Abstract

Network coding is an emerging technology that promises to foster new distribution models for digital content. Network coding has been a popular research topic, but until recently it had not seen widespread uptake in terms of mass usage. The UUSee system [6] is one of the first network coding applications to garner a large user base, as it was used to deliver streaming video on demand to millions of users in China during the Beijing Olympics. Despite its success, the authors of the UUSee system acknowledge some inefficiencies in communication that take place during streaming sessions in their protocol. The work presented in this paper proposes an improvement to reduce these inefficiencies, by using a formula to predict when message segments will be successfully received instead of waiting for explicit acknowledgment messages. This *completion prediction* method is shown to reduce communication redundancies in the UUSee system, and is also potentially applicable in other network coding systems.

1 Introduction

Network coding grew out of the general observation that inserting targeted redundancy in network communication helps avoid some routing and feedback problems that arise in networked systems [2, 7]. The redundancy is in the form of erasure codes, where small pieces of a message are merged into a "coded" message that combines information from all its constituent pieces. Frequently this takes the form of adding the message pieces in a finite field (equivalent to XOR for a 2-element field), which is known as *linear network coding*. For example, if a server sends message pieces A, B, and $A \oplus B$, the receiver is able to recover the original message segments A and B, no matter which two messages he receives. In general, for any message $M = m_1, m_2, m_3, ..., m_n$, we want the senders to be able to send any sequence of n coded messages that enable the receiver to recover the original message M.

The network coding model allows receivers to reconstruct the original message. But what benefits does this model provide? The benefits can vary depending on the application, but typically include robustness to dropped packets (any packet can be replaced with any other) and decentralized coordination (that is, the server only needs to know that the recipient wants a certain message, he doesn't need to know exactly which pieces of the message the recipient has already received).

The UUSee system was designed to take advantage of these properties of network coding, in a streaming Video-on-Demand setting. The basic architecture of UUSee consists of an overlay network that seeds a large number of videos. Each video is divided into 300KB-500KB segments, and each segment is further divided into 1KB blocks. When a user requests a video, a tracker in the overlay network assigns it a collection of nodes to download from, which all send a stream of coded blocks from the desired video segments to the user. The coded blocks are generated using random linear codes, with byte-wise operations performed using the finite field $GF(2^8)$. Streaming packets are sent via UDP, because the redundancy provided by network coding means the user need not worry about individual dropped packets. Once the segment is completely downloaded and decoded, the user's client requests the next segment from its neighbors, and the process repeats.

Perhaps the primary motivation for using network coding in the streaming Video-on-Demand scenario is that it allows each sender/receiver pair to only communicate local information. That is, each sender knows only which segment the receiver requires, and he can stream coded packets from that segment simultaneously to other seeds, without any central coordination, or requiring any acknowledgments.

The UUSee authors describe inefficiencies in their protocol that take place during the transition from one segment to another, which they call *braking redundancy*. Braking redundancy is a

byproduct of their brake messages, which are only generated once a peer has completely decoded a segment. In the time between the last useful packet for segment i arriving and the first useful packet for segment $i+1$ arriving, many redundant packets for segment i will arrive, only to be discarded. Using trace data from the live system, they find that the average redundancy percentage is about 2.5% per segment. They justify this as being fairly low overhead, and only representing 1-2 redundant blocks per server-to-peer stream. However, their choice of 300-500 blocks per segment is expressly designed to minimize this already. A smaller segment size would reduce the CPU load and memory consumption for each peer, since the decoding time for each segment is $O(n^3)$. So if we can reduce the braking redundancy in the system, we can materialize the gains both in terms of faster throughput, less computation for each peer, and finer-grained segments. The authors pose the question whether additional feedback messages can be utilized to reduce or eliminate the braking redundancy, but leave the implementation as future work. I propose adding an additional piece of state to the system – the receivers will now keep track of an "estimated segment finishing time" for each associated sender – which will allow the system to reduce some of the packet loss inefficiencies caused by braking redundancy.

2 System Design

I developed a proof-of-concept system to test whether completion prediction can indeed reduce redundancy in a network coding system. The network coding aspect of my system is modeled on the published description the UUSee system [6], although some details are not provided, such as the specifics of receiver-to-sender communication and the optimizations allowing efficient message decoding.

The communication pattern is formalized in Algorithm 1. The server waits for incoming requests from peers, after which it sends a stream of coded blocks from the requested segment. The blocks are composed using random linear network codes (an $O(n^2)$ process), and the communication overhead is reduced by sending the pseudo-random number generator seed value instead of all the block coefficients. The server continues to stream blocks to the client until it receives a completion message.

The receiver's client program is supplied with the server address, after which it requests the data being served, starting with the initial segment of the file, and enters into the peer loop shown in Algorithm 2. The naive implementation progressively decodes a segment as each packet arrives (see Figure 1), but waits until the entire segment is decoded before sending a request for the next segment to the server. Logic is added to both the server and peer software to enable completion prediction, as shown in Section 2.1.

The implementation described below follows many of the details presented by the authors of the UUSee paper, including their optimizations of arithmetic operations in $GF(2^8)$. However there are still some optimizations that can be made, but for which fewer details are supplied. A viable network coding system will need to take advantage of available hardware optimizations, such as those mentioned by Shojania and Li [8]. They describe SIMD hardware acceleration techniques using standard, off-the-shelf x86 and PowerPC processors. However, the merits of limiting braking redundancy can be tested without such optimizations.

2.1 Progressive Decoding

An important feature of the system that makes it feasible for streaming use is that we can perform some amount of decoding work after each packet arrives, but before all the packets for a segment have arrived. This means that segments can be played immediately upon receipt of their last coded block. It also means that we know, after each packet is received, how many more

linearly independent packets must be received to complete the segment (this value will be useful for completion prediction). Figure 1 displays a simple example of this process, with small segment and block sizes (20 and 5 bytes respectively).

2.2 Completion Prediction

To reduce the braking redundancy found in UUSee, it would be helpful to give the server an estimation of when it should transition to the next segment. To enable this, my implementation of network coding with completion prediction adds another field to the packets being sent from the servers to the clients, and allows the client to send an additional type of packet back to the server.

Instead of the server streaming packets comprising the tuple (*segment index*, *PRNG seed*, *coded bytes*), we add a *server counter* value to each streamed packet. This increases by 1 for each packet the server sends from a segment.

Completion Prediction Packet Contents	
segment index	4 bytes
server counter*	4 bytes
PRNG seed	4 bytes
coded bytes	1024 bytes
* added for completion prediction	

Then, the client uses these values to estimate what the value of the server counter will be when it receives the last useful block of the current segment, and it communicates this estimate back to the server. It computes the estimate by periodically recording a snapshot of the server counter value and the number of degrees of freedom remaining in the segment (i.e., the number of new packets needed to fully decode the segment). Then, it can estimate the server counter value as $est = y + (\Delta y/\Delta x) \times x$, where y is the current server counter value, Δy is the change in the server counter value since the last snapshot, x is the current number of degrees of freedom, and Δx is the change in degrees of freedom since the last snapshot. This is simply a linear extrapolation of the completion time, which gives us an estimate of when the segment will complete, assuming the counter and degrees of freedom increase at the same rate as they did since the snapshot.

The client sends these estimates to the server periodically (the interval δ is set to 50ms in my implementation, but the estimate is only sent if the prediction has changed since the last transmitted estimate). There are multiple reasonable reactions the server might have to make use of this estimate. In my implementation, once the server counter has passed the most recent completion estimation value for segment i, but before it has received a completion acknowledgment from the client for segment i, it randomly selects which segment to send next. Specifically, it sends a coded block from segment i with probability $1 - \epsilon$, otherwise it sends a coded block from the next segment requested by the client (typically segment $i + 1$). My implementation uses $\epsilon = 0.9$.

3 Experiments

To test the effectiveness of completion prediction in this system, I recorded several metrics during the transfer of a file using the system (UUSee with and without completion prediction). The results are shown in Figures 2 and 3.

Figure 2 shows the braking redundancy recorded in four separate testing configurations. Braking redundancy is measured as the percentage of all packets received which belong to an already completely decoded video segment. We would like to minimize this amount to achieve an efficient protocol. Two of the tests are performed without completion prediction and the other two have

Incoming packets

$$\Downarrow$$

$$\text{After packet 1:} \left|\begin{array}{cccc|ccccc} 1 & 109 & 120 & 138 & 10 & 53 & 124 & 45 & 115 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{array}\right|$$

$$\Downarrow$$

$$\text{After packet 2:} \left|\begin{array}{cccc|ccccc} 1 & 0 & 72 & 200 & 241 & 254 & 154 & 22 & 145 \\ 0 & 1 & 57 & 48 & 26 & 35 & 126 & 254 & 166 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{array}\right|$$

$$\Downarrow$$

$$\text{After packet 3:} \left|\begin{array}{cccc|ccccc} 1 & 0 & 0 & 14 & 9 & 153 & 143 & 94 & 73 \\ 0 & 1 & 0 & 206 & 139 & 12 & 87 & 199 & 237 \\ 0 & 0 & 1 & 188 & 34 & 233 & 193 & 1 & 3 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{array}\right|$$

$$\Downarrow$$

$$\text{After packet 4:} \left|\begin{array}{cccc|ccccc} 1 & 0 & 0 & 0 & 72 & 101 & 108 & 108 & 111 \\ 0 & 1 & 0 & 0 & 32 & 119 & 111 & 114 & 108 \\ 0 & 0 & 1 & 0 & 100 & 46 & 32 & 45 & 32 \\ 0 & 0 & 0 & 1 & 77 & 97 & 114 & 99 & 111 \end{array}\right|$$

$$\Downarrow$$

$$\text{Output: } [\ 72 \quad 101 \quad 108 \quad \ldots \quad 114 \quad 99 \quad 111\ ]$$

Figure 1: Progressive decoding example. The decoding matrix is shown 4 times, having been returned to *reduced row echelon form* after each new packet arrives. Each packet represents a linear system of equations with coefficients from the field $GF(2^8)$. The coefficients are shown on the left of the matrix dividing lines. As each packet is received, it is appended to the decoding matrix before applying row operations. The result is the concatenation of all block values (on the right of the dividing line), which is a byte string from the data being transferred.

Algorithm 1 Server Loop

```
Require: ε, 0 ≤ ε < 1
  Initialize array S (holds active segment index for each peer)
  Initialize array SC (holds server counter for each segment)
  Initialize array SC-EST (holds estimated completion SC value for each peer)
  loop
    for all received messages do
      if m is a REQ message from peer i, requesting segment s then
        S_i ← s
      else if m is a EST message from peer i, with completion estimate e then
        SC-EST_{S_i} = e
      end if
    end for
    for all active file segments s do
      SC_s ← SC_s + 1
      Randomly select seed_s for PRNG
      Generate coded block b_s using seed_s
    end for
    for all active peers i do
      r ← random_float_between(0, 1)
      if (SC_{S_i} < SC-EST_{S_i}) or (r < (1 − ε)) then
        s ← S_i
      else
        s ← S_i + 1
      end if
      send message m = (s, SC_s, seed_s, b_s) to peer i
    end for
  end loop
```

Algorithm 2 Peer Loop

```
Require: δ > 0
  for each received message m = (s, SC_s, seed_s, b_s) do
    if segment s is already complete then
      m is redundant. Discard it.
    else
      Add b_s to decoding matrix for segment s and perform progressive decoding
      DOF_s ← degrees of freedom in decoding matrix
    end if
    if DOF_s = 0 (i.e., segment s is complete) then
      Mark segment as complete and write decoded segment to file.
      Send REQ message to active servers, requesting segment s + 1.
    end if
    if time > last_time + δ then
      SC-EST ← SC_s + (SC_s − last_SC_s)/(DOF_s − last_DOF_s) * DOF_s
      Send EST message to active servers, with completion estimate SC-EST
      last_time ← time
      last_SC_s ← SC_s
      last_DOF_s ← DOF_s
    end if
  end for
```

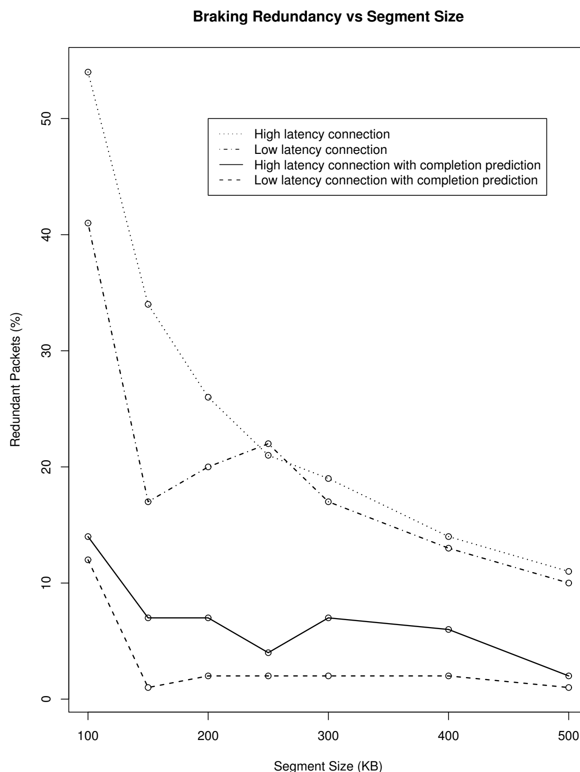

Figure 2: Braking redundancy of a network coding client. The top two lines use the naive braking method and thus experience a high level of redundancy. The lower lines use predictive braking to estimate completion times, resulting in lower levels of redundancy. The low latency tests were performed on a local network (avg. latency: 0.2ms) and the high latency tests over the Internet (avg. latency: 52ms)

completion prediction enabled. Each was tested between a pair of nodes on a low latency network (with an average round trip time of 0.2ms) and a higher latency connection over the Internet (average RTT of 52ms). The tests were performed with a range of segment sizes, ranging from 100KB to 500KB. For comparison, the UUSee system runs with variable segment sizes, ranging from 300KB-500KB depending on the video's bit-rate.

For the tests without completion prediction, the rates of braking redundancy range from 10.71% for a segment size of 500KB over the low latency link to a very high 54.10% for a segment size of 100KB over the high latency link, which are all considerably higher than the 2.5% value reported by the UUSee authors [6]. This may have several causes. Namely, as mentioned in Section 2, my implementation does not include hardware acceleration techniques used by UUSee, so the decoding rate may be considerably slower than the official UUSee implementation, which may lead to delayed feedback from the receiver to the sender, thereby introducing latency in the segment completion notifications. Also, the exact nature of braking messages is not fully specified by the UUSee authors, so my implementation uses a UDP message, which is repeated every 50ms until the segment counter is incremented. This combination of factors may explain the discrepancy.

Despite any disadvantages inherent in my implementation, the tests of a client and server using

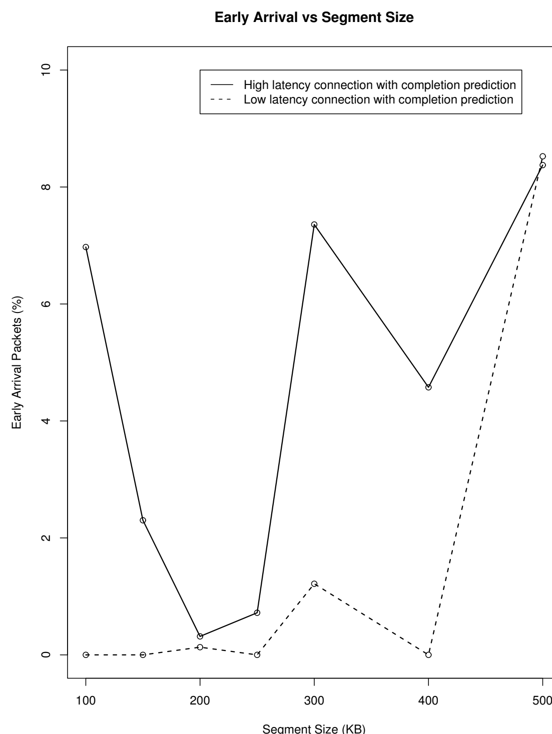

Figure 3: Packets arriving while a previous segment is still incomplete. Ideally the percent of early arrivals will remain low when using completion prediction, so that segments finish quickly and the streaming rate is not affected. The low latency tests were performed on a local network (average latency: 0.2ms) and the high latency tests over the Internet (average latency: 52ms)

completion prediction performed much better in terms of braking redundancy. For segment sizes above 100KB, the low latency results are all between 1.43% and 2.78%, while the high latency results fall between 2.63% and 7.87%. Moreover, they remain relatively flat between 150KB and 500KB, which indicates that smaller segment sizes can be used without severely harming system efficiency.

Figure 3 measures the orderliness of the packets in tests with completion prediction enabled, which creates the possibility of the sender transmitting blocks from a segment before the receiver has completely finished the previous segment. The results here are less conclusive, but still indicate that only a relatively small fraction of packets from any segment i are received before all segments j are finished where $j < i$. The correlation to segment size also appears to be small. However, a more meaningful metric is whether this rearrangement of packet arrival order would affect video playback. A careful study of the latency introduced by packet reordering is left as future work.

4 Related Work

The use of feedback channels in the context of network coding scenarios has been previously explored in some circumstances [1, 3, 4, 9, 10, 11].

Bakshi and Effros assume unlimited feedback from sinks (receivers) to source [1]. They analyze the problem from an information-theoretic perspective, showing that the bounds on the sending

rates can be improved for certain network topologies by adding a feedback channel.

Fragouli et al. illustrate the use of a feedback channel as a method for parameter adaptation [4], including a description of a system in which receivers need to decode data within a preset delay, or the segment/generation is useless, similar to UUSee. They also discuss the throughput improvement gains that result from limiting the receiver communication to a single ACK at the end of transmission to signal "stop transmission" sends $1/(k*(1+\epsilon))$ as many feedback packets as a scheme that sends one after each packet. They discuss the trade-offs that extra feedback introduces to a network coding system.

Drinea et al. show that network coding with feedback outperforms, in terms of delay, both a Forward Error correction scheme and a normal scheduling scheme for broadcasting applications without network coding [3].

In addition to single-seed delivery networks, network coding has also been put to use in peer-to-peer content distribution networks. The addition of network coding to previous peer-to-peer systems provides an increase in the likelihood that a packet sent from one peer will be useful to another. Avalanche implements this in a system similar to BitTorrent [5]. However, since they do not use segments to subdivide their files, the application of predicting completion times to such systems is not directly apparent.

5 Future Work

The completion prediction method reduces braking redundancy, but adds the server counter to every packet. The overhead is relatively small, as it contributes only 4 bytes to a 1032 byte packet for 1KB blocks. However, this overhead may be reduced. For example, if the PRNG seeds follow a known sequence, the server counter values in received packets could be used by the receiver to regenerate the PRNG seed values, which means the 4 byte PRNG seed values could be removed from the packets.

Completion prediction, as described in this paper, relies on two parameters: δ and ϵ. Values for these parameters were hand-selected for the experiments in this paper, but fine-tuning the values could have an even greater effect on the results. Additionally, instead of using a constant value for ϵ, a function of the server counter could be used instead.

Another direction for future enhancements would be to directly compare with the performance of UUSee by implementing the peer-to-peer overlay and the hardware-accelerated matrix operations mentioned by the UUSee authors [8].

6 Conclusion

The completion prediction method presented here has the potential to greatly reduce the braking redundancy present in network coding systems like UUSee. The modification to the UUSee protocol is minor. The receiver maintains a past snapshot of the download progress and server counter, which it uses to predict when the sender can transition to serving the next segment and sends this prediction to the sender. The only change to the sender, aside from appending a short counter to all messages, is to accept these predictions from the receiver and transition to a probabilistic phase between the predicted completion time and the confirmed completion time. However, this small change was shown to greatly reduce the redundant braking messages in a test environment, and may be able to provide similar benefits to the real UUSee system.

References

[1] M. Bakshi and M. Effros. On feedback in network source coding. In *ISIT'09: Proceedings of the IEEE International Symposium on Information Theory*, pages 1348–1352, June 2009.

[2] J. W. Byers, M. Luby, M. Mitzenmacher, and A. Rege. A digital fountain approach to reliable distribution of bulk data. In *SIGCOMM'98: Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, Vancouver, British Columbia, Canada, August 1998.

[3] E. Drinea, C. Fragouli, and L. Keller. Delay with network coding and feedback. In *ISIT'09: Proceedings of the 2009 IEEE international conference on Symposium on Information Theory*, pages 844–848, Piscataway, NJ, USA, 2009.

[4] C. Fragouli, D. Lun, M. Médard, and P. Pakzad. On feedback for network coding. In *CISS'07: Proceedings of the 41st Annual Conference on Information Sciences and Systems*, pages 248–252, March 2007.

[5] C. Gkantsidis and P. Rodriguez. Network coding for large scale content distribution. In *INFOCOM'05: Proceedings of the 24th Annual Joint Conference of the IEEE Computer and Communications Societies*, pages 2235–2245, Miami, FL, March 2005.

[6] Z. Liu, C. Wu, B. Li, and S. Zhao. UUSee: Large-scale operational on-demand streaming with random network coding. In *INFOCOM'10: Proceedings of the 29th Annual Joint Conference of the IEEE International Conference on Computer Communications*, pages 2070–2078, San Diego, CA, March 2010.

[7] L. Rizzo and L. Vicisano. A reliable multicast data distribution protocol based on software FEC techniques. In *HPCS'97: Proceedings of the Annual Conference on High Performance Computer Systems*, pages 23–25, 1997.

[8] H. Shojania and B. Li. Parallelized network coding with hardware acceleration. In *IWQoS'07: Proceedings of the 15th International Workshop on Quality of Service*. IEEE, June 2007.

[9] J. K. Sundararajan. *On the role of feedback in network coding*. PhD thesis, Massachusetts Institute of Technology, 2009.

[10] J. K. Sundararajan, D. Shah, and M. Médard. Feedback-based online network coding. Submitted to IEEE Transactions on Information Theory, April 2009.

[11] J. K. Sundararajan, D. Shah, M. Médard, M. Mitzenmacher, and J. Barros. Network coding meets TCP. In *INFOCOM'09: Proceedings of the 28th Annual Joint Conference of the IEEE Computer and Communications Societies*, pages 280–288, Miami, FL, April 2009.