Rateless Codes and Big Downloads

Petar Maymounkov and David Mazières, NYU

Abstract

This paper presents a new algorithm for downloading big files from multiple sources in peer-to-peer networks. The algorithm is compelling with the simplicity of its implementation and the novel properties it offers. It ensures low hand-shaking cost between peers who intend to download a file (or parts of a file) from each other. Furthermore, it achieves maximal file availability, meaning that any two peers with partial knowledge of a given file will almost always be able to fully benefit from each other's knowledge– i.e., overlapping knowledge will rarely occur. Our algorithm is made possible by the recent introduction of linear-time rateless erasure codes.

1 Introduction

One of the most prominent functions of peer-to-peer systems is the download of files. More often than not, people tend to download big popular files (like movies) that are available in full or in part at more than one node on the network. This observation has inspired a number of different algorithms for multi-source file download. Some are already in use [1]. Some are in the form of proposals [3].

The basic multi-source download setting is simple. A set of nodes, called *source nodes*, of the underlying peer-to-peer system, have complete knowledge of a certain file, which is being requested by another set of nodes on the system. The effective goal is to devise an algorithm that allows the requesting nodes to obtain the file in a timely and bandwidth-efficient manner. The realistic setting, however, is much more elaborate. Nodes can join or leave the system at any point, and new requests for files may be issued while older requests are being serviced. As a result, a download algorithm can make almost no assumptions about the uptime of nodes, their available bandwidth, and so on.

Most previous multi-source download algorithms have taken the same approach. When a node needs a file, it looks up contact information for other peers who have partial or full knowledge of the file and contacts as many of them as necessary. For each peer it contacts, the node must reconcile the differences in its knowledge of the file with that of the peer, then download or exchange the non-overlapping information. This common approach poses two main challenges to the algorithm designer. First, the algorithm needs to ensure that whenever a node contacts another to download a file, the two are useful to each other. This is the case when the amount of overlapping information they are likely to have is minimized. We call this property the *availability* aspect of the algorithm, because it ensures that nodes complete their downloads in a more timely manner. Second, the reconciliation phase should be as bandwidth-efficient as possible. This reconciliation is an instance of the more general set reconciliation problem, which does not yet have a practical solution. Existing set reconciliation algorithms [8, 4, 9, 2] suffer from one of two drawbacks in the context of practical usage. They are either too complicated to implement, or are not optimal in terms of message complexity.

In this paper, we propose an algorithm that combines near-optimal availability with a simple yet practical reconciliation phase not based on general set reconciliation. Our approach is based on the way file content is dispersed over a set of requesting peers. Our algorithm is made possible by the recent introduction of locally-encodable, linear-time decodable, rateless erasure codes. This paper is written in terms of a new algorithm called on-line codes [7]. Luby has recently published an algorithm called LT-codes, which is similar to on-line codes, but suffers from $\mathrm{O}(n \log n)$ running time, as compared to linear time for on-line codes.

Section 2 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

Like most erasure codes, on-line codes encode a message of n blocks into an encoding of more than n blocks, so that any subset of n or slightly more than n encoded blocks is sufficient to recover the original message. A block is the smallest logical unit of information depending on the context. It can be any fixed-length bit-string, e.g. a bit, a byte, a word, or even a 512-byte array for use with UDP. Conventional erasure codes have a *rate* R, which means that the encoding they generate is fit to suffer losses of up to $1 - R$ fraction of the encoding blocks, while still ensuring that the remaining R fraction of the blocks will be enough to recover the original message. The term "rate" comes from the fact that codes are most commonly used to transfer information across a lossy communication channel with a given rate, where the rate of the channel is defined to be the fraction of packets transfered through the channel that actually get received on the other end. An erasure code of rate R takes a message of size n blocks and produces an encoding of size n/R, so another way of looking at it is to think of the code as expanding the message by a factor of $1/R$.

We can subdivide erasure codes into optimal and near-optimal. Optimal codes can recover the original message from any subset of the encoding that is of size n blocks – the message size. Near-optimal codes can recover the message from any subset of size $(1 + \epsilon)n$ blocks, for any fixed $\epsilon > 0$ – slightly more than the message size. A big open problem of coding theory is whether optimal codes with linear time to encode and decode exist. However, near-optimal codes that achieve linear-time encodability and decodability have been devised.

Linear-time near-optimal erasure codes have already found an application in multi-source downloads [3]. The main idea is that whenever a source node is asked to initiate a download of a file's contents to a requesting node, the source node starts sending a random permutation of a rate $R < 0.5$ (i.e. big expansion factor) encoding of the file. This permutation is chosen in some way dependent on the node's ID. Using this technique, two nodes that each downloaded $0.6n$ blocks of a file's encoding from some source nodes will be very likely, albeit not entirely likely, to recover the file on their own by exchanging each other's knowledge of the file's encoding. This way, even if the source nodes for a given file disappear from the network, there still will be sufficient information available in the network to recover the file. This is why we say that this technique provides higher file availability. This fails to be true as the number of nodes with partial knowledge grows larger, unless $1/R$ (the expansion factor) grows proportionally with it. Unfortunately, even state-of-the-art near-optimal linear-time erasure codes become extremely expensive and impractically memory-intensive as for very low code rates.

A new kind of erasure code, on-line codes [7], addresses the problem. On-line codes are near-optimal, "rateless," linear-time decodable, "locally-encodable" and very simple to implement. The two novel properties of these codes – rateless-ness and local encodability, go hand by hand. Rateless-ness means that each message n has practically infinite encoding. Whereas local-encodability means that any one encoding block can be computed quickly and independently of the others. Replacing conventional fixed-rate codes with on-line codes in the above scenario and making some additional use of the unique properties that rateless codes offer leads to the algorithm that we propose in section 4 of this paper. The next section describes the implementation of on-line codes in greater detail.

3 On-line codes

Here we explain how to implement a variant of on-line codes, presented in [7], and shortly explain how they work. The code consists of two parts. Part one is a near-optimal, rateless code which can recover a fixed, arbitrarily large fraction $1 - \delta$ of the message, but not the whole message. This is why we refer to it as the *incomplete code*. Part two is a recipe for how to add a small fraction of "auxiliary blocks" to the original message to produce a new composite message. Any $1 - \delta$ fraction of the composite message will be enough to recover the entire original message with fixed, arbitrarily high probability.

3.1 Part one

We now proceed to describe how to generate the encoding blocks (we call them *check blocks* from now on) of the incomplete code, and how to decode most of the underlying message using a sufficient number of these blocks. Given a message of size n, every check block is generated randomly and independently as follows. A check block is the XOR of d randomly chosen message blocks. The degree d is chosen first according to a given distribution $\rho(\epsilon, \delta) = (\rho_1, \rho_2, \ldots, \rho_F)$, such that the degree is d with probability ρ_d and the maximum degree F is a constant (independent of n). Knowledge of any $(1+\epsilon)n$ check blocks generated according to the above procedure will recover $1-\delta$ fraction of the original message. The distribution ρ is a function of ϵ and δ as follows: $F = (\ln\delta + \ln(\epsilon/2))/\ln(1-\delta)$, $\rho_1 = 1 - (1+1/F)/(1+\epsilon)$, and $\rho_i = (1-\rho_1)F/((F-1)i(i-1))$ for $2 \leqq i \leqq F$. The price to pay for a smaller ϵ and δ is an increase in the constant factor in the decoding time. Specifically, the decoding time is proportional to $n \ln F \approx n \ln(1/\epsilon)$ (when $\epsilon \sim \delta$ are small).

The decoding process is no more difficult than the encoding. We call the message blocks that comprise a check block its *adjacent* message blocks. Decoding consists of one basic step: Find a check block, all of whose adjacent message blocks have been recovered except for one, and solve for it. Repeat this step until the entire message has been recovered. The main idea is that initially only a few check blocks will be useful to decode some message blocks. Namely this will be the check blocks of degree 1, which are in effect direct copies of message blocks. After the degree-1 check blocks have been processed, more check blocks will become useful, and so forth. This way most of the message will be recovered in a cascading fashion. To see that this process takes linear time, following the appraoch of [5, 6], we think of the n message blocks and the $(1+\epsilon)n$ 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 are 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. We would also like to make a point of the fact that using the above decoding procedure one can easily decode on-line, i.e. as the check blocks arrive.

Since the receiving party needs to know how the received check blocks were created in order to use them properly, each check block needs to have an ID (or a sequential index) which could be the seed of the pseudo-random function used to generate it. We are going to use 160-bit check block ID's, which will mean that the effective number of conceivable unique check blocks is 2^{160}, given that the message is reasonably big.

3.2 Part two

To make sure that we recover the whole message, instead of applying the above code to the actual message that we want to encode, we apply it to a composite message. The composite message will consist of the actual message followed by a small number of auxiliary blocks which, just like the check blocks from the previous section, will act as a kind of parity check blocks. Like check blocks, every auxiliary block will be the XOR of some number of message blocks. Only this time, each message block randomly chooses which q auxiliary blocks will be adjacent to it. To ensure that knowledge of any $1-\delta$ fraction of the composite message will recover the entire original message with probability $1-\delta^{q+1}$, at least $1.1q\delta n$ auxiliary blocks should be used.

By picking a good configuration of ϵ, δ and q, one can practically alleviate the probability of failing to recover the whole message. Since these codes are proven to be good asymptotically (in n), there is some lower limit on how small the encoded messages could be. Using this two-part code design, we have achieved codes that recover messages of as few as 1000 blocks, with only 3% overhead and probability of failure 10^{-8}. Our non-optimized, 150-line Java implementation encoded and decoded a message of size 1 million blocks in roughly 10 seconds. For the experiment, we used blocks of size 0, so that we can measure the intrinsic time of encoding and decoding without factoring in disk access and XOR-ing costs (which depend on the block size).

4 The algorithm

We start by defining a *stream* with ID $s^* \in \{0,1\}^{160}$ to be the infinite sequence of 160-bit numbers $a_0, a_2, \ldots$, where $a_i = \text{SHA1}(s^* \cdot i)$. Consequently, we use "a stream with ID s^* of a file's encoding" to mean the sequence of check blocks of this file's encoding with ID's taken from the above infinite sequence.

Each peer node, which is in the process of downloading a file, will keep a small amount of state information represented by a table of pairs of the form *(stream ID, last position)*. If the pair (s^*, p) is in the list, it will mean that the node has the first p check block of the stream with ID s^* of the file's encoding. The pair (q^*, r), where q^* is the node's peer-to-peer ID, will always be in this table, even if $r = 0$.

We are going to assume that each requesting node v has some external way of finding out which other nodes are source nodes for the requested file, and which other nodes are in the process of downloading the file, i.e. have partial knowledge. Of all these other nodes, v will choose one or more, according to some open-ended choice procedure, and will initiate an interaction with each of them. Each interaction will proceed as follows.

If v is interacting with a source node, then first v sends its (v^*, r) pair to that node, where v^* is v's node ID. Then the source node starts sending to v all check blocks from the stream with ID v^* starting from position r. For the sake of simplicity of exposition, we are going to assume that the recipient knows the stream ID and position of each check block. One can achieve this, e.g., by sending the stream ID, starting position and length of the stream before they start transferring the stream. The interaction terminates whenever one of the two parties disconnects or when v has collected the required number of blocks to recover the file. In case the interaction terminates before v is able to recover the file, v updates its state information table appropriately.

On the other hand, if v is interacting with a node w that has partial knowledge of the file, v first proceeds to send w its entire state information table. This allows w to get a complete picture of what check blocks v has. Then w proceeds to send v check blocks that v doesn't have, while making sure that for each encoding stream the check blocks are sent in consecutive order (different streams could be interleaved), starting from the position after the last position in the stream for which v has a check block. We note that there is a lot of freedom in the ordering of the check blocks that are sent to v. This freedom allows for a variety of optimizations that we discuss later. This completes the definition of the algorithm.

5 Discussion

Rateless codes seem to be a promising new tool for the peer-to-peer world. They offer improved file availability and simplified reconciliation protocols. Already in plain vanilla form, we expect the above algorithm to outperform most currently implemented or proposed multi-source download schemes. When it comes to availability, i.e. guarantees that peers with partial file knowledge have as little overlap in their knowledge of the file as possible, the use of rateless codes achieves most of what one could hope for. Two nodes can have overlapping information only if earlier down the line this information came from the same one node that had partial information of the file. The possibility of such occurances are low. They can be reduced even further by carefully designing the way in which nodes with partial file knowledge choose the ordering of the check blocks that they exchange among each other. To properly understand the implications of rateless codes and further optimize their use, we emphasize a few open questions.

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 to be more cost effective compared to algorithms using compact summary data structures [2] for set reconciliation.

To prove that this algorithm scales, though, one needs to consider the case when the number of nodes with partial file knowledge increases dramatically. In this case, we believe that one can improve the above algorithm to make peers form small but sufficiently big clusters within which they will help each other with the download. Then the size of the state information table will be proportional to the size of these clusters. How big these clusters need to be, and how to design the algorithms for forming these clusters poses an open question.

Another open question asks whether availability guarantees can be improved even further. The specification of our 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 blocks 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 of approach becomes important if the connection between the two nodes is unexpectedly interrupted. By choosing what specific approach to use, 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, the consistency of our algorithm is based on a common, unspoken assumption that nodes communicate over TCP. This ensures that receiving nodes won't have holes in their knowledge of the streams, because packets don't get dropped until the connection is open. It is an interesting question whether there is an equivalent of this algorithm that works just as well over a lossy UDP channel. There is, of course, a handful of hacks that can get around this problem, however most of them will increase the number of rounds in the protocol which is undesirable. We believe that possible solutions to this problem may make use of global properties of the flow of information in the peer-to-peer network.

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.

References

[1] EDonkey2000. http://www.edonkey2000.com/.

[2] 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.

[3] J. Byers, J. Considine, M. Mitzenmacher, and S. Rost. Informed Content Delivery Across Adaptive Overlay Networks. In *SIGCOMM*, 2002.

[4] 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.

[5] M. Luby, M. Mitzenmacher, A. Shokrollahi, D. Spielman, and V. Stemann. Practical Loss-Resilient Codes. In *STOC*, 1997.

[6] M. Luby, M. Mitzenmacher, and A. Shokrollahi. Analysis of Random Processes via And-Or Tree Evaluation. In *SODA*, 1998.

[7] Petar Maymounkov. Online Codes. Technical Report TR2002-833, New York University, October 2002.

[8] Y. Minsky, A. Trachtenberg, and R. Zippel. Set Reconciliation with Nearly Optimal Communication Complexity. In *International Symposium on Information Theory*, 2001.

[9] Y. Minsky and A. Trachtenberg. Practical Set Reconciliation. In *40th Annual Allerton Conference on Communication, Control, and Computing*, 2002.