Abstract—TCP over wireless networks is challenging due to high random losses and data-ACK interference. To address both problems, a new hybrid network coding scheme, ComboCoding, is proposed. ComboCoding combines intra-flow and inter-flow coding to provide robust communications over TCP in disruptive wireless networks. Intra-flow coding mitigates losses by using a modified random linear coding scheme that introduces adaptive redundant transmissions. The channel efficiency is enhanced by introducing also inter-flow coding, which performs opportunistic XOR of TCP data and ACK flows. Simulation results show that under high loss scenarios, ComboCoding delivers the highest throughput and achieves acceptable overhead. ComboCoding also features a novel adaptive redundancy control algorithm that adapts to unstable lossy links.

Keywords—Network Coding, Random Linear Coding, XOR Coding, TCP, Wireless Multihop, Lossy Channels

I. INTRODUCTION

The Transport Control Protocol (TCP) is the most commonly used reliable transport protocol in the Internet. In addition to end-to-end reliable transmission, TCP also provides fair congestion control for better sharing of network resources. Based on how it detects congestion, the control algorithms in TCP are categorized as loss-based or delay-based congestion control. Among all TCP variants, the most well-known and most widely-used is TCP-NewReno, which adopts a loss-based congestion control algorithm. A loss event is inferred by the source upon receiving 3 duplicate acknowledgements (ACK), which are sent by the TCP destination whenever it receives a DATA packet with non-consecutive sequence number. TCP-NewReno assumes packet losses are due to router buffer overflow. This assumption was always true in the environment it was originally designed for, the wired Internet, where most links are point-to-point.

However, the assumption that buffer overflow is the only reason behind packet loss no longer holds in wireless multihop networks, where a significant amount of loss is due to the interference and unpredictable wireless links quality. It has been shown that in wireless multihop scenarios, TCP significantly suffers from misinterpreting random errors as congestion. Also, TCP ACK and DATA contend for the shared wireless medium, which causes self-induced collisions. However, there are several approaches to address these two issues separately, including other forms of congestion control, tuning TCP parameters or protocol optimization.

One method to improve loss-based congestion control in wireless networks is to deploy Loss Discrimination Algorithms (LDA) [1-4]. Another is to optimize TCP parameters. [5] pointed out TCP congestion window to be key in improving TCP performance in wireless networks. [6] went further to show that controlling the maximum congestion window size is even more effective. Yet another optimization is to reduce the interference between ACKs and DATA packets by reducing the ACK frequency [7]. Intelligently controlling the so-called “Delayed Acks” can reduce ACK packets in the network, thus reducing interference at the inter-flow level.

A recently proposed approach to help TCP in wireless networks is to exploit network coding. Network coding has been proposed in the past for a variety of network environments and traffic scenarios [8]. [9-10] further enhanced the practical design of coding packets by introducing random linear coding. It has been shown that network coding helps in alleviating wireless interference by reducing the number of transmissions in multicasting or multi-flow environments [11-12]. In addition, network coding also helps in effectively overcoming high loss rate in wireless networks for unicast traffic as shown in [13-15].

Nevertheless, coding overhead has so far been a significant problem as pointed out in publications on the subject. In [11], the authors point out that network coding does not significantly improve TCP. The authors in [16] claim that this is mainly because the study in [11] does not take into account bi-directionality, i.e., the fact that TCP DATA flow and TCP ACK flow are naturally in opposite directions. Therefore, they propose to XOR TCP DATA and ACK opportunistically to reduce transmissions. This is what we refer to as inter-flow coding, i.e., coding of two flows is in opposite directions. A similar idea is proposed in [17], where the throughput is improved by opportunistically XORing the TCP ACK and DATA flow. Following the work presented in [17], [18] proposed a MAC layer modification to provide a better channel
access scheme. One of the earliest proposals to improve TCP in lossy networks is [19], in which authors studied an intra-flow random linear coding scheme. Intra-flow implies that only packets within one flow are coded, which in this case is the data flow. As a result, TCP is significantly improved, but their scheme requires TCP modifications at both the senders and the receivers.

All the above solutions address either the interference problem or the high loss problem, but not both. In this paper, we present a hybrid network coding scheme that is (1) transparent to TCP, and (2) addresses both interference and random loss problems. The proposed coding scheme, ComboCoding, provides a robust, comprehensive solution for TCP in lossy wireless scenarios. ComboCoding conceptually combines the idea of PiggyCode [17], an implementation of inter-flow coding, and a revised version of intra-flow random linear coding [20-21]. The proposed scheme opportunistically encodes the ACK flow and DATA flow together to reduce interference, exploits packet redundancy, and further improves TCP performance.

The contribution of ComboCoding is four-fold: (1) ComboCoding combines inter- and intra-flows coding to address both high loss rate and self-interference; (2) ComboCoding features an novel, adaptive redundancy algorithm that effectively handles transient, unstable link conditions; (3) ComboCoding is implemented in the network layer and is transparent to TCP and other reliable protocols at the upper layers; (4) ComboCoding does not rely on any new or modified MAC layer protocols. This is different from [18], where the authors propose a MAC layer modification to further improve coding gain. We aim to provide a solution that is transparent and requires no hardware or software modifications to the MAC layer.

The rest of the paper is organized as follows. Section II reviews the fundamentals of the underlying coding schemes in ComboCoding. The design and implementation of our protocol are discussed in Section III. Section IV evaluates the performance of ComboCoding and the paper is concluded in Section V.

II. RELATED WORK

This section provides a background overview of the underlying coding schemes in ComboCoding. We first specify a general network coding scheme that uses a random linear coding approach. Secondly, we introduce a modified version of network coding, Pipeline Coding, which is compatible and transparent to TCP. Lastly, the concept of PiggyCoding is illustrated. Table I below provides a summary of terminology we adopt in this paper.

A. Pipeline Coding

The idea of Pipeline Coding has been introduced in [20], where the goal was to reduce the overall coding delay. While this scheme still uses the concept of packet generations, encoding and decoding occur progressively.

Assuming an application produces a sequence of equal-sized packets $p_1, p_2, p_3, \ldots$. Let $k$ be the number of packets in a single generation. A coded packet $c$ in the $i$th generation is then defined as:

$$c = \sum_{j=1}^{m} c_j p_{i+k+j}$$

where $m$ is the number of data packets currently in the generation buffer, $c_j$ is randomly selected from a particular Galois field $\mathbb{F}_2^{12}$, and $t \times k$ is the total number of packets transmitted before the $i$th generation. Throughout this article, we use lowercase boldface letters to denote vectors, frames, or packets, uppercase letters to denote matrices, and italics to denote variables or fields in the packet header. Every arithmetic operation is over $\mathbb{F}_2^{12}$, so that data packets $p_i$ and coded packets $c$ are also regarded as vectors over $\mathbb{F}_2^{12}$. Conceptually, Eq. (1) says that upon receiving a new data packet, the source will instantly trigger the encoding process based on the currently received data packets. Let $r$ denote the source coding redundancy, where $r \geq 1$ so that for each generation the source produces $k \times r$ coded packets. If all coded packets are delivered successfully, destinations can construct the following lower triangular matrix without any extra computation:

$$\begin{bmatrix} c_1 \\ \vdots \\ c_k \\ \vdots \\ c_{m-k} \\ \vdots \\ c_{m-1} \\ c_m \end{bmatrix} = \begin{bmatrix} e_1^{(1)} & 0 & \cdots & 0 \\ \vdots & \ddots & \vdots & \vdots \\ e_k^{(1)} & \cdots & e_k^{(m-k)} & 0 \\ \vdots & \vdots & \ddots & \vdots \\ e_{m-k}^{(1)} & \cdots & \cdots & e_{m-k}^{(m-1)} \end{bmatrix} \begin{bmatrix} p_1 \\ \vdots \\ p_k \end{bmatrix}$$

(2)

The above linear equation (2) can be solved progressively without waiting for generation completion. For example, upon receiving $c_i$, destinations can decode $p_i$, and so on. Relays also participate in reencoding, as explained in [22], in order to minimize the information loss per hop. In addition, relays also have a forwarding redundancy to determine how many reencoded packets to transmit upon receiving each innovative packet. A feedback-based redundancy control algorithm for Pipeline Coding is introduced and will be discussed in Section III.

Fig. 1 shows an example of Pipeline Coding, with generation size $k = 4$ and source coding redundancy $r = 1.25$. Packet re-encoding part is not shown for simplicity. Data packets are encoded instantly upon arrival and decoded immediately at the destination. This allows Pipeline Coding to avoid triggering the retransmission timeout of DATA packets, which is critical to TCP. This is the major reason why Pipeline Coding is compatible with TCP, while conventional batch network coding is not.

In addition, Pipeline Coding can partially recover a subset of the data packets in a generation and deliver them to the upper layer. This is a significant difference from batch-based coding, which either delivers an entire generation to the upper layer or discards the whole generation. For example, assuming that $c_{10}$ of Fig. 1 is lost, data packet #7 and #8 will never have a chance to be decoded, regardless of which coding scheme is used. However, without Pipeline Coding, none of the data packets in the 2nd generation can be decoded, whereas with Pipeline Coding, we can still decode data packets #5 and #6.
Note that Pipeline Coding does not require any TCP modification since it functions in the network layer, which is the major difference between Pipeline Coding and the coding scheme proposed in [19]. The approach in [19] encodes all packets in the congestion window and redefines the semantic of TCP acknowledgements, while Pipeline Coding uses its own coding generation buffer that has no relationship with the TCP congestion window. However, as Pipeline Coding does not provide reliable transmission, TCP still needs to retransmit lost packets in the presence of a loss or timeout event.

<table>
<thead>
<tr>
<th>Term</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Network Coding</td>
<td>Source-side and relay coding.</td>
</tr>
<tr>
<td>Batch Coding</td>
<td>Generation based coding scheme.</td>
</tr>
<tr>
<td>Pipeline Coding</td>
<td>Coding scheme that incrementally encodes and decodes once a new packet arrives</td>
</tr>
<tr>
<td>Generation</td>
<td>A set of packets that are encoded or decoded together</td>
</tr>
<tr>
<td>Coding Vector (Encoding Vector)</td>
<td>A vector of coefficients that reflect the linear combination of data packets</td>
</tr>
<tr>
<td>Rank (Degree of Freedom)</td>
<td>Number of linearly independent packets in the generation</td>
</tr>
<tr>
<td>Innovative Packet</td>
<td>A packet that increases the rank of a generation</td>
</tr>
<tr>
<td>Coding Redundancy</td>
<td>Number of coded packets sent per generation divided by generation size</td>
</tr>
<tr>
<td>Delay</td>
<td>The time difference between packet reception by destination application and generation at source application</td>
</tr>
</tbody>
</table>

### III. PROTOCOL DESIGN

As mentioned previously, ComboCoding consists of two different types of network coding: inter-flow coding and intra-flow coding. Throughout this paper, we refer to inter-flow coding as a modified version of PiggyCode, and the intra-flow coding as Pipeline Coding. We also refer to a PiggyCoded packet as the result of a bitwise XOR of TCP DATA and TCP ACK. As explained above, the main goal of performing an XOR is to deliver both packets in one transmission. The major difference between the modified PiggyCode implementation and the original design is that we do not rely on a MAC layer buffer. For transparency to the MAC layer and to avoid modifying MAC standards, we introduce a buffer in the network layer that queues DATA packets for a given time $T$. If an ACK happens to arrive when there exists at least one DATA packet in this buffer, the network layer module XORs both packets and sends a PiggyCoded packet to the lower layer. If no ACK arrives within this time $T$, DATA packets are forwarded normally, and the buffer is freed.

For intra-flow coding we use Pipeline Coding, because to the best of our knowledge it is the best random linear coding scheme that is compatible and transparent to TCP. However, as a tradeoff, Pipeline Coding requires a little higher redundancy to mitigate losses. A detail discussion is given in the simulation results section.

We implemented the proposed ComboCoding in QualNet 4.5 [23] as a “routing protocol”. The following sections first present the logic followed by the source, destination, and relays. A redundancy control algorithms is then given, followed by a brief discussion of the chosen (standard) channel access scheme.
A. Coding Flow Charts

Fig. 2 below shows the coding flow chart at a source. If the packet is from the upper layer, the source has no chance to XOR anything, and it forwards the packet to the Pipeline Coding module. This module will generate the desired number of redundant packets and deliver them to the lower layer. Meanwhile, all generated coded packets are stored in a local limited-capacity buffer so that they can be later used in decoding PiggyCoded packets. The source module has a chance to receive a PiggyCoded packet, since PiggyCoded packets by default will carry the source and destination IP address of the DATA flow. Therefore, the PiggyCoded packet might be delivered to the source, in which case it will be forwarded to the destination handler function instead.

Fig. 2 Coding Flow Chart at Source

Fig. 3 gives the decoding flow chart at the destination. If the packet is PiggyCoded, it has to be decoded from the packet previously sent and stored in the buffer. If the packet is not found in the buffer for any reason, the packet will be dropped. After decoding the PiggyCoded packet, ComboCoding first examines whether the received packet is innovative or not, which is determined by examining the packet’s coding coefficient vector. If the packet has a linearly independent coding coefficient vector, then it is linearly independent to all received coded packets in the same generation. ComboCoding will then store and decode the packets and deliver them to the upper layer.

Fig. 3 Coding Flow Chart at Destination

Fig. 4 shows the coding flow chart for relay nodes, which has the same PiggyCode decoding and Pipeline Coding innovative checking as the destination. A PiggyCoded packet will first be decoded and delivered to the Pipeline Coding module. If the received data packet is innovative, it then examines whether the packet is marked as “queued” (up to a timeout $T$). In our simulations, TCP source always marks DATA as queued and TCP destination always marks ACK as NOT queued. ACKs are not queued at intermediate nodes since TCP needs feedback to be delivered as soon as possible. This implies that if the ACK cannot find a DATA packet to XOR, it will be transmitted as an explicit ACK (no piggybacking).

If the packet is an ACK, it proceeds to look up the PiggyCode queue. If any DATA that has not yet been mixed is in the queue, the ACK cancels the timer for that DATA packet and performs an XOR to generate a PiggyCoded packet, which is then sent out.

Fig. 4 Coding Flow Chart at Relay
B. Redundancy Control Algorithm

Since the link quality in wireless networks varies significantly over time, we propose a feedback-based redundancy control algorithm to dynamically control the coding and forwarding redundancy. In wireless multihop communications, the redundancy should ideally be set to 1/(1-p), where p is the link loss probability. Therefore, it is crucial to estimate the loss rate for each link in order to adaptively adjust the redundancy. In our experiment, we found that the TCP ACK flow must use the same redundancy as the TCP DATA flow due to the use of symmetric links. Therefore, the following algorithm estimates only the loss rate on the TCP DATA flow. Note that the algorithm is specifically for a single path (string) topology.

To estimate per link loss rate, we first add a field in the header of each coded packet to track the number of coded packets received in the corresponding generation at node i, which is denoted by N_i. This N_i will be carried in the TCP ACKs, in addition to the reencoded TCP DATA packets. Nodes receive reencoded TCP DATA packets by overhearing neighbor nodes, and receive TCP ACKs through unicasting. Assuming node i+1 is the next hop of node i in the TCP DATA flow, the instantaneous link loss rate from node i to node i+1 is estimated as follows:

\[ P_i^0 = \frac{M_i - N_{\text{ack}}}{M_i}, \]  

where M_i is the number of reencoded packets sent from node i to node i+1, which is recorded locally at node i.

Since the loss rate may vary significantly over time, a smoothed loss rate is calculated by taking the exponential moving average of the instantaneous link loss rate as follows:

\[ \overline{P_i} = \alpha \times (P_i^0) + (1 - \alpha) \times \overline{P_i}, \]  

where \( \alpha \) is the smoothing factor, which is set to 1/6 in our simulation. The redundancy for the link from node i to node i+1 is thus estimated as follows,

\[ R_i = (K - 1) + \frac{1}{1 - \overline{P_i}}, \]  

where \( K_i \) is the base redundancy that is needed at node i in the absence of losses. \( K_i \) is used to introduce extra redundancy to recover packets that have been lost and to compensate future potential packet losses. In our simulation, \( K_i \) is set to 1.4.

C. Channel Access Scheme

The choice of channel access scheme is critical in changing wireless network behavior. Since ComboCoding is designed specifically for delivering TCP traffic, it is important to have lower layer reliability support to reduce losses due to MAC layer collisions. Consequently, Pseudo-Broadcast was chosen in our implementation. In other words, nodes unicast a coded packet to an intended receiver, and all nodes are set in promiscuous mode. As mentioned previously, the original PiggyCode is limited by the lack of dual-ACK support in the MAC layer [18]. Therefore, it is important to choose the intended receiver such that the lack of dual-ACK has minimal impact on TCP. Since TCP ACKs are cumulative and DATA is not, all PiggyCoded packets in ComboCoding are destined for the next hop of the DATA flow.

RTS/CTS is also an important issue in configuring a desired channel access scheme. A previous study in [24] suggested that RTS/CTS is not effective in ad hoc networks, as it introduces overhead while not entirely helpful in preventing hidden terminals. Similar observations are found by most of the network coding work in [11-12, 17]. As a result, ComboCoding disables RTS/CTS, and our simulations confirm that this choice provides better performance.

IV. SIMULATION RESULTS

The proposed ComboCoding is tested on QualNet 4.5 [23], where the ComboCoding module is implemented in the network layer. Therefore, the PiggyCode mentioned in this section is a modified version that works in the network layer. The simulation topology is given in Fig. 5, which is a 3-hop string topology. Each node is 250 meters apart, and the Physical and MAC layers are standard 802.11g. RTS/CTS is disabled by configuring a large RTS/CTS threshold in QualNet. As discussed in section 3, each node communicates with each others by pseudo-broadcasting. The transport layer protocol is TCP-NewReno and the application traffic is a Generic-FTP, which will always keep TCP occupied. Table 2 summarizes the configuration of the simulation described above.

![Fig. 5 Simulation Topology](image)

<table>
<thead>
<tr>
<th>Table 2 Simulation Configuration</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Parameter</strong></td>
</tr>
<tr>
<td>Node Distance</td>
</tr>
<tr>
<td>Channel Bit-rate</td>
</tr>
<tr>
<td>Channel Access Control</td>
</tr>
<tr>
<td>RTS/CTS Disabled</td>
</tr>
<tr>
<td>Transport and Application Layer</td>
</tr>
<tr>
<td>Per Link Packet Loss Rate</td>
</tr>
<tr>
<td>Packet Size</td>
</tr>
<tr>
<td>Generation Size</td>
</tr>
</tbody>
</table>

A. ComboCoding Evaluation

In this set of simulations we assume RTS/CTS is disabled, and an optimal PiggyCode timer of 4ms is used for the cases that PiggyCode is enabled. Both Pipeline Coding and ComboCoding use a generation size of 16 packets in random linear coding. The dynamic redundancy control algorithm is turned off in this set of simulations in order to demonstrate the performance gain of coding without being affected by other factors.
Fig. 6 presents the throughput-to-loss curve of TCP-NewReno without any coding, with PiggyCode, with Pipeline Coding, and with ComboCoding. In Fig. 6, we notice that in the absence of random losses, PiggyCode outperforms all other schemes as it reduces interference without introducing significant coding overhead. Pipeline coding performs the worst, because in order to reduce coding delay, it adopts a non-uniform inclusion of original packets into a coded packet. Specifically, the number of transmission is not equal for each uncoded packet within the same generation. For example, the first uncoded packet has the highest chance to be included and transmitted in coded packets, but the last uncoded packet has only 1 chance. Due to this property, Pipeline Coding requires a relatively higher redundancy as discussed in the original paper [20], where it was shown that a coding redundancy of 2.5 was needed. A higher redundancy implies more interference, and thus low loss rate cases it does not perform any better than non-coded TCP-NewReno. This anomaly can be addressed by modifying Pipeline Coding to eliminate the non-uniform inclusion of original packets. However, this is a problem that we are working on and will report on in future work.

However, under low random loss, ComboCoding still achieves the same amount of throughput since PiggyCode reduces the overhead of redundant transmissions. Because of PiggyCode, in the low loss cases where Pipeline Coding performs worst, ComboCoding does not suffer from the same redundancy it needed by using Pipeline Coding. It will be shown later in the overhead evaluation that PiggyCode significantly reduces the transmission overhead introduced by Pipeline Coding. This is the major reason for the improvement in the case of low random error rate.

As the packet error rate increases, the performance of TCP-NewReno with no coding help deteriorates, particularly after the loss rate increases beyond 30%. This is because without redundant packet transmission, TCP throughput is inversely proportional to the square root of the packet loss rate as shown in [25]. With the use of redundant packets, both Pipeline Coding and ComboCoding are more robust to losses. Most importantly, ComboCoding is greatly helped by PiggyCode, and the throughput is consistently higher than in the Pipeline Coding case by 10%–100%.

Note that both Pipeline Coding and ComboCoding are configured with the optimal coding redundancy that is experimentally discovered in a reasonable parameter space. Theoretically, the coding redundancy is a function of packet loss. Previous work in [12], ETX [26] is used to estimate how many packets have to be sent to make 1 successful delivery. However, since packets in Pipeline Coding do not have equal chances to be transmitted as explained previously, it needs a relatively higher coding redundancy [20]. This reflects in the transmission overhead that will be discussed later.

---

1 We measure throughput in the application layer, so all the graphs are actually showing TCP GOODPUT rather than throughput.

---

We next evaluate the delay performance as shown in Fig. 7. We observe that the delay for all cases is a function of loss rate, where a higher loss rate results in a higher delay. The intuition behind this is the more packets get lost, the more time it takes to deliver a packet to the destination. We also notice that for all cases, once the throughput drops to almost zero, the delay increases dramatically. Since ComboCoding still achieves about 400Kbps under 50% packet loss rate, the delay of ComboCoding never increases beyond 2 seconds and is consistently lower than the other cases.

As network coding relies on redundant packet transmission to compensate for packet losses, it is important to evaluate the transmission overhead of ComboCoding. The transmission overhead is designed to address the impact of reducing interference with PiggyCode packets, so we define the term “transmission overhead” as:

\[
\sum_{i=1}^{N} S_{it} / D,
\]

where \( N \) is the total number of nodes, \( D \) is the total number of DATA packets received by the TCP destination, and \( S_{it} \) is the
number of signals physically transmitted by node $i$. This metric is based on the signals transmitted in the physical layer rather than packets sent in the network layer. This is because the reduction of transmitted packets also implies a lower probability of interference with other nodes. Consequently, packet collision probability is reduced and thus fewer signals need to be retransmitted. The number of physical signals transmitted is obtained from the simulation statistic reported by QualNet. Eq. (6) is the average number of physical transmissions needed per each successful TCP DATA packet delivery.

As mentioned previously, Pipeline Coding requires a relatively high coding redundancy, and consequently results in the highest transmission overhead as shown in Fig. 8. In the case of perfect links, Pipeline Coding still needs 18 transmissions in order to deliver 1 TCP DATA packet, which explains why it has the worst throughput when no loss is present. In contrast, PiggyCode has low overhead since it does not introduce any redundant packets, and further attempts to reduce transmissions by mixing DATA and ACK opportunistically.

Without random loss, TCP-NewReno still needs 15 transmissions for a single DATA packet delivery, because TCP source and TCP destination are hidden from each other. Potential collisions significantly increase the average number of transmissions required per successful packet delivery. The power of PiggyCode is shown in the low loss rate cases, where both PiggyCode and ComboCoding reduce the number of transmissions by 50%. The overhead of ComboCoding eventually increases, because the optimal coding redundancy requires more coded packets to be sent to recover more losses. From figures 6 and 8, ComboCoding is shown to be robust to losses while reducing redundant packet transmission overhead by up to 30% when compared to Pipeline Coding.

**B. Redundancy Control Evaluation**

We next evaluate the performance of our redundancy control algorithm by comparing TCP-NewReno under different loss rates in the following configurations: without coding, PiggyCode, Pipeline Coding (with and without redundancy control), and ComboCoding (with and without redundancy control). The application starts sending packets at time 20 seconds, and during time 20–50 the packet error rate for all links is 0%. As shown in Fig. 9, TCP-NewReno and PiggyCode outperform all other coding schemes with perfect links. Pipeline Coding and ComboCoding perform worse because of the extra overhead due to the redundancy of random network coding. The two configurations with the adaptive redundancy control algorithm perform slightly worse than those without, because the adaptive algorithm reacts to short-term losses and thus takes time to lower redundancy.

In the interval from 50 to 80 seconds, a 40% packet error rate is introduced to every link. During this interval, all TCP variants without adaptation drop to almost zero throughput, while the adaptive configurations still achieve around 1Mbps. Moreover, adaptive ComboCoding delivers about 20% higher throughput than adaptive Pipeline Coding.

From 80 to 110 seconds, the per link packet error rate is lowered from 40% to 20%. We notice that both TCP-NewReno and PiggyCode have the highest instantaneous throughput, but are both very unstable due to the lack of network coding redundancy. Moreover, Pipeline Coding and ComboCoding without adaptive control take a longer time to stabilize to the same level as the adaptive versions. This is because random linear coding needs time to discard undecodable generations resulting from loss. In contrast, adaptive Pipeline Coding and ComboCoding instantly adapt to the loss rate. As seen before, ComboCoding delivers about 20% higher throughput than Pipeline Coding at steady state.

**V. CONCLUSION**

In this paper, we present a novel coding scheme, ComboCoding, which combines intra-flow Pipeline Coding and inter-flow PiggyCode. By exploiting the advantage of intra-flow coding, ComboCoding reduces the interference between data and ACKs in the same TCP session while remaining robust to link loss rates. The simulation results show that in a 3-hop string topology, ComboCoding successfully achieves 400Kbps throughput with 50% packet error rate per link. Moreover, compared to the original Pipeline Coding,
ComboCoding reduces transmission overhead by 30% under perfect link conditions and by 10% overhead in most other cases. By combining the benefits of intra- and inter-flow coding, ComboCoding achieves the highest throughput and lowest delay in all lossy scenarios. An adaptive redundancy control algorithm is also introduced to adapt to unstable links with very promising preliminary results. Generalized topologies, better optimized coding schemes, analytic models, and measurements experiments remain to be explored and will be in the subject of future studies.

REFERENCES