Rateless Codes and Big Downloads

Petar Maymounkov and David Mazières
{petar,dm}@cs.nyu.edu
NYU Department of Computer Science

Abstract. This paper presents a novel algorithm for downloading big files from multiple sources in peer-to-peer networks. The algorithm is simple, but offers several compelling properties. It ensures low handshaking overhead between peers that download files (or parts of files) from each other. It is computationally efficient, with cost linear in the amount of data transfered. Most importantly, when nodes leave the network in the middle of uploads, the algorithm minimizes the duplicate information shared by nodes with truncated downloads. Thus, any two peers with partial knowledge of a given file can almost always fully benefit from each other's knowledge. Our algorithm is made possible by the recent introduction of linear-time, rateless erasure codes.

1 Introduction

One of the most prominent uses of peer-to-peer systems is to download files—often very large files, such as movies [1]. More often than not, these files are available at least in part at more than one node on the network. This observation has inspired a number of different algorithms for multi-source file download [2], including some that have already been deployed [3].

The basic multi-source download problem is simple. A set of nodes, called *source nodes*, have complete knowledge of a certain file. A set of nodes we call *requesting nodes* wish to obtain a copies of that file. The goal is to transfer the file to the requesting nodes in a fast and bandwidth-efficient manner. In practice, the task is complicated by the fact that nodes can join or leave the system, aborting downloads and initiating new requests at any time. Thus, a download algorithm can make very few assumptions about the uptime or bandwidth capacity of participating nodes.

Most multi-source download algorithms take the same general approach. When a requesting node needs a file, it first locates a set nodes with full or partial knowledge of that file. It then contacts as many of them as necessary to download the file efficiently. For each source node the requesting node contacts, the two must reconcile the differences in their knowledge of the file. Then either the requesting node downloads any non-overlapping information, or often both nodes exchange any non-overlapping information they have about the file.

An effective multi-source download algorithm should meet two main challenges. First, it should maximize the utility of nodes with partial knowledge of a file to each other. This, in turn, means minimizing the amount of overlapping information nodes are likely to have. We call this property the *availability* aspect of the algorithm, because it allows nodes with truncated downloads to reconstruct a file even in the event that every source node with the complete file has left the network.

The second challenge of a multi-source download algorithm is to make the reconciliation phase as bandwidth-efficient as possible. This phase is an instance of the more general set reconciliation problem [4–7]. Unfortunately, existing set reconciliation algorithms are not practical for multi-source download algorithms. They are either too computationally costly, suboptimal in terms of message complexity, or simply too complicated to implement.

In this paper, we propose an algorithm that combines near-optimal availability with a simple yet practical reconciliation phase not based on the general set reconciliation problem. Our approach is made possible by the recent introduction of locally-encodable, linear-time decodable, rateless erasure codes. It exploits particular properties of the way file contents tend to disperse over nodes in a peer-to-peer system. The paper is presented in terms of a new erasure code called on-line codes [8]. The recently published LT-codes [9] are similar to on-line codes, but have $O(n \log n)$ running time, compared to linear time for on-line codes.

The next section gives an overview of erasure codes and their use in multi-source downloads and introduces on-line codes. Section 3 details on-line codes and their implementation. Section 4 describes our multi-source download algorithm. Section 5 discusses aspects of the algorithm and open questions.

2 Loss-resilient codes

Erasure codes transform a message of n blocks into an encoding of more than n blocks such that one can recover the message from only a fraction of the encoded blocks. A block is just a fixed-length bit string, with block size a parameter of the algorithm. Many erasure codes, including the on-line codes in this paper, support blocks of arbitrary size, from a single bit to tens of kilobytes or more. Choice of block size is driven by the fact that a fragment of an encoded block conveys no useful information about the original message. Thus, blocks should be small enough that aborted block transfers do not waste appreciable bandwidth.

Conventional erasure codes also have a *rate* parameter, specifying the fraction of the encoded output blocks required to reconstruct the original message. An *optimal* erasure code with rate R transforms a message of n blocks into n/R blocks such that any n suffice to decode the original message. Because of the cost of optimal codes, people often employ *near-optimal* codes, which require $(1+\epsilon)n$ output blocks to reconstruct the message for any fixed $\epsilon > 0$. The cost of a smaller ϵ is increased computation. Currently there is only one instance of linear-time, near-optimal, conventional erasure codes, called Tornado codes [10].

Linear-time, near-optimal erasure codes have already been used for multi-source downloads [2]. The basic approach is for source nodes to encode files with rate $R < 1/2$ to achieve a large expansion factor. When a node downloads files, the source node sends a pseudo-random permutation of the encoded blocks to the requesting node, deriving the permutation from the requesting node's ID. Using this technique, two nodes that each have downloaded $0.6n$ encoded blocks of an n-block file will likely have enough information between them to reconstruct the file. Thus, the file will remain available even if all source nodes leave the network. To generalize the technique to more requesters, however, the expansion factor $1/R$ would need to grow proportionally to the number of truncated downloads. Unfortunately, even Tornado codes become impractically expensive and memory-intensive for rates $R < 1/4$.

A new class of erasure codes, *rateless, locally-encodable codes*, addresses the problem. The two novel properties of these codes—ratelessness and local encodability, go hand-in-hand. Ratelessness means that each message of size n has practically infinite encoding. Local encodability means that any one encoding block can be computed quickly and independently of the others. Replacing conventional fixed-rate codes with rateless, locally-encodable ones in the above scenario and making some additional use of the unique properties of rateless codes leads to the multi-source download algorithm presented in section 4 of this paper.

There are two instances of practical rateless codes, LT codes [9] and on-line codes [8], both recently proposed. We present our algorithm in terms of on-line codes because they are more efficient, requiring $O(1)$ time to generate each encoding block and $O(n)$ time to decode a message of length n. LT codes, in contrast, take $O(\log n)$ and $O(n \log n)$ time respectively (though they are asymptotically optimal and require no preprocessing, which may make them more convenient for other settings). The next section describes the implementation of on-line codes in greater detail.

3 On-line codes

This section explains how to implement on-line codes. A more detailed description and analysis of the algorithm is available in [8]. On-line codes are characterized by two parameters, ϵ and q (in addition to the block size). ϵ determines the degree of suboptimality—a message of n blocks can, with high probability, be decoded from $(1 + 3\epsilon)n$ output blocks. q, discussed subsequently, affects the success probability of the algorithm—it may fail to reconstruct the message with negligible probability $(\epsilon/2)^{q+1}$.

The overall structure of on-line codes has two layers, depicted in Figure 1. To encode a message, in an outer encoding, we first generate a small number of *auxiliary blocks* and append them to the original message to produce a *composite message*. The composite message has the property that knowledge of any $1-\epsilon/2$ fraction of its blocks is sufficient to recover the entire original message. Next, in a second, inner layer, we continuously generate blocks to form an infinite, rateless encoding of the composite message. We call these blocks *check blocks*, because they serve as a kind of parity check during message reconstruction.

The decoding process is the inverse of the encoding. We first recover a $1-\epsilon/2$ fraction of the composite message from (received) check blocks, then recover the entire original message from the this fraction of the composite message. In practice, auxiliary blocks and check blocks are similar in nature, allowing implementations to combine both layers of the decoding process.

3.1 Outer encoding

The first step of the encoding process is to produce a composite message by generating $0.55q\epsilon n$ auxiliary blocks and appending them to the original message. Each auxiliary block is computed as the XOR of a number of message blocks, chosen as follows. We first seed a pseudo-random generator in a deterministic way. Then, using the pseudo-random generator, for each block of the original message, we chose q auxiliary blocks, uniformly. Each auxiliary block is computed as the XOR of all message blocks we have assigned to it. We append these auxiliary blocks to the original message blocks, and the resulting $n' = (0.55q\epsilon + 1)n$ blocks form the composite message.

With this construction, knowledge of any $1 - \epsilon/2$ fraction of the composite message is sufficient to recover the entire original message with probability $1 - (\epsilon/2)^{q+1}$. The decoding process is described at the end of this section, though the analysis is beyond the scope of this paper and described in [8].

3.2 Inner encoding

We now describe how to generate check blocks from the composite message. The inner encoding depends on global values F and $\rho_1, \ldots, \rho_F$ computed as follows:

$$F = \left\lceil \frac{\ln(\epsilon^2/4)}{\ln(1-\epsilon/2)} \right\rceil$$

$$\rho_1 = 1 - \frac{1 + 1/F}{1 + \epsilon}$$

$$\rho_i = \frac{(1-\rho_1)F}{(F-1)i(i-1)} \quad \text{for } 2 \leq i \leq F$$

Each check block is named by a unique identifier, taken from a large ID space, such as 160-bit strings. The check block is computed by XORing several blocks of the underlying composite message. These blocks of the composite message are chosen as follows, based on the check block ID.

We begin by seeding a pseudo-random generator with the check block ID. Using the pseudo-random generator, we chose a *degree* d from 1 to F for the check block, biased such that that $d = i$ with probability ρ_i. We then pseudo-randomly and uniformly select d blocks of the composite message and set the check block to the XOR of their contents.

Any set of $(1+\epsilon)n'$ check blocks generated according to this procedure will be sufficient to recover a $1-\epsilon/2$ fraction of the underlying composite message. The price to pay for a smaller ϵ is an increase by a constant factor in the decoding time. Specifically, the decoding time is proportional to $n \ln(1/\epsilon)$.

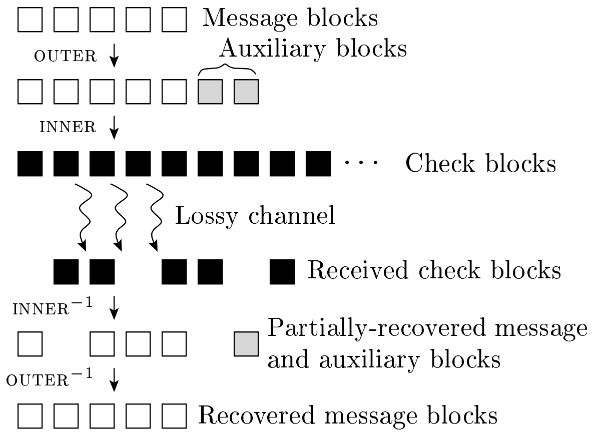

Fig. 1: Overall design of online codes.

3.3 Decoding

We call the message blocks that were XORed to produce a check or auxiliary block its *adjacent* message blocks. Decoding consists of one basic step: Find a check or auxiliary block with exactly one unknown adjacent message block and recover the unknown block (by XORing the check block and all other adjacent message blocks). Repeat this step until the entire original message has been recovered.

The decoding process is depicted in Figure 2. Initially, the only useful check blocks will be those of degree 1, which are just copies of composite message blocks. Once the degree-1 check blocks have been processed, more check blocks become usable. As the number of decoded message blocks continues to increase, more and more higher-degree check and auxiliary blocks will become usable. Note that one can execute this decoding procedure on-the-fly, as check blocks arrive.

To see that the decoding process takes linear time, following the approach of [10, 11], we think of the composite message blocks and the check blocks as the left and right vertices, respectively, of a bipartite graph G. A check block has edges to and only to the message blocks that comprise it in terms of the XOR. We say that an edge has *left* (respectively *right*) degree d if the left-end node (respectively right-end node) of this edge is of degree d. Using the graph language, the decoding step is: find an edge of right degree 1 and remove all edges incident to its left-end node. In the graph context, decoding completes when all edges have been removed. Since the total number of edges is bounded by $(1 + \epsilon)Fn'$ (specifically it is roughly equal to $n' \ln F$), the decoding process runs in linear-time. A similar argument applies to the auxiliary blocks as well.

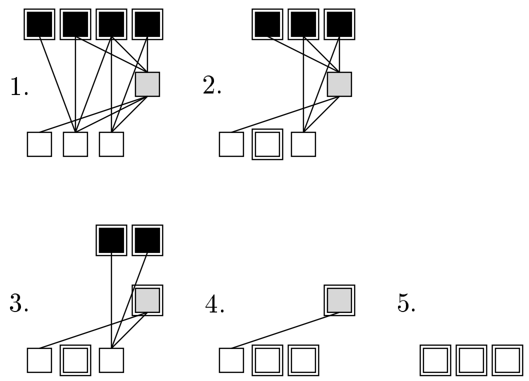

Fig. 2: Evolution of the decoding process of an example 3-block message. Squares with double-boundaries represent blocks that are known (recovered).

3.4 Practical considerations

On-line codes are proven to be good asymptotically in n. Thus, as messages get too small, the suboptimality of the erasure code increases. However, with $\epsilon = 0.01$ and $q = 3$, one can decode messages of as few as 1,000 blocks after receiving only 3% more check blocks than the size of the message, and with probability of failure 10^{-8}.

To estimate the performance on large files, we used a non-optimized, 150-line Java implementation to encoded and decoded a message of size 1 million blocks. The decoding took roughly 10 seconds for blocks of size zero, and would have required approximately 11 million block XORs for a non-zero block size.

We recommend the parameters $\epsilon = 0.01$ and $q = 3$ to implementors, resulting in $F = 2114$ and an average check block degree of 8.23.

4 Multi-source download

We now address the question of how to implement multi-source download with on-line codes. One approach might be for source nodes simply to send check blocks with random IDs to requesting nodes. This solution yields nearly optimal availability. However, if source nodes disappear and requesting nodes begin exchanging check blocks, two communicating nodes may still have significant overlap in their knowledge of check blocks. (This can only happen, if the overlapping check blocks came from the same third node earlier in time. Avoiding this effect is within the scope of another topic—deciding how nodes choose whom to exchange information with in the first place.) Exchanging only useful blocks would essentially boil down to the set reconciliation problem.

To improve on the above solution, a multi-source download algorithm should allow nodes to produce concise descriptions of large numbers of check blocks.

Fortunately, requesting nodes tend to download large numbers of blocks from the same source before being interrupted. Thus, we can divide a node's check blocks into a small number of *streams*—one for each aborted download from a source node to a particular requesting node.

We define a *stream with ID* $s^* \in \{0,1\}^{160}$ to be the sequence of 160-bit numbers $a_0, a_1, a_2, \ldots$, where $a_i = \text{SHA-1}(s^*, i)$. For a given file f, we refer to a sequence of check blocks with IDs $a_1, a_2, \ldots$ as the *encoding stream with ID* s^*.

Each requesting node downloading a file f can keep a small state table of pairs *(stream ID, last position)*. A pair (s^*, p) is in the table if and only if the node has the first p check blocks of the encoding stream with ID s^*. The pair (q^*, r), where q^* is the node's ID in the peer-to-peer system, is always in this table, even if $r = 0$.

To download a file, a requesting node, v, first uses a peer-to-peer lookup system to locate and select a node with information about the file. Preference is always given to a source with complete knowledge of the file, if one is available and has spare capacity. When downloading from a source node, a requesting node always downloads the encoding stream with ID equal to its own node ID in the peer-to-peer system. When resuming an aborted download, the requesting node simply informs the source of which stream position to start at.

When a requesting node v downloads a file from another node w that has only incomplete information, v begins by sending its entire state information table to w. In this way, w can send v only streams or stream suffixes that v does not already have. Furthermore, v can optionally request that w simultaneously upload any of its streams or stream suffixes. Blocks can be transfered one stream at a time, or else multiple streams can be interleaved, so long as each encoding stream is sent in order and starting from the correct position. There is a lot of freedom in the ordering of streams transfered, allowing for some optimizations we discuss later.

5 Discussion

Rateless codes are a promising new tool for the peer-to-peer community, offering the prospect of improved file availability and simplified reconciliation of truncated downloads. We expect even the simple multi-source download algorithm in the previous section to outperform most other schemes, either proposed or implemented. Rateless codes guarantee that almost every data block transmitted by a source node contains unique information about a file, thus minimizing the amount of duplicate information amongst nodes with partial file information. Moreover, the freedom to chose encoded block IDs from a large identifier space allows files to be encoded in concisely specifiable streams so that nodes can inexpensively inform each other of what blocks they know.

5.1 Open questions

We speculate that the reconciliation costs upon initiation of interaction between two nodes are minimal. The message cost of reconciliation between two nodes is no bigger than the cost of sending the state information table, whose size is directly proportional to the number of different streams from which a node has check blocks. This number is generously upper-bounded by the total number of nodes that had partial knowledge of the file within the life-span of the download. In our experience, this number has actually never exceeded 20. As a result, the reconciliation data sent upon initiation of interaction between two nodes will in practice always fit in one IP packet. This is likely more efficient than algorithms based on compact summary data structures [7] for set reconciliation.

To make the algorithm truly scalable, however, one needs to consider scenarios with dramatically larger numbers of nodes with partial file knowledge. In this case, we believe we can limit the growth of state tables by clustering peers into smaller groups within which nodes exchange partial file information. How big these clusters need to be, and how to design the algorithms for forming these clusters poses an open question.

Another open question is whether availability guarantees can be further improved. The specification of the algorithm in Section 4 leaves some freedom of interpretation. When a node v requests help from a node w with partial knowledge, w can choose the order in which it sends streams to v. For example, w could send blocks from one stream until the stream is exhausted, or it could interleave blocks from different streams. The choice becomes important if the connection between the two nodes is unexpectedly interrupted. By choosing what specific approach to use, and which stream(s) to send first, one can pick a favorable trade-off between higher reconciliation costs and higher file availability in the presence of unexpected disconnects. It is an open problem to find good strategies and understand the nature of this trade-off.

Finally, our multi-source download algorithm uses TCP. One could alternatively imagine UDP-based downloads. In particular, people often want peer-to-peer traffic to have lower priority than that of other network applications. A user-level UDP download protocol less aggressive than TCP could achieve this. With erasure codes, such a protocol might also avoid the need to retransmit lost packets, but at the cost of complicating state tables with gaps in streams.

6 Conclusion

We hope that this paper will motivate further studies of applications of rateless codes to peer-to-peer problems. Our experiments show that due to their simplicity of implementation and speed, on-line codes are a good candidate for practical solutions.

The download algorithm that we propose shows that rateless codes offer increased file availability and decreased reconciliation costs. Interestingly, the decrease of reconciliation costs is due to the limit on how many streams a cluster of nodes may need. This shows that one can avoid difficult information-theoretical problems, like set reconciliation, by making use of a wider range of properties of the underlying peer-to-peer system. Moreover, since the limit on the number of streams is, in some sense, a global property of the multi-source setting, further research should be done to better use other such global properties.

Acknowledgments

We thank Yevgeniy Dodis, Srinivasa Varadhan and Maxwell Krohn for helpful feedback on this work.

This research was supported by the National Science Foundation under Cooperative Agreement #ANI–0225660 (`http://project-iris.net`), and by DARPA and the Space and Naval Warfare Systems Center under contract #N66001–01–1–8927.

References

1. Saroiu, S., Gummadi, K.P., Dunn, R.J., Gribble, S.D., Levy, H.M.: An analysis of internet content delivery systems. In: Proceedings of the 5th Symposium on Operating Systems Design and Implementation. (2002) 315–327
2. J. Byers, J. Considine, M. Mitzenmacher, and S. Rost: Informed Content Delivery Across Adaptive Overlay Networks. In: SIGCOMM. (2002)
3. McCaleb, J.: (EDonkey2000) `http://www.edonkey2000.com/`.
4. Y. Minsky, A. Trachtenberg, and R. Zippel: Set Reconciliation with Nearly Optimal Communication Complexity. In: International Symposium on Information Theory. (2001)
5. M. Karpovsky, L. Levitin, and A. Trachtenberg: Data verification and reconciliation with generalized error-control codes. In: 39th Annual Allerton Conference on Communication, Control, and Computing. (2001)
6. Y. Minsky and A. Trachtenberg: Practical Set Reconciliation. In: 40th Annual Allerton Conference on Communication, Control, and Computing. (2002)
7. J. Byers, J. Considine, and M. Mitzenmacher: Fast Approximate Reconciliation of Set Differences. In: Draft paper, available as BU Computer Science TR 2002-019. (2002)
8. Petar Maymounkov: Online Codes. Technical Report TR2002-833, New York University (2002)
9. Michael Luby: LT codes. In: The 43rd Annual IEEE Symposium on Foundations of Computer Science. (2002)
10. M. Luby, M. Mitzenmacher, A. Shokrollahi, D. Spielman, and V. Stemann: Practical Loss-Resilient Codes. In: STOC. (1997)
11. M. Luby, M. Mitzenmacher, and A. Shokrollahi: Analysis of Random Processes via And-Or Tree Evaluation. In: SODA. (1998)