How Practical is Network Coding?

Mea Wang, Baochun Li

***Abstract*— With network coding, intermediate nodes between the source and the receivers of an end-to-end communication session are not only capable of relaying and replicating data messages, but also of coding incoming messages to produce coded outgoing ones. Recent studies have shown that network coding is beneficial for peer-to-peer content distribution, since it eliminates the need for content reconciliation, and is highly resilient to peer failures. In this paper, we present our recent experiences with a highly optimized and high-performance C++ implementation of randomized network coding at the application layer. We present our observations based on an extensive series of experiments, draw conclusions from a wide range of scenarios, and are more cautious and less optimistic as compared to previous studies.**

I. Introduction

Network coding has been originally proposed in information theory in 2000 [1], and has since received extensive research attention. The essence of network coding is a paradigm shift to allow coding at intermediate nodes between the source and the receivers in one or multiple communication sessions. The fundamental insight of network coding is that information to be transmitted from the source in a session can be *inferred*, or *decoded*, by the intended receivers, and does not have to be transmitted verbatim. It is a well known result that network coding may achieve better network throughput in certain network topologies [2].

To practically implement the paradigm of network coding, Ho *et al.* [3] has been the first to propose the concept of *randomized network coding*, in which an intermediate node transmits on each outgoing link a linear combination of incoming messages, specified by independently and randomly chosen *code coefficients* over some finite field. In Wu *et al.* [4], it has been shown that randomized network coding can achieve "close to the theoretically optimal performance." *Avalanche* [5] has further proposed that randomized network coding can be used for bulk content distribution, in competition with *BitTorrent*, one of the most successful P2P content distribution protocols. It claims that "the performance benefits provided by network coding in terms of throughput can be more than 2-3 times better compared to transmitting unencoded blocks."

Nevertheless, the claims have not been viewed to be sufficiently conclusive, as they are based on simulation studies (in both [4] and [5]), rather than real-world implementations. Reasonable doubts have been raised with respect to the feasibility of implementing network coding, especially with respect to the additional computational overhead of coding operations. Recent work from Avalanche has sought to demonstrate the feasibility of network coding with a real-world implementation in C# [6]. With its experiments in a long-lived distribution session lasting around 38 hours to distribute a 4.3GB file, it has been observed that "network coding incurs little overhead, both in terms of CPU and I/O, and it results in smooth and fast downloads." In particular, with respect to the computational overhead of network coding, the conclusion was drawn from the observation that each of the clients only consumes about 20% – 40% of its CPU throughout the session. This work, however, did not make any comparisons between using and not using network coding for content distribution.

Despite recent efforts of studying the practicality of applying network coding in peer-to-peer networks, there are still no clear and well-justified answers to the following question: Given a content distribution session in peer-to-peer networks, is network coding indeed able to offer a better throughput — best measured in the time to complete downloading at the peers, as compared to using a protocol without coding (such as BitTorrent)? In other words, *how practical is network coding, as compared to content distribution without coding?* In this paper, we seek to systematically study the advantages and drawbacks of implementing network coding in practice, and to offer an impartial view of the computational and communication overhead introduced by network coding. To achieve this objective, we resort to a controlled environment of a cluster of high-performance servers, and evaluate the practical implications of randomized network coding in a highly optimized and high-performance C++ implementation at the application layer. We have implemented a *bandwidth-emulated* environment, in which bandwidth limits of typical DSL and high-bandwidth institutional peers are emulated. We unfold our journey through an extensive series of experiments. Though some of our experiences echo some of the previous observations on CPU overhead [6], our conclusions are, nevertheless, *more cautious* and *less optimistic* that previous studies.

II. How Practical is Network Coding?

The focus of our study is on the practicality, performance and overhead of randomized network coding, as compared to not using coding to distribute data. To achieve our objectives, over a 12-month period, we have implemented a bandwidth-controlled and repeatable experimental testbed in a dedicated 50-node cluster of high-performance dual-CPU servers, interconnected by Gigabit Ethernet.

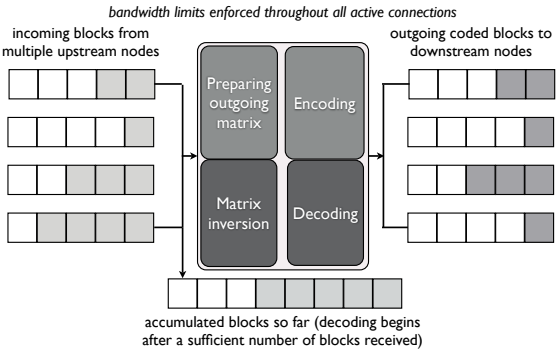

Fig. 1. The architecture of a bandwidth-emulated peer.

Mea Wang and Baochun Li are affiliated with the Department of Electrical and Computer Engineering, University of Toronto. Their email addresses are {*mea,bli*}*@eecg.toronto.edu*.

On each participating peer in a peer-to-peer session, the architectural design of our implementation is best illustrated in Fig. 1. The core of the architecture is our highly optimized implementation of the encoding and decoding routines of randomized network coding. To feed the encoding routines, we allow multiple TCP connections from multiple upstream peers. To transmit the result of the encoding process, multiple TCP connections to their corresponding downstream peers are established. In the lifetime of each neighboring peer, we use a persistent TCP connection to minimize overhead. To guarantee the most optimized binary with the best performance, the system is implemented in C++ (with about 10,000 lines of code), and is compiled with full optimization (`-O3`).

We briefly summarize the concept of randomized network coding [3], [4], [5], [7]. The original content on the source is segmented into n *blocks* $[b_1, b_2, \ldots, b_n]$, each b_i has a fixed number of bytes k (referred to as the block size). At the time of encoding for downstream peer p, a peer (including the source) independently and randomly chooses a set of coding coefficients $[c_1^p, c_2^p, \cdots, c_m^p](m \leq n)$ in the Galois field GF(2^8) for the downstream peer p. It then randomly chooses m blocks — $[b_1^p, b_2^p, \ldots, b_m^p]$ — out of all the blocks it has received so far (all the *original* blocks if it is a source of the session), and produces one coded block x of k bytes by computing $x = \sum_{i=1}^{m} c_i^p \cdot b_i^p$, where the ratio m/n is referred to as *density* in this paper. The n coding coefficients can easily be computed by multiplying $[c_1^p, \cdots, c_m^p]$ with the $m \times n$ matrix of coding coefficients embedded in the incoming blocks $[b_1^p, b_2^p, \ldots, b_m^p]$. As the session proceeds, a peer accumulates coded blocks received from its upstream peers into its local buffer, and encodes new coded blocks to serve its downstream peers. To maximize linear independence among coded blocks transmitted in the network, a peer independently and randomly chooses a new set of coding coefficients for each of its downstream peer.

As soon as a peer has received a total of n linearly independent coded blocks $\mathbf{x} = [x_1, x_2, \ldots, x_n]$, it starts the decoding process. To decode, it first forms a $n \times n$ matrix $\mathbf{A}$, using the n coding coefficients embedded in each of the n coded blocks it has received. Each row in $\mathbf{A}$ corresponds to n coded coefficients of one coded block. It then recovers the original blocks $\mathbf{b} = [b_1, b_2, \ldots, b_n]$ by computing $b = \mathbf{A}^{-1}\mathbf{x}^T$. In such a decoding process, it first computes the inverse of $\mathbf{A}$ using Gaussian elimination, and then multiplies $\mathbf{A}^{-1}$ with $\mathbf{x}$, which takes $n^2 \cdot k$ multiplications of two bytes in GF(256).

A. Coding Performance

The first question that one would naturally ask is: "What is the performance of randomized network coding?" To answer this question, we establish a single connection between one source and one receiver, each hosted by a dedicated dual-CPU server (Pentium IV Xeon 3.6GHz). We show the bandwidth of the encoding and decoding functions in Fig. 2, in bytes per second. The figure shows the results of a series of experiments, with different configurations with respect to the number of blocks and the block size.

In the figure, the x axis shows the number of blocks used, and bars with different grayscale represent different block sizes. We vary the block size from 64 bytes to 1 MB, and vary the number of blocks from 10 to 1500. There are a number of important observations we can derive from these results.

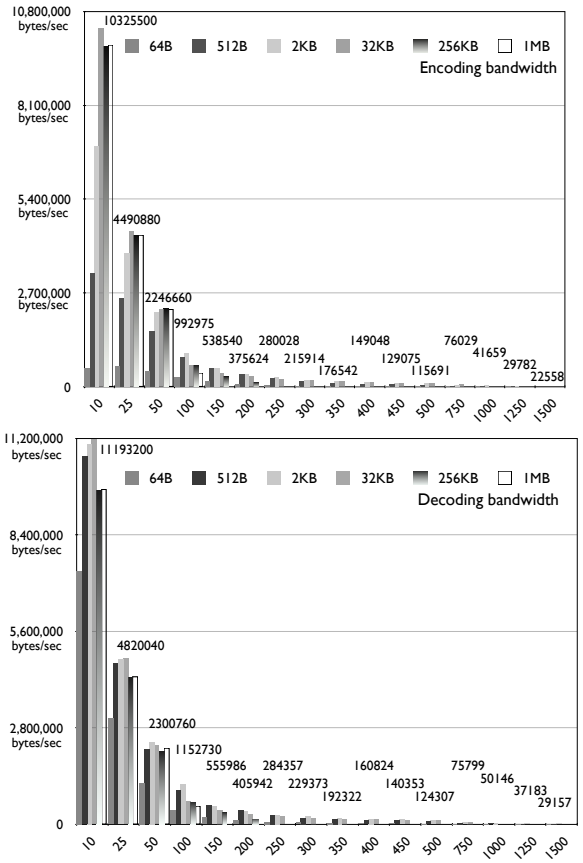

Fig. 2. Performance of randomized network coding: encoding (left) and decoding (right) bandwidth in bytes per second.

First, the absolute values of both encoding and decoding bandwidth are impressive, especially when the number of blocks is smaller than 100. As we have shown, when there are only 10 blocks, the coding bandwidth exceeds 10 MB per second! Even with 1500 blocks, we can still observe a coding bandwidth of more than 20 KB per second, which is about the same bit rate as a typical multimedia stream.

Second, the coding bandwidth rapidly decreases as the number of blocks increase, but it does not vary significantly as the block size varies. This observation justifies the use of a small number of blocks (such as 100). Indeed, the peer-to-peer experiments in Avalanche divide 4.3 GB files into *groups* (also referred to as *generations* [4]), and uses 80 blocks per group. Each block in Avalanche has approximately 2.3 MB. From our results, we can reflect that with 80 blocks and 2.3 MB per block, the coding bandwidth is between 1 and 2 MB per second. Since most peers are on a slower connection, the coding thread does not have the data readily available to code at its full bandwidth. This explains why CPU load is around 20% in Avalanche experiments.

Third, as the block size varies, there is a "sweet spot" when the coding bandwidth is maximized. This optimal block size shifts upwards as the number of blocks increases, but it is around 2 KB – 32 KB. This justifies the use of small block sizes less than 32 KB. Indeed, even BitTorrent uses 16 KB as

its block size.

Finally, in each configuration, the encoding and its corresponding decoding bandwidth are practically the same. At first glance, this seems to be hardly surprising, as both of them have been previously shown to be a multiplication of a matrix and a vector, in GF(256). As we further study the decoding algorithm, however, we note that it involves matrix inversion using Gaussian elimination, which has a complexity of $O(n^3)$. Shouldn't decoding be much less efficient than encoding?

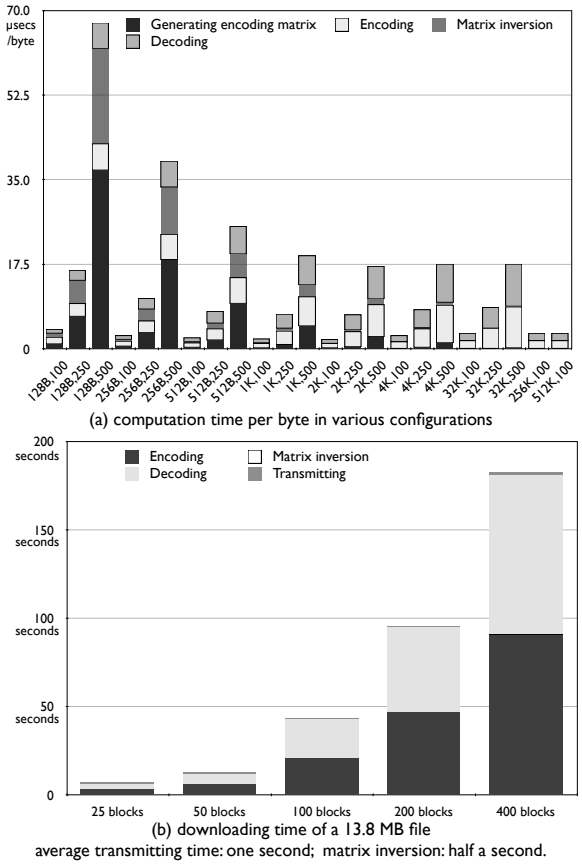

Fig. 3. Performance of randomized network coding: (a) computational time per byte in various configurations of block size and number of blocks; (b) time needed to download a 13.8 MB file in an one-to-one session, with the average transmitting time around one second, and matrix inversion around half a second.

To obtain further insights, we show the time to generate encoding matrices, encode, inverse a matrix and decode. We show these times, normalized to microseconds per byte, in Fig. 3(a). Again, we observe that the coding performance degrades significantly as the number of blocks increase from 100 to 500, regardless of the block size. We can also observe that the time taken to perform matrix inversion is negligible as we increase the block size k to anything beyond 2 KB. Larger blocks are actually more reasonable, as the header overhead to carry n coding coefficients in a coded block would be too significant if the block size is smaller than 2 KB.

What if we wish to transmit a fixed-size file from the source to the receiver? In Fig. 3(b), we show the time it takes to transmitting a 13.8 MB file using network coding in an one-to-one session, over Gigabit Ethernet without bandwidth limits. It appears that it takes, on average, only a second to transmit the coded blocks and half a second to inverse the matrix $\mathbf{A}$, both of which are negligible as compared to the encoding and decoding time. We have previously noted that it takes $n^2 \cdot k$ multiplications in GF(256) to multiply $\mathbf{A}^{-1}$ and $\mathbf{x}$. If the number of bytes in the data to be distributed, *i.e.*, $n \cdot k$, remains fixed, then the number of multiplications on GF(256) that a peer needs to perform (after computing $\mathbf{A}^{-1}$) scales linearly with n. This precisely matches the results shown in Fig. 3(b).

B. Density and Aggressiveness in More Realistic Topologies

Now that we have an idea about coding performance, we wish to deploy randomized network coding in more realistic network topologies, and compare its performance to *naive broadcast*, a scheme we choose as the *worst* possible protocol one can use to distribute content over peer-to-peer networks. In naive broadcast, upon receiving a block, each peer simply forwards it to all its downstream neighbors. All incoming duplicated blocks will simply be discarded. Our experiences have shown that it usually leads to 300% additional bandwidth consumption as compared to using a better pull or push-based protocol to reconcile content. We use naive broadcast as the baseline benchmark that we use to evaluate the practicality of network coding, with the mentality that if it is challenging for network coding to compete with naive broadcast, it will have greater difficulties competing with a well-tuned protocol such as BitTorrent.

To experiment with more realistic topologies in our 50-node dual-CPU server cluster, we wish to construct bandwidth-emulated peer-to-peer topologies with around 50 – 100 peers, each with a dedicated CPU in the server cluster. We implement the strategy of *staged* construction of such topologies, in that peers are added in batches, as each new peer joins, it randomly selects a certain number of upstream neighbors from the existing peers in the network. The topologies are designed to contain 30% Ethernet peers and 70% DSL peers. The uplink and downlink bandwidth limits of an Ethernet peer are set at 1 MB/sec, and a DSL peer has an uplink bandwidth limit of 50 – 80 KB/sec, and a downlink bandwidth limit of 100 – 200 KB/sec. These topologies are designed to emulate real-world peer-to-peer networks, and are referred to as *mixed* topologies hereafter. Further, when we evaluate performance, we are mostly concerned with the *average downloading time* of a peer-to-peer content distribution session, which is measured as the time period from the starting point of the session (before encoding starts at the source) to the point that a peer finished decoding (or recovering) all the original blocks.

To optimize the performance of a peer-to-peer session using network coding, one needs to tune its operational parameters so that they are optimal for this purpose. One such parameter is *density* m/n, in which m is the number of blocks a peer randomly selects to encode a new coded block for its downstream peers, and n is the number of original blocks in the session. In Fig. 4(a), we vary density from 100% to 2%, and measure the average downloading time in a mixed topology of 80 peers. We used 100 blocks and 32 KB per block, as they have been shown to offer superb performance in our previous experiments.

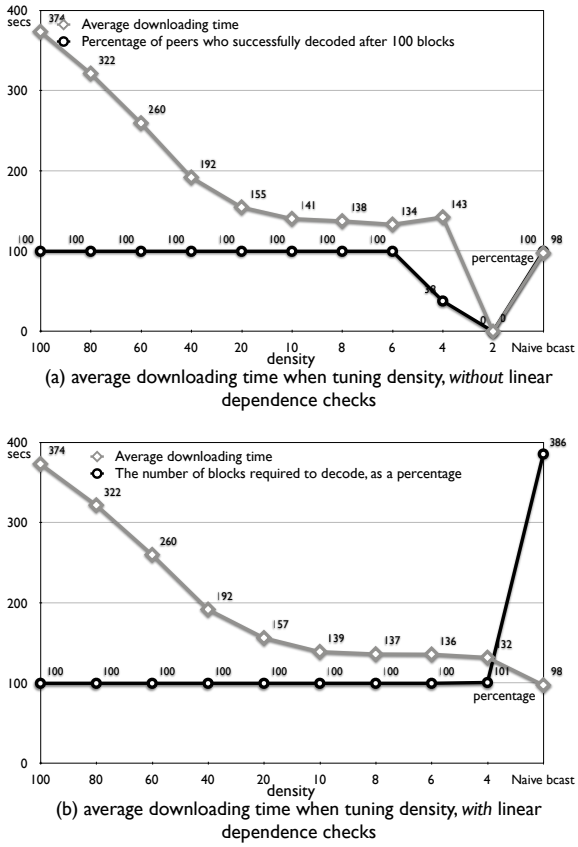

Fig. 4. Average downloading times when density varies from 100% to 2%, as compared to naive broadcast. (a) without linear dependence checks, also shown is the percentage of nodes that have successfully decoded after receiving 100 coded blocks; (b) with linear dependence checks, also shown is the number of blocks required for successful decoding, as a percentage of the number of original blocks The network has 80 peers, each with 10 upstream neighbors, 30%/70% mixed topology, aggressiveness is 100%, 100 blocks, 32 KB per block.

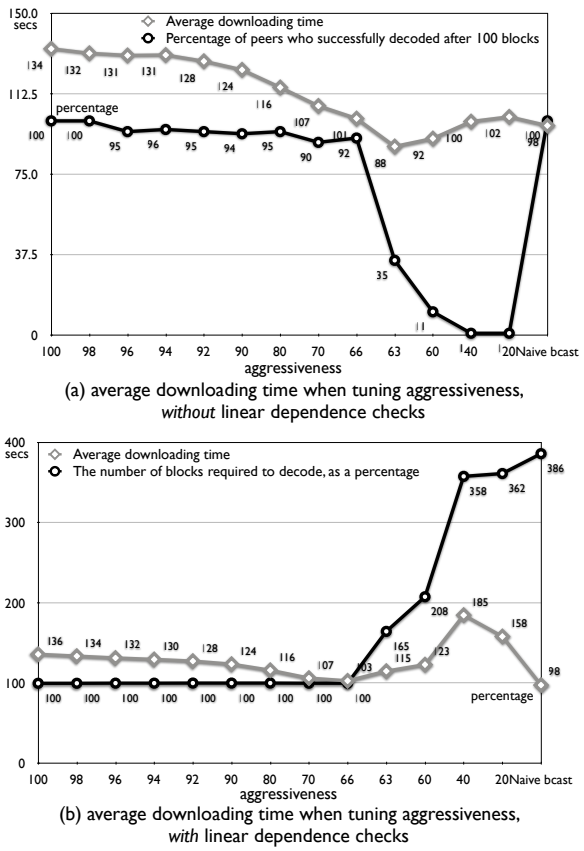

Fig. 5. Average downloading times when aggressiveness varies from 100% to 20%, as compared to naive broadcast. (a) without linear dependence checks, also shown is the percentage of nodes that have successfully decoded after receiving 100 coded blocks; (b) with linear dependence checks, also shown is the number of blocks required for successful decoding, as a percentage of the number of original blocks. The network has 80 peers, each with 10 upstream neighbors, 30%/70% mixed topology, density is 6%, 100 blocks, 32 KB per block.

From Fig. 4(a), we observe that as density decreases, the average downloading time steadily decreases. Indeed, this observation matches our intuition, as when density decreases, each peer has fewer blocks to encode, thus leading to a higher coding bandwidth, and smaller computational overhead of encoding. That said, when we tune density to lower than 6%, we can see that a substantial number of peers is not able to successfully decode after receiving the first 100 coded blocks, due to linear dependence among these blocks. This is not surprising, as the corresponding coding matrix would be too sparse to be full rank if the density is too low. It is indeed *surprising*, however, when we compare network coding with naive broadcast. We can see that naive broadcast, being the worst possible non-coding protocol, actually enjoys a better average downloading time than coding at any density using network coding! We wish to postpone our discussion of Fig. 4(b).

We now try to find remedies to improve the performance of network coding. When we review our implementation of network coding, it appears that each peer only starts to produce and serve coded blocks to its downstream peers when it has buffered n coded blocks itself. This is not efficient at all since it can easily start to produce coded blocks much earlier. We then implement a tunable parameter referred to as *aggressiveness* (for lack of a better word), which represents the number of blocks it needs to buffer before starting to produce and serve new coded blocks, normalized by the number of original blocks (as a percentage). We tune aggressiveness from 100% — which is what we use in the density experiment — down to 20%, in the hope that it can lead to shorter downloading times, as each peer is now more eager to start serving.

We start our experiment with the same configuration as the previous experiment, but with density set at 6%, the best we have observed in Fig. 4. Our results, shown in Fig. 5(a), indeed show a trend of *slowly* decreasing downloading times, which may not be as promising as we have expected. To make matters worse, as we start to monitor the number of peers who fail to decode after receiving 100 blocks, we are surprised to see that, even with aggressiveness set to as high as 96%, there still exist a few peers who cannot decode, which implies that they have received *linearly dependent blocks*. All previous work (*e.g.,* [7]), however, theoretically maintained that the blocks should be linearly independent with high probability.

Going to one extreme, we have run a simple test with 10 peers and aggressiveness set to 1%, *i.e.,* a peer sends a new coded block to its downstream peer whenever it receives a new coded block from its upstream peers. Fig. 6 shows a simplified

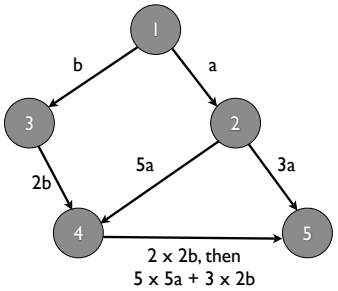

Fig. 6. Linearly dependent blocks may be received by a peer if aggressiveness is below 100%: an example when aggressiveness is 1%, *i.e.,* a peer sends a new coded block to its downstream peer whenever it receives a new coded block from its upstream peers.

version of what we have observed from experiment traces. As the source, peer 1 sends two linearly independent blocks, a and b, to peer 2 and 3, respectively. Peer 2 immediately sends a to its own downstream peers 4 and 5, scaled with two random code coefficients, say $5a$ and $3a$. Peer 3 also forwards $2b$ to peer 4 around the same time. If peer 4 receives $5a$ first, it will immediately send $k \cdot 5a$ to peer 5, which is linearly dependent with $3a$ that peer 5 has already received from peer 2. If peer 4 receives $2b$ first, it will first send $k_1 \cdot 2b$ to peer 5, and then another coded block $k_2 \cdot 5a + k_3 \cdot 2b$ upon receiving $5a$ from peer 2, where k_1, k_2 and k_3 are randomly chosen coefficients. Naturally, the three coded blocks that peer 5 has just received are not linearly independent from one another.

The above example confirms the informal intuition that, when peers are too "eager" to serve other peers with new coded blocks, there are not enough linearly independent blocks to go around the system. Our solution to this challenge is to perform *linear dependence checks* at each peer: when a coded block is received at a peer, we attempt to run it through a dependence check routine. It is discarded if it is linearly dependent with any of the previously received blocks. Our experiences have shown that (details omitted due to space constraints) the computational overhead of such linear dependence checks increases as the number of buffered blocks increases, which implies that such linear dependence checks is only practical if we have a small number of blocks buffered in each peer. In a session with fewer than 300 blocks, we have observed that, when comparing the downloading time of a 13.8 MB file in an one-to-one session, the downloading time is practically the same with or without linear dependence checks. This observation is further confirmed when we re-run our density tuning experiment with linear dependence checks activated. Shown in Fig. 4(b), the average downloading times are unchanged when compared to Fig. 4(a), without the checks. This confirms, in a more realistic topology, that the overhead of linear dependence checks is quite negligible if the number of blocks is small.

Since we only have 100 blocks in the aggressiveness tuning experiment, we run the experiment again, this time with linear dependence checks. Shown in Fig. 5(b), the average downloading times slowly decrease as aggressiveness becomes lower. However, we note that as aggressiveness reaches a certain critical point (around 66%), the number of blocks required to decode takes a sharp turn upwards, which leads to longer, rather than shorter, downloading times. This corresponds to the same point in Fig. 5(a) without checks, as the number of peers that fail to decode increases dramatically around 63 – 66%. Even with 66% aggressiveness, it can only offer a 25% advantage as compared to when aggressiveness is 100%. Nonetheless, the brighter side of the story is that, with aggressiveness at 66% and density at 6%, the performance of network coding with linear dependence checks *approaches* that of naive broadcast (*but still inferior*).

III. Concluding Remarks

This paper is written, for all practical purposes, to present our experiences with network coding in a comprehensive and unbiased manner. We are content with the raw one-to-one bandwidth of coding when the number of blocks is small and when the block size is appropriate, but have noted the presence of linearly dependent blocks when we seek to tune density and aggressiveness to speed it up in more realistic topologies. We have discovered that a coded peer-to-peer session offers poorer performance in terms of downloading times, as compared to the worst possible non-coding protocol, *naive broadcast*.

We believe that network coding offers inferior performance to naive broadcast due to two of its key characteristics. *First*, though encoding takes little computational overhead if we use a low density (such as 6%), decoding still requires full matrix multiplication. *Second*, due to its requirement to buffer a certain number of coded blocks before it can serve new coded blocks, the latencies of such block accumulation are not negligible. Since it is not possible to use a very small aggressiveness value due to linear dependence constraints, such buffering latencies cannot be completely eliminated. We should keep in mind that naive broadcast would never be used in realistic peer-to-peer applications; in reality, BitTorrent has little bandwidth overhead, due to its elaborate protocols performing content reconciliation. If network coding cannot even match naive broadcast, it is hard to imagine it offering the same performance as *BitTorrent*, which is approximately 300% more efficient based on our observations! As such, our conclusion with respect to the benefits of network coding is more cautious and *less optimistic* than previous studies.

References

[1] R. Ahlswede, N. Cai, S. R. Li, and R. W. Yeung, "Network Information Flow," *IEEE Transactions on Information Theory*, vol. 46, no. 4, pp. 1204–1216, July 2000.

[2] R. Koetter and M. Medard, "An Algebraic Approach to Network Coding," *IEEE/ACM Transactions on Networking*, vol. 11, no. 5, pp. 782–795, October 2003.

[3] T. Ho, R. Koetter, M. Medard, D. Karger, and M. Effros, "The Benefits of Coding over Routing in a Randomized Setting," in *Proc. of International Symposium on Information Theory (ISIT 2003)*, 2003.

[4] P. Chou, Y. Wu, and K. Jain, "Practical Network Coding," in *Proc. of Allerton Conference on Communication, Control, and Computing*, October 2003.

[5] C. Gkantsidis and P. Rodriguez, "Network Coding for Large Scale Content Distribution," in *Proc. of IEEE INFOCOM 2005*, March 2005.

[6] C. Gkantsidis, J. Miller, and P. Rodriguez, "Anatomy of a P2P Content Distribution System with Network Coding," in *Proc. of the 5th International Workshop on Peer-to-Peer Systems (IPTPS 2006)*, 2006.

[7] T. Ho, M. Medard, J. Shi, M. Effros, and D. Karger, "On Randomized Network Coding," in *Proc. of Allerton Conference on Communication, Control, and Computing*, October 2003.