
v w x Y Z

Upon interleaving, we get the following interleaved packets

Interleaved packet before substitution: `AMV, BNW, COX, DPY, EQZ`

Interleaved packets after substitution: `AMv, BNw, Cox, DpY, EqZ`

Errors introduced: 1 1 2 1 1

In order to propagate the change, we need to send the parity information (twice the error), required for correcting the maximum number of errors viz. 2. Thus we need to send 2*2 bytes for all the interleaved blocks, even when the number of errors is 1 in most of the interleaved blocks. We refer to our inability to scatter the changes in such a way that, all the interleaved blocks have the same number of errors as imbalance in interleaving.

Choosing the interleaving offset plays a major role with imbalance in interleaving. For example an interleaving offset of 3 will result in a 2 byte error in all the interleaved blocks. However with varying size and location of modification the best option would be to choose the smallest of the most commonly occurring bursts.

However, when *size of the object > interleaving_offset*2* ^{n} where n is the symbol size, using the estimate of total change to determine the change in each of the source blocks becomes a non-trivial task. Also, we lose generality, as the parity information we send becomes specific to the source block. Hence we must choose the interleaving offset so as to preserve the generality of the parity information. In our implementation we used a source symbol of size 8 and hence faced the problem of losing the generality of parity information due to break up of data into multiple source blocks. Thus a source symbol of size 16-bit would be a very good choice. The above solution of reducing the interleaving depth has been realized and a implementation of a re-programmable Reed-Solomon Decoder over GF(2^{16}) on a digital signal processor was completed by Christof Paar and Olaf Hooijen [Paar94]. However the decoding time should be significantly high for 16-bit Reed Solomon codes [Byers98].

Tornado codes [Luby97, Luby98] prove to be much faster than standard erasure codes for large symbol size [Byers98]. However they require negligibly more information than Reed-Solomon codes.

Let us consider the following algorithm for correcting substitution errors.

SENDER:

1. Split the file into small blocks of fixed size same as or smaller than that of interleaving depth.
2. For each block calculate CRC.
3. Send the CRC to the receiver.
4. Send the parity information to correct the erasures obtained by interleaving and encoding.

RECEIVER:

1. Split the file into small blocks of fixed size same as or smaller than that of interleaving depth.
2. For each block calculate CRC.
3. Receive the CRC sent by the sender.
4. Compare the received CRC with the corresponding calculated CRC.
5. Each unmatched CRC correspond to location of erasure of size same as that of block size.
6. Receive the parity information.
7. With the interleaved blocks and parity information correct the erasures.
8. De- interleave the file to construct the updated file.

As you may have noticed we do not determine the exact location of the erasure, as the amount of information and computation time needed to locate the exact location is significantly high. Also the amount of information needed to correct the erasure is the same as the number of erasures. Hence the new approach would also be relatively bandwidth efficient for changes greater than 4 percent of file size if the block size chosen were small. However for smaller changes our earlier approach would be better as the amount of data required to locate the erasure is 2 percent of the file size for a block size of 100 if we use a 16-bit CRC. As you may have noticed if the CRCs of the blocks match due to random collision decoding will fail. For CRC collision factor refer section that provides background information.

3.5 Parameters and Consequences

3.5.1 Weak Signature

Clue:

Smaller the clue, larger the number of mismatches and hence more the CRC values computed.

Illustration:

Assume a smaller clue of say 3 bytes on a typical text file. The number of occurrences of triplets like “The“, ”if ”and ”of “ is significantly more. When these triplets are chosen to be the weak signature, the number of matches and hence the number of CRC computation done will be high.

Larger the clue, more is the data transferred.

CRC:

If the number of consecutive bytes involved in the CRC computation is smaller, larger is the possibility of a mismatch. Hence the number of consecutive bytes involved in the CRC computation is dependent on the repetitive patterns in the object to be transferred.

If the block size involved in the CRC computation is larger, not all modified blocks could be recovered.

Illustration:

Consider the extreme case of the whole block being involved in the CRC computation. The block will not be identified, even if one of the byte in the whole block of a few thousands/hundreds of bytes.

3.5.2 Interleaving depth

Larger the interleaving depth more is the imbalance in interleaving and hence more is the parity information we send.

However when the source block (*interleaving depth*(255-2*T)*) is smaller than the object to be transferred, imbalance in interleaving increases.

3.5.3 Block Size

Larger the block size, greater will be the pseudo errors (errors induced in the process of converting insertions and deletions to substitutions).

Smaller the block more will be the data transferred for weak signature. Hence size of the clue must be in proportion to the block size in addition to the above-mentioned constraints.

3.6 Handling inability of the decoder to detect the error

Throughout our description of the algorithm we have assumed that the decoder will either correct the error or report that it had failed to correct the error. However, its not always true. There is a possibility that the Reed Solomon decoder will neither correct the error nor report the failure. Henceforth, the file we obtained may not be the same as the original version. A mechanism is needed to verify that the file obtained after processing is the same as the updated file. We use a 128-bit MD4 checksum to verify that the two files are the same. The node with the updated file should send the MD4 value of the updated file along with the parity information to the node with the outdated file. The received MD4 value must be matched with the MD4 value of the generated file by the node with the outdated file. If the received MD4 value and the generated MD4 value matches the update has been completed successfully. If the MD4 values are different, then a re-transmission request must be sent and the whole file is transferred.

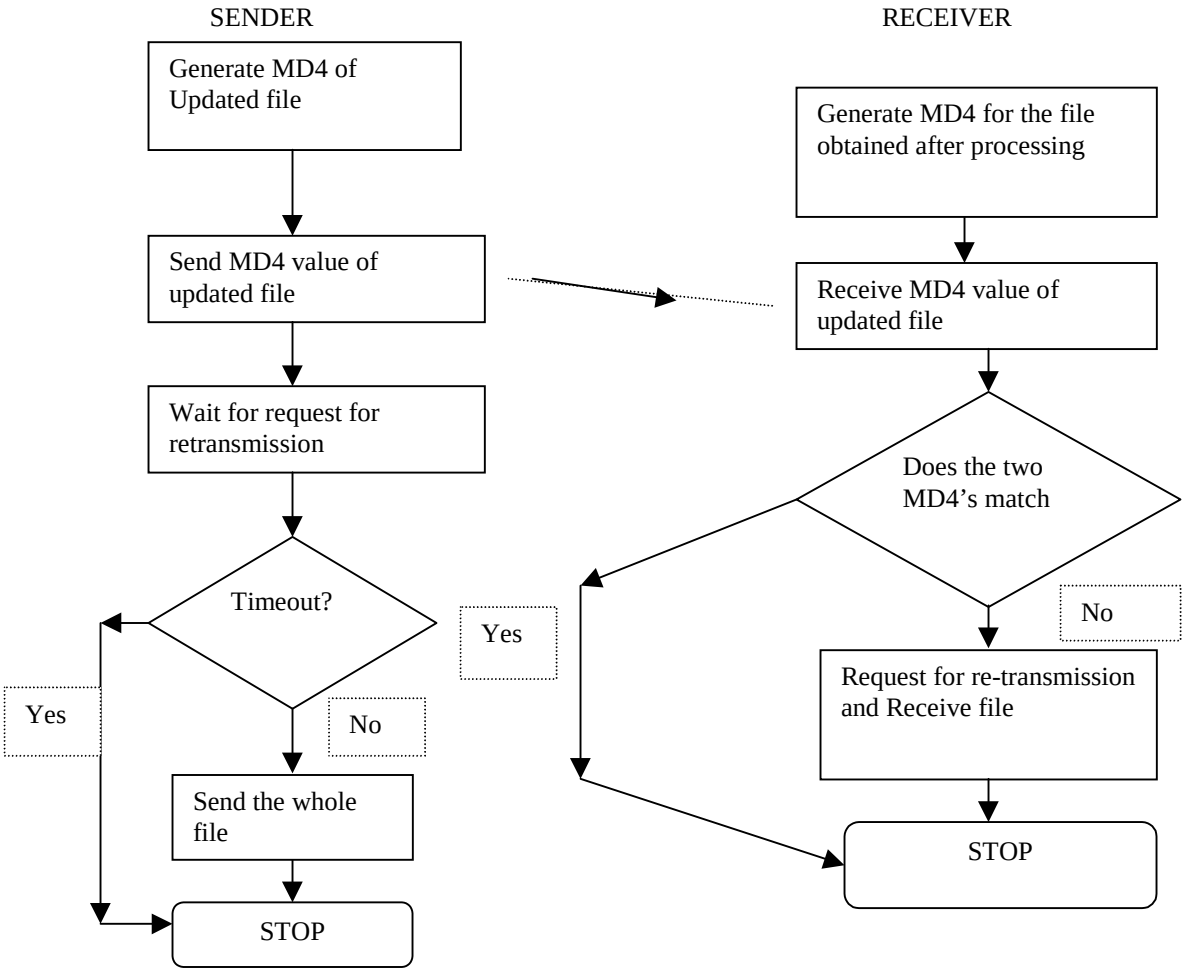


Figure 3-5 Detecting inability of decoder to detect errors

3.7 Specific data patterns of object that makes the approach inadaptable

1. Objects where a byte sequence occurs very frequently:

   In such cases the possibility of such byte sequence being a weak signature in very high and the probability of mismatch of weak signature is very high as well. Hence more pseudo errors are induced during the conversion of insertions and deletions to substitution.

2. Highly Scattered modification:

   For a modification of given size, the possibility of a weak signature being part of the modification increases with the degree of scatter. For a very high scatter of modification with burst size in the range of 1-50, the approach is unacceptable due to excessive damaged weak signature.

3.8 Improvements

In the above description of our final version of the algorithm we make use of CRC to detect the location of erasure. If the CRCs generated are not the same as expected(received) CRCs we mark the location as a location of erasure.

Consider the following block.

```
A B C D E F G H I J
K L M N O P Q R S T
0 1 2 3 4 5 6 7 8 9
```

Let the character inserted be “X”. The resultant most recent block will be as follows.

```
A B C D E F G H I J
K X L M N O P Q R S T
0 1 2 3 4 5 6 7 8 9
```

The weak signature transferred will be AK0. Match will be found for all the weak signatures. Hence the resultant block obtained after substitution will be as follows.

```
A B C D E F G H I J
K L M N O ? P Q R S T
0 1 2 3 4 5 6 7 8 9
```

Now let the interleaving depth be 2 characters for our example. CRC will be sent for each of the 2 characters.

The CRCs received will be for the blocks
AB, CD, EF, GH, IJ, KX, LM, NO, PQ, RS, T0,12, 34, 56, 78, 9.

The CRCs generated will be for the blocks
AB, CD, EF, GH, IJ, KL, MN, O?, PQ, RS, T0, 12, 34, 56, 78, 9 .

Note that the characters *L,M,N* and *O* are not recoverable. However if we have used the RSYNC(an update algorithm that we will discuss later) approach to compute the CRC for every byte offset viz. KL, LM, MN, NO, O?, ?P, PQ, QR, RS, ST and found the matching CRCs of received CRCs by searching through the above CRCs (those generated for every byte offset) we would have recovered L,M,N,O as CRCs when generated for every byte offset will include CRCs for blocks LM and NO.

As you may have realized CRC computation for every block byte offset is very expensive. However rolling check sums similar to those used in RSYNC that could be computed cheaply at every byte offset would solve the problem. Richard Taylor [Taylor] has made a comparative study of various fast signatures with the rolling property.

In this chapter, we explained the various versions of our algorithm and discussed the need for revision. We discussed about pseudo errors and imbalance in interleaving. We explained the algorithm with examples and have suggested improvements that we believe would further reduce the required bandwidth.

4 Simulation Study

4.1 Input modeling

To model our input we had to consider what we wanted to investigate. We were interested in investigating an estimate of the amount of data we need to transfer for a given amount of change in data. In other words we needed to determine the extent to which factors like imbalance in interleaving and pseudo errors influence the deviation of our throughput from the ideal case.

Stochastic modeling of the input was done, by identifying the typical characteristics of a modification. Basically the components involved in modeling the input were the burst size and the distance between subsequent bursts. Let us first consider the burst size. Typically a change in an object has some bursts sizes which are more likely and some bursts which are more unlikely. In a replicated dynamic HTML document of size in kilobytes the probability of the whole or most part of the file being modified at a single burst is very small. Even modifying half of the file at a single burst is unlikely but it is not as unlikely as modifying the whole file at a burst. However the probability of 10 bytes – 1000 bytes is more common. As another scenario, consider a replicated web server. Changing the whole file system in the web server is very unlikely. Changing a part of the file system is likely. Replacing an existing file with a completely new file or adding a new file is more frequent. Hence a few hundred bytes to a few ten thousand bytes are more common. Thus we need to choose a probability distribution with the PDF curves as shown in Figure 4.1.

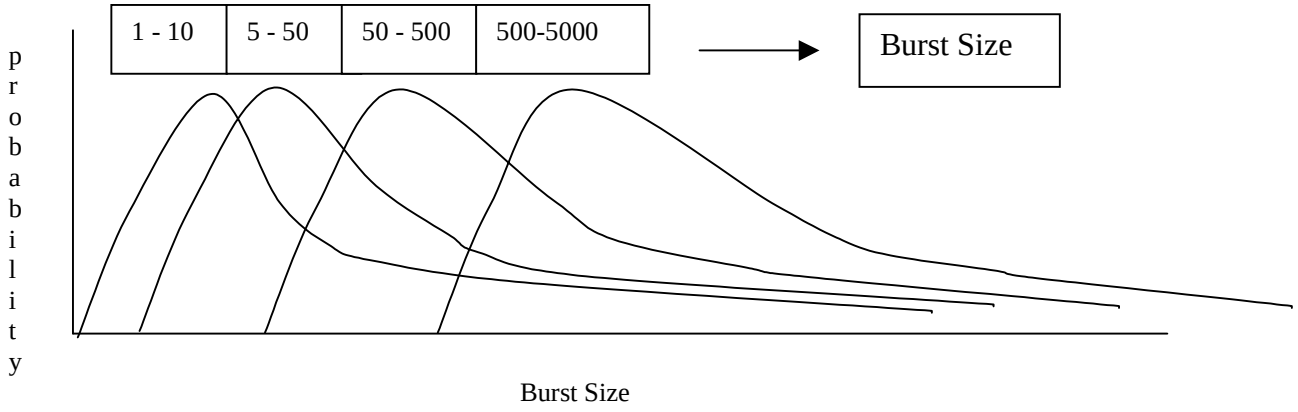


Figure 4-1 Burst Size distribution

We chose a distribution from the family of gamma distribution namely Chi-square distribution. Now, we are left with many Chi-square curves with different degrees of freedom. However a burst of size m or a burst size of *n*interleaving_offset + m* will result in the same imbalance in interleaving, for

any given integer m and n. Hence we can estimate the excessive data to be sent due to imbalance in interleaving with any of these values. After trying different degrees of freedom, we picked the degree of freedom(DOF) to be 3 and multiplied the random number generated by 100 to get the most likely burst sizes in the range of 200-500 bytes for an interleaving offset of 500 i.e. *burst size* < *interleaving_offset*. We chose such a parameter, as it would give us the worst case effect of both Macro pseudo errors and Micro Pseudo errors.

Due to lack of any identifiable characteristics in the gap between the bursts, we chose uniform distribution, as it would keep the modeling simple. The distance between the bursts is modeled using uniform distribution, as the probability of every character being the start of burst is equal. While this may not be the best choice, its simplicity and the lack of any significant characteristics about the beginning location of the burst makes it a better choice.

The parameters we chose are specific to the size of the object we used for testing and the percentage of modification. We modeled the two extreme characteristics in which the change is scattered throughout the file and the characteristic in which the change is scattered but very close to one another. The extent of closeness increases as the percentage of change increases in the earlier case.

4.2 Simulation

We determine the value of T required to propagate the change for the document by comparing the two files (the original file and the file obtained after converting insertions and deletions to substitution) on a byte-by-byte basis. We calculate the amount of data transferred from our value of T, the file size and the block size. We did our estimation for the case of insertions and substitutions only as deletions require only fewer amounts of data to be transferred.

4.2.1 Test Data

Throughout the analysis we have used a text file of size 1.3 MB with the number of repetitive patterns significantly greater than a typical text file. The weak signature is composed of 10 bytes of clue and 2 bytes of the CRC. However, we believe that the 8-bit CRC would be sufficient as well. For a detailed discussion refer to the background information on CRC. Our initial interleaving offset was 1000 bytes. As we would see in the later sections we use an interleaving depth of 200 bytes for the case of highly scattered substitutions and insertions to prove our arguments.

4.2.2 Test Results

In the rest of this chapter we will be illustrating the bandwidth efficiency in terms of the graphs and strengthen our argument with additional test results.

We will be looking at plots between efficiency factor(bytes changed/ parity information sent) and the bytes change in object in this section for the cases of substitutions that are closely scattered, insertions that are closely scattered, substitutions that are highly scattered and insertions that are highly scattered in the order said and the revisions done on the algorithm. *The graphs obtained are based on the assumption that the optimum value of T can be derived from the estimated number of bytes changed.*

Figure 4-2 is the efficiency factor graph for substitutions with low scatter(substitutions are closely spaced and are restricted to first quarter of the test file). The test was based on the algorithm with arbitrary block substitution in case of a mismatch.

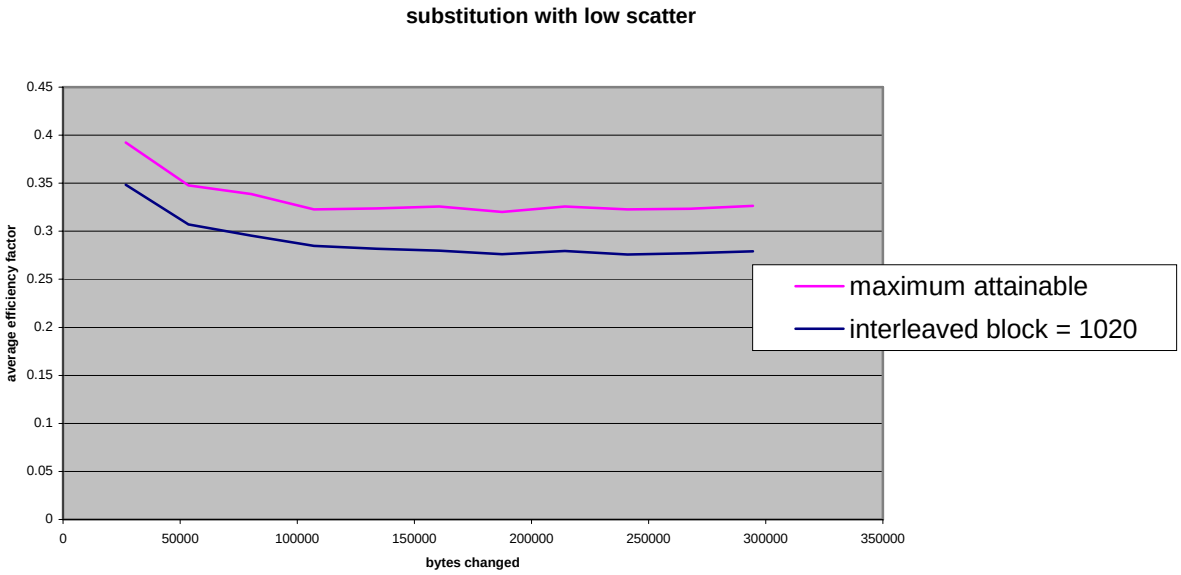


Figure 4-2 Plot of Average Efficiency Factor for Substitutions with Low Scatter

As we need to transfer 2 bytes of data in order to correct a single byte, the maximum attainable efficiency factor is .5 (1/2) i.e. for a change of n bytes we need to transfer at least 2*n bytes in order to propagate the change. However this ideal limit is not achievable as we induce pseudo modifications in the process of converting insertions and deletions to substitutions. Hence our maximum attainable average efficiency factor is in the range of .35 to .325. Therefore we need to transfer on an average data of size about 3 times the change. However this average efficiency factor is not achievable either due to

the imbalance in interleaving. For a block size of 1020 bytes, the average efficiency factor is in the range of .3 – .275. Hence we send data of about 4 times the change to propagate the change. However the insertions have better efficiency factor, as the loss of headers is not possible

Figure 4-3 is a efficiency factor graph for insertions with low scatter. The test was performed with best guess strategy as described earlier. Note the increase in maximum attainable average efficiency factor.

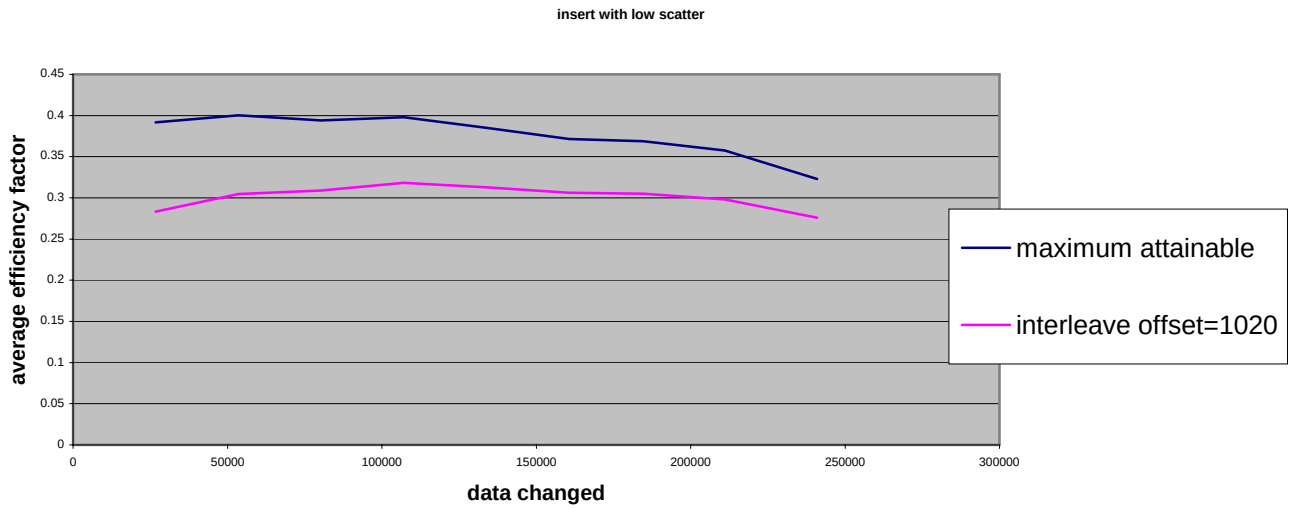


Figure 4-3 Plot of Average Efficiency Factor for Insertions with Low Scatter

As we know, we need to transfer 2 bytes of data in order to correct a single byte and hence the maximum attainable efficiency factor is .5 (1/2) i.e. for a change of n bytes we need to transfer at least 2*n bytes in order to propagate the change. However this ideal limit is not achievable as we induce pseudo modifications in the process of converting insertions and deletions to substitutions. Hence our maximum attainable average efficiency factor is in the range of .4 to .35. The high variation in the average efficiency factor relative to substitutions of same size and scatter (earlier graph) is due to increase in trailer errors caused by insertions. The trailer error also accounts for the gradual degradation of average efficiency factor with data changed. Hence we need to transfer on an average data of size about 3 times the change. However this average efficiency factor is not achievable either due to the imbalance in interleaving. For a block size of 1020 bytes, the average efficiency factor is in the range of .325 – .3. Hence we send data of about 4 times the change to propagate the change. We defer the

discussion of increase in trailer errors till we discuss the graph with highly scattered substitutions and insertions. Consider Figure 4-4, efficiency factor graph for highly scattered substitutions(the changes are scattered in blocks of a few hundred bytes across the whole file).

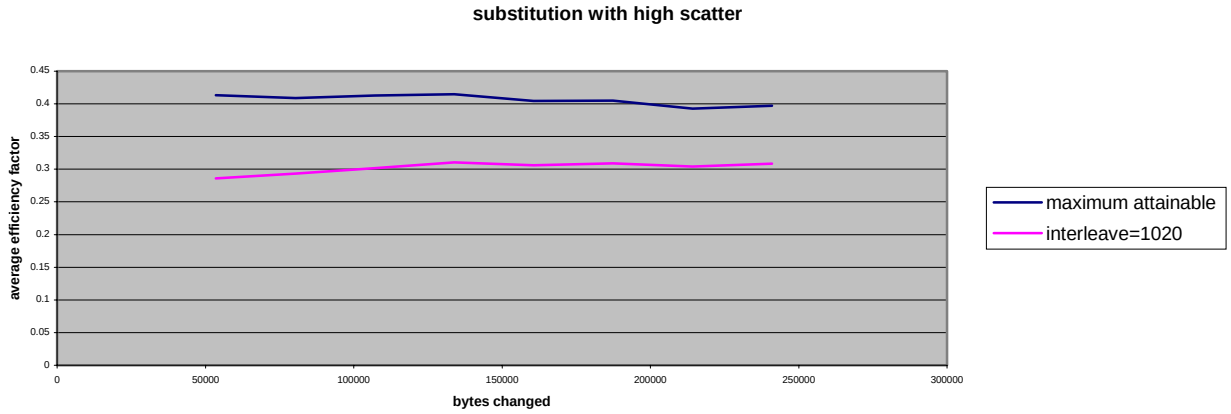


Figure 4-4 Plot of Average Efficiency Factor for Substitutions with High Scatter

Comparison with low scattered insertions

While the decrease in maximum attainable average efficiency factor is insignificant, the average efficiency factor for interleaving offset of 1020 has significantly drifted down due to increase in imbalance in errors. Hence interleaving depth of 205 was used in the simulation in combination with a symbol size of 16-bits. The plot obtained is close to the maximum attainable average efficiency factor as shown in the Figure 4.5. Hence we send almost 3 times the change to propagate the substitutions.

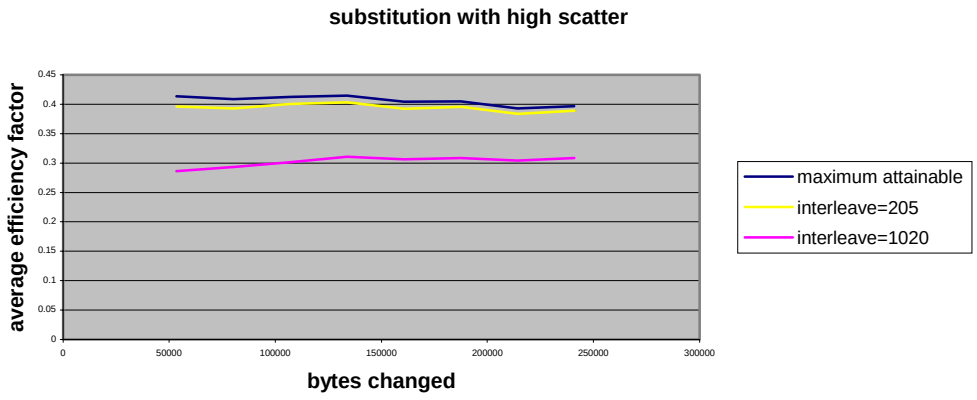


Figure 4-5 Plot of Average Efficiency Factor for Substitutions with High Scatter

Consider the efficiency factor graph for insertions with high scatter.

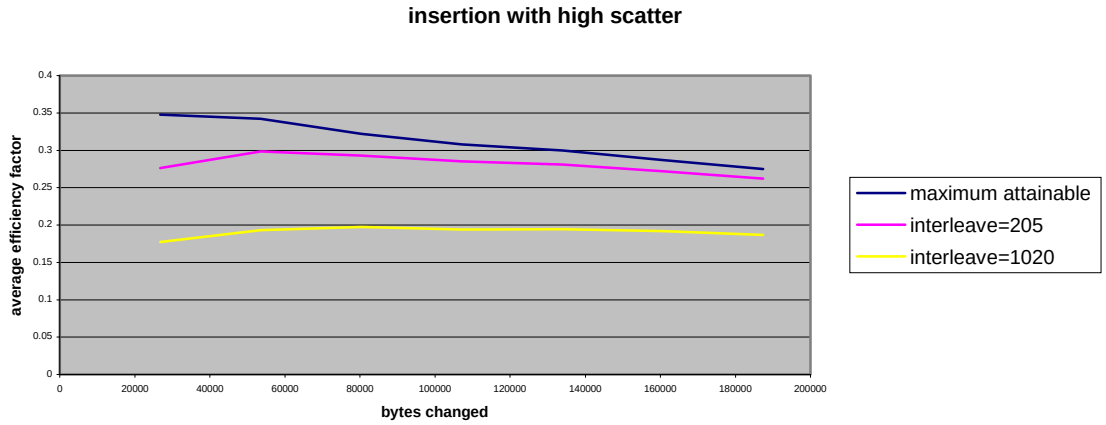


Figure 4-6 Plot of Average Efficiency Factor for Insertions with High Scatter

As you may have noted, the maximum attainable average efficiency factor is very high due to the increase in trailer error and hence we need to send about 4 times the change in data. As said earlier the lesser the interleaving depth lesser is the imbalance due to interleaving and hence lesser is the data transferred.

Considering the high pseudo errors introduced, the fact that for correcting every byte we need to send two bytes of data and that the computation cost increases with increase in the size of symbol, we proposed an algorithm based on erasure correction. Figure 4-7 is the result of simulation of such an approach for highly scattered substitutions and proves to be effective. We believe that similar improvements will hold for insertions and deletions.

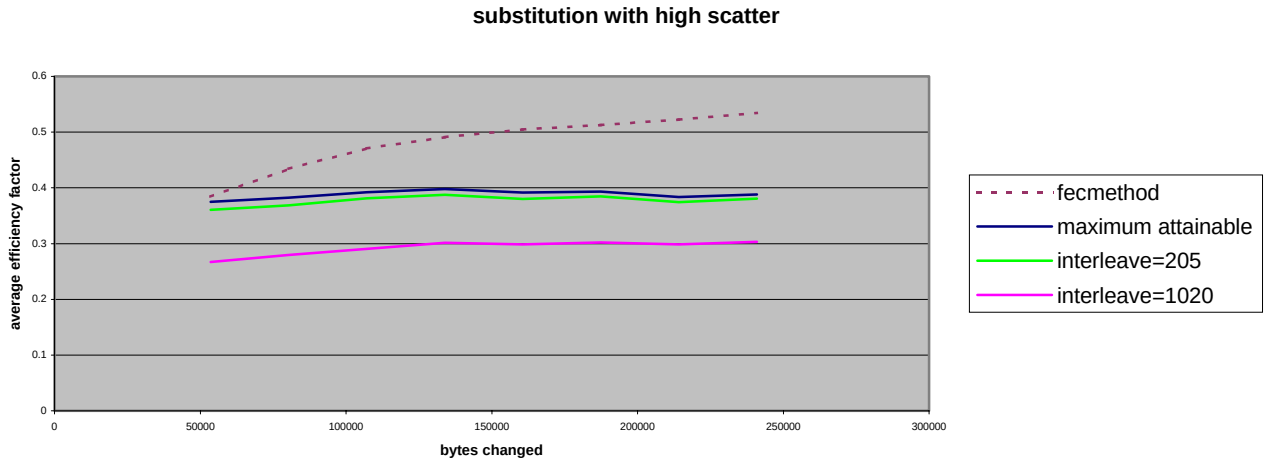


Figure 4-7 Plot of Average Efficiency Factor for Substitutions with High Scatter

This chapter rationalized our choices on probability distributions for modeling the burst size and location of burst. We analyze the efficiency factor graphs obtained for various cases and strengthen our argument on need for smaller interleaving depth and the reduced bandwidth requirement of erasure correction approach over error detection approach.

5 Relevant Work and Future Work

5.1 Relevant work

Simplest possible update propagation is the transfer of the updated object to the node with the outdated object. Typically, an update will change some part of the data and leave the rest unmodified. In our simplest approach of transferring the whole object in compressed or uncompressed format, the unmodified part of the data needs to be transferred as well. Update propagation techniques that exploit the availability of partial data could be broadly classified as

- Entropy based Algorithm
- Anti-Entropy based Algorithm

5.1.1 Entropy Based Update Propagation Algorithm

Typically, the node with the updated file determines difference between the updated file and the outdated file and encodes the changes as insertions, deletions and substitutions and transmits them across the network. Such encoding is referred as delta encoding. The node with the outdated file constructs the updated file using the outdated file and the delta encoding. Standard diff algorithms were used to determine the difference between the updated and outdated file. These diff algorithms required both the updated file and the outdated file to be available at the node with the updated file. Focus of these algorithms has been on finding the longest matching sequence. Version control software CVS and Distributed Operating systems Munin and Treadmark [Keleher96] use diff based update propagation of memory blocks. The difference computation between the outdated and updated object does not necessarily require a diff algorithm. The application that manipulates the object could record the difference as well, if the operations involved are only insertions, deletions and substitutions. One such proposed idea involves propagation of java object [Cohen00]. When multiple versions of the document exist among the replicas, management of replicas becomes a non-trivial task and the amount of memory occupied is more. When we are not interested in the various versions and their relations but just the recent version, maintaining multiple versions of the object is asking for trouble.

Fortunately, Mathematical work proving that the amount of data that need to be transferred, when the updated node does not have a copy of the outdated data is negligibly small, when a three-way

handshake communication is involved was done by Orlitsky, A 1991. RSYNC update algorithm, a computationally feasible practical implementation of the work was done by Andrew Tridgelle. The algorithm is widely used specifically in update propagation of mirrored sites. Our algorithm is much similar to that of RSYNC. Some of our design decisions were influenced by RSNC update algorithm. A brief comparison of our approach with RSYNC will be done in the following section. Our approach could be considered as entropy based update propagation algorithm with a generic coding of the difference so as to update not a specific version but many versions of the outdated object. However unlike the early diff algorithms that expected a copy of outdated and updated object at the same location, our focus will be in propagating the update with the minimum data transfer and not on obtaining the exact difference. In other words, there must be a balance between the amount of data transferred to determine the diff and the accuracy of the difference.

5.1.2 Anti-Entropy based update propagation algorithm

Another way of propagating updates is the anti-entropy approach that typically requires all the nodes with the replicas to be available throughout the process of collaboration. Such approaches are widely used in distributed collaborating systems and transaction based Database Systems. Operation based update propagation has been proposed for Mobile File Systems [Lee99] as well. With the increase in availability of computing power and demand for bandwidth, this approach seems rational for many applications. Changes are coded in the form of operations that need to be made on the object of interest in order to update the object. The codes representing the operations are propagated rather than the data. The operations are redone on all the replicas to obtain the updated data. They are very optimal in terms of bandwidth utilization for propagating changes. However they expect all the nodes to have the same capability in terms of operations that could be performed on the object. The update propagation may be done by the application making the change or by a specific daemon process. While the use of daemons generalize the update propagation mechanism (ie. Non-application specific), they do require the co-ordination of the applications that modify the objects of interest. However, unlike entropy algorithms they are not sensitive to the degree of scatter of modification.

5.1.3 RSYNC update algorithm

RSYNC [Tridgelle] update protocol is a popularly used update propagation protocol mainly used in synchronizing mirror sites. Currently a proposal by the author of rsync, to use rsync update

algorithm in HTTP in case of dynamic web pages is being made. The relevant material is available at http://rproxy.smaba.org.

The following description of RSYNC is picked from the rsync site http://rsync.samba.org on 02/27/00.

The rsync algorithm consists of the following steps:

1. Let Node2 splits the file B into a series of non-overlapping fixed-sized blocks of size S bytes. The last block may be shorter than S bytes.

2. For each of these blocks calculates two checksums: a weak ``rolling'' 32-bit checksum (a block signature with less computation cost that is not so reliable) and a strong 128-bit MD4 checksum (signature with high computation cost that is very reliable).

3. Node2 sends these checksums to Node1.

4. Node1 searches through A to find all blocks of length S bytes (at any offset, not just multiples of S) that have the same weak and strong checksum as one of the blocks of B. This can be done in a single pass very quickly using a special property of the rolling checksum described below.

5. Node1 sends Node2 a sequence of instructions for constructing a copy of A. Each instruction is either a reference to a block of B, or literal data. Literal data is sent only for those sections of A which did not match any of the blocks of B.

Comparison of approach and objective:

RSYNC could be considered as a dynamic Diff algorithm that does not require the updated and outdated copy of the object in order to propagate the update. RSYNC detects the changes and transmits the change. Such differences are specific to the two versions of the object involved in the computation. In places where multiple versions of the data exist, the same difference cannot be used to propagate the change across any two versions of the file. Hence ideally we require a coding technique that would generate encoded data that could reconstruct any version of the file. However the volume of such an encoded data will be more than the size of the file. Hence we chose to move from the ideal target of correcting any file to correcting many files. The number of versions we could correct depends on the amount of data we send.

Both the computation cost of RSYNC and the data sent by RSYNC are significantly less than our approach. We compromise on the computation cost and data sent on a per-transfer basis in order to achieve generality. The major advantages of such a generic coding follow.
The parity information can be cached and used for correcting other versions of the document unlike the difference produced by RSYNC and other diff algorithms
Can be used in multicast where all the receiver nodes may not have the same data.

While RSYNC does not pose any limit on the amount of modification, we are only interested in modifications that are less than 30% of the object as the amount of data transferred increases in proportion with amount of modification.

The purpose of weak checksum in RSYNC is to reduce computation cost. The purpose of our weak signature is to convert insertions and deletions to substitutions. Our weak signature is not dependent on the whole block but on the first n bytes of the block. Weak signature of RSYNC is dependent on the whole block. We deviated from RSYNC in order to provide graceful degradation in performance for scattered modifications. However due to the sensitivity of our weak signature to modifications occurring at the beginning of the block we could not achieve significant benefits and will propose other weak signature for future work that may alleviate the sensitivity to the modifications occurring at the beginning of the block.

5.1.4 Outline of Orlitsky's algorithm for handling restricted edit problem:

The protocol [Orlitsky91] is based on an (n,k) linear t-error correcting code with parity-check matrix H that is known a priori to both Px (node with updated object) and Py (node with outdated object) . X is the updated object and Y is the outdated object. The number of bits inserted is T.

1) Px transmits XH^T , the (n-k)-bit syndrome of X.
2) Py computes $XH^T – YH^T$, the syndrome of X-Y.
3) Py finds an n-bit sequence Z with Hamming weight<=t such that $ZH^T = (X-Y)H^T$ and decides that X is Z + Y.

Restrictions include T<<N and only insertions and deletions occur. Also insertions and deletions do not introduce new runs[5] or delete existing runs. The above algorithm has been analytically proven but has not been practically implemented. The above algorithm has been presented in this document, due to the similarity with our approach in the phase of correcting substitution errors or erasures.

5.2 Future Work

In this section we discuss the problems that need to be solved in order to achieve practicality and the problems that would make our approach adaptable to various applications.

5.2.1 Obtaining the parameter T of Reed-Solomon code

No matter how bandwidth efficient we make our approach, the estimate of the parameter T provided by the user has the final say on the effectiveness of the approach. If the user wants to be very cautious, he/she may provide an estimate that is much more than the needed. In an effort to obtain bandwidth efficiency if the user makes an estimate that is lower than the required a retransmission will be triggered. Also in many applications where manipulation of the object like pictures is involved, we cannot expect the users to provide an estimate, as they may not be aware of the internal representation of the object. Hence we need an estimator that would measure the amount of change and arrive at a value of T. We have suggested some approaches below that we believe to be worth trying. There is no best approach and the requirement of the application would be the deciding factor.

One of the valuable clues is the change in file size. The change in file size indicates a minimum change that has definitely occurred. The substitutions and the cancellation of the effect of insertion and deletion on the object size make the estimate unreliable. We believe that by making an additional two-way communication we can obtain an approximate estimate of the number of blocks modified using the weak checksum of RSYNC. The sender will have to send the weak checksum and the receiver will have to find the matching blocks. The number of unmatched blocks and the change of object size could be used to obtain a value of T. Due to back and forth communication, complexity is introduced in protocol design and more data needs to be transferred.

[5] A run in a string is a continuous sequence of zero's and one's

An alternative is to learn by failure. In such an approach we can make a guess on the parameter T of Reed-Solomon code and increase the value of T if re-transmission is triggered and decrease the value of T if retransmission is not triggered. Such a failure based learning works only if the frequency of updates is more.

An alternative is to run a daemon process and expect the application to provide an estimate of the change and keep track of changes of multiple versions. We believe that this approach will blend easily with the consistency mechanism. This estimate could be used to obtain a value of T.

As there are multiple versions of the document multiple estimate corresponding to each of the outdated version will be produced. Although by choosing the maximum modification we may be able to correct all the versions of the document, such a decision may not be appropriate always. The variance and the average of the estimates must be considered as decision factors in determining the value of T of Reed-Solomon code.

5.2.2 Concurrency and Communication protocol design

Although concurrency and communication protocol design are two separate problems the establishment of concurrency often poses restriction on the communication sequence. Hence the problems preferably must be dealt together. Careful design of the communication sequence to overlap decoding process and transfer of parity packets would introduce negligible delay due to communication. Among the values that need to be communicated are the updated file size to the node with the outdated file, the generator polynomial to be used for CRC and RS codes.

5.2.3 Adapting to scattered small insertions and deletions

Adapting to scattered changes of a few bytes is a relatively difficult problem to solve. We believe that we can adapt to scattered substitutions effectively by removing the process of converting insertions and deletions to substitutions. However, our approach will be inadaptable to scattered insertions and deletions in bursts of a few bytes to a few ten bytes. The three major problems that we face in propagating such modifications are the pseudo errors due to header loss, high trailer errors caused by scattered changes and the imbalance in interleaving. The first two problems could be removed without any effort only when our system does not have to deal with insertions and deletions.

The imbalance in interleaving could be tolerated by reducing the interleaving depth and adapting to large symbol size with tornado codes. Adapting to a new weak signature that is based on document finger printing techniques to identify matching blocks in the process of converting insertions and deletions to substitutions may alleviate the problem of pseudo errors due to header loss. However we will have to send more statistical information and deal with false alarms. Unfortunately, the costs of trailer errors dominate the cost of the rest of the problems.

Illustration of increase in induced trailer errors with highly scattered data: Consider a block of 10 characters. Let X denote the location of insertions.

Outdate block: A B C D E F G H I J

Updated block: A X B C D E F G X H I J

After identifying that the blocks match and assuming that the weak signature detects the beginning of next block, we pick a corresponding substitution block. The substitution block will be

Our Result block: A B C D E F G H I J

Ideal Result block: A X B C D E F G X H

As you may have noticed, scattered single-byte insertions *within the same block* make all the bytes in between the new insertions unrecoverable. We propose using RSYNC's weak signature (rolling checksum) in recovering the blocks that reside in between the scatter. Let us say we split the block into equal parts of size same as that of interleaving offset say 2 bytes for our example. We are interested in determining the beginning location of each of the five parts. By identifying the master block(the block composed of all the four parts) using the weak signature(document fingerprint techniques) we have identified the beginning of first part. Now our task is to identify the beginning location of the rest of the four parts. Let us send the rolling checksum for each of the four parts(BC,DE,FG,XH) and follow RSYNC's approach in determining the location of matching blocks. By using this information we can construct our object with substitution errors and lesser trailer errors. In our case the resultant block with character Y representing induced trailer errors will be as follows.

Our Result Block: A X B C D E F G Y Y

Ideal Result block: A X B C D E F G X H

However, we are not sure if the document fingerprinting technique scale well in identifying similar blocks of a few hundred bytes.

5.2.4 Reducing the retransmission frequency

We can follow the lines of multicast solutions [Rizzo, Nonnenmacher96, Rizzo97] that make use of FEC to adapt with links of varying packet loss rate. The technique is to generate redundant parity information (more than what is needed) to propagate the estimated change in number of bytes. However we transmit just the parity information required to propagate the modification rather than transmitting all the generated parity information. In case of receiving a re-transmission request, we could retransmit rest of the generated parity information. By this way the information we sent earlier will not be wasted in case of a decoder failure. Unfortunately the computation time spent in decoding cannot be recovered. However in order to adapt to such a technique, the source symbol size should be large enough to accommodate the whole object in a single source block for a small interleaving offset.

Our focus in this chapter has been on the related work by Tridgelle on RSYNC update algorithm approach and Orlitsky's solution to the limited edit problem. We also discussed the open problems viz. adapting to scattered changes and determining the value of T from user's estimate on the number of bytes modified.

6 Conclusion

The objective of our approach was to propagate the change across multiple versions of the document by one-way communication and to achieve bandwidth efficiency. We assumed a correct estimate of the amount of change in bytes from the user. We did not want to pose any restriction on the characteristics of modification.

Unfortunately we are unable to generalize our update algorithm to all possible modifications of the file. We recognize as stated earlier in the chapter on *Future Work* that our approach will not be efficient for modifications that are scattered in the form of a few bytes to a few ten bytes.

We believe that the algorithm has met the rest of our goals. In addition to the advantages mentioned by Alon Orlitsky on obtaining a one-way communication based update protocol we describe an outline of a distributed architecture that may reduce the typical memory requirement and provides a balance between the server's load and the bandwidth efficiency. Consider a distributed architecture with *write-at-primary-read-from-any* principle structured as shown in the Figure 6-1.

The clients will read from the replications while writing only to the primary copy. Hence we need to store only the parity information for correcting say X% of the object and the weak signature in the replicas instead of the whole object. We have to provide a mechanism that would request a read from the primary copy in case of a decoder failure. The value X can be determined on a trial and error basis by choosing an initial value and incrementing for every n read request arrived, as read requests indicate that a decoder failure has occurred. The value of X provides a balance between the server's load and the bandwidth efficiency (amount of parity information on the server).

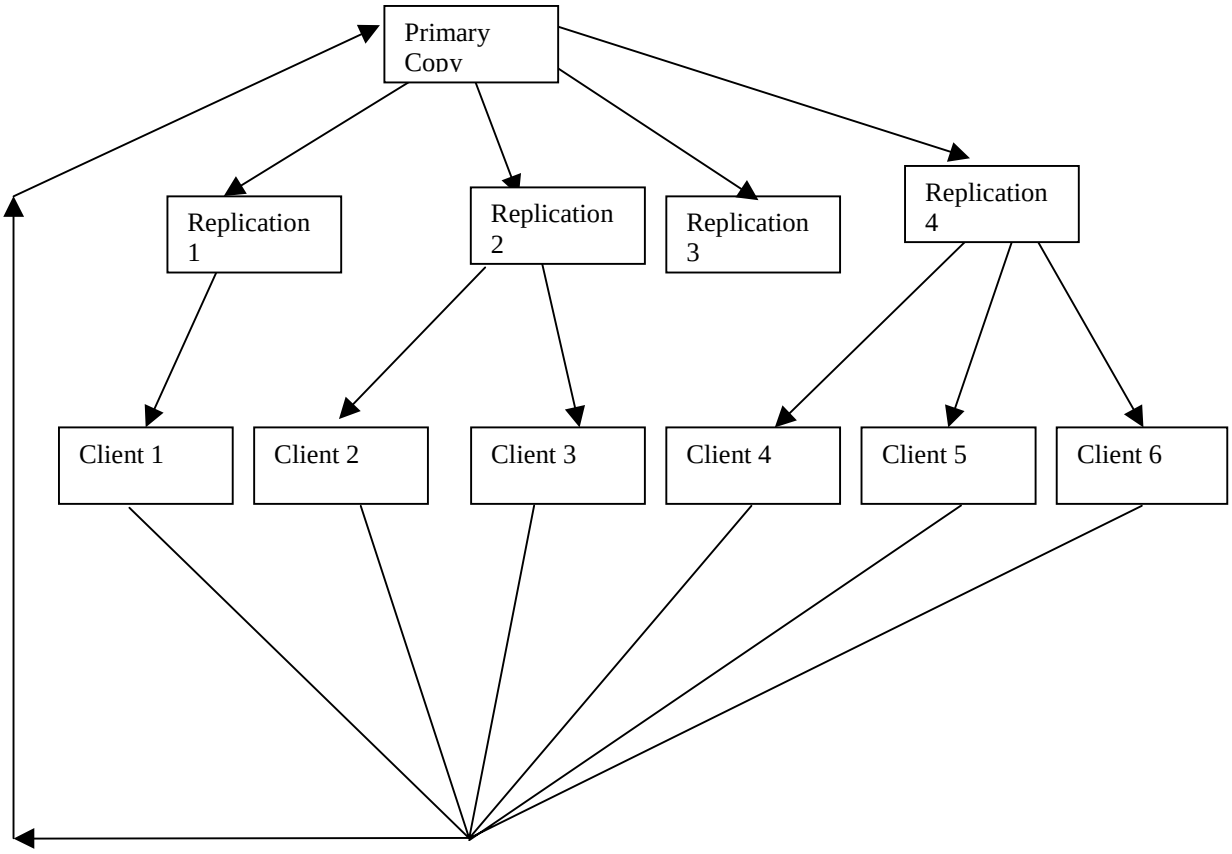


Figure 6-1 General Distributed Architecture

S. Williams [Williams] has proposed an extension to HTTP 1.1 to allow a WWW client (or cache server) to receive ***only*** the differences between a cached copy of a document and the current version. Sending the differences potentially reduces the latency to the client, the number of bytes sent and the number of packets sent by the server. Mogul, Douglas, Feldmann, Krishnamurthy have quantified the potential benefit for delta encoding and data compression [Mogul97].

The following are the throughput graphs obtained as a result of simulation. In the simulation we have assumed Reed-Solomon codes. Hence usage of tornado codes will require a little more data than shown here. The graphs are abtained based on the assumption that the optimum value of T can be derived from the users estimate on number of bytes changed.

Consider relatively larger insertions (larger than block size) occurring close to one another in a small portion of the object. Figure 6-2, throughput graphs for interleaving offsets of 1020 and 205 provide a measure of effectiveness of the approach.

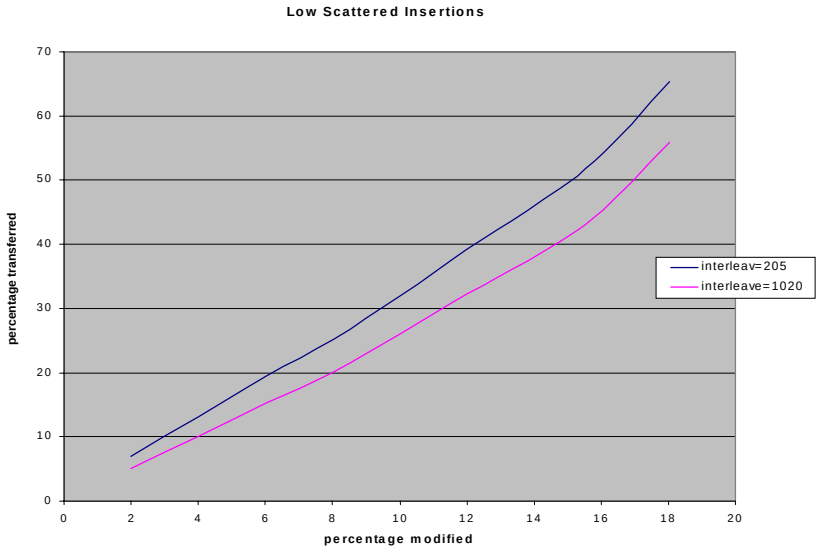


Figure 6-2 Throughput Graph for Low Scattered Large Insertions

As you may observe from the graph the throughput is a linear function of the amount of modification. We must send parity information of about 3.3 times the data modified for an interleaving offset of 1020 and about 2.5 times the data modified for an interleaving offset of 205 in order to propagate the change.

Consider relatively small insertions (smaller than block size) spread uniformly throughout the object. Figure 6-3, throughput graphs for interleaving offsets of 1020 and 205 provide a measure of effectiveness of the approach.

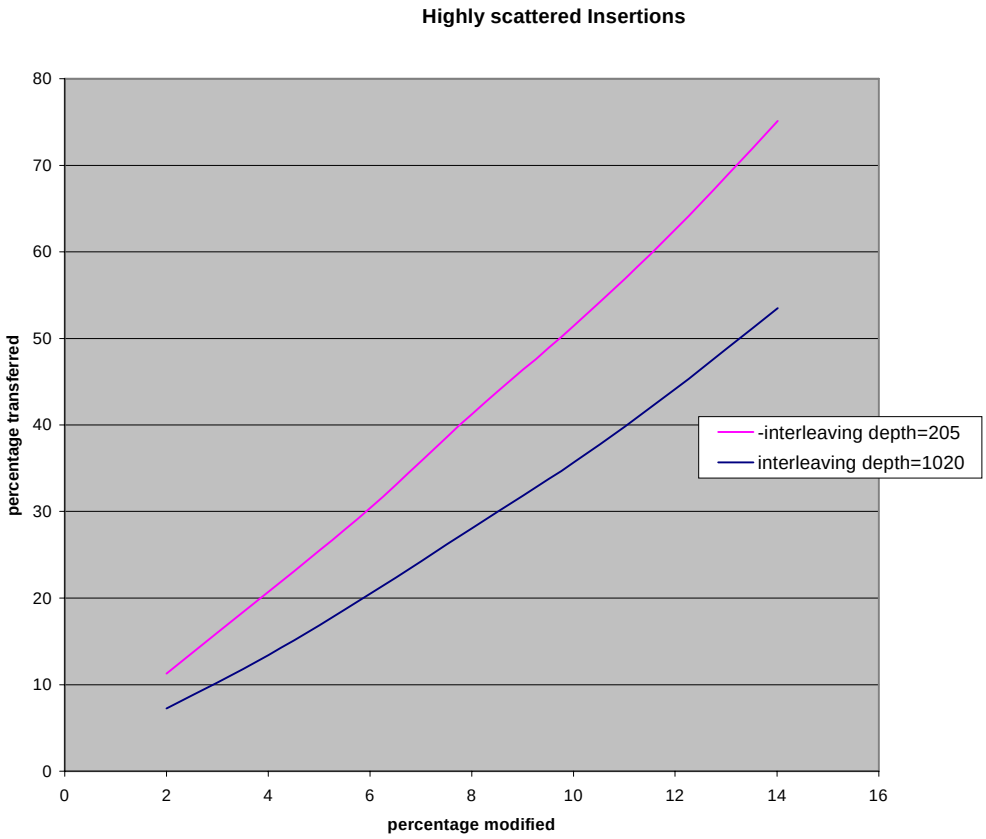


Figure 6-3 Throughput Curve for Highly Scattered Small insertions

We can infer from the graph that the throughput is a linear function of the amount of modification. Insertion of X bytes that is scattered in blocks of few hundred bytes across the whole file requires about 6*X bytes of parity information for an interleaving offset of 1020 and about 4*X bytes of parity information for an interleaving offset of 205 to propagate the change. The increase in the slope of the plots in comparison with the plots for large modifications is due to the high pseudo errors in the small modifications that are highly scattered.

Consider the highly scattered substitutions occurring in small bursts (smaller than block size). Figure 6-4 is the throughput graph for an interleaving offset of 1020 and 205.

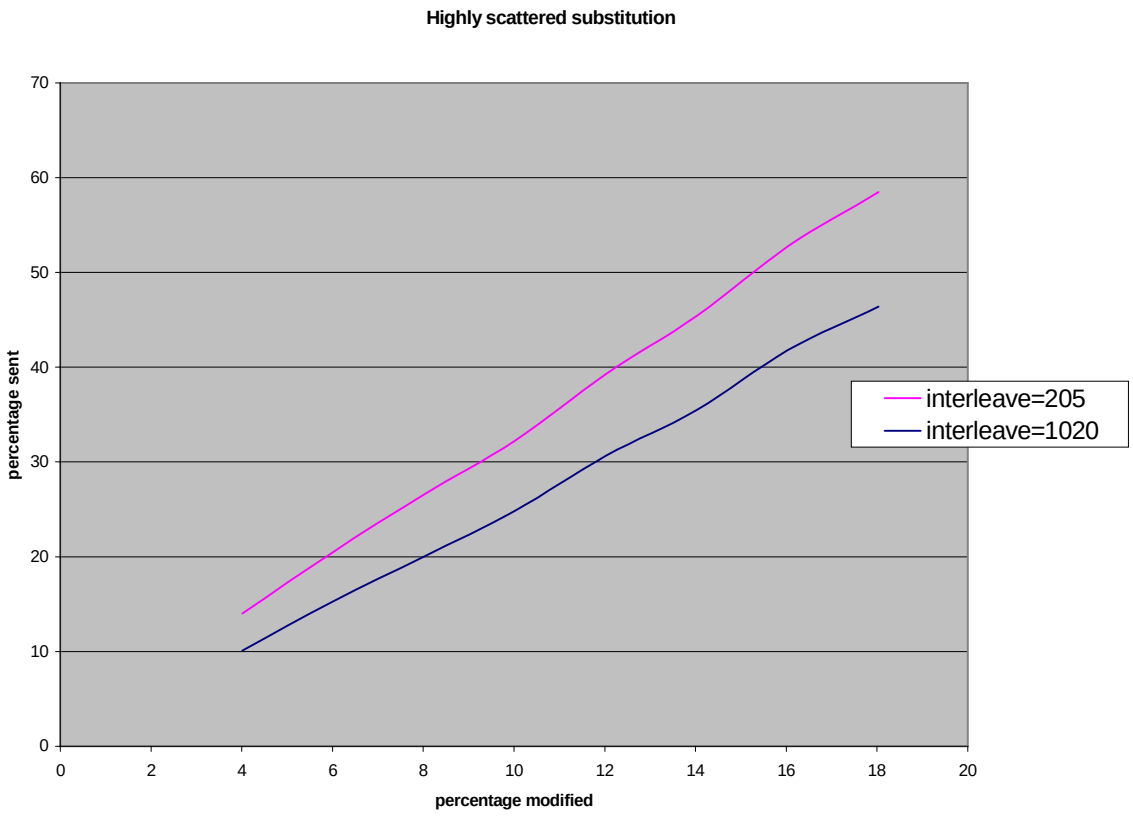


Figure 6-4 Throughput Curve for Highly Scattered Substitutions

Here again the graph is linear i.e. the throughput is a linear function of the amount of modification. In order to propagate X bytes of substitutions we must send 3.3*X bytes of parity information for an interleaving offset of 1020 and about 2.5*X bytes of parity information for an interleaving offset of 205. The decrease in the slope of the plots in comparison with the plots for smaller insertions is the effect of trailer errors in small insertions that are highly scattered.

We believe that the computation time involved in the decoding process of Reed-Solomon code is not tolerable for many applications. Considering the fact that Tornado codes are much faster than Reed Solomon Codes for large symbol size, we believe that the algorithm that involves approximate erasure location and erasure correction would be better than error detection and correction for modification that occur in bursts of a few hundred bytes as for every byte error we need to propagate due to the high parity information required for error detection.

Figure 6.5 is the throughput graph for substitutions with high scatter of our Erasure correction based algorithm and our earlier plots that use error detection and correction. In our simulation we have assumed the use of Reed-Solomon erasure correction. However, tornado codes would require negligible amount of extra parity information than Reed-Solomon codes.

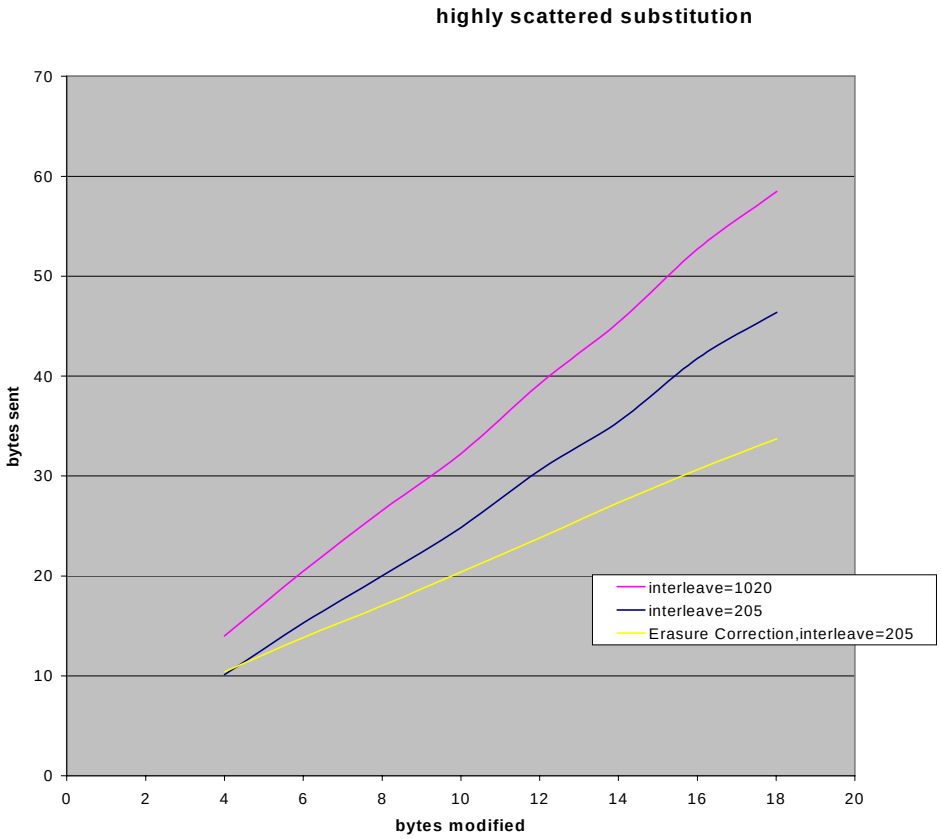


Figure 6-5 Throughput curve for highly scattered substitutions using Error and Erasures correction

From the graph we can infer that, the erasure correction approach proves to be more bandwidth efficient than Error detection and correction approach for modifications greater than 5%. The in-

efficiency in modification below 5% is due to the 2% CRC data sent for locating erasure. We would like to highlight the fact that the parity information required for erasure detection does not vary with the percentage of modification. This invariance in amount of erasure detection information accounts for the non-linear nature of the throughput graph of erasure correction method. The ratio of percentage of data sent to percentage of data modified varies from 1.83 – 2.5 decreasing with increase in modification.

7 APPENDIX

Appendix A-Implementation details

Implementation Details:

Implementation was done in C++. The implementation has been tested in binary and text file in Solaris platform. The input modeling was done in Java. The COLT numerical package was used in the generation of random numbers.

The description of the package, the API used and the algorithms used in generation of random numbers are available at
http://tilde-hoschek.home.cern.ch/~hoschek/colt/V1.0.1/doc/overview-summary.html.

The implementation of Boyer-Moore search algorithm was obtained from the following web page.http://www-igm.univ-mlv.fr/~lecroq/string/node14.html#SECTION00140

The MD-4 implementation was obtained from RFC1320.

The Reed-Solomon code was obtained from Zhao Hui Chen. http://www.hrl.harvard.edu/~zhchen. Decoding is done using Euclid's Algorithm.

Simulation:

Block size used was 1020 and the clue size was 10 bytes. CRC used was a 16-bit CRC. First 30 characters of every block were used for CRC calculation. The MD4 was 128 bit. The interleaving depths used were 1020 and 205 for a symbol size of 8-bit and 16-bit respectively. A file of 1.3 MB was used throughout the simulation. The file used was a temporary output file generated by our program in the development phase. The file was chose due to the innumerable number of patterns in the file.

Appendix B-Plot of T across number of bytes substituted

Figure A-1 is the plot of T (parameter of Reed-Solomon code) against bytes modified for highly scattered substitution for a symbol size of 16 bits.

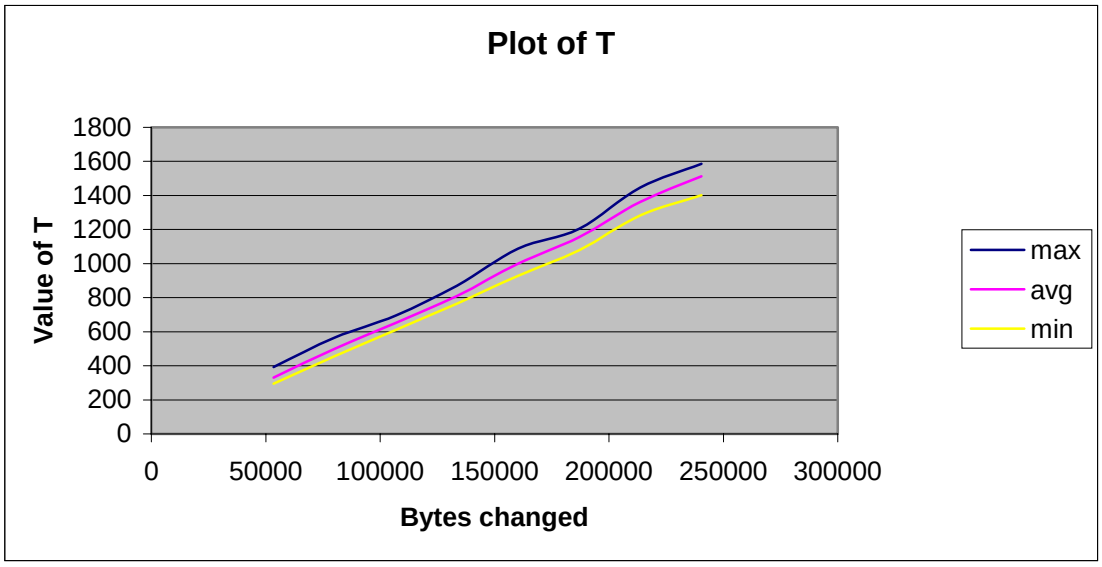


Figure A- 1 Plot of T for highly scattered smaller substitutions

Appendix C- Throughput curve for applications that allow substitutions only

Figure A-2 is a throughput graph of highly scattered substitutions for applications that handle fixed sized objects and do not have the concept of insertions and deletions. We do not have the need to handle the problem of converting insertions and deletions to substitutions. Hence no pseudo errors are introduced.

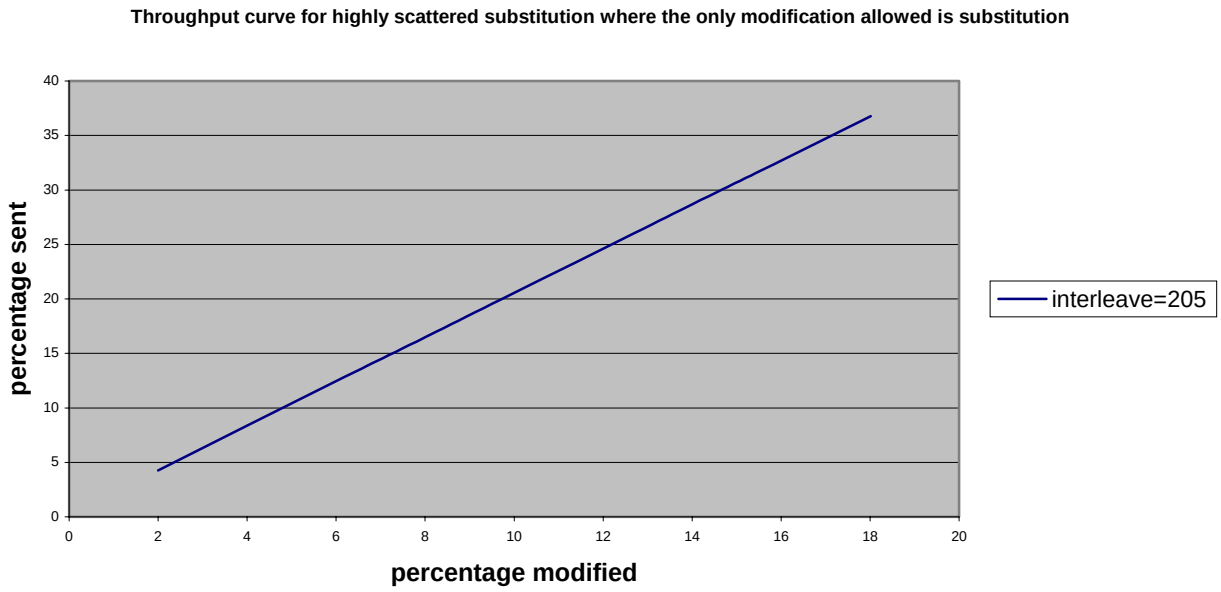


Figure A- 2 Throughput curve for highly scattered substitutions for applications that handle fixed sized objects

Appendix D– Presentation Slides

Slide 1

Propagation of Updates Using Error Correcting Codes

- Committee Chair: Dr. James Mooney
- Committee members:
  - Dr. Jagannathan
  - Dr. Mathew Valenti
  - Dr. Sumitra Reddy

Slide 2

Need and Objective

- Need for replicas.
  - With the increasing need for quick access of data, replication techniques like caching and mirroring have been used increasingly.
  - To increase the reliability and availability of the system backups are being used.
- Objective
  - Our objective is the propagate of changes made in one of the replicas to the rest of the copies

Slide 3

Classification of Existing Consistency Mechanism

- Entropy based update propagation
  - The difference between the outdated copy and the updated copy is determined. The difference is propagated. *Our mechanism is entropy based.*
- Anti-Entropy based update propagation
  - The operations made in a replica is propagated to be rest and the operation is redone. Effective bandwidth utilization. Cannot expect all the nodes to redo the operation. Requires the application that modifies the object to be alive at the time of establishment of consistency.

Slide 4

Entropy based update propagation

- Static diff based algorithms
  - We find the difference between the outdated copy and the updated copy of the data.
  - Assumption: Every outdated version should be known to the nodes that share the data.
  - Or a producer initiated approach should be adapted where a change is propagated regardless of its needs to the rest of the nodes.

Slide 5

Entropy based update propagation

- Dr. Orlitsky's mathematical proofs:
  - The amount of extra data to be transferred in order to propagate an update when the outdated copy is not available at the node with the updated copy is negligent, *if a three-way communication is used*. RSYNC a highly practicable update algorithm was implemented that satisfies this property by Dr. Tridge.
  - The amount of data to be transferred in order to propagate an update for the above case *when a one-way communication* is used is *utmost twice* the data required when the outdated data is available at the node with the updated data.

Slide 6

Continued…

- RSYNC update algorithm
  - A dynamic diff algorithm which establishes the difference between the existing file and the outdated file.
  - Handles insertions , deletions , substitutions and moves of data blocks very efficiently.
  - Better bandwidth utilization than our approach.
  - We are interested in a one-way communication. RSYNC uses 3-way communication

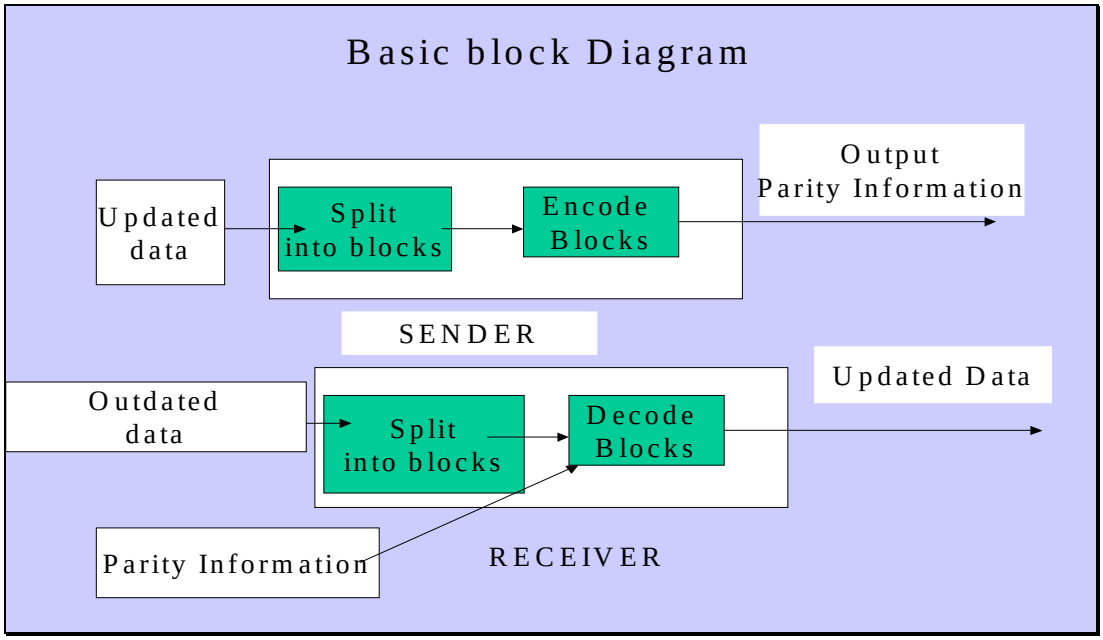
Slide 7

Assumed Environment for the system

- The maximum change in data is in the range of 20-30%.
- An estimate of the maximum of amount of change in bytes is available.
- Volume of data is of the order of Mega bytes.
- Cost of computation << cost of bandwidth.
- The outdated and updated files are not available at the same location.

Slide 8

Basic block Diagram



Slide 9

Handling insertions and deletions

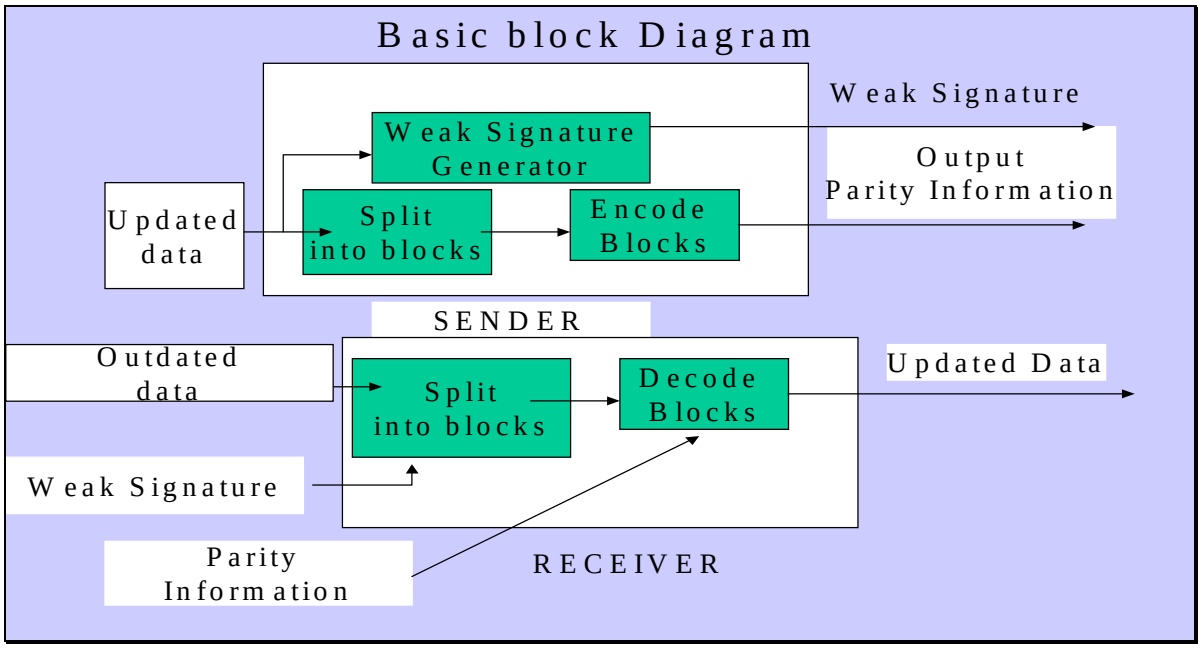
- What will happen if an insertion occurs at the beginning of the block?
  - Illustration:Updated Data: **abcdXefg…n bytes**

    Outdated Data:**abcdefg…n bytes**

    Number of errors: n-4

    Number of changes : 1 insertion
- Solution:
  - We identify each block with a weak signature. In our case the weak signature is the first X bytes of a block.
  - We convert insertions and deletions to substitution.
  - During the process of converting insertions and deletions to substitutions we introduce errors referred in the future as pseudo errors.

Slide 10



Slide 11

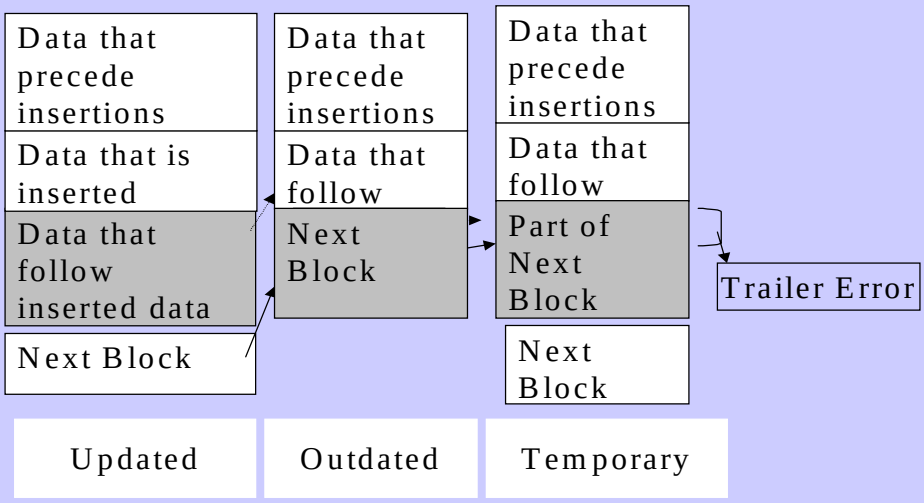
Weak signature and Pseudo Errors

- Our weak signature would be the first N bytes(clue) and the CRC of first n+m bytes of the block.
- Major causes, for the inability to recognize a block completely follow
- - The weak signature by itself could be a new data that has been inserted, deleted or substituted.
- - Due to repetitive patterns in the file, the weak signature could be wrongly matched.
- Major cause, for the inability to recognize a block partially follow

- Often due to *insertions* or *deletions* that span within a block we only partially recover the block as the characters that follow the insertion are shifted. We refer to such errors as trailer errors.

Slide 12

Trailer Error Illustration



Slide 13

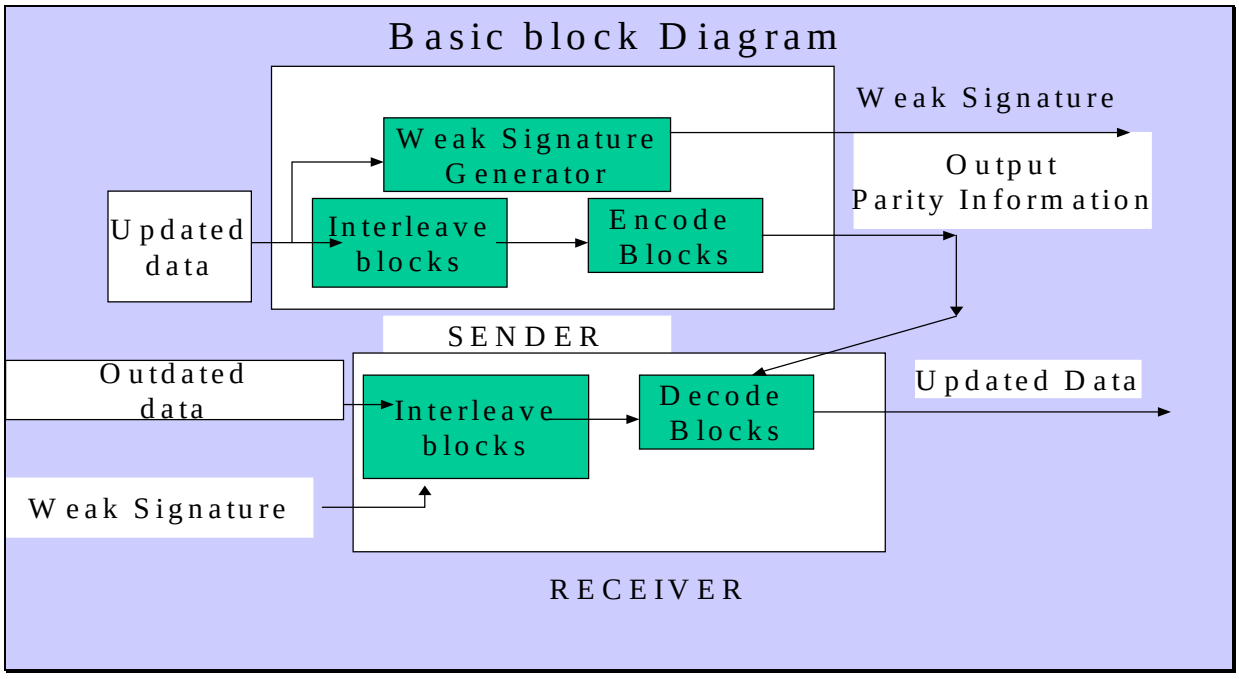
Dispersing Errors

- Consider the scenario of block n being completely changed. In order to propagate the change we transfer parity information that is twice the file size in order to correct any change of one block(not just the n^{th} block)$^{+}$.
- Solution: Interleaving of data. Instead of forming a block by grouping 1000 consecutive bytes we group the n^{th} byte of the blocks to form an input to the encoder/decoder. By this way the change gets scattered.Such interleaving is referred as Matrix interleaving.
- N ranges from 1 to interleaving depth(parameter we choose)
- Maximum Burst size
  = Inerleaving depth*(maximum_correctable_blocks)

+ Our objective is to correct multiple versions of outdated objects

Slide 14

Basic block Diagram



Slide 15

How much parity information?

- Encoding and Decoding : Reed Solomon Encoder and Decoder
  - Encoder accepts K = 255 – 2*T
  - T is the maximum number of errors that could occurs in K bytes
  - 2*T is the size of the parity information transferred needed to propagate a change of utmost T bytes for every K bytes to be propagated (best case).
- Hence for a change of X bytes we need to transfer 2*X bytes if no pseudo error occurs(ideal case).
- In addition for every pseudo error introduced we need to send two bytes.
- Besides the computation time involved in decoding was very high for larger percentages of modification.

Slide 16

Basis for Modification

- Basis for the change
  - Cost of imbalance in interleaving increases with interleaving depth, hence restricting us to smaller interleaving depth. Smaller interleaving depth results in splitting of data into multiple source blocks. When the object is split into multiple source blocks obtaining value of T for each source block from the users estimate of amount of change is non-trivial. Hence codes that are efficient with large values of K of a (K,T) should be preferred.
  - Tornado Erasure Correcting codes are erasure codes that are extremely faster than Reed-Solomon codes. However they require negligently more parity information that Reed-Solomon codes
  - In order to correct an erasure we need one bytes but to correct an error we need two bytes.

Slide 17

New Algorithm for Correcting substitution Errors

- *SENDER:*

1. Split the file into small blocks of fixed size same as or smaller than that of interleaving depth.For each block calculate CRC.
2. Send the CRC to the receiver.
3. Send the parity information obtained by interleaving and encoding to correct the erasures.

- *RECEIVER:*

1. Split the file into small blocks of fixed size same as or smaller than that of interleaving depth.For each block calculate CRC.
2. Receive the CRC sent by the sender.
3. Compare the received CRC with the corresponding calculated CRC.
4. Each unmatched CRC correspond to location of erasure of size same as that of block size.
5. Receive the parity information.With the interleaved blocks and parity information correct the erasures.
6. De- interleave the file to construct the updated file.

Slide 18

Effectiveness Of the Erasure Correction approach

Advantages:

- The approach will be bandwidth efficient as we sent only single byte per erasures.
- We expect the computation time to be significantly lower in comparison with the earlier approach.

Disadvantages:

- The approach is ineffective when the amount of CRC sent to detect erasures will be more, relative to the change in bytes. For a 16-bit CRC and a interleaving depth of 100 bytes amount of CRC sent is about 2% of the file size.
- For applications that deal with only substitutions we do not require weak signatures to convert insertions and deletions to substitutions. In such applications we can use our earlier approach effectively for highly scattered substitutions but not the later.

Slide 19

Parameters and Tuning

- Weak Signature
  - Clue
    - » large clue size has high Bandwidth cost
    - » Small clues- many CRC computation due to false match
  - CRC
    - » Probability of random collision and hence block mismatch increases for smaller CRC
- Block size
  - Larger the block size larger is the loss due to pseudo errors.
  - Smaller the block size more is the number of clues we send and hence more is the bandwidth needed.

Slide 20

Parameters and Tuning

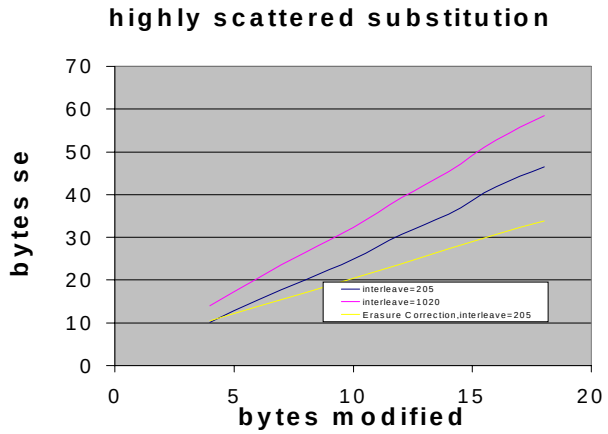
Interleaving depth

–Larger the interleaving depth larger is the amount of parity information we send due to imbalance in dispersion of change. Hence larger values of K in a (K,T) code are required. This is the major reason for adapting to Tornado codes.

–Smaller interleaving depths may result in breaking of object into more than one source block results in loss of generality as well as makes the process of determining the value of T from the user's estimate non-trivial.

Slide 21

Throughput Graph for Highly Scattered Substitutions



Slide 22

Conclusion

- Multicast propagation across multiple versions
- Caching of just the parity information if the updated copy and outdated copy are at the sender and receiver respectively unlike the diff based approach.

Future Work

- Mechanism for auto-detection of T-value
- Adapting to smaller insertions and deletions
- Concurrency and Communication protocol

Table of Figures

References

Bahl75 - Lalit Bahl and Frederick, *Decoding for Channels with Insertions, Deletions and Substitutions with Applications to Speech Recognition*, IEEE transactions on Information Theory, vol IT-21, July 1975

Balenson - David Balenson, David Carman, Michael Heyman and Alan Sherman, *Adaptive Cryptographically Synchronized Authentication model and analysis*, TIS Report# 0758, TIS labs at Network associates, Inc .

Berlekamp65 - Berlekamp, E.R., *On Decoding Bose-Chaudhuri-Hocquenghem Codes*, IEEE Trans. Information Theory, Oct 1965 IT-11: 577-579

Boyer77 - Boyer R.S. and Moore J.S. *A fast string searching algorithm*, Comm. ACM 20,1977, pp. 762-772.

Byers98 - John W. Byers, M. Luby, M. Mitzenmacher and Ashutosh Rege, *A Digial Fountain Approach to Reliable Distribution of Bulk Data*, ACM SIGCOMM, sep 1998 pp. 56-57

Chien64 - R.T. Chien, *Cyclic Decoding Procedures for BCH codes,* IEEE Trans. Information Theory, IT-10, Oct 1964. pp. 357-363

Cohen00 - Norman H. Cohen, *A Java frame work for mobile data synchronizations*, 7th International Conference on Co-Operative Information systems, Sep 2000.

Forney65 - G. D. Forney Jr., *On decoding BCH codes*, IEEE Transactions on Information Theory, vol. IT-11, October 1965

Imai90 – *Hideki Imai, Essentials of error-control techniques*, Academic press, 1990. pp. 93-95

Keleher96 - Pete Keleher, Alan L. Cox, Sandhya Dwarakas and Willy Zwaenepoel, *TreadMarks: Distributed shared memory on standard Workstations and Operating Systems*, IEEE Computer, Vol.29, Feb 96. pp. 18-28

Lee99 - Lee, Leung, Mahadev Satyanarayanan, *Operation based Update Propagation in a Mobile File System*, USENIX Annual Technical Conference, June 1999

Luby97 - M. Luby, M. Mitzenmacher, A. Shokrollahi, D. Spielman and V. Stemann, *Practical Loss-Reslient Codes*, 29th AVM Symposium on Theory of Computing, May 1997

Luby98 - M. Luby, M. Mitzenmacher and A. Shokrollahi, *Analysis of Random Processes via And-Or Tree Evaluation*, IEEE Information Theory Workshop, Feb 1998

Mogul97 - Jeffrey Mogul, Fred Douglas, Anja Feldman , Krishnamurthy. *Potential benefits of delta encoding and data compression for HTTP*, Proceedings of ACM SIGCOMM'97 Conference, Sep 1997. pp. 181-194 .

Nonnenmacher96 - Jorg Nonnenmacher, E.W.Biersack, *Reliable Multicast: Where to use Forward Error Correction*, Proc. 5th Workshop on Protocols for high speed networks, Oct 1996.

Orlitsky91 - Alon Orlitsky, *Interactive Communication of Balanced Distributions and of Correlated files*, 32nd normal Symposium on foundation of Computer Science, Oct 1991. pp 229 - 238

Paar94 - Christof Paar and Olaf Hooijen, *Implementation of a Reprogrammable Reed-Solomon Decoder over GF(2^{16}) on a Digital Signature Processor with External Arithmetic Unit*, Fourth International European Space Agency(ESA) workshop on Digital signal processing techniques applied to space communications, sep 1994

Patterson96 - John Patterson, Mark Day and Javok Kucan , *Notification Servers for Synchronous Groupware*, Proc. ACM 1996 Conference on Computer Supported Co-operative work, Nov 1996, pp: 122-128

Rivest - R. Rivest, *The MD4 Message Digest Algorithm* RFC1320

Rizzo - Luiggi Rizzo, *Effective Errasure Codes for Reliable Computer Communication Protocols*, DEIT Technical report, LR - 970115

Rizzo97 - Luiggi Rizzo and Lorenzo Vicisano, *A Reliable Multicast Data distribution Protocol based on Software FEC techniques*, Proc. 4th IEEE Workshop on Architecture and Implementation of High Performance Communications Systems, June 1997.

ShivaKumar98 - Shivakumar and Garcia-Molina, *Finding near-replicas of documents on the web*, Sixth International Word Wide Web Conference, Vol. 29, pp:157-66, Sep 1997

Taylor - Taylor, R., Jana R. and Grigg, M., *Checksum Testing of Remote Synchronization tool* , Technical report 0627(Nov), Defence Science and Technology Organization, Canberra, Australia (p.72)

Tridgelle - Andrew Tridgelle, *Efficient algorithm for Sorting and Synchronization* (Chapter 3 and 4) , http://samba.org/~tridge/phd_thesis.pdf

Williams - S.Williams, *HTTP Delta-Encoding Notes*, http://ei.cs.vt.edu/~williams/DIFF/prelim.html