An Efficient Implementation of Tornado Code for Fault Tolerance

JianJun Lei* Gu In Kwon*

Abstract This paper presents the implementation procedure of encoding and decoding algorithms for Tornado code that can provide fault tolerance for storage and transmission system. The degree distribution satisfying heavy tail distribution is produced. Based on this distribution, a good random irregular bipartite graph is attained after plenty of trails. Such graph construction is proved to be efficient, and the experiments also demonstrate that the implementation obtains good performance in terms of decoding overhead.

Keywords: Tornado Code, Degree Distribution, Graph Construction

1. Introduction

In a distributed storage system, data preservation is a primary requirement of storage. The storage system must use a fault tolerance mechanism for reliable store data despite of physical device failures [1]. Existing systems generally use data replication for fault tolerance: every disk is protected by mirroring in a RAID array. Replication is a simple approach, while it must increase entire replicas to provide high durability for reliable storage system [2].

In data communication system, FEC(Forward Error Correction) and ARQ (Automatic Repeat reQuest) are two traditional data robust transmission mechanism for reliable peer to peer or multicast system. The FEC system uses error correction code by adding redundancy and corrects errors directly in receiver. So it must predict the condition of channel in advance to improve the efficiency of channel. In addition, the complexity of encoding and decoding is deadly for some constrained power systems [3]. The ARQ system corrects errors by retransmission. Nevertheless, end to end retransmission is inefficient, or in some cases impossible due to infinite buffer requirement for potential retransmission request or long latency for real time application.

The development of Tornado code tackles the dilemma from another angle. Tornado code provides redundancy without the overhead of strict replication. It divides an object into k symbols and encodes them into n symbols, where $n > k$, we call $r = n/k$ as stretch factor. The key property of Tornado code is that original symbols can be reconstructed from any a little more than k symbols. Where, each symbol is a single bit or a packet of many bits. After this invention, all kinds of application based on storage and communication spring up like mushrooms. It provides much more fault tolerance than systems using RAID, FEC or ARQ.

2. Tornado Code

Luby first presented Tornado code as a mechanism for reliable and efficient multicast for file distribution system [4]. The basic unit of Tornado code is a bipartite low density parity check (LDPC) graph (figure 1). A series of left nodes represent the original data symbols and right nodes that are derived from such original data symbols are referred as check symbols to provide redundancy. Edges of graph describe which data symbols are used when calculating check symbols using logical XOR operations. Each right node is assigned two or more left nodes to be its neighbors, and the contents of

‡This research was supported by a grant(07KLSGC05) from Cutting-edge Urban Development-Korean Land Spatialization Research Project funded by Ministry of Construction & Transportation of Korean Government.

* Ph .D Candidate, Inha University, School of Computer and Information Engineering, dannylei@hotmail.com
** Assistant Professor, Inha University, School of Computer and Information Engineering, gikwon@inha.ac.kr(corresponding author)

Manuscript received : 2009.05.12 Revised : 2009.06.19 Accepted : 2009.06.20

check symbol is set to be the bit wise XOR of value of its neighbors[5]. Reconstruction is performed in reverse: if any right node has exactly one missing left node, the missing left node can be reconstructed. Parameters involved in the construction of a Tornado code include the number of data symbols, the stretch factor, and the left and right edges distributions.

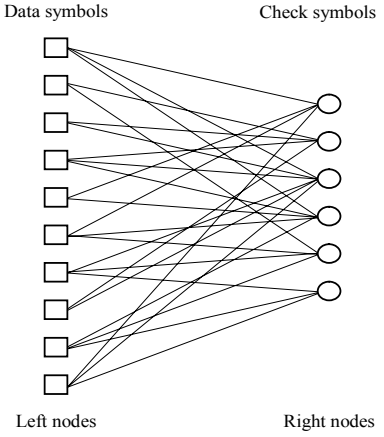

Fig. 1. Structure of Tornado Code

We retained Luby's preference for working with degrees of edges instead of degrees of nodes: an edge has degree i on the left if its adjacent node on the left has degree i, and an edge with degree j on the right is defined similarly. The graphs are described in the following terms: for each graph, there is a corresponding left degree vector λ and a right degree vector ρ. The value λ_i represents the fraction of edges with degree i on the left, and similarly the value ρ_i represents the fraction of edges with degree j on the right. Note that given a vectorλandρcan constructs a graph with the correct edge fractions for any number of nodes [6].

An important intuition that irregular graph prove useful for construction of Tornado code is demonstrated by Luby: from the point of view of a data node, it is best to have high degree, since the more information it gets from its check nodes the more accurately it can judge what its correct value should be; in contrast, from the point of view of a check node, it is best to have low degree, since the higher the degree of a check node, the more likely it is to transmit incorrect guesses to the data node [7]. These two competing requirements must be appropriately balanced. For irregular graphs, there is more flexibility in balancing these competing requirements. Actually, for our following experiments, data nodes with high degree tend to their correct value quickly, and almost all undecodable data nodes only have degree 2 when the decoding is failed. Hence irregular graphs are a necessary component of Tornado code.

3. Implementation

In this section, we describe some detail of construction of Tornado code and explain some principles behind Tornado code.

3.1. Degree Distribution Graph Construction

We now present a method to construct degree distribution graph with "double heavy tail distribution". We use R denote the rate for code, which is the ratio of number of check symbols to number of original symbols, it will affect the amount of data overhead that must be gotten in order to reconstruct original data; D denote the maximum degree for the left nodes. The value of D can affect both encoding and decoding time. The first simple constrain from our previous definition of degree is: $\Sigma_i \rho_i/i = R * \Sigma_i \lambda_i/i$, the following lemmas are easily proved too.

Lemma1 : The average degree of left nodes $A_L = H(D) * (D+1) / D$, and the average degree of right nodes $A_R = A_L / R$.

Proof : The fraction of edge of degree i on the left is given by $\lambda_i = 1/(H(D) * (i-1))$, where $H(D) = \Sigma_1^D 1/i$ is the harmonic sum truncated at D , this result is approximately equal to $\ln(D)$. Assume the total number of edges is E, by definition, the number of edges of degree i on the left is $E\lambda_i$, therefore the number of left nodes of degree i is $E\lambda_i$, the total number of left nodes is $\Sigma_2^D E\lambda_i/i$ (no node of degree 1 in our graph construction), the average degree of left nodes is calculated as:

$$A_L = \frac{E}{\Sigma_2^D \frac{E\lambda_i}{i}} = \frac{1}{\Sigma_2^D \frac{\lambda_i}{i}} = \frac{1}{\Sigma_2^D \frac{1}{H(D)*(i-1)*i}} = H(D)*(D+1)/D$$

Lemma2 : the fraction of left nodes with degree i to be $Node\_\lambda_i = A_L * \lambda_i / i = A_L / (H(D) * i * (i-1))$. The fraction of nodes with degree j on the right equals to $Node\_\rho j = A_R * \rho_j / j = A_R * e^{-\alpha} * a^{j-1} / j!$

Proof : The right degree is defined by the poisson distribution, the fraction of edges of degree i on the right equals to $\rho_j = e^{-\alpha} * a^{j-1} / ((j-1)!)$, where a is chosen to guarantee that the average degree on the right is A_R , in the other word, a should satisfy $a * e^{\alpha} / (e^{\alpha} - 1) = A_R$. A specific node degree distribution with code rate

$R = 0.5$ and the maximum degree for the left nodes $D = 15$ is depicted in figure 2.

In implementation, since computer arithmetic is not exact, we may need to adjust the degree of certain

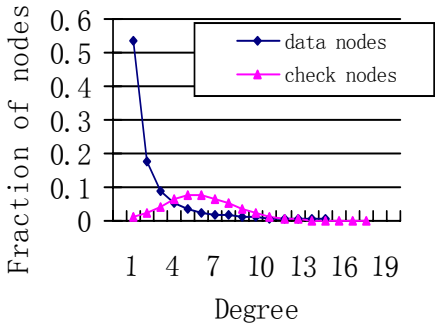

Fig. 2. Degree Distribution of Nodes(D=15, R=0.5)

nodes so that the total number of edges leaving the left nodes is equal to the number of edges leaving the right nodes. The "correct" number of edges in the graph should be $A_L * k$. We compare the number of left edges and right edges to see which comes closer to this correct value and then adjust the other side of the graph to match the "winner's" edge count. When computing the value for fraction of left nodes and right nodes, we also should calculate the number of nodes of each degree to ensure that the total number of nodes in those calculations matches the number that we had to work with. When adjustments are necessary, we pick a node out at random and increment or decrement that node's degree by one. We continue to pick nodes until the desired node and edge counts are correct. We first adjust the node counts before adjusting the edge counts.

The generated degrees are assigns randomly to left nodes and right nodes. Then we connect the edges with a left node and a right node: for each edge, randomly pick a left node and randomly pick a right node. If either node already has all of its edges, then select another node at random. In most cases the last few left nodes have multiple edges to the same right node due to imperfections in the randomization process, which will produce poor results. In our implementation, we will avoid this defect by constructing these graphs many times and choose one good construction of them. Although we spent a considerable amount of computing time on the optimization it is clear that any given degree distribution graph can be further improved given enough patience.

3.2. Encoding Algorithm

For encoding, we set the value of each right node to be the XOR of the value of its left neighbors. At the same time, the sequences of all left neighbors are recorded in the head of check nodes and the sequences of all right neighbors are recorded in the head of data nodes. In the following subsection, we will explain how such solution improves the efficient of decoding.

3.3. Decoding Algorithm

The decoding process is symmetric to the encoding process, except that the check nodes are now used to recover their left neighbors. A check node can recover data node only if the content of the check node is known and only one left neighbor of that node is missing. The result that the content of check node is XORed with the contents of its left neighbors is assigned to the missing neighbor [5]. To reduce the effect of certain types of network data loss behavior (e.g. bursts) and minimize the overhead of decoding, the nodes are transferred over the network in random order. In our implementation, the certain amount of data nodes are chosen randomly as missing nodes, then pick up check node one by one until all missing data nodes are recovered. The decoding algorithm is listed for receiver in the following box.

Algorithm of decoding:

1. *Randomly receives part of data nodes as known nodes, the amount is dependent on loss rate of channel.*
2. *Pre_decode check nodes: recover part of check nodes from received data nodes.*
3. *Receives one check node, if such check node has not been recovered by Pre_decode procedure and its degree is one, then start to decode:*
   a. *Recovers one data node, if all data nodes are recovered, stops and reports SUCCESS.*

b. Decreases the degree of all right neighbors (check nodes) of the recovered data node by one, if there is any check node of degree one and the check node has been recovered, repeat step a.

4. If all check nodes have been received or recovered, stops and reports FAILURE, otherwise repeats step 3.

The algorithm of decoding can improve time efficiency by recording all right neighbors of data nodes without requirement of global research for check nodes of degree one in every decoding. See figure 3 for a recovery example. Obviously, the success of the above algorithm depends only on the graph and the specific set of loss data node.

4. Experimental Results

The most important metric for Tornado algorithm is decoding overhead, which is the ratio of the number of received nodes to the number of original data nodes. The three parameters that affect this metric are the number of data packet k, the code rate R and the loss rate of data packets. Actually, the maximum degree of left node also affects the decoding overhead slightly. In our experiments, we have observed that degree 15 as maximum degree of left node and code rate R=0.5 result in good results, so these parameters will be fixed in all following experiments.

We do not perform an actual encoding, but instead for each trail use an initial message consisting entirely of zeros, which is reasonable assumption because we only focus on how to construct a good graph so that the decoding overhead is as small as possible. A different random graph was constructed for each trial, we then get the average decoding overhead from 1000 decoding tests. The best average decoding overhead will be chosen as the final result after 5000 random graph construction trial. We expect that our results would be slightly better if more trials were conducted. This is a time consuming work due to the imperfection of randomization process for graph construction. So, present implementation has been tested for k less than 200. One option would have been to process the left nodes sequentially, and only choose the right nodes at random. We found that the process does run faster, but it produces poor results. We plot the decoding

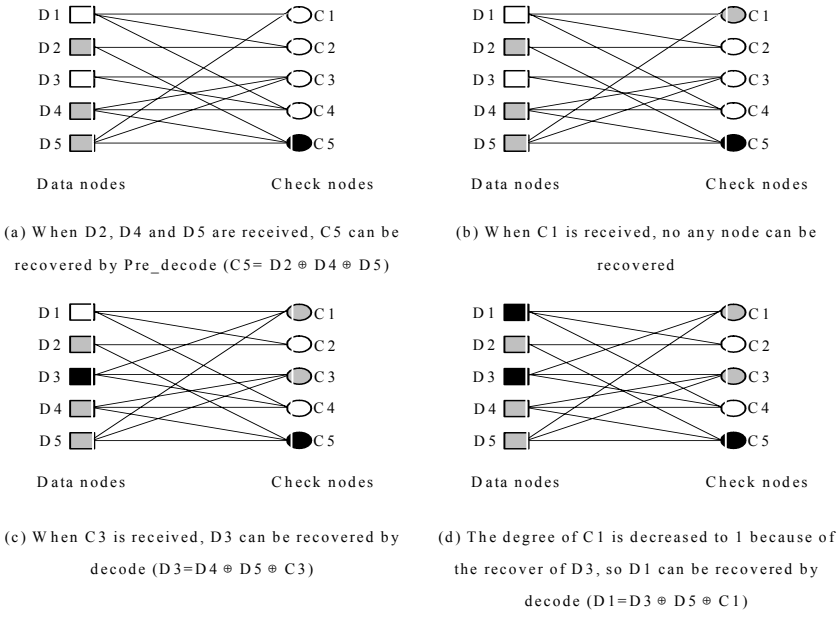

Fig. 3. An example of decoding

overhead for different number of data nodes in figure 4.

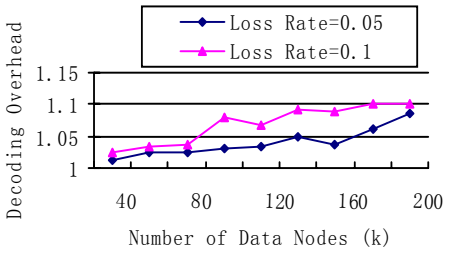

Fig. 4. Decoding Overhead with Increase of k

More trials should be conducted with bigger value of k to obtain smaller decoding overhead, since it is higher probability that generate bad edges connections when random graph construction. The defect of bad graph is the case where a set of left nodes shared the same right nodes. In this case, some check nodes that carry redundant check information contained by other check nodes result in the inefficiency of algorithm. We can adjust such bad graph by graph optimization process instead of constructing good graph endlessly, which will be one issue of our future works.

5. Conclusions

Tornado code provides fault tolerance for reliable data storage and transmission. As the first step in Tornado code, we produce degree distribution and construct bipartite graph according to it. These random graphs are tested and the best ones are chosen. The decoding experiments demonstrate that the implementation of algorithm obtains good performance in terms of decoding overhead in spite of more trials will need to be conducted for the case of bigger number nodes.

The next step of our plan is to detect and migrate the bad connect edges from the random graph instead of constructing good graph by endless random graph construction. Another neglected problem in this paper is decoding inefficiency, which also will be an important issue in the future.

References

[1] Matthew Woitaszek and Henry M. Tufo, “Fault Tolerance of Tornado Codes for Archival Storage”, 15th IEEE International Symposium on High Performance Distributed Computing, 2006, pp. 83-92.

[2] Hakim Weatherspoon and John Kubiatowicz, “Erasure Coding Vs. Replication: A Quantitative Comparison”, Revised Papers from the First International Workshop on Peer to Peer Systems, 2002, pp. 328-338.

[3] A. G. Dimakis, V. Prabhakaran, and K. Ramchandran, “Distributed data storage in sensor networks using decentralized erasure codes”, in Proc. Asilomar Conf. Signals, Systems, and Computers Signals, November 2004, pp.1387-1391.

[4] J. W. Byers, M. Luby, M. Mitzenmacher and A. Rege, “A Digital Fountain Approach to Reliable Distribution of Bulk Data”, SIGCOMM, 1998, pp. 56-67.

[5] Matthew Delco, Hakim Weatherspoon and Shelley Zhuang, “Typhoon: An Archival System for ToleratingHigh Degrees of File Server Failure”, University of California, Berkeley project report, December 1999, http://www.cs.cornell.edu/~hweather/Typhoon/.

[6] Michael Luby and Michael Mitzenmacher and Amin Shokrollahi and Daniel Spielman and Volker Stemann, “Practical Loss Resilient Codes”, Proceedings of the twenty ninth annual ACM symposium on Theory of computing, 1998, pp. 150-159

[7] Michael G. Luby, M. Amin, Z Daniel and A. Spielman, “Analysis of low density codes and improved designs using irregular graphs”, Proceedings of the thirtieth annual ACM symposium on Theory of computing, 1998, pp. 249-258.

JianJun Lei

2000 Chongqing Institute of Technology(B.S.)
2006 Chongqing University of Posts and Telecommunications(M.S.)
Present Ph.D. candidate, School of Computer and Information Engineering at Inha University
Research Interests : Wireless Communication, Sensor Network.

Gu In Kwon

1995 Dept. of Computer Science, Inha University(B.S.)
1998 Dept. of Computer Science, City University of New York, Queens College(M.S.)
2005 Dept. of Computer Science, Boston University(Ph.D.)
Present Assistant Professor, School of Computer and Information Engineering, Inha University
Research Interests : Multicast, Congestion Control, Overlay Network, and Sensor Networks