UNIVERSITY OF CALIFORNIA

Los Angeles

TCP in Error-Prone, Intermittent MANETs: Exploiting Codes and Multipath

A dissertation submitted in partial satisfaction of the

requirements for the degree Doctor of Philosophy

in Computer Science

by

Chien-Chia Chen

2012

Abstract of the Dissertation

TCP in Error-Prone, Intermittent MANETs: Exploiting Codes and Multipath

by

Chien-Chia Chen

Doctor of Philosophy in Computer Science

University of California, Los Angeles, 2012

Professor Mario Gerla, Chair

Wireless communications over error-prone, intermittent networks are challenging, and prevent many applications from working properly. Network coding has been successful in providing low-cost robust communications in such challenged wireless networks. Nevertheless, existing schemes do not take into account constraints from applications, such as delay for streaming applications and for TCP timeouts, fairness, and data reordering. This study aims to bridge the gap between coding protocols and applications. We consider two types of important applications, multicast streams and unicast TCP. In the context of multicast streams, we propose a novel coding scheme, Pipeline Coding. Compared to the conventional batch coding scheme, our analysis shows that Pipeline Coding improves both packet delivery ratio (PDR) and end-to-end delay. Pipeline Coding is also shown via simulations and testbed experiments to improve the throughput, and significantly reduce coding delay. In addition, Pipeline Coding is tested in a heterogeneous VANET testbed, where vehicles upload streams through WiMAX and

WiFi links via roadside access points (AP) and base stations (BS) to a static client. The experiment results demonstrate that Pipeline Coding not only achieves better PDR but also utilizes heterogeneous links without explicit scheduling.

For TCP over static, error-prone environments, a combined intra-/inter-flow coding scheme, ComboCoding, is proposed. ComboCoding combines inter-flow and intra-flow coding and features an adaptive redundancy control to provide robust, fair TCP communication in challenged wireless networks. Simulation results show that TCP with ComboCoding delivers higher throughput than other coding options in high loss scenarios. Moreover, we study the behavior of multiple TCP sessions intersecting and interfering with each other in the same ad hoc network. The results show that ComboCoding consistently provides better fairness among multiple co-existing TCP sessions when compared with TCP without coding.

For TCP over challenged mobile ad-hoc networks (MANETs), we propose a network coded multipath scheme for conventional TCP—CodeMP. CodeMP adapts to frequent link changes in MANET and requires no explicit control messages. The scheme exploits multiple-path redundancy and maintains total transparency to transport layer protocols. The proposed coding scheme is based on three components: (1) random linear coding scheme with adjustable redundancy, (2) multipath routing, (3) ACK Piggy coding. Simulation results show that in an extreme MANET scenario where two TCP sessions co-exists and nodes move as fast as 25 m/s with up to 40% packet error rate (an environment in which regular TCP collapses completely), CodeMP achieves at least 700Kbps aggregate TCP goodput, with a Jain's fairness index of 0.99.

The dissertation of Chien-Chia Chen is approved.

Mani Srivastava

Songwu Lu

Danijela Cabric

M.Y. Sanadidi

Mario Gerla, Committee Chair

University of California, Los Angeles

2012

To My Mom, Dad,

and My Grandparents

Table of Contents

List of Figures

List of Tables

ACKNOWLEDGEMENTS

Ph.D. study is a long journey that could not be made without guidance, assistance, and companions. An immense gratitude goes to my advisor, Dr. Mario Gerla, who gave me guidance and inspiration during my doctoral study and provided me resources to conduct research. This study would have been much less in depth and breadth without his everlasting advice, support, and encouragement.

I would like to also specially thank Dr. M. Y. Sanadidi, who offered the first class I took in UCLA and opened my world of networking research. His thoughtful advice and enduring patience were always important and helpful. I also appreciate the other members in my doctoral committee, Dr. Mani Srivastava, Dr. Songwu Lu, and Dr. Danijela Cabric for their keen insight and advice making the work much deeper and more complete than it would otherwise have been.

A special appreciation goes to every NRL member, including Dr. Joon-Sang Park, Dr. Uichin Lee, Dr. Soon Oh, Dr. Giovanni Pau, Mr. Brandon Pancost, Mr. Jui-Ting Weng, Ms. Yu-Ting Yu, and Mr. Lee Aik Tuan, for their assistance, ideas, and support. Another special appreciation goes to Dr. Yong Chiang Tay, who provided guidance and assistance on many of my analytical studies. Also, I would like to thank all of my project partners, Chieh-Ning Lien, Cliff Chen, Phillip Tao, and Guruprasad Tahasildar. Without their help, many ideas would have taken much longer to be implemented and evaluated.

The Ph.D. study would have been much tougher without friendship. I feel grateful to all of my friends, who accompanied me on this long journey and had so much memory with me at many places in UCLA, from Boelter Hall, Drake Stadium, IM Field, Wooden Center, to Sunset Recreation Center.

Lastly, the study could not be done without the support of our sponsors, NSF, ONR, and Singapore DSO National Laboratories.

VITA

2006	B.S. (Computer Science), National Tsing Hua University, Hsinchu, Taiwan.
2009	M.S. (Computer Science), University of California at Los Angeles, Los Angeles, California, USA.
2012	Ph.D. (Computer Science), University of California at Los Angeles, Los Angeles, California, USA.

Publications

Journal Paper

C.-C. Chen, C. Chen, S. Y. Oh, J.-S. Park, M. Gerla, M. Y. Sanadidi, "ComboCoding: Combined Intra/Inter-Flow Network Coding for TCP over Disruptive MANETs," *Journal of Advanced Research*, vol. 2, pp. 241-252, 2011.

Conference Proceedings

C.-C. Chen, G. Tahasildar, Y.-T. Yu, J.-S. Park, M. Gerla, M. Y. Sanadidi, "CodeMP: Network Coded Multipath to Support TCP in Disruptive MANETs," submitted to IEEE MASS 2012.

C.-C. Chen, C. Chen, J.-S. Park, S. Y. Oh, M. Gerla, M. Y. Sanadidi, "Multiple Network Coded TCP Sessions in Disruptive Wireless Scenarios," in *Proc. of IEEE MILCOM 2011*, Nov. 2011.

C.-C. Chen, C. Chen, S. Y. Oh, M. Gerla, M. Y. Sanadidi, "ComboCoding: Combined Intra/Inter-Flow Network Coding for TCP over Multihop Lossy Wireless Netowkrs," in *Proc. of ACITA 2010*. Sept. 2010.

C.-C. Chen, S. Y. Oh, P. Tao, M. Gerla, M. Y. Sanadidi, "Pipeline Network Coding for Multicast Streams (Invited Paper)," in *Proc. of the 5th International Conference on Mobile Computing and Ubiquitous Networking (ICMU 2010)*, April 2010.

B. Pancost, C.-C. Chen, M. Y. Sanadidi, M. Gerla, "Buffer Estimate Filtering Using Dispersion Deltas," in *Proceedings of the 7th International Workshop on Protocols for Future, Large-Scale and Diverse Network Transports (PFLDNeT 2009*), May 2009.

Demonstrations

P. Tao, C.-C. Chen, S. Y. Oh, M. Gerla, M. Y. Sanadidi, "Demo Abstract: Pipeline Network Coding for Multicast Streams," in *Proc. of IEEE INFOCOM 2010*, March 2010.

C.-C. Chen, C.-N. Lien, U. Lee, S. Y. Oh, "CodeCast: Network Coding Based Multicast in MANETs," in *Demos of the 10th International Workshop on Mobile Computing Systems and Applications (HotMobile 2009)*, Feb. 2009.

CHAPTER 1

Introduction

Wireless communications over extremely lossy networks are known to be challenging since packet error rates as high as 50% will kill all kinds of applications. Since ARQ schemes are not suitable in streaming applications [1], recent research has turned to coding approaches, such as erasure codes and network coding. Network coding, which mostly stands for coding at the packet level, has attracted significant interest since [2]. Originally, the main aspect considered in the pioneer network coding work [2-7] was to exploit the broadcast nature of wireless links such that the multicast capacity is maximized [8]. Authors in [9] further pointed out the fact that in addition to multicast flows, network coding is also of benefit to unicast flows by providing a low-cost reliable transmission scheme over a disruptive network environment. Although several attempts have been made to propose coding protocols for multicast streams [10] and TCP [11], none of them takes into account realistic application constraints, such as delay constraints for streaming applications and latency issues for TCP. This study aims to propose novel approaches to network coding to bridge the gap between coding and transport/application protocols. We first propose an innovative low-delay coding scheme, Pipeline Coding, which is ideally suitable for multicast streams. Based on the proposed low-delay coding, we next propose an efficient coding approach, ComboCoding, which is compatible and transparent to existing TCP variants. We further design and implement a multipath network coding scheme, CodeMP, which equips with an adaptive multipath and redundancy control to support TCP traffic in disruptive MANETs.

In the context of multicast streams, delay and jitter are critical measures in addition to throughput. However, existing network coding approaches introduce high coding delay and high delay jitter. The price of using network coding to provide robust transmissions is one order of magnitude increase in end-to-end delay [12]. This is because conventional network coding is based on a batch, or a generation model [3-4, 6, 8, 10, 13] that is inherited from erasure coding [14] and fountain codes [15-16]. Such type of coding schemes are referred to as "batch coding" schemes throughout this dessertation.

In the batch coding scheme, each coded packets is a linear combination of all data packets in a generation. Successful decoding is guaranteed once enough coded packets are received. However, there are two critical drawbacks in the batch coding scheme. First, it introduces encoding and decoding delay that proportionally increases with generation size. As a result, the quality of real-time streaming degrades due to delay and jitter. Furthermore, because of the generation based coding, a generation is decodable only once enough linearly independent (i.e., innovative) packets are received; otherwise, the entire generation is discarded. In other words, throughput may significantly degrade due to frequent generation discards in a lossy network.

To address these batch coding problems, we propose to use "Pipeline Coding," a scheme in which both encoding and decoding can proceed progressively [17-18]. Instead of waiting for all packets in a generation to be received before coding and transmitting them, the source start encoding and sending coded packets whenever a new data packet arrives. At the destination, data packets can also be reconstructed "progressively," i.e., incrementally. Consequently, Pipeline Coding achieves (1) a lower coding delay, (2) a higher throughput, (3) transparency to higher layers (UDP, TCP, or other applications), and (4) no special hardware requirement.

As for TCP, we propose another novel coding scheme, ComboCoding, which consists of intra-flow coding and inter-flow coding. By intra-flow coding, we specifically refer to Pipeline Coding as it provides a low-delay robust communication in disruptive network environments. In addition, we borrow the idea from previous work on coding schemes for TCP to encode TCP DATA and ACK packets and thus reduce cross interference [7, 19-21]. We refer to the latter as "inter-flow coding" throughout this dissertation.

The first work on the inter-flow coding for TCP was reported in [7], in which the authors point out that network coding does not significantly improve TCP. Authors in [19] were prompt to remark that this is mainly because the study in [7] 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 [20], where the throughput is improved by opportunistically XORing TCP ACK and DATA flow. Following the work presented in [20], [21] proposed a MAC layer modification to provide a better channel access scheme. One of the earliest proposals to improve TCP in lossy networks is [11], in which authors studied an ***intra-flow*** random linear coding scheme. Intra-flow implies that only packets within one flow, the data flow in this case, are coded. Hence 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 study, we present a hybrid network coding scheme that is (1) transparent to TCP, and (2) addresses both interference and random loss problems. ComboCoding is an important step forward in the state of the art in many directions: (1) ComboCoding combines inter- and intra-flows coding into an efficient, robust coding approach that addresses both high random loss rate and self-interference. (2) ComboCoding is implemented in the network layer and is transparent to TCP or other reliable protocols at the upper layers. This is contrast to [11], where the authors entirely redesigned a new TCP variant that is incompatible with existing TCP protocols. (3) ComboCoding does not rely on any new or modified MAC layer protocols. This is different from [21], in which the authors propose a MAC layer modification to further improve coding gain.

While ComboCoding works for a disruptive environment with a static topology, in disruptive MANET scenarios, similar to [11, 43], it suffers from frequent routes breaks and highly unstable links. To address these new challenges in disruptive MANET environments, we next propose CodeMP, a multipath network coding scheme. Following similar design principles as in ComboCoding, CodeMP is transparent to transport layer protocols such as TCP and requires no explicit control messages. CodeMP provides a practical and efficient approach to exploit multiple-path and coding redundancy in disruptive MANETs. It also achieves better fairness in the presence of multiple coexisting TCP sessions in the MANET scenarios.

In the rest of this thesis, we will first present the design of Pipeline Coding as well as the corresponding evaluations through both analysis and simulations in chapter 2. Chapter 3 proceeds to unicast TCP case and introduces ComboCoding with a loss adaptation algorithm, followed by our first order analysis for the inter-flow XOR coding and intensive simulation evaluations. Chapter 4 further extends to disruptive MANET scenarios and proposes CodeMP to support TCP applications in such a disruptive mobile environment. Chapter 5 presents a testbed implementation for Pipeline Coding, in which we validate the results measured via simulations and conduct a number of experiments in our Campus VANET Testbed (C-VeT). Finally, the dissertation is concluded in chapter 6.

CHAPTER 2

Pipeline Coding for Multicast Streams

Due to the nature of wireless communications, wireless ad-hoc networks are vulnerable to channel errors, interference and jamming. There are two well known approaches for effective error recovery; Forward Error Correction and ARQ [1]. Since the targeted application here is real time multicast streaming, the ARQ scheme is not appropriate. In order to efficiently control losses, recent research in this area has exploited FEC-based coding schemes, such as erasure coding [14] and network coding [2-7].

Erasure coding was first proposed in [14], in which the authors referred the word "erasures" at the network layer means "missing packets in a stream". Erasure coding is classified as FEC coding for binary erasure channels since a source achieves error correction by injecting redundant data. Conventionally, erasure coding refers to a general coding scheme consisting of source-only coding. In order to make distinction, we define a new term "batch erasure coding" that generates encoded packets only from the same batch. A source generates *n* packets from *k* original ones. The original data can be reconstructed from a subset of *n* packets. A stream of packets is split into *k* packets called "generation" or "batch" [3-4, 6, 8, 10, 13] and the source produces *n* coded packets for each generation using random linear coding. We define *r=n/k* as the "coding redundancy" in this dissertation. At destinations, any subset of *k* linearly independent coded packets is sufficient to reconstruct the original generation. In other words, *n*-*k* losses are allowed in a group of *n* coded packets. The difference between Erasure coding and network coding is that in the former the coding is done only at the source while in the latter the coding is done also at intermediate nodes. In fact, Erasure coding can be viewed as a subset of network coding.

The above erasure coding scheme is called “pre-defined rate coding” since coding redundancy is decided before transmission. Another form of erasure coding is called “rateless coding,” which can generate infinite encoded packets. Namely, there is feedback from destination to source and the source will adaptively adjust the redundancy depending on reception quality at destination etc. However, end to end feedback can be slow and is not practical in multicast/broadcast because of feedback control message “explosion” Fountain Codes [15-16] and Raptor Codes [22] are this type of coding schemes. In this chapter, we only consider pre-defined rate coding. In this respect, it should be noted that network coding allows dynamic adjustment of redundancy WITHOUT expensive end to end feedback since it can just rely on downstream neighbor feedback to adjust retransmissions.

In disruptive networking environments, end-to-end erasure coding is not sufficient to achieve reliable packet transmission [23]. Thus, researchers have applied network coding to such challenged environments, in which the broadcast nature of wireless medium renders network coding particularly effective [6]. Unlike source coding, as mentioned earlier all nodes in the network participate in the encoding process. In this dissertation, we use “batch network coding” to refer to a network coding scheme where source and relay nodes encode data packets in the same generation using random linear coding.

In the rest of this chapter, we first provides fundamentals of batch coding schemes, followed by the design of a novel low-delay coding scheme, Pipeline Coding. Section 2.3 presents an analytical model for Pipeline Coding. The encoding and decoding flows are given in Section 2.4. We evaluate Pipeline Coding through simulations and present results in Section 2.5.

2.1 Backgroud—Batch Coding

This section gives the mathematical definition of batch coding. Table 1 below summarizes the terminology we adopt throughout the dissertation. As defined previously, in this dissertation, batch coding refers to both batch erasure coding and batch network coding. Assuming at the source, an application generates a series of equal-sized packets $\mathbf{p_1}$, $\mathbf{p_2}$, $\mathbf{p_3}$,…,. Let k be the number of packets in a single generation. A coded packet $\mathbf{c}$ in ith generation is then defined as:

$$\mathbf{c} = \sum_{j=1}^{k} e_j \mathbf{p}_{i\times k+j}, \tag{1}$$

where e_k is a particular element in a particular finite Galois field $\mathbb{F}$, and $i\times k$ is the total number of packets transmitted so far in the file, before the ith 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 $\mathbb{F}$ so that data packets $\mathbf{p_i}$ and coded packets $\mathbf{c}$ are also regarded as vectors over $\mathbb{F}$. Let r denote the coding redundancy, where $r\geq1$. For each generation, the source will produce $k\times r$ coded packets. At destination side, after receiving k linearly independent packets, the destination can then reconstruct the original content by solving the following linear system of equations as given in eq. (2) using Gaussian Elimination.

$$\begin{bmatrix} \mathbf{c}_1 \\ \vdots \\ \mathbf{c}_k \end{bmatrix} = \begin{bmatrix} e_1^{(1)} & \cdots & e_k^{(1)} \\ \vdots & \ddots & \vdots \\ e_1^{(k)} & \cdots & e_k^{(k)} \end{bmatrix} \begin{bmatrix} \mathbf{p}_1 \\ \vdots \\ \mathbf{p}_k \end{bmatrix}. \tag{2}$$

Table 1 Definitions of Terms Used in the Dissertation

Term	Definition
Erasure Coding	Source-side only coding.
Network Coding	Source-side and relay coding.
Batch Coding	Every coded packet will encode all data packets within the same generation. Coding and decoding begins only when the generation rank is "full"
Pipeline Coding	Coded packets will be generated upon every new data packet arriving. Destinations decode the data packets progressively if possible.
Generation	A set of packets that are encoded or decoded together as a unit.
Coding Vector (Encoding Vector)	A vector of coefficients that reflect the linear combination of data packets.
Rank (Degree of Freedom)	Number of linearly independent combinations of data packets.
Innovative Packet	A packet that increases the rank.
Coding Redundancy	Number of coded packets sent per generation divided by generation size.
Delay	The time difference between packet reception by destination application and transmission from source.

In batch erasure coding, only the source encodes packets and other nodes relay encoded packets while in batch network coding, relays also participate in coding. Upon receiving a coded packet, relays first check whether it is innovative. Non-innovative packets are discarded while innovative packets are stored in the generation buffer. For each newly arrived innovative packet, relays generate and send out a new coded packet, which is a new linear combination of all received innovative coded packets in the same generation. This procedure is called "re-encoding." Note that relays do not attempt to recover the original data packets. The re-encoded packets, by induction, are still linear combination of original data packets. This simple innovative checking and re-encoding on relays will make a major difference in performance, as will be shown in Section 2.3.

Since each coded packet must be a linear combination of ALL data packets in the same generation, at the source a delay is incurred until all data packets in a generation arrive. Further, at the destinations, either all data packets in a generation will be decoded successfully, or none will be, under batch coding.

In order to formulate the coding delay, we first define the "delay" as the time difference from the moment when a packet is received by the application at the destination, to when it is delivered to the application at the source. Let t_i denote the time that the source generates $\mathbf{data_i}$ and t_{ci} denote the time that i^{th} coded packet is sent out. Assume the four delay components - processing, transmission, propagation, and queuing delay - are constant. Let d_n be the constant delay in the network including all delay components, and d_c be the constant decoding delay. Assume each generation has k data packets. Therefore, in lossless links, the delay to deliver $\mathbf{data_i}$ will be:

$$t_{c\lceil i/k \rceil * k} - t_i + d_n + d_c. \tag{3}$$

For example, the delay for delivering $data_1$ will be t_{c4}-t_1+d_n+d_c as shown in Fig. 1, where we assume the generation size is $k=4$ and the coding redundancy is $r=1.25$. Therefore, $k*r=5$

coded packets will be sent out for every generation. Note that the figure does not show the re-encoding process because the latter does not significantly affect the coding analysis while at the same time creates a graph that is too complicated to draw in limited space. A detailed description of re-encoding can be found in [4].

Assuming now that packet losses are possible over the communications path, data packets can still be decoded as long as any subset of 4 linearly independent coded packets out of the 5 transmitted by the source is received. The generation #2 in Fig. 1 shows an example of loss that was recovered and resulted in a successful decoding. However, if the redundancy level is not enough to compensate for losses, none of the data packets in this generation will be decoded as depicted in Fig. 2. Here the loss of two packets renders it impossible to decode this generation.

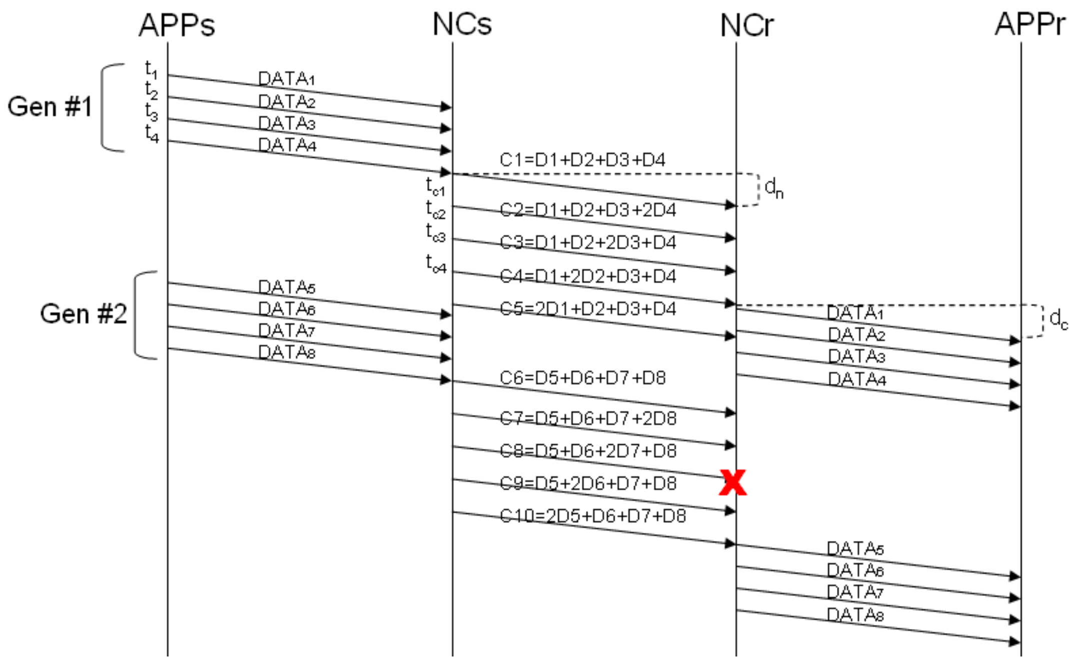


Fig. 1 Batch Coding Example

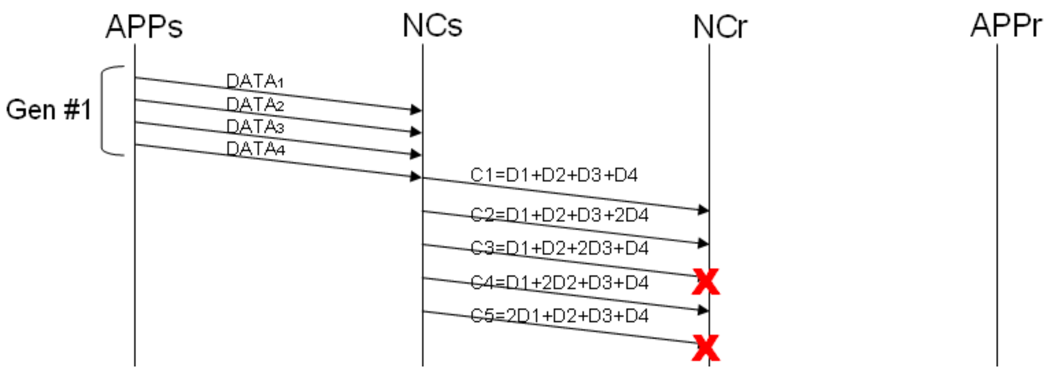


Fig. 2 Batch Coding: Undecodable Case

2.2 Pipeline Coding and Decoding

Pipeline coding scheme aims to reduce the coding delay as well as to further improve the throughput. Normally, packets are not sent from an application all at once. Therefore, in Pipeline Coding, we relax the limitation of waiting for all data packets of a generation to be received from the application. Adopting the same notation in section 2.2, the encoding function is then changed as following:

$$\mathbf{c} = \sum_{j=1}^{m} e_j \mathbf{p}_{i \times k + j}, \tag{4}$$

where the new variable m is the number of data packets currently present in the generation buffer. In other words, upon receiving a new data packet, the source will instantly trigger the encoding process based on currently received data packets. If all coded packets are delivered successfully, destinations can construct the following lower triangular matrix without any extra computation:

$$\begin{bmatrix} \mathbf{c}_1 \\ \vdots \\ \mathbf{c}_k \end{bmatrix} = \begin{bmatrix} e_1^{(1)} & 0 & \cdots & 0 \\ e_1^{(2)} & e_2^{(2)} & & \vdots \\ \vdots & & \ddots & 0 \\ e_1^{(k)} & e_2^{(k)} & \cdots & e_k^{(k)} \end{bmatrix} \begin{bmatrix} \mathbf{p}_1 \\ \vdots \\ \mathbf{p}_k \end{bmatrix}. \tag{5}$$

The above linear equation can be solved progressively without waiting for generation completion. For example, upon receiving $\mathbf{c_1}$, destinations can decode $\mathbf{p_1}$, and so on. Coding redundancy is applied at the source in order to mitigate losses, in a slight different way from batch coding. Let r be the coding redundancy, where $r \geq 1$. Per each data packet, with a probability of $r - \lfloor r \rfloor$, $r+1$ coded packets are generated and sent; with a probability of $1-(r-\lfloor r \rfloor)$, r coded packets are generated and sent. In other words, redundancy is uniformly distributed among every data packet. Also, if network coding is used, relays will all participate in coding, as explained in [4], in order to produce further redundancy of coded packets.

In the batch coding example (Fig. 1), the delay of data packets is shown in eq. (3), which is two constant delays plus the time difference between the data packet sent from the source application and the last innovative packet sent from the source. Following the same notation, t_i denotes the time when the source generates **data$_\mathbf{i}$** and t_{ci} denotes the time when the i^{th} coded packet is sent out. If Pipeline Coding is adopted, in lossless links, the delay of **data$_\mathbf{i}$** will become $t_{ci}- t_i+ d_n+ d_c$, which is much lower than eq. (3) delay in most cases. Thus, the delay is greatly reduced.

Fig. 3 presents an example of Pipeline Coding, with generation size, $k = 4$ and source coding redundancy, $r = 1.25$. The re-encoding part is not shown for simplicity. As shown in the figure, data packets are encoded instantly upon arrival. Similarly, at the destination, coded packets can be decoded immediately once a new innovative packet arrives. For example, if $\mathbf{c_1}$ is delivered successfully, it is decoded immediately at the destination. This gives us a delay of t_{c1}- t_1+ d_n+ d_c.

In the case of packet losses as shown in the 2nd generation of Fig. 3, the destination will store un-decodable packets and wait for the next packet with which, hopefully, the previous loss can be recovered. As soon as a new innovative coded packet arrives and thus Gaussian Elimination can decode new unknown packets, newly decodable packets will then be delivered to upper layer. In Fig. 3, after receiving $\mathbf{c_9}$ and $\mathbf{c_{10}}$, $\mathbf{data_7}$ and $\mathbf{data_8}$ will be decoded. Let t_{c10} denote the time that c_{10} is sent. The delay in this case will be $t_{c10} - t_7 + d_n + d_c$ for $\mathbf{data_7}$ and $t_{c10} - t_8 + d_n + d_c$ for $\mathbf{data_8}$. These two data packets will have the same amount of delay as in batch coding scheme, but $\mathbf{data_5}$ and $\mathbf{data_6}$ still benefit from Pipeline Coding.

Pipeline coding can partially recovers a subset of the data packets of a generation and deliver them to the upper layer. This is a significant difference from batch coding, which either delivers an entire generation to the upper layer, or discards the whole generation. For example, assume that $\mathbf{c_{10}}$ of Fig. 3 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 2nd generation can be decoded, while with Pipeline Coding, we can still decode data packets #5 and #6.

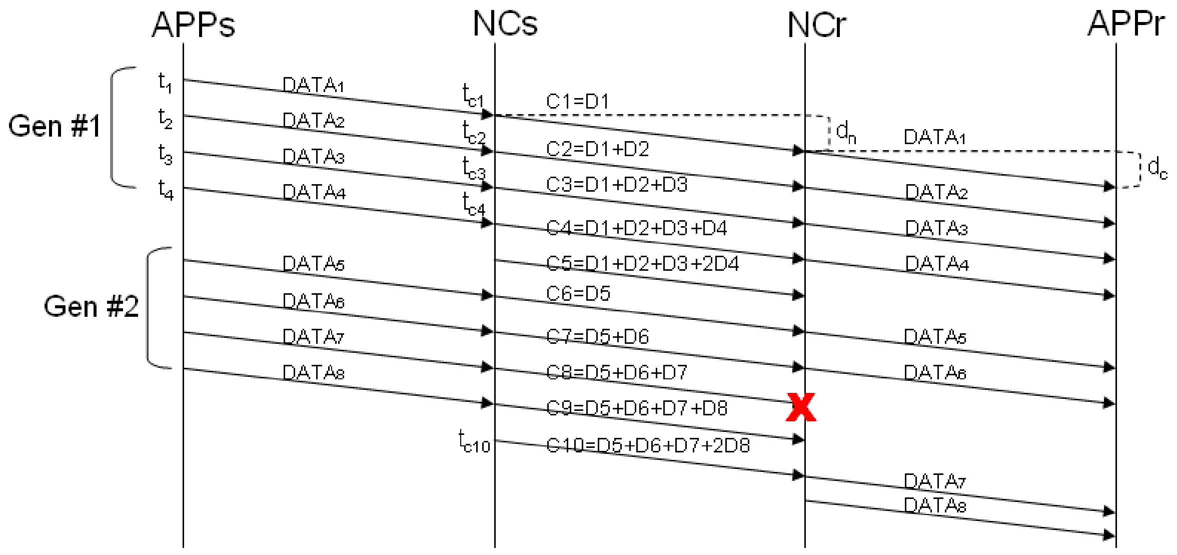


Fig. 3 Pipeline Coding Example

The ability of "partial" generation recovery is important in real time stream applications. It would not be useful of course in reliable data applications, in which the entire generation must be received ("all or nothing").

2.3 Encoding and Decoding Procedure

Fig. 4 shows the encoding procedure of our proposed Pipeline Coding. The coding module is implemented at the network layer and it does encoding, decoding, and broadcasting. When the source receives a data packet from the transport layer, it first stores the data in the generation buffer. Thereafter, based on the coding redundancy, a number of coded packets are generated and sent out. For each generated coded packet, the source first randomly generates the coding coefficients. Secondly, it checks the generation buffer. For a missing packet, which has not arrived yet, the corresponding coefficient is set to zero. Finally, the source encodes the data packets based on this encoding vector, and sends it to the lower layer. Note that the '**if**' statement in the while loop is to make sure redundant packets are generated uniformly instead of always in the end of the generation.

Fig. 5 is the decoding procedure. A destination first examines whether the received coded packet is innovative or not. An innovative packet will be stored in the generation buffer. Afterward, the decoding module invokes Gaussian Elimination routines and attempts to decode data packets. After decoding, newly decoded data packets will be stored into another decoding buffer and then delivered to the upper layer. Note that in some rare cases, a few packets might be decoded out-of-order. In order to reduce the impact to TCP, we have a function to avoid out-of-order data packet delivery. However, this contributes only little to this performance study, and thus the detail is not shown in the pseudo-code in this dissertation.

Encoding Procedure

```
Function Handle_Packets_from_Transport(data)
Store data to generation buffer
Coded_Packet_Gen(coding_redundancy)
Function Coded_Packet_Gen(num)
  i := 0
  while i < num do
    if (num - i < 1 and rand() % 100 > num - i) then
      break
    Generate Coding Vector
    for j := 0 .. generation_size - 1do
      if generation_buffer[j] is NULL then
        coding_vector[j] := 0
    Encode packet
    Send Coded Packet to MAC layer
    i := i + 1
```

Fig. 4 Pipeline Coding—Encoding Procedure

Decoding Procedure

```
Function Handle_Packets_from_MAC(data)
if (not innovative(data)) then
  Message_Free(data)
  return
store data to generation buffer
Gaussian_Elimination(generation)
if (decodable packets cause no reordering) then
  Deliver_to_Tranport(newly decoded data packets)
```

Fig. 5 Pipeline Coding－Decoding Procedure

2.4 Pipeline Coding Analysis

Conventional network coding with batch mode has been analyzed extensively. We present here preliminary analysis based on probabilistic models that compare batch and pipeline coding. We assume that the same generation size G is used for both. Moreover, for simplicity, a unit capacity point-to-point lossy link is considered. In batch coding, the sender waits until it collects G packets and transmits $G \times R$ encoded packets, where $R \geq 1$ is the redundancy ratio, and G is the generation size. In pipeline coding, upon arrival of every new application packet, a coded packet is generated using all the packets belonging to the same generation that are "currently" in the buffer. When a link loss occurs, the packet is dropped with random, independent probability p. In batch coding, a packet received at the destination is "helpful" with high probability. This has been proven in [38], and for the sake of completeness, we include the result in Lemma 1.

Lemma 1: Suppose node v transmits a coded packet to node u. Let S_u^- and S_v^- denote the subspaces spanned by the code-vectors with u and v respectively at the beginning. Let S_u^+

denote the subspaces spanned by the code-vectors by *u* after receiving a coded packet from *v*. Then, $\Pr(S_u^+ > S_u^- \mid S_v^- \not\subseteq S_u^-) = 1 - 1/q$, where *q* is the size of the field.

A newly received packet is helpful if its code vector is linearly independent from previous received packets' code vectors. Thus, when a coded packet is transmitted over the channel the probability of receiving a helpful packet is $(1 - p) * (1 - 1/q)$. In order to be able to decode and recover the original data when using batch network coding, one must collect more than *G* or more packets with independent code vectors out of a total of $G * R$ transmitted packets. We calculate the probability for a destination to receive greater than or equal to *G* packets out of $G * R$ packets with independent code vectors as follows:

$$P_{batch}(G,R,p,q) = \sum_{k=G}^{G\times R} \binom{G\times R}{k} (1-r)^{(G\times R-k)} r^k \, , \tag{6}$$

where $r = (1 - p) * (1 - 1/q)$, which is the probability of successfully (i.e., without error) receiving a helpful packet. In short, the probability is $F(G * R\text{-}G;\ G * R,\ r)$ where $F(k;\ n,\ p)$ is the cumulative distribution function of the binomial distribution $K \sim B(n, p)$. Then the expected throughput of the batch network coding is $1/R * F(G * R - G;\ G * R,\ r)$ assuming that the link capacity is 1. 1/R is the reduction factor due to the redundancy implemented in the batch network coding.

In the pipeline coding, if a destination receives greater than or equal to *G* packets out of $G * R$ transmitted packets, the destination can decode and recover the original data, similar to the batch coding case. Moreover in pipeline coding one can decode even if less than *G* packets are received due to the "partial decodability" property. For example, if a receiver receives only the first two encoded packets then it can decode and recover the first two original data packets, which is not possible in batch coding. Considering this property, the throughput of pipeline coding can be expressed as:

$$P_{pipeline}(G,R,p,q) = P_{batch}(G,R,p,q) + P_{partial-decode}(G,R,p)\,. \tag{7}$$

The second term reflects the throughput increase due to partial decodability, which can be obtained by enumerating all combinations that the destination receives only some of the first G packets. Fig. 6 below shows an example of such an enumeration with G = 4, where the superscript of a state denotes the next packet to be transferred and the subscript denotes the packets received successfully. With this state transition diagram, the partial decoding probability is then approximated as follows:

$$P_{partial-decode}(G,R,p) \approx \sum_{k=1}^{S} \frac{S_k}{G}(1-p)^k p^{\lceil G\times R\rceil - k}, \tag{8}$$

where S is the total number of the terminal states and S_k is the number of decaodable packets for a particular terminal states. Since there is no closed form to derive S_k, we approximate the partial decoding probability state by state. However, such a partial decoding probability is always greater than zero in most cases, so the throughput in pipeline coding is always higher than in batch coding.

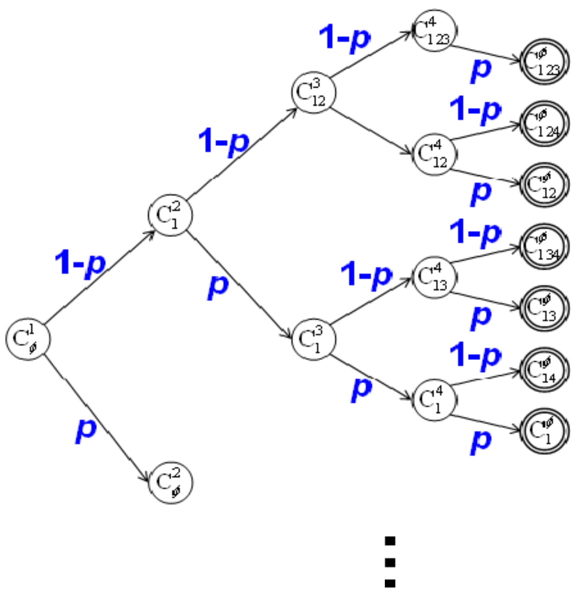


Fig. 6 Enumeration of Received Pipeline Coded Packets

To validate the accuracy of the above first-order analysis, we ran a number of testbed experiments with our Pipeline Coding router using a 2-hop, 4-laptop braid topology as shown in Fig. 7. In the testbed experiments, the server sends a sequence of streaming packets over UDP and the client collects the packet delivery ratio (PDR) at the application level, which is equivalent to the decoding probability. We use a generation size of 8 packets and a fixed redundancy of 1.25. Fig. 8 below presents the PDR measurements collected from testbed experiments and the predicted PDR using the above model. We find that both the model and the testbed measurements show the PDR of batch coding drops faster than that of Pipeline Coding as the random error rate increases. The model, however, tends to be more optimistic since it does not take into account collisions and channel access details, which could lead to higher loss rates than the one we manually introduced to the experiments.

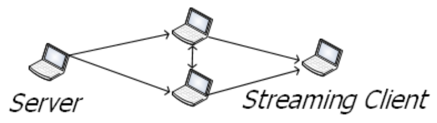


Fig. 7 Pipeline Coding Analysis Validation

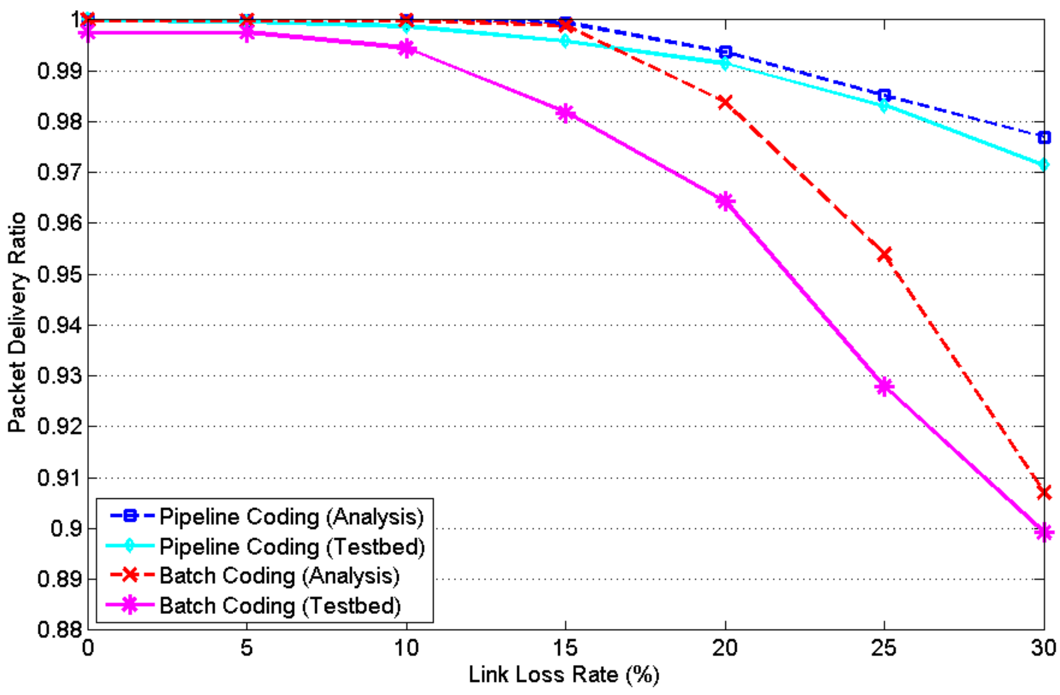
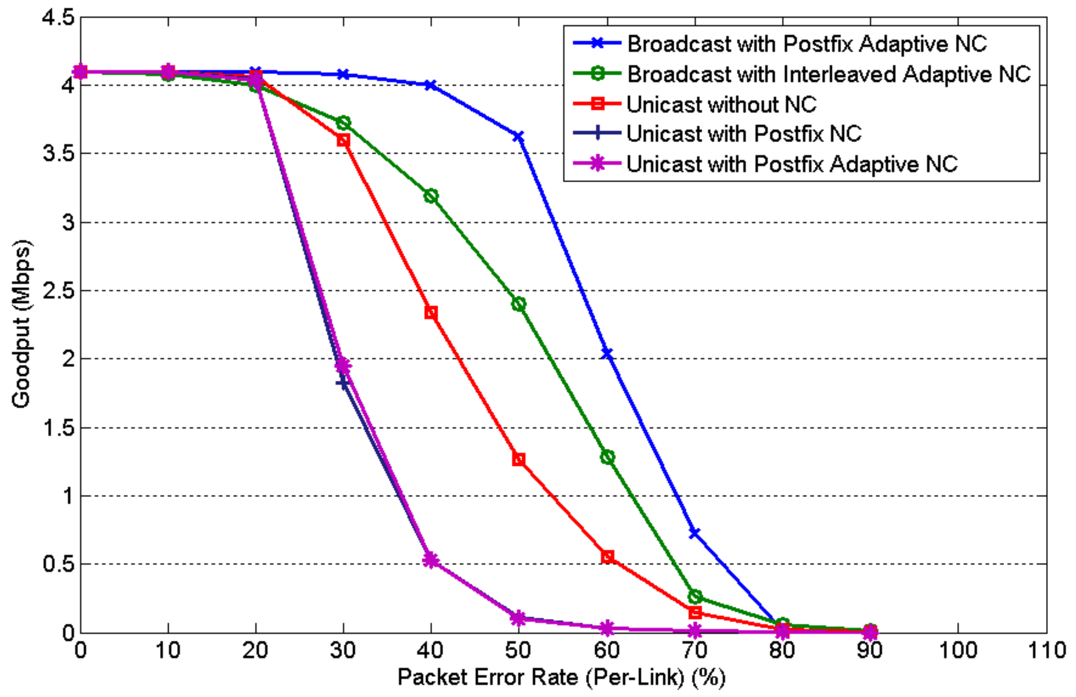


Fig. 8 Pipeline Coding Analysis Validation (*G* = 8, *R* = 1.25)

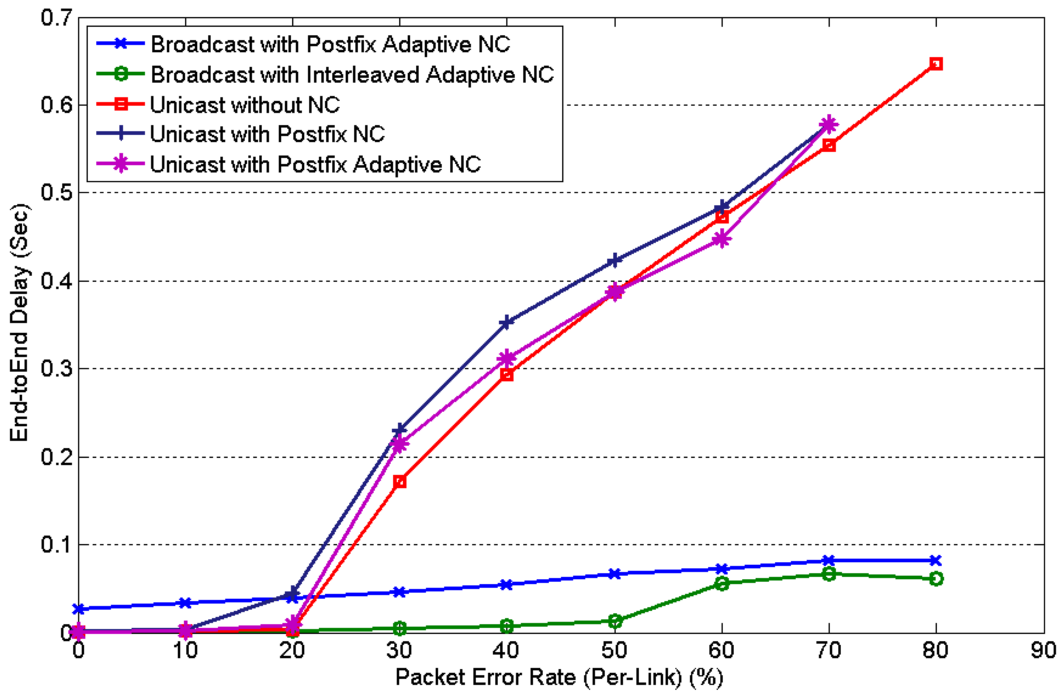
The above analysis shows the first-order behavior of the pipeline coding scheme, where the additional partially decoded generation helps outperform batch coding scheme at all times. As we mentioned earlier, we propose to study multiple options to inject redundant packets such as postfix or interleaved. We also did some preliminary simulation to evaluate the performance gain and the tradeoffs among different pipeline coding strategies. We ran UDP flows over a 4-node single path topology. The offered load of the CBR application is 4.1Mbps. We injected random errors on each link and studied the performance gain and robustness of the following redundancy strategies: (1) broadcast with postfix adaptive NC, (2) broadcast with interleaved adaptive NC, (3) unicast without NC, (4) unicast with postfix NC, and (5) unicast with postfix adaptive NC. The "adaptive" here refers to the loss adaptive redundancy control used in Section 3.4. In these settings, standard 802.11g is used with RTS/CTS disabled. When using broadcast, MAC layer performs no retransmissions; when using unicast, MAC layer retransmits a frame up to seven times when a MAC ACK times out.

Fig. 9(a) shows the goodput vs. link error results. We first found that among all cases, postfix with broadcast performs the best, since it provides the ability to partially decode generations and include all data packets in the redundant coded packets, which matches our analytical result. The interleaved with broadcast case performs slightly worse, but it is still better than the unicast without NC case, which relies solely on the MAC reliability. We also found that when using postfix coding scheme, enabling MAC layer retransmission (unicast) is harmful. This is because using unicast converts the postfix redundancy back to the interleaved mode and further, all the "interleaved" redundant packets generated by the MAC layer are identical, which reduces the probability of delivering linearly independent packets.

Fig. 9(b) shows the end-to-end delay vs. link error results, which indicate a clear robustness vs. delay tradeoff. We found that although broadcast with postfix is the most robust to link errors, when under 20% link error rates, it achieves one magnitude higher end-to-end delay than all other schemes. This is because even if it provides a higher probability of partially decoding a generation, when losses are high, the destination has to wait for redundant packets to arrive, which reduces the delay improvement provided by pipelining packets. This preliminary study inspires us in designing an adaptive coding scheme that when under low error rates, unicast or broadcast with interleaved redundancy is chosen to maximize the goodput while maintaining low delay; when under extreme losses, it switches to broadcast postfix NC to provide robustness with relatively lower delay.



(a) Goodput vs. Link Error



(b) End-to-End Delay vs. Link Error

Fig. 9 Simulation Results of Different Redundancy Strategies

2.5 Simulation Results

The simulation topologies under study are the single path string topology shown in Fig. 10 and the multipath braided topology shown in Fig. 11. Nodes on both topologies are placed on grids, with grid edge =150m. For each topology, we use a single traffic flow from a single source to a single destination. The generated traffic is CBR/UDP traffic. Note that since the proposed coding scheme exploits a pure-broadcasting channel access protocol, multicast/broadcast can also be supported. Namely, all nodes in the network can hear the coded packets and thus can decode the stream if they are multicasting clients.

For the channel access protocol, standard 802.11b (CSMA/CA) is used in "single path without coding" scenario. For all the remaining cases, a pure-broadcasting scheme is used. Pure-broadcasting means all packets are sent in standard 802.11b broadcast mode. The reason why we do not adopt the pseudo-broadcasting approach proposed in [7] is that such design is not supported by all hardware [19]. However, since RTS/CTS will not be used in broadcast mode, we exploit the idea from [10], in which a small amount of random delay is added before delivering the packets to MAC layer. This short random delay effectively avoids phasing, namely the situation where all nodes receive an innovative packet, finish re-encoding, and attempt to send it out at the same instant.

The following subsections will discuss the simulation results of UDP. Further details of the simulation configuration are given in table 2.

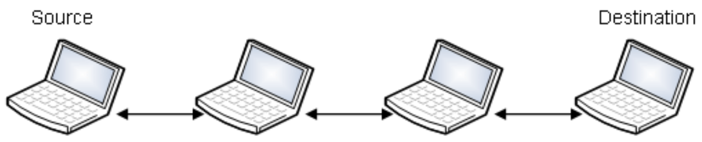


Fig. 10 String Topology

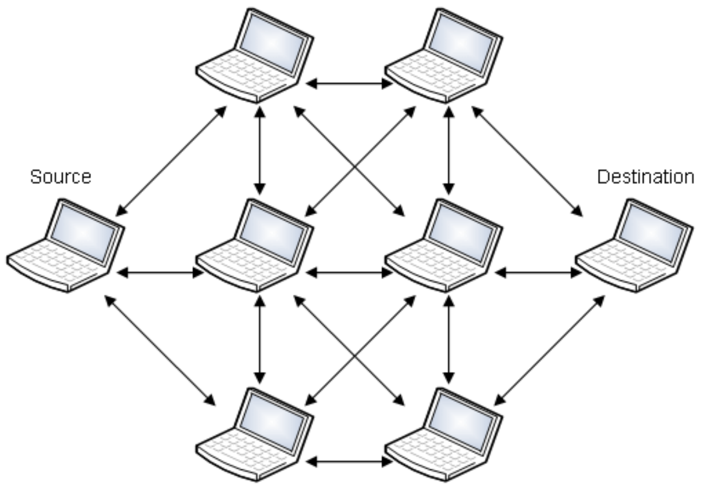


Fig. 11 Braided Topology

Table 2 Pipeline Coding Simulation Configuration

Parameter	Value
Grid Distance	150 m
Channel Bit-rate	11 Mbps
Channel Access Control	802.11b +RTS/CTS for string topology , no coding; 802.11b broadcast mode, for all other cases.
Transport and Application Layer	CBR/UDP (820Kbps)
Per Link Loss Rate	0%~60%
Packet Size	1500Bytes
Generation Size	8 Packets

A CBR application of 820Kbps sending rate is configured at the source in our simulation study. 820Kbps is below the saturation throughput of the 3-hop wireless network, thus it leaves room for redundant packets. Topologies and coding schemes tested using CBR/UDP traffic are summarized in table 3.

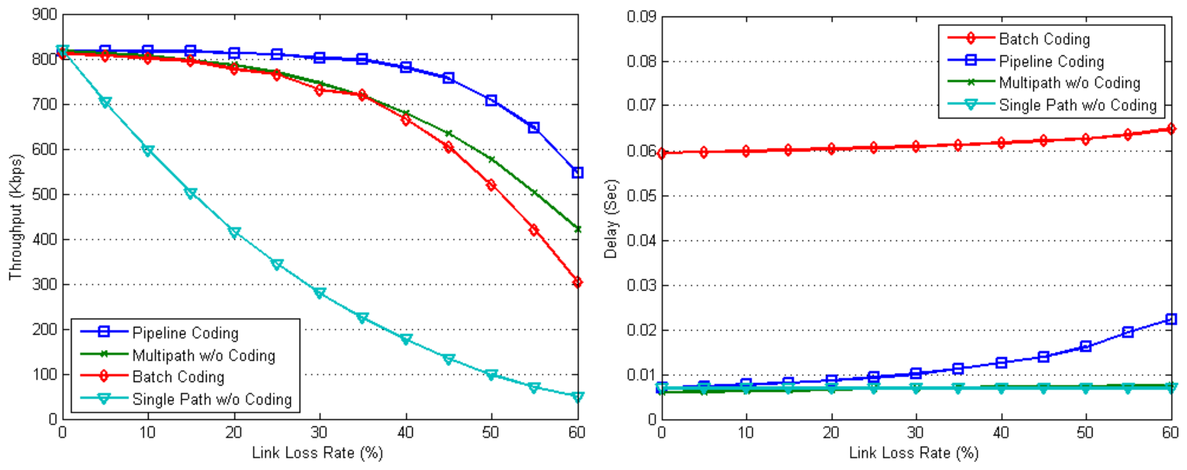
Note that the coding redundancy is chosen to be 2.5 based on a previous simulation study, which shows that a redundancy level of 2.5 is about the right level to compensate losses without congesting the network.

Table 3 Pipeline Coding CBR/UDP Simulation Configuration

Traffic Type	Topology	Coding Scheme	Coding Redundancy
CBR/ UDP	String	No coding (Unicast)	No Coding
		No Coding (Broadcast)	No Coding
	Braided	Batch Network Coding	2.5
		Pipeline Network Coding	2.5

The throughput-to-loss plot is demonstrated in Fig. 12(a). The single path without coding case is included as the base case. As shown in Fig. 12(a), generally, as the loss rate increases, all throughput curves drop. Also, all braided cases significantly outperform the string without coding. Pipeline network coding performs the best regardless of the link loss rate. The throughput of pipeline network coding shows no degradation for loss rate under 35%. Multipath without coding achieves 2nd highest throughput, which is slightly better than multipath with batch network coding case. This fact was also noted in [12]. Based on these simulation results, we notice that supporting partial decoding of a generation can significantly improve throughput compared to batch network coding.

Fig. 12(b) presents the delay-to-loss plot for the same set of configurations. As in Fig. 12(a), the single path no coding case is shown as the baseline case. From Fig. 12(b), we notice that, multipath without coding has almost the same delay as single path, while Fig. 12(a) shows that the throughput is greatly improved. Also, pipeline network coding reduces the delay significantly from batch network coding. In addition, we observe that network coding delay increases as the loss rate increases.



(a) Throughput (b) Delay

Fig. 12 UDP Results (braided topology)

CHAPTER 3

ComboCoding for TCP over Multihop Wireless Networks

We next move onto TCP traffic scenarios and further propose a combined intra/inter-flow coding approach, which is designed specifically for TCP over wireless multihop networks [24-25]. The Transport Control Protocol (TCP) is a 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 that packet losses are due to router buffer overflow, which was always true for the wired Internet, for which TCP-NewReno was originally designed, in which most networks links are point to point links.

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 seriously suffers from reacting to random losses wrongly assuming that they are congestion indicators. Also, due to broadcast channel sharing in wireless networks, it is highly likely that a TCP ACKs interfere with its DATA packets causing self-induced collisions. There are several approaches to address these two issues separately, including other forms of congestion control, tuning TCP parameters or protocol optimization.

One way to improve loss-based congestion control in wireless networks is to deploy Loss Discrimination Algorithms (LDA) [26-29]. Another way is to optimize TCP parameters. K. Su et al. pointed out that TCP congestion window is a key to improve TCP performance in wireless networks [30]. In [31], J. Li et al. went further showing that controlling the *maximum* congestion window size as the most effect. Yet another optimization is to reduce the interference between ACKs and DATA packets by reducing the ACK frequency [32]. Intelligently controlling the so-called "Delayed Acks" can reduce ACK packets in the network, thus reducing the inter-flow interference level.

As mentioned previously, considerable previous work has attempted to exploit network coding to help TCP in wireless scenarios. However, none of them takes into account both problems of lossy networks and self-interference. We present a hybrid network coding scheme that consists of intra- and inter- flow coding schemes to provide an efficient and robust coding scheme. The proposed coding scheme, ComboCoding, is compatible and transparent to existing TCP variants.

In ComboCoding, the intra-flow coding is employed to provide low-cost robust communication over disruptive environments and the inter-flow coding is used to alleviate the interference of the TCP DATA and ACK flows. 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. With Pipeline Coding, the total amount of information transmitted is less than with batch coding. This implicitly suggests that Pipeline Coding requires a higher forwarding redundancy, which has been confirmed through simulation results.

In the following subsections, the original design of PiggyCode, the inter-flow coding scheme we adapt, is introduced first in Section 3.1. The ComboCoding protocol is presented in Section 3.2. We next present a first-order analysis on our inter-flow coding scheme. A loss adaptation algorithm is given in Section 3.4, followed by a brief discussion of the chosen channel access scheme in Section 3.5. Lastly, ComboCoding and the proposed loss adaptation algorithm are evaluated through QualNet [33] simulations, in which ComboCoding is implemented as a "rouging protocol." The simulation results are shown in Section 3.6.

3.1 Backgroud—PiggyCode

The idea of PiggyCode is similar to COPE [7]. Namely it is an **inter-flow** coding scheme. However, it is re-designed specifically here for a special type of bi-directional traffic—TCP [20]. It is known that in wireless multihop scenarios, TCP DATA flow and ACK flow may create interference with each other, which decreases the throughput. The main goal of PiggyCode is to improve TCP performance by opportunistically XORing TCP DATA and ACK packets at intermediate nodes as shown in Fig. 13. Upon receiving a TCP ACK, the intermediate node checks its MAC layer buffer. If there exists a TCP DATA packet, the MAC layer performs an XOR of both packets with additional appropriate identification and transmits the PiggyCoded packet; otherwise, the ACK is sent out without encoding. All packets are buffered before being sent out. Upon receiving a PiggyCoded packet, both receivers perform another XOR operation with the buffered packets to decode the original packets.

The advantage of PiggyCode is it requires no TCP modification. However, as the recent study in [21] points out, the major challenge here is that a PiggyCoded packet is conceptually a "dual-cast" packet in the link layer, meaning there are two intended receivers that should receive the packet correctly. Due to the lack of dual-ACK support in 802.11, the authors noticed a limitation of the PiggyCode throughput gain. In order to improve performance, the authors propose a MAC layer modification to introduce dual-ACK support so that both intended receivers will send a MAC-ACK to the PiggyCode sender. It has been shown in their research that with this special MAC-layer support, PiggyCode can improve TCP throughput by as much as 100%.

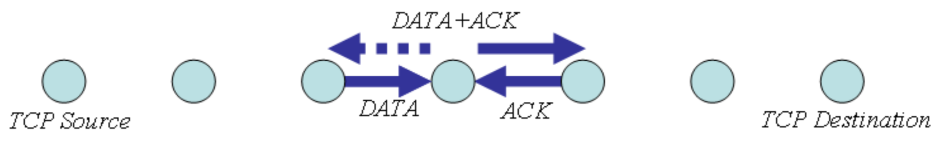


Fig. 13 PiggyCode Example:

DATA+ACK is the bitwise XOR result of DATA and ACK packets.

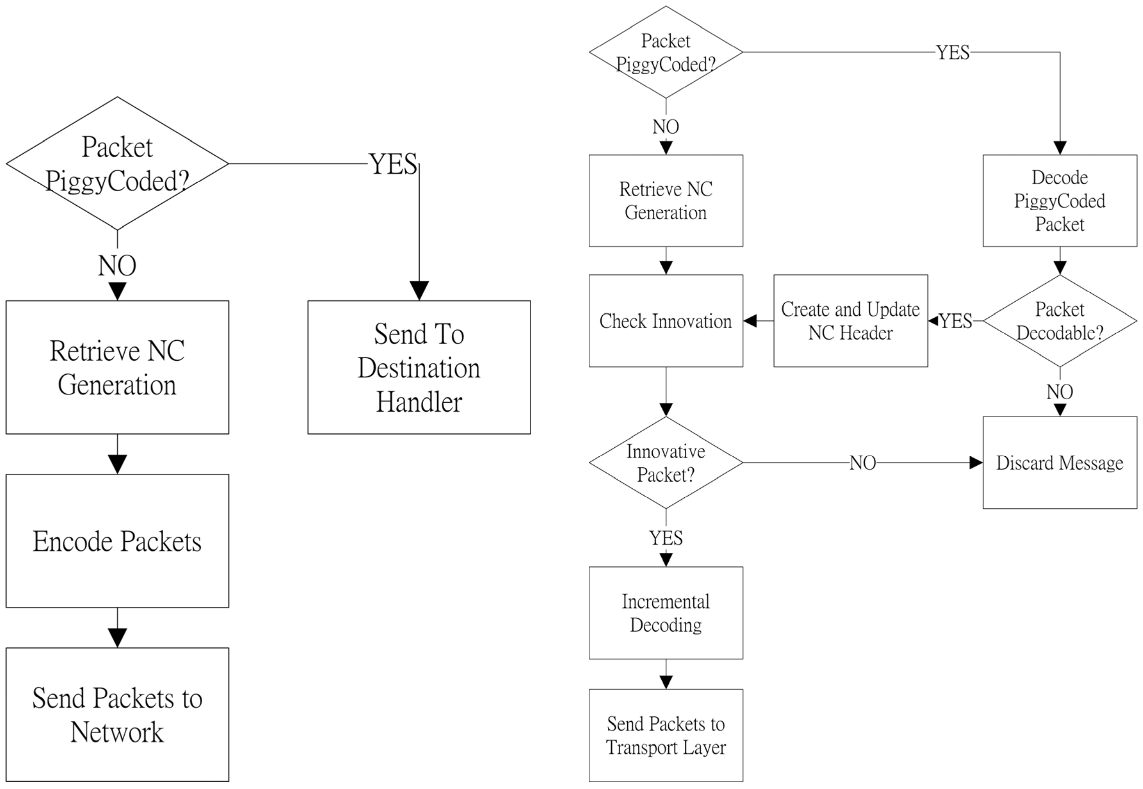
Dotted line stands for receiving by overhearing.

3.2 Coding Flow Chart

As mentioned previous, ComboCoding consists of two coding schemes, intra-flow coding and inter-flow coding. We choose Pipeline Coding as our intra-flow coding, which is applied only to the TCP-DATA flow. A modified version of PiggyCode is chosen as our inter-flow coding scheme. The major difference between the original PiggyCode and our modified PiggyCode is the original PiggyCode implementation sits inside MAC layer, but our implementation manages a separate coding buffer in the network layer to queue up TCP DATA packets. This modification helps ComboCoding to be transparent to both higher and lower layers. Throughout this dissertation, we refer PiggyCode in ComboCoding as the modified PiggyCode that works in the network layer. The original version of PiggyCode is not considered, implemented nor tested in our simulation.

Fig. 14(a) below shows the coding flow chart at sources. 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 packet might be delivered to the source, in which case it will be forwarded to the destination handler function instead.

Fig. 14(b) gives the decoding flow chart at destination. If the packet is PiggyCoded, it has to be decoded from the packet we 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 receiving coded packets in the same generation. ComboCoding will then store and decode the packets and deliver them to the upper layer.



(a) at Source (b) at Destination

Fig. 14 Coding Flow Chart at Source and Destination

Fig. 15 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 be first decoded and delivered to the Pipeline Coding module. If the received packet is innovative, it first checks whether PiggyCode is enabled on this node or not. If the node is allowed to perform

PiggyCoding, it first examines whether the packet should be queued for a given time *T* or not, which is set by the packet source. In our case, TCP source always marks DATA as queued and TCP destination always marks ACK as NOT queued. The reason behind this design choice is that TCP needs feedback to be delivered as soon as possible in order to properly react.

If the packet has to be queued, it will be inserted into the PiggyCode queue, and a timer of the given buffering time *T* is scheduled. If the ComboCoding module does not have any ACK to perform an XOR with the queued DATA, the DATA will proceed to the "Reencode Packet" decision block once the timer expires.

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

If a PiggyCoded packet is not generated, the relay proceeds to determine whether the packet has to be reencoded or not. This reencoding step is designed for the extension of braided path support, which has been shown to be extremely helpful in mobile wireless networks [12]. However, to perform ComboCoding in a braided path and minimize the information loss per hop, relays in network coding have to perform re-encoding, which produces a new packet. The new re-encoded packet will be known only to the node that produced it. As a result, if we allow both DATA and ACK to be re-encoded at each relay, once the packets are PiggyCoded, only the two original senders can decode them. This significantly reduces the redundancy and increase unnecessary overhead. Therefore, ComboCoding does not allow the packet that has not been queued to be re-encoded. For our purposes, the ACK will never get re-encoded to allow the DATA a better chance to be decoded, because it is piggybacked on an ACK.

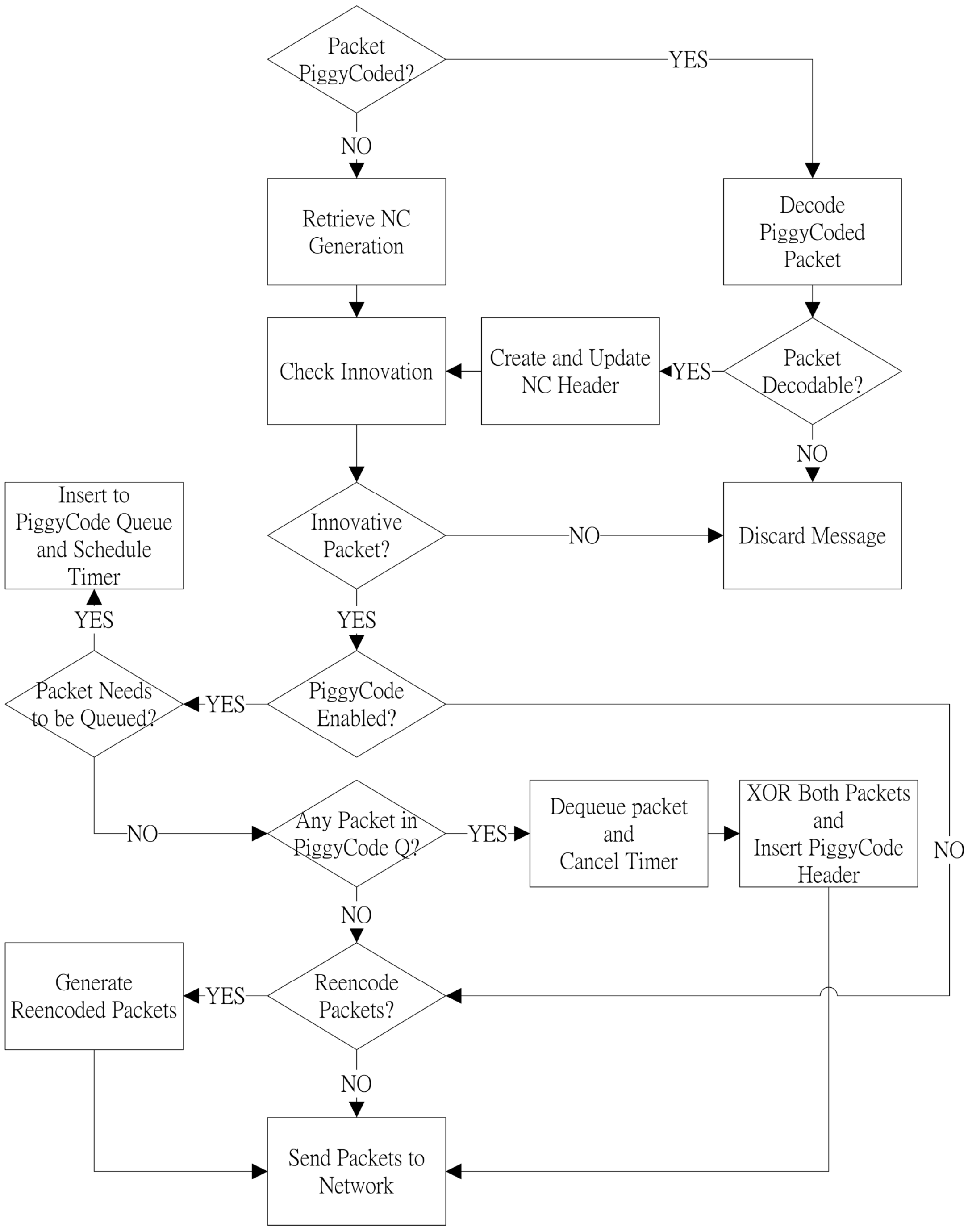

Packet PiggyCoded?
YES
NO
Retrieve NC Generation
Decode PiggyCoded Packet
Check Innovation
Create and Update NC Header
YES
Packet Decodable?
NO
Insert to PiggyCode Queue and Schedule Timer
Innovative Packet?
NO
Discard Message
YES
YES
Packet Needs to be Queued?
YES
PiggyCode Enabled?
NO
Any Packet in PiggyCode Q?
YES
Dequeue packet and Cancel Timer
XOR Both Packets and Insert PiggyCode Header
NO
NO
Generate Reencoded Packets
YES
Reencode Packets?
NO
Send Packets to Network

Fig. 15 Coding Flow Chart at Relay

3.3 PiggyCode Performance Analysis

Our inter-flow coding design is different from the original PiggyCode in that we rely on an additional network layer queue to control XOR opportunities. We next present a first-order model to analyze the performance gain of our inter-flow coding scheme and provide guidance on selecting the queue timer value. We first analyze TCP-NewReno throughput in a wireless multi-hop scenario as shown in Fig. 16 (3-hop string topology). Using the resulting model, we further model the throughput of PiggyCode under the same configuration. Note that in our analysis and in the simulations for model validation, we use standard 802.11b with RTS/CTS disabled and a maximum MAC retransmission of 7 times. The channel bitrate is set to 11Mbps. Also, we assume there is no random error and majority of the losses are due to collisions. Table 4 below summarizes the notations used in the following analysis.

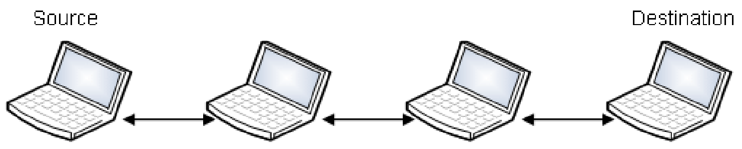


Fig. 16 3-Hop String Topology

Table 4 Inter-Flow Coding Throughput Model Notation

Variable	Definition
RTT	Round trip time that measured by TCP.
n	Number of hops.
T_0	The transmission timeout. Should be somewhere between 3*RTT~10*RTT.
B	Throughput in packet/*t*, where *t* is a given observation period.
$P_{data\text{-}link}$	Link loss rate for data flow
$P_{ack\text{-}link}$	Link loss rate for ack flow
$P_{collision}$	Collision probability.
λ	Packet sending probability at each node. At the source, this should be B/BW, where BW is the MAC layer transmission speed (bandwidth) in packet/t. The theoretical upper bound of λ should be BW/3 in a string topology
λ_{eff}	Effective packet sending probability in the presence of PiggyCode packet.
$E[RTX_d]$	Expected number of transmissions for a data packet.
P_{data}	Overall path loss rate for data flow.
P_{ack}	Overall path loss rate for ack flow.
P_{pc}	Probability of a packet get PiggyCoded.

We first adopt the well-known TCP throughput model proposed by J. Padhye as follows [35]:

$$B(RTT, P_{data}, P_{ack}) \approx \frac{1}{RTT\sqrt{\frac{2P_{data}}{3}} + T_0 \bullet \min(1, 3\sqrt{\frac{3P_{ack}}{8}})P_{ack}(1+32P_{ack}^2)} . \qquad (9)$$

In eq. (9), we assume the average round trip time, RTT, is measureable from the simulation or the traffic trace and the RTT variance due to queueing delay is negligible. We also assume T_0 is a measureable value from simulation or the traffic trace and is in the rage of 3*RTT to 10*RTT. To simplify the model, we assume that TD (triple dup-ack) event, which is the first term of the dominator, is not a function of P_{ack}. The intuition is that the loss of an ACK will never trigger dup acks and thus this term should consider only the data loss rate. The last assumption is the second term of the dominator, the TO (timeout) event, is affected mainly by P_{ack}. This implies the TO is mainly due to a number of consecutive ack losses, whose probability is much higher than losing a number of data consecutively. This is especially true in our PiggyCode implementation since a PiggyCoded packet is sent to the TCP DATA flow direction via unicast and relies on overhearing on the ACK flow direction.

<u>*Data Loss Probability*</u>

We next model the overall path data loss probability as follows:

$$P_{data} = 1 - (1 - P_{data-link})^n , \qquad (10)$$

where $P_{data\text{-}link}$ is the data loss probability per each link. As we focus on the first order behavior, we simplify our model by assuming all links have the same loss probability. $P_{data\text{-}link}$ can be further modeled as:

$$P_{data-link} = (P_{collision})^7 , \tag{11}$$

where P$_{\text{collision}}$ is the collision probability of every packet. In a string topology, it is:

$$P_{collision} = \binom{3}{2}(1-\lambda_{eff})\lambda_{eff}^2 + \binom{3}{3}\lambda_{eff}^3 , \tag{12}$$

where λ_{eff} is

$$\lambda_{eff} = (1-P_{pc})\lambda + \frac{P_{pc}}{2}\lambda = (1-\frac{P_{pc}}{2})\lambda , \tag{13}$$

where P$_{\text{pc}}$ is the probability of a packet get PiggyCoded. The intuition behind eq. (12) is that for every three consecutive nodes, there is a collision if more than one node transmits at a given time instant. Eq. (13) represents the fact that PiggyCoded packets indeed carry two packets in one transmission and thus reduce the number of transmission by half. In theory, λ should be $\frac{B}{BW}$, where BW is the maximum transmission rate in packet per a given time period. Since λ depends on *B* and *B* depends on TCP control feedback, in our first order analysis, we simply select a reasonable constant for λ as the input to our model.

Ack Loss Probability

Similar to the data loss probability, we have the overall path loss probability for ACK flows as follows:

$$P_{ack} = 1-(1-P_{ack-link})^n , \tag{14}$$

where $P_{\text{ack-link}}$ is the ack loss probability per each link. To simplify, we again assume all links have the same loss probability. $P_{\text{ack-link}}$ is further modeled as:

$$P_{ack-link} = (1-P_{pc})(P_{collision})^{7} + P_{pc}(P_{collision})^{E[RTX_d]}, \tag{15}$$

where E[RTX$_d$] is the expected number of transmissions for a data packet, which is:

$$E[RTX_d] = \left(\sum_{i=1}^{7} i \bullet (1-P_{collision}) \bullet (P_{collision})^{i-1}\right) + 7 \times \left(1 - \sum_{i=1}^{7} (1-P_{collision}) \bullet (P_{collision})^{i-1}\right). \tag{16}$$

Here I assume the maximum number of MAC layer retransmission is 7 times.

Model Validation

Fig. 17 shows the n-P_{pc}-throughput chart in 3-D. The range of n (number of hops) is 3~10 and the range of P_{pc} is 0~0.5. Generally speaking, the throughput drops as the number of hops n increases. Also, for all n, they have a similar curve that the throughput increases first as P_{pc} increases but after a certain threshold, the throughput starts dropping. We will next show that from the simulation measurements, the operation range of P_{pc} is less than 0.3. Since if P_{pc} goes close to 0.3, TCP will be unstable and thus system becomes difficult to model.

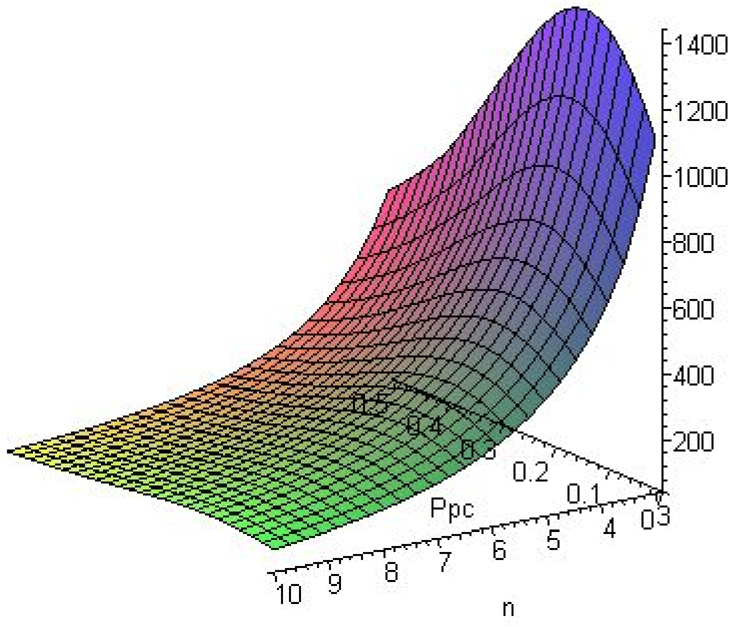


Fig. 17 *n*-*P_{pc}*-throughput (throughput in bps)

The next step to to measure P_{pc}, *RTT*, and T_0 from the simulation in order to validate our proposed model since those are the input values to our model. The simulations are conducted by the following configuration. As shown in Fig. 16, the topology is a 4-node (3-hop) string topology. Each node is 500 meters apart so each of them can hear only its 1-hop neighbors. The MAC layer uses standard 802.11b unicast mode with RTS/CTS disabled. The MAC-layer retransmission limit is set to 7 and the transmission rate is 11Mbps. Transport layer runs standard TCP-NewReno without delayed acknowledgement. The application layer is a generic FTP, which backlogs TCP by continuously generating packets of 1536 bytes. The simulations are done by varying the PiggyCode timer to increase the P_{pc} and measure *RTT*, T_0, resulting in different throughput.

Fig. 18 shows the Timer-P_{pc} plot. P_{pc} is measured by summing up the total number of PiggyCoded packets sent by the relays divided by the total number of data packets and acknowledgements sent by TCP server and client. We notice that as we increase PiggyCode timer, P_{pc} increases dramatically in the first half. The curve becomes flatter after 31ms, which might imply the system reaches its limit and thus TCP is unable to transmit fast enough for more packets to be PiggyCoded.

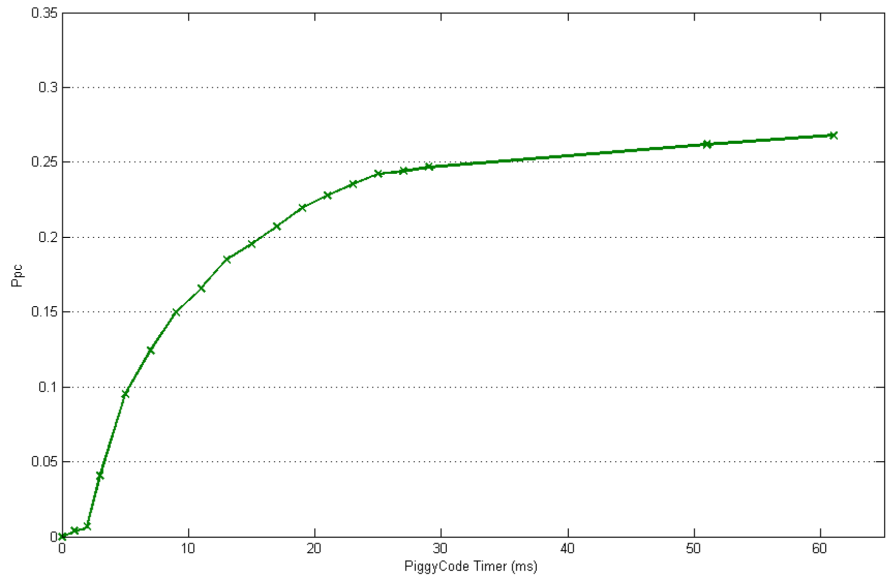


Fig. 18 Timer-*P_{pc}* Plot (Simulation Measurements)

Figures 19 and 20 plot *RTT* and retransmission timer (T_0) for different PiggyCode timer settings. Note that since P_{pc} is not linearly to the timer, in order to show figures in a better scale, all the rest figures use P_{pc} as the x-axis instead of the timer, which can be converted back to PiggyCode timer values by cross-checking Fig. 18. We observe that *RTT* decreases slowly in the beginning and tends to be unstable in the end. Fig. 20 also reflects the same fact in T_0 that the retransmission timer tends to increase extremely fast in the end, which implies TCP is not in a normal operation range. From figures 18-20, we notice that a timer of 51ms (equivalent to a P_{pc} of 0.25) has already pushed the system to its limit as curves turns unstable.

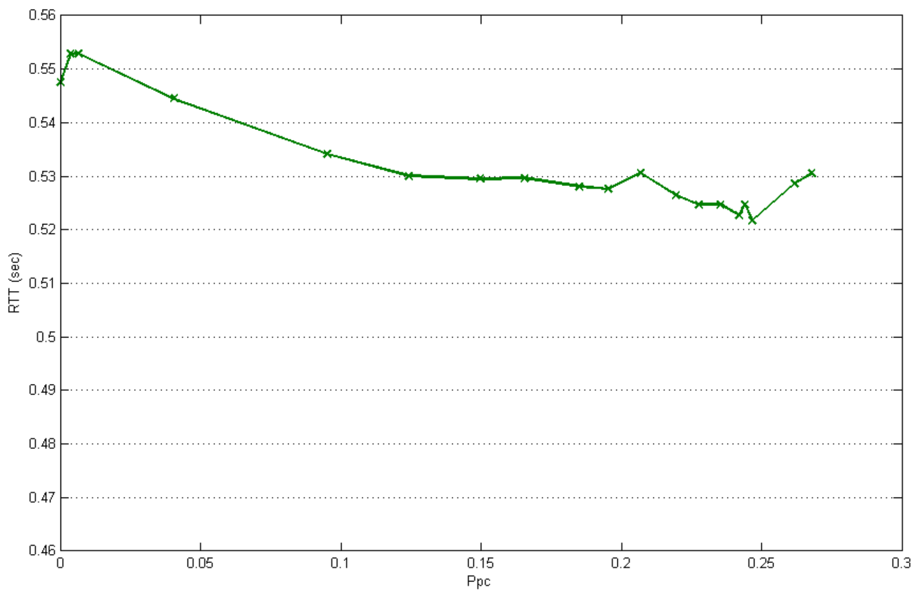


Fig. 19 P_{pc}-RTT Plot (Simulation Measurements)

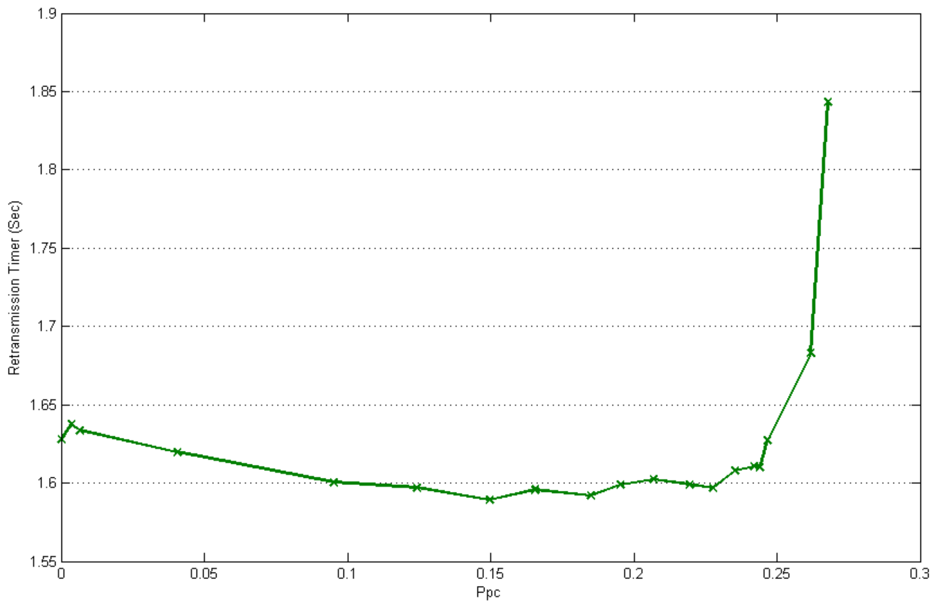


Fig. 20 P_{pc}-T_0 Plot (Simulation Measurements)

Ideally, we should next validate our PiggyCode throughput model by the above values observed. However, as we omit the channel access scheme details, the λ becomes a term that can neither be obtained from the simulation nor be obtained by the model. Therefore, we experimentally pick a constant value of 0.65 for λ, which is within a reasonable range and fits the best to our model.

Fig. 21 shows the link collision probability validation by using a λ of 0.65. The collision probability measured from the simulation is obtained by taking the total number of MAC-layer retransmission divided by the total number of frames sent at all nodes. We observe that the collision probability measured from the simulation drops faster than the model prediction. Also, our model always estimates a higher collision probability. The reason why the model has a higher estimated collision probability is because the measured collision probability considers only the case that a data frame is sent but no MAC-layer ack is received, i.e., an actually collision. However, in 802.11b, MAC layer will also perform random backoff when it senses a busy channel without even sending out the data frame. Since my model considers both cases as "collisions," it results in a higher predicted loss probability. Note that the increment after P_{pc} = 0.25 is because the system is out of its normal operation range.

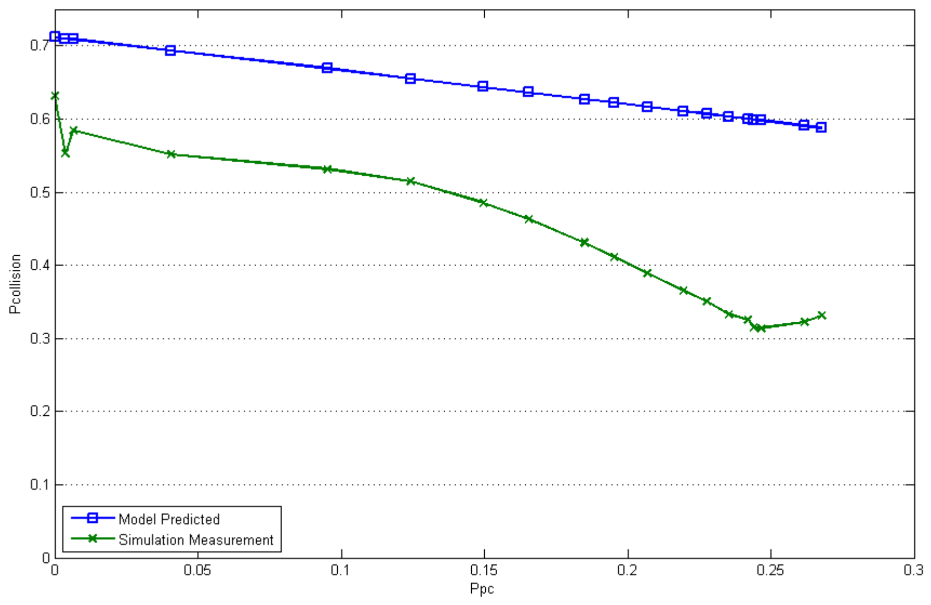


Fig. 21 $P_{collision}$-P_{pc}

Putting together all these values collected from simulation measurements, Fig. 22 shows the throughput measured from simulations and predicted by our PiggyCode throughput model. The estimation is obtained by feeding the measured P_{pc}, RTT, and T_0 values and then times by the size of a data packet (1536 bytes). The need of multiplying by the packet size is because in the original model given in eq. (8), its throughput unit is in number of packets per a given time. Again, we observe that the model has about 10%~30% error in the operation range. The reason why the model is always higher than the measured throughput is because we over-estimate the collision probability and thus the PiggyCoded ACKs have more chances to be retransmitted. However, if we choose a smaller λ resulting in a lower collision probability, then both the peak of the throughput and the turning point will be shifted, which results in a less accurate prediction. A better approach is to further consider the channel access scheme such as binary exponential backoff and the channel sensing behavior. By modeling the channel access scheme in more detail, the model should be able to estimate the throughput more accurately. Nevertheless, our PiggyCode throughput model provides a first order analysis on how to select a PiggyCode timer. We have shown that there is an optimal value for PiggyCode timer and it is a function of the topology and network load. Also, we show that in a 3-hop string topology, a timer value of 5ms to 20ms (P_{pc} of 0.1 to 0.2) affects only less than 8% of the throughput. In other words, the operation range of PiggyCode timer is relatively broad and needless to be fine-tuned case by case.

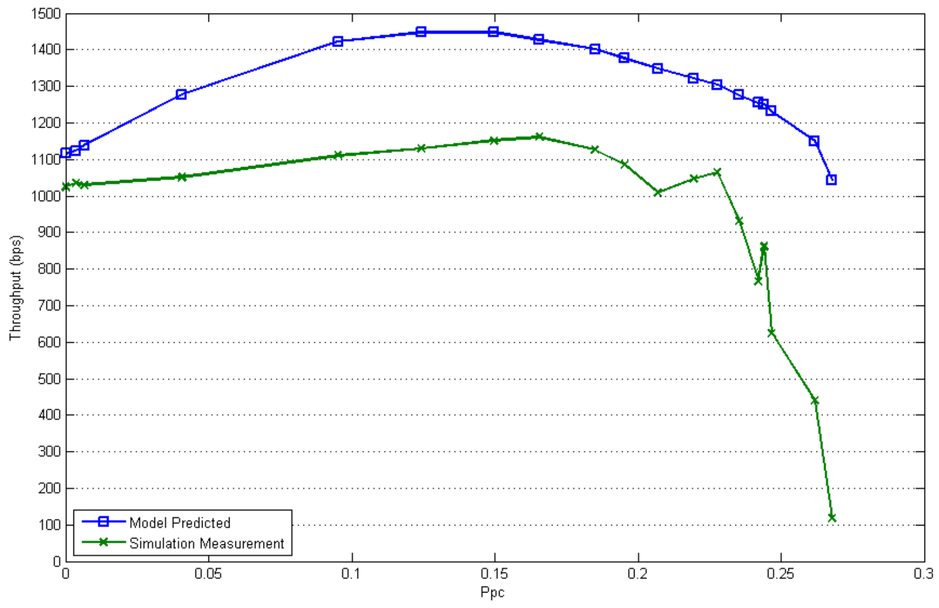


Fig. 22 Throughput-P_{pc}

3.4 Loss Adaptation Algorithm

Since the link quality in wireless networks varies significantly over time, we propose a feedback-based loss adaptation algorithm to dynamically control the coding and forwarding redundancy. In wireless multihop communication, 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 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 nexthop of node *i* in the TCP DATA flow, the instantaneous link loss rate from node *i* to node *i*+1 is estimated as follows:

$$P_i^0 = \frac{M_i - N_{i+1}}{M_i}, \tag{17}$$

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:

$$\overline{P_i}' = \overline{P_i} + \alpha \times \left(P_i^0 - \overline{P_i}\right), \tag{18}$$

where α 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,

$$R_i = (K_i - 1) + \frac{1}{1 - \overline{P_i}}, \tag{19}$$

where K_i is the base redundancy that is need 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.

3.5 Channel Access Scheme

The choice of channel access scheme is always a crucial issue that affects wireless network behavior under any conditions. 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. PseudoBrodcast is also a good fit with PiggyCoding. As mentioned previously, the original PiggyCode is limited by the lack of dual-ACK support in the MAC layer [21]. 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 sent *destined* to 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 [34] 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 [6-7, 20]. As a result, ComboCoding disables RTS/CTS, and our simulations confirm that this choice provides better performance.

3.6 Simulation Results

The proposed ComboCoding scheme was tested on QualNet 4.5 [33], where the module is implemented at the network layer. We compare ComboCoding with the original PiggyCode [20] and the unmodified Pipeline Coding. The simulation topology is a string as shown in Fig. 23. Nodes are 250 meters apart, and the Physical and MAC layer protocols are standard 802.11g, with RTS/CTS disabled. The channel bit-rate is 54Mbps. To exploit the MAC layer reliability, nodes communicate using pseudo-broadcast as in [7]. The transport layer protocol is TCP-NewReno and the application traffic is a Generic-FTP, which continuously generates packets of 1500 bytes and keeps TCP backlogged. The generation size of the random linear coding (intra-flow coding) module is set to 16 packets. The inter-flow coding timer is set to 4ms, i.e., TCP DATA packets will be buffered in the inter-flow coding module for up to 4ms. In all simulation sets, variable link loss rates of up to 50% are introduced in order to simulate a challenged environment subject to random interference and jamming. Note that we assume routing is a given input to our problem and thus all simulations are based on single-path topologies with variable hops and different number of flows in the network. Table 5 summarizes the configuration of the simulation described above.

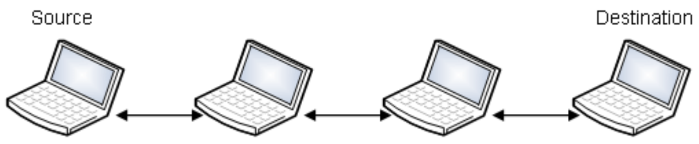


Fig. 23 Simulation Topology

Table 5 ComboCoding Simulation Configuration

Parameter	Value
Node Distance	250 m
Channel Bit-rate	54 Mbps
Channel Access Control	802.11g (CSMA/CA) RTS/CTS Disabled
Transport and Application Layer	Generic FTP/TCP-NewReno
Per Link Packet Loss Rate	0%~50%
Packet Size	1500 Bytes
Generation Size	16 Packets

3.6.1 ComboCoding Overall Comparison

We first evaluate the performance gain of ComboCoding. 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 loss adaptation 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. 24 presents the throughput-to-loss[1] curve of TCP-NewReno without any coding, with PiggyCode, with Pipeline Coding, and with ComboCoding. In Fig. 24, 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 [17], where it was shown that a coding redundancy of 2.5 was needed. A higher redundancy implicitly means more interference and thus in low loss rate cases it does not perform any better than non-coded TCP-NewReno. This anomaly can be addressed by a modification of Pipeline coding to eliminate the non-uniform inclusion of original packets, which is a problem that we are working on and will report on in future papers.

However, under low random loss rates, ComboCoding still achieves the same amount of throughput in the sense that the coding overhead and the number of redundant packets it sends is compensated for by PiggyCode. Because of PiggyCode, in the low loss rate 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

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

PiggyCode significantly reduces the transmission overhead introduced by Pipeline Coding, which 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 goes 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 [35]. 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 Pipeline Coding 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 [6], ETX [36] is used to estimate how many packets have to be sent in order 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 [17]. This reflects in the transmission overhead that will be discussed later.

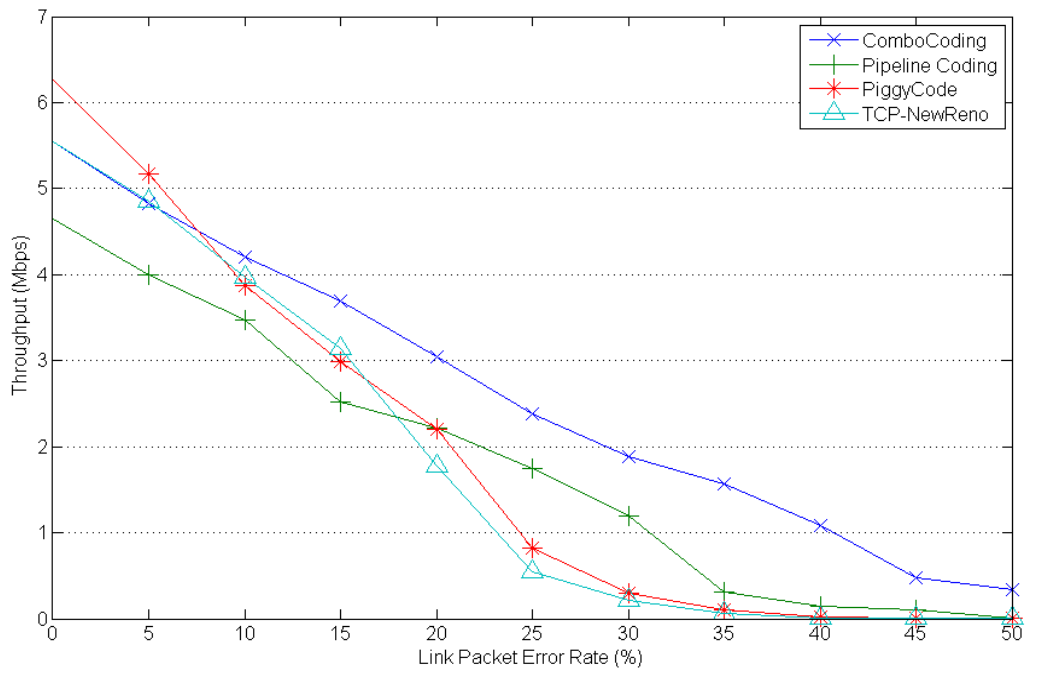


Fig. 24 Throughput-to-Loss

We next evaluate the delay performance as shown in Fig. 25. 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 has about 400Kbps under 50% packet loss rate, the delay of ComboCoding never increases beyond 2 seconds and is consistently lower than the others.

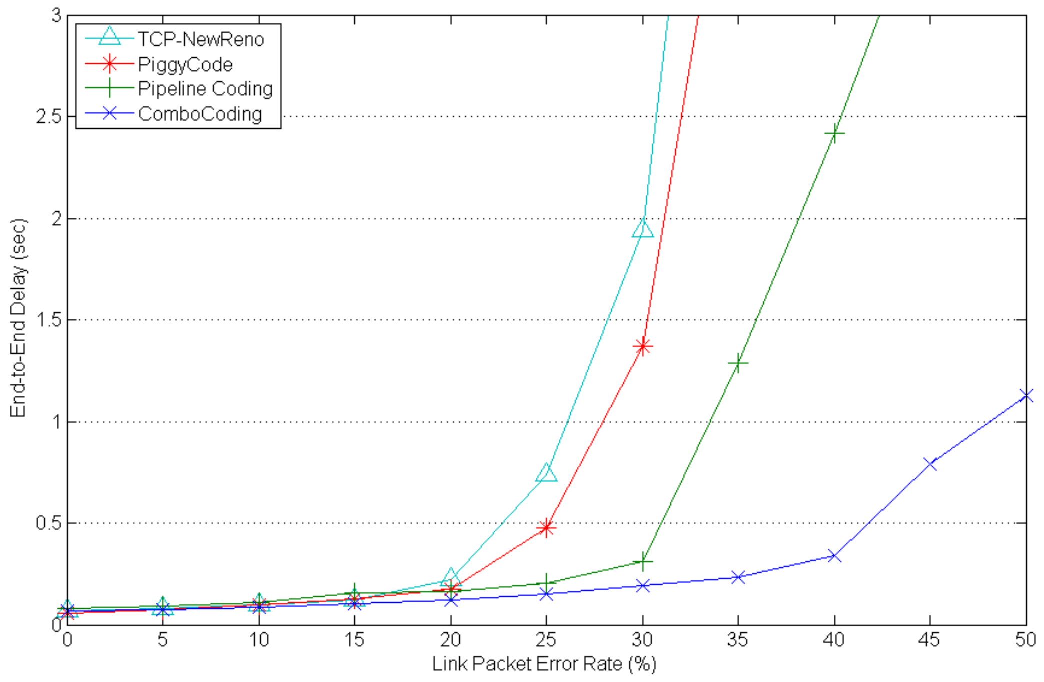


Fig. 25 Delay-to-Loss

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 take into account the impact of reducing interference with PiggyCode packets, so we define the term "transmission overhead" as:

$$\frac{\sum_{i=1}^{N} S_{tx_i}}{D}, \tag{20}$$

where N is the total number of nodes, D is the total number of DATA packets received by the TCP destination, and S_{tx_i} is the number of signals physically transmitted by node i. This metric is based on signals transmitted in the physical layer rather than packets sent in the network layer, since the reduction of transmitted packets also implies a lower probability to interfere with other nodes. Consequently, packet collision probability is reduced and thus fewer signals need to be retransmitted. The value of physical signals transmitted is obtained from the simulation statistic reported by QualNet. Eq. (20) 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. 26. 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 mix 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 best coding redundancy requires more coded packets to be sent in order to recover more losses. From figures 24 and 26, ComboCoding is shown to be robust to losses while reducing the redundant packet transmission overhead by up to 30% when compared to Pipeline Coding.

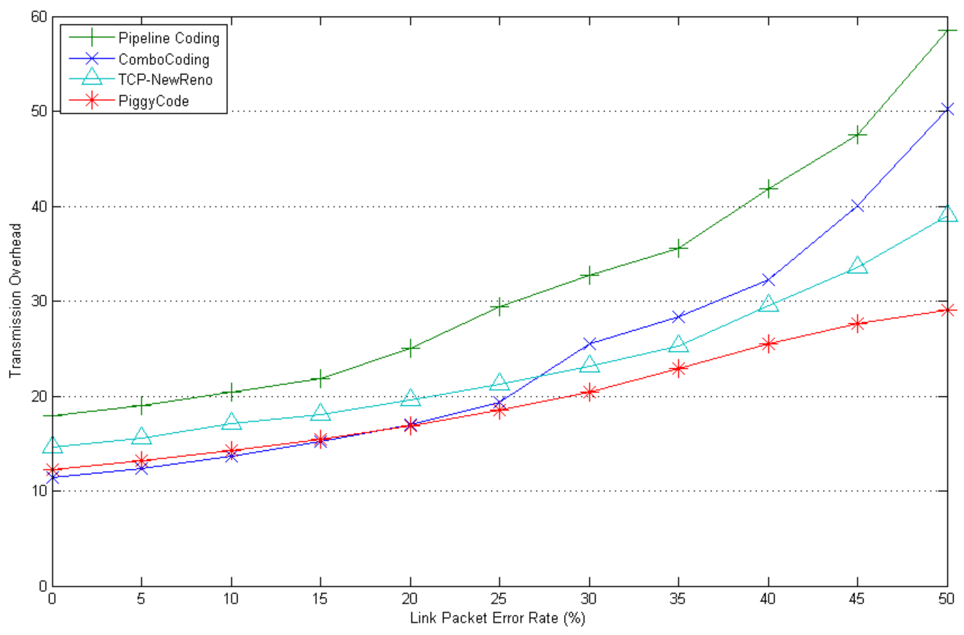


Fig. 26 Overhead-to-Loss

3.6.2 Loss Adaptation Evaluation

We next evaluate the performance of our loss adaptation algorithm. We compare TCP-NewReno without coding, PiggyCode, Pipeline Coding with and without loss adaptation, and ComboCoding with and without loss adaptation in the following loss rate configuration. The application starts sending packets at time 20 and during time 20~50, the packet error rate for all links is 0%. As shown in Fig. 27, TCP-NewReno and PiggyCode outperform all other coding schemes when we have perfect links. Pipeline Coding and ComboCoding perform worse because of the extra redundancy needed. The two configurations with the loss adaptation algorithm perform slightly worse than those without, because the algorithm reacts to short-term loss and thus takes time to lower redundancy.

At time 50~80 seconds, a 40% packet error rate is introduced to every link. During this period, all cases that do not have the adaptation algorithm drop to almost 0bps throughput, while both configurations with adaptation still achieve around 1Mbps. Specifically, ComboCoding with loss adaptation delivers about 20% higher throughput than PipelineCoding with loss adaptation.

At time 80~110 seconds, the per link packet error rate is lowered back 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 coding redundancy. In the absence of dynamic coding redundancy control, both Pipeline Coding and ComboCoding take a longer time to stabilize. This is because random linear coding needs time to give up undecodable generations resulting from loss. In contrast, loss adaptation helps both Pipeline Coding and ComboCoding to efficiently adapt to the loss rate and stabilize. In particular, ComboCoding delivers about 20% higher throughput than Pipeline Coding.

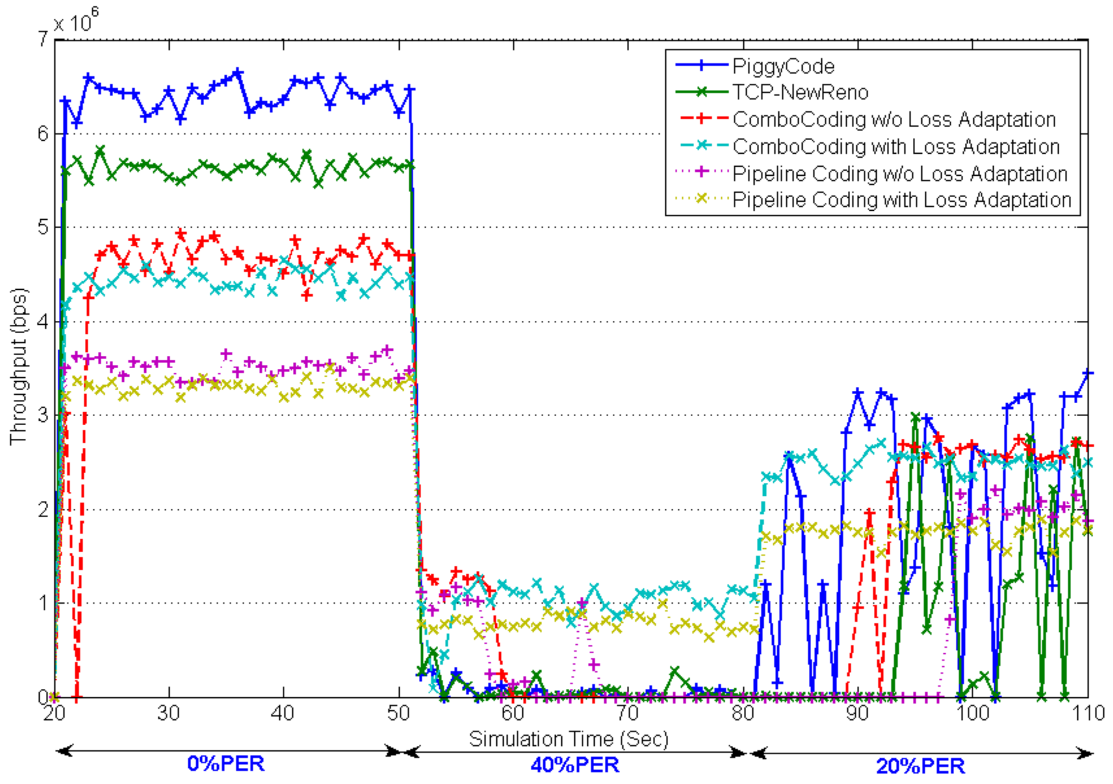
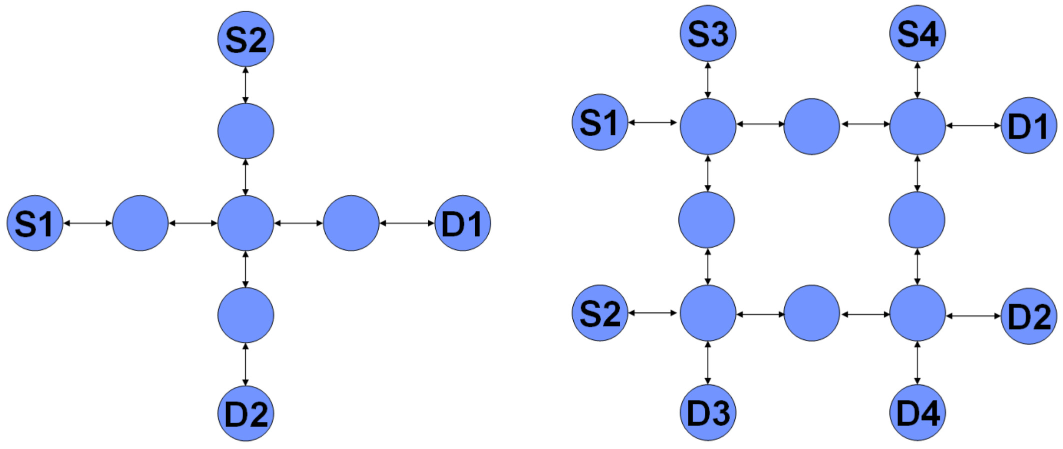


Fig. 27 Goodput over Time for Loss Adaptation

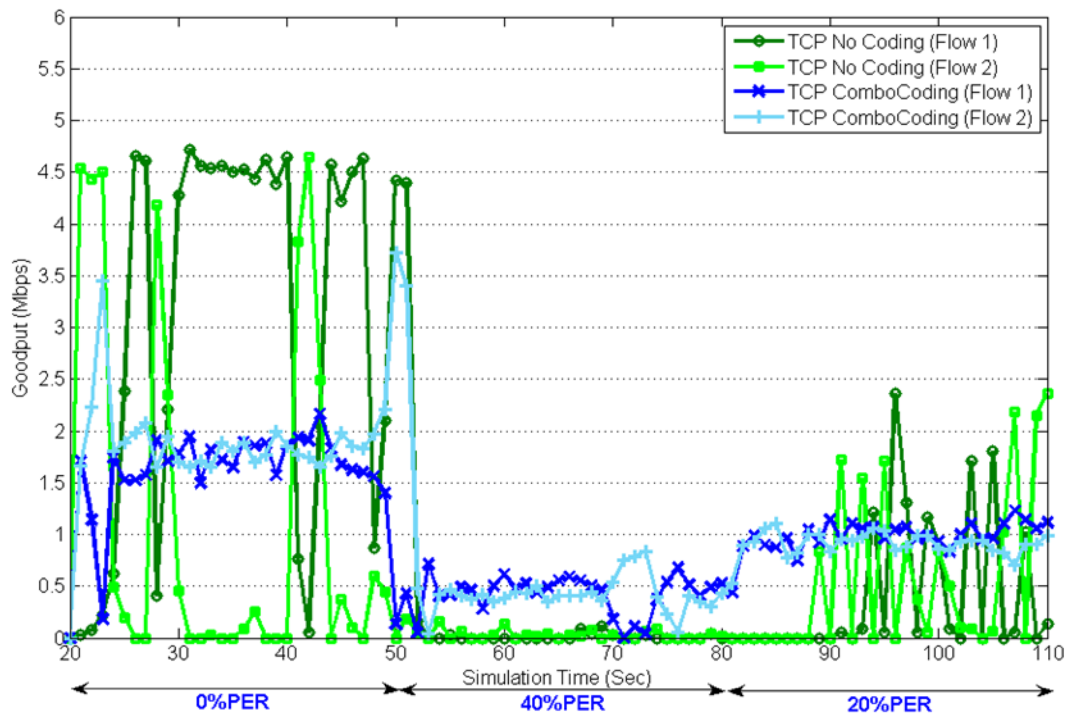
3.6.3 Multiple-Session Scenarios

In out last set of simulations, more coexisting TCP sessions are introduced to further evaluate the adaptive ComboCoding performance in complex topologies. Two topologies, as shown in Fig. 28, are used: X-topology and grid topology. In the X-topology, there are two coexisting sessions, and in the grid, there are four. All other simulation parameters remain the same. Note that nodes can hear diagonally so at most five nodes can fall within the same collision domain. Fig. 29 shows the goodput for different coding options under the 2-flow X-topology.

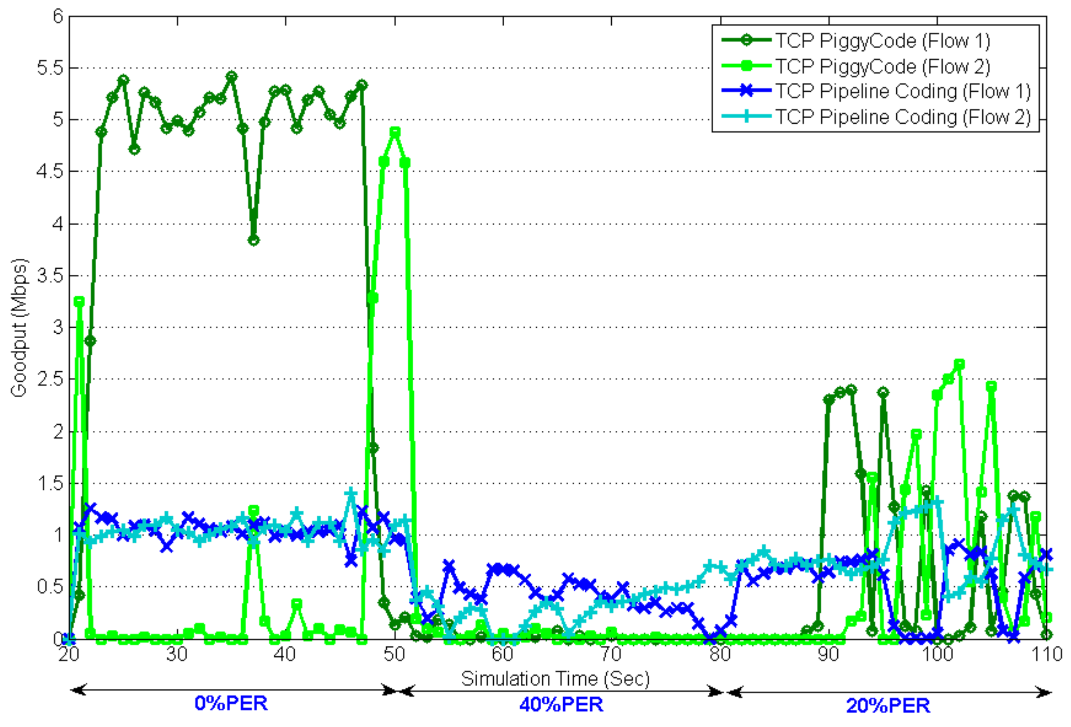


(a) X-Topology (b) Grid Topology

Fig. 28 Multi-flow Topology



(a) No Coding and ComboCoding

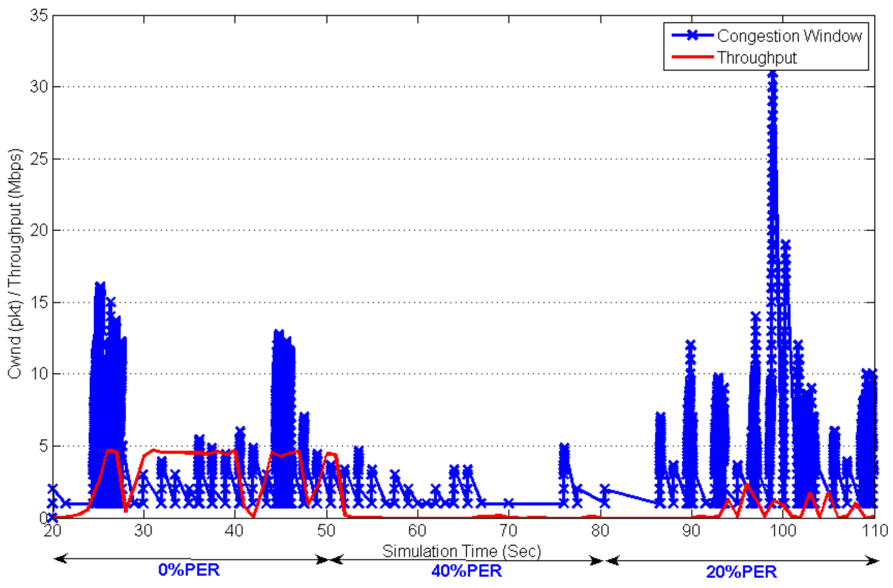


(b) PiggyCode and Pipeline Coding

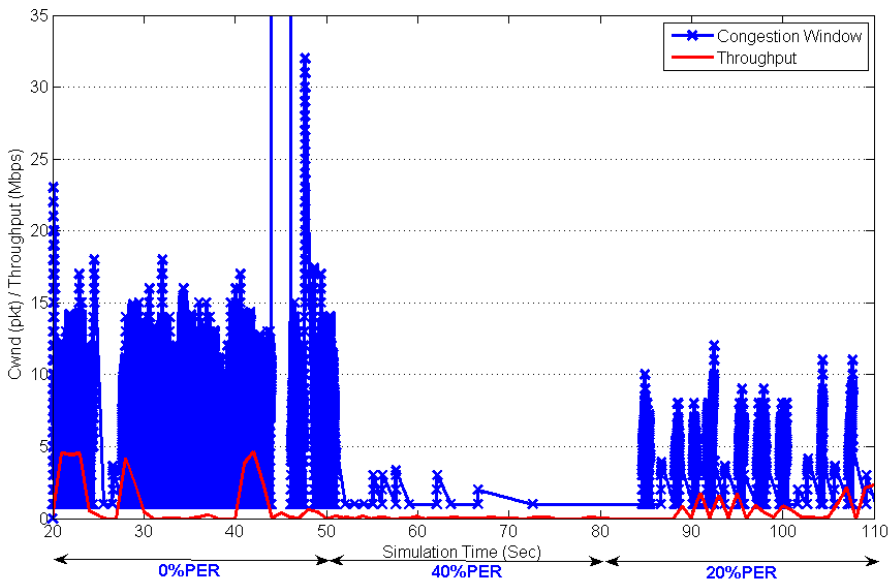
Fig. 29 X-Topology Goodput

We notice that without random linear coding, TCP No Coding and TCP PiggyCode both encounter severe capturing problems. This was expected since TCP is prone to capture as demonstrated in several previous studies [46]. Also, as expected, the aggregate throughput is high, i.e., it is the same as the single TCP throughput since only one flow is transmitting at a time. Unfortunately, this solution is unacceptable, as it may shut off one TCP session for an indeterminate amount of time. Once the random linear coding is enabled, the two flows start sharing the bandwidth fairly. The aggregate throughput is slightly lower than for the non-coded TCP, since now some of the bandwidth is wasted to manage the fair sharing among the flows. ComboCoding further outperforms Pipeline Coding thanks to the inter-flow coding.

To understand capture, note that due to extreme losses, TCP No Coding and PiggyCode suffer frequent timeout events and thus are both in the slow start phase all the time. This was observed from the congestion window time series plot as shown in figures 30 and 31 for TCP No Coding and ComboCoding respectively. We observe that both two sessions get frequent timeout events and thus stay mostly in the slow-start stage. Due to exponential window expansion during slow start, bottleneck buffers tend to be filled up by one of the flows, resulting in capture. In contrast, TCP over ComboCoding, with the assistance of random linear coding, encounters only few timeouts. By inspecting the congestion window behavior of ComboCode, we noticed that these flows stay mostly in the congestion avoidance phase. This leads to more stable flows, and results in better fairness.

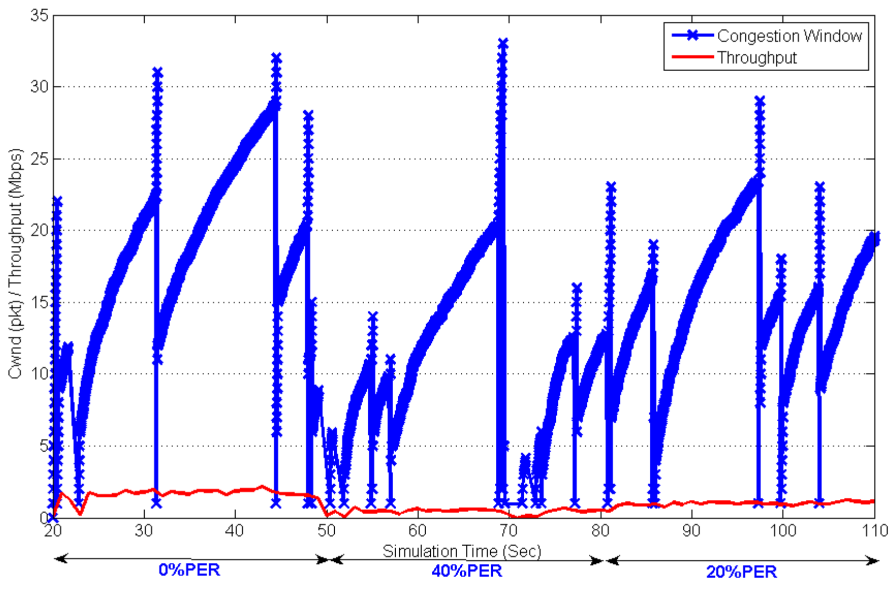


(a) TCP No Coding—Session 1

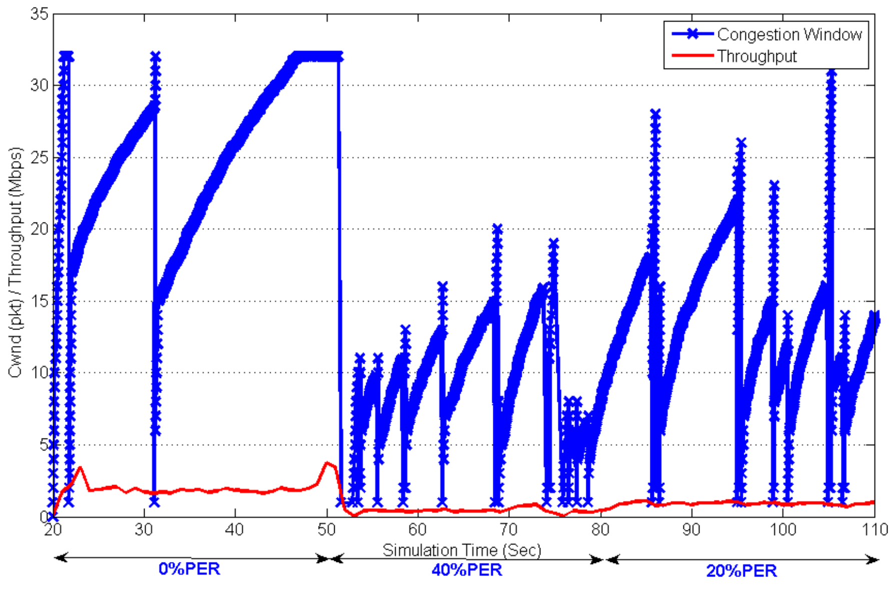


(a) TCP No Coding—Session 2

Fig. 30 Congestion Window Time Series for TCP No Coding



(a) ComboCoding—Session 1



(a) ComboCoding—Session 2

Fig. 31 Congestion Window Time Series for TCP over ComboCoding

We next extend the topology to a 4-flow grid as shown in Fig 28(b). The results in figures 32 and 33 show the overall goodput of all four flows and the instantaneous Jain's fairness index respectively [45]. Consistent with the X-topology, both TCP No Coding and PiggyCode provide a higher aggregate throughput, but a much poorer fairness among flows. In both non-coding runs, flows 1 and 2 and flows 3 and 4 capture the channel in turns, as those two flows do not interfere with each other. For ComboCoding and Pipeline Coding, the overall throughput is lower but the fairness is almost optimal since losses caused by either interference or link quality are overcome by coding, so all four flows are transmitting and sharing the bandwidth.

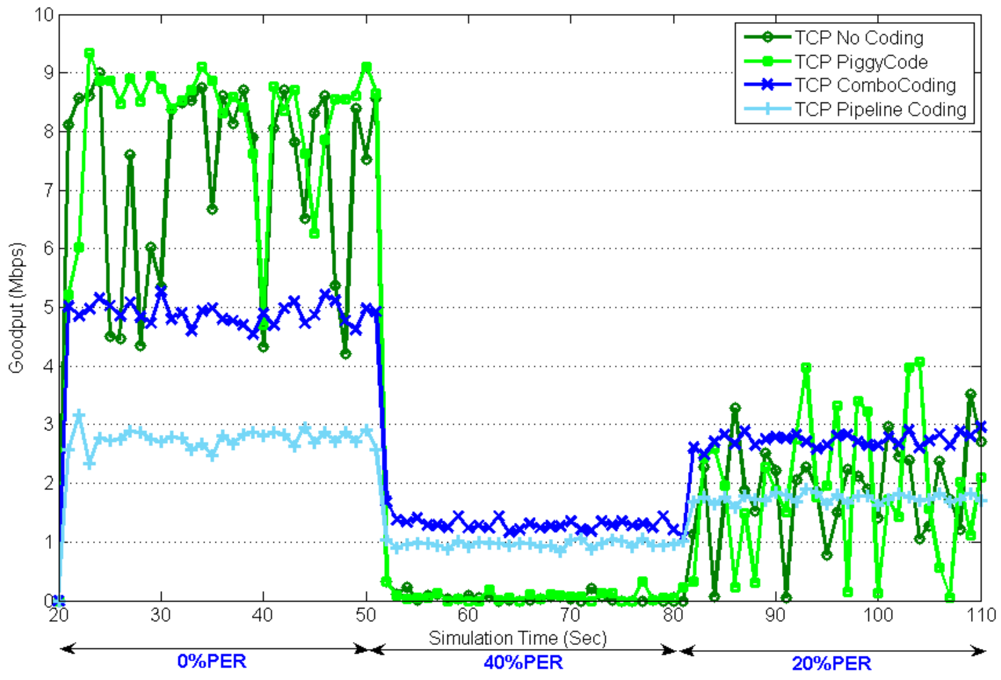


Fig. 32 Grid Topology Aggregate Goodput

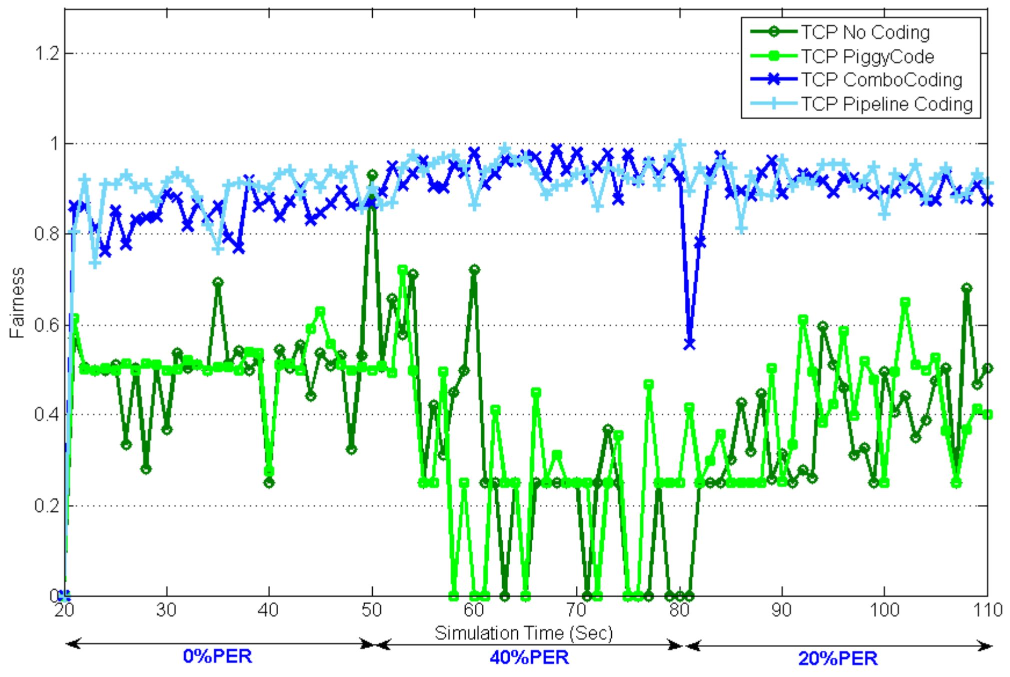


Fig. 33 Grid Topology Fairness

(Note that we mark the fairness as 0 if the overall throughput is 0.)

CHAPTER 4

CodeMP: Network Coded Multipath for TCP Support in Challenged MANETs

In disruptive MANET environments, TCP not only encounters challenges described above, random errors, unstable links, interference, etc, but also suffers from frequent route breaks due to mobility. In order to overcome the problem of frequent route breaks and highly unstable links in disruptive MANETs, multipath routing is inevitable. In [6], S. Chachulski et al. propose a multipath coded routing scheme, called MORE. Subsequent to MORE, many extensions have been proposed [39-41] to further improve its performance. Although these studies have shown the effectiveness of multipath coded routing, most of them rely on explicit control messages and/or explicit ACKs at the network layer. The dependence on explicit control messages can lead to excessive overhead in MANET environments where most of the network parameters, such as number of nodes, network topology, number of flows, buffer length at all nodes, and buffer occupancy by every flow, are changing at all times. Furthermore, all these approaches do not claim to be compatible with and transparent to the TCP layer.

In this study, we address the above problems with conventional TCP running on an adaptive network coded multipath scheme. While totally transparent to TCP, the combination of conventional TCP over multipath, which we refer to as CodeMP, provides a robust solution for TCP in disruptive MANET environments by mitigating random losses via intra-flow coding, and reducing interference through encoding the ACK and DATA packets together. Moreover, without explicit control messages, CodeMP adjusts coding redundancy and multiple-path redundancy adaptively to further deal with route breaks and unstable wireless link quality in MANET scenarios.

The contributions of the proposed adaptive multipath coding scheme are as follows. (1) It provides a practical and efficient approach to exploit multiple-path redundancy in disruptive MANETs. (2) It adapts the redundancy to network changes using a heuristic-based link estimate algorithm. (3) The adaptation algorithm requires no explicit control messages. (4) The design and implementation are transparent to transport layer protocols such as TCP. (5) The proposed coding scheme achieves better fairness in the presence of multiple coexisting TCP sessions in the MANET scenarios.

Note that in order to deploy CodeMP, proper network layer modifications at the wireless nodes are inevitable as for any network coding implementation. However, modifying the network layer at the node itself is well accepted and is part of the MANET design [42].

4.1 Related Work on Supporting TCP in MANETs

Recent related work on network coding strategies for lossy wireless networks mainly falls in two categories: multipath coded routing and single-path coded routing (generally with TCP). Numerous research studies have shown that Network Coding has helped improve the throughput or robustness or both. We review the most popular schemes below. Anticipating the conclusions, we report that no previous study has addressed the problem of TCP performance in lossy, mobile MANETs using multiple paths (for redundancy) and no previous study has evaluated fairness among competing flows.

Multipath Coded Routing

J.-S. Park et al. proposed and implemented CodeCast, one of the first Network Coding schemes designed for a disruptive, ODMRP-mesh based, multicast environment [10]. CodeCast was successfully demonstrated in lossy environments particularly for multicast streaming applications. S. Oh et al. later extended CodeCast to unicast UDP traffic [12], where relays can dynamically switch between random linear coding or plain multipath routing based on network loss conditions. CodeCast and its successor work [12], however, assume fixed redundancy and are thus not very suitable for disruptive MANET environments.

MORE, proposed by S. Chachulski et al., like CodeCast, uses multipath coded routing. It adapts coding redundancy to changing link loss rates [6]. In MORE, relays opportunistically form multiple paths on which packets are re-encoded and forwarded. Based on ETX, relays adaptively adjust their redundancy. Relays also rely on explicit network layer ACKs to determine when a generation is completed so that a new generation can be started. As a result of this “stop-and-wait” design, MORE induces potentially high delay and delay jitter that is not suitable for some TCP applications. Also, MORE uses explicit control messages that tend to cause excessive overhead in disruptive MANETs. In [40], Y. Lin et al. point out that the design of MORE does not fully utilize the available capacity. They introduce an enhanced, window-based flow control algorithm, CodeOR. CodeOR improves the performance of MORE, but it is still not designed for MANETs and still suffers from a significant amount of explicit control message overhead.

Along a similar path, but with a more theoretical emphasis, X. Zhang et al. propose an extension of MORE called Dice [41]. Dice considers the sharing of network resources among two or more UDP sessions, and using a game theory framework, optimizes resource allocation for better efficiency-fairness tradeoffs. The environment is static and loss free and thus the scheme is hardly applicable to MANET scenarios. Another MORE variant is later proposed by the same group in [39]. Different mathematical optimization techniques are used to address the resource sharing problem, this time with random errors. Both work in [39] and [41] provide a good theoretical study of MORE properties using optimization-based approaches. The schemes, however, offer little insight into routing and flow control in a constantly changing MANET. In addition, as in MORE, the dependence on explicit control messages remains a major problem.

In the family of network coding solutions, one must also mention the inter-flow coding solutions, like COPE [7], which have the goal of opportunistically saving spectrum in multi-flow scenarios, generally trading bandwidth for robustness. These schemes do not claim to be suitable for disruptive MANET operations.

Single-Path Coded TCP

While the above multipath proposals address mainly robust unicast and multicast applications without TCP, there have been several single path proposals that are directly targeting TCP. The reason for preferring single paths for TCP is quite obvious—the avoidance of out-of-order packets due to multiple paths. Yet, the single path configuration renders TCP performance precarious in presence of mobility and high errors. In [11], J. K. Sundararajan et al. propose one of the pioneer schemes for single-path coded TCP over lossy links. They show that random linear coding improves TCP performance in lossy scenarios after clever modifications to TCP acknowledgement semantics are made. The work in [11] assumes a fixed Network Coding redundancy. H. Seferoglu et al. extend the above work in I2NC, where, using Network Utility Maximization (NUM) techniques, the network coded TCP protocol design is formulated as an optimization problem [43]. I2NC also employs inter-flow coding, in addition to intra-flow coding, to mitigate interference. Nevertheless, since I2NC is based on NUM framework, it has the inherent limitation of requiring the fine tuning of several parameters, such as number of nodes, network topology, number of flows, buffer size at all nodes and buffer occupancy by every flow. As a consequence, re-optimization must be triggered whenever a parameter changes value. Similar to the approach presented in [43], we propose a practical network coding design, ComboCoding, which uses both intra- and inter-flow coding and adapts the redundancy to the estimated loss rates. As described in chapter 3, ComboCoding further eliminates all explicit control messages and relies completely on overhearing. As we shall see, this combination strengthens the protocol and improves its fairness in multi-flow disruptive scenarios.

Most of the above single-path coded TCP studies, however, assume static scenarios with a pre-determined single-path route from the TCP source to the TCP destination. This is totally impractical in MANET environments. Besides, although each of the above schemes supports multiple concurrent sessions and works in lossy wireless networks, none has been tested in a disruptive MANET where mobility and time-varying jamming and random errors are the norm. Given these premises, it appears that no study has addressed the situation where TCP must run on a multiple paths (for redundancy); moreover no study has reported on the ability for two or more TCP flows to share resources fairly in a multipath, network coded scenario. In this study, we approach this problem leveraging our prior experience with TCP on single path. Our aim is to leverage single-path coded TCP and the multipath coded routing results to design a practical scheme for TCP traffic in disruptive MANETs.

4.2 CodeMP Design and Implementation

CodeMP consists of three main elements: random linear coding (RLC) with adaptive redundancy, adaptive multipath routing, and ACK Piggy coding. The first two components interact closely with each other to provide efficient and reliable communications. We chose Pipeline Coding as presented in chapter 2 as the RLC scheme for CodeMP. The ACK Piggy coding is used to mitigate TCP DATA-ACK interference.

4.2.1 Adaptive Redundancy Control and Multipath Routing

The redundancy and multipath adaptation algorithm consists of four parts: slice assignment, forwarding control, cross-slice loss estimates, and intra-slice redundancy distribution. The main idea of our adaptive control algorithm is to first group together the nodes within the same distance from the flow source. In our design, such a group is called a “virtual slice” or a “slice.” We use the number of hops from the flow source as our distance metric. Once nodes assign themselves a slice number, they then start learning and estimating the network conditions and adjust the redundancy accordingly. If a node notices that it delivers only a few packets to the adjacent downstream slice or it is not within a reasonable distance to the destination, the node suppresses all of its packet forwarding until the network conditions change. Note that all the control and adaptation are done per “flow,” so each flow will form their own multipath forwarding set with different redundancy at each node. In our design, TCP DATA flow and ACK flow within the same session are treated as two separate flows.

Slice Assignment and Forwarding Control

To determine the number of hops a node is away from the flow source, an extra 1-byte field called "hop-count" is added to our coded packet header. It is initialized to zero by the flow source and incremented at each relay. Note that all these procedures are applied only to "innovative packets," to avoid loops. Fig. 34 below presents an example of a snapshot of a sliced network topology.

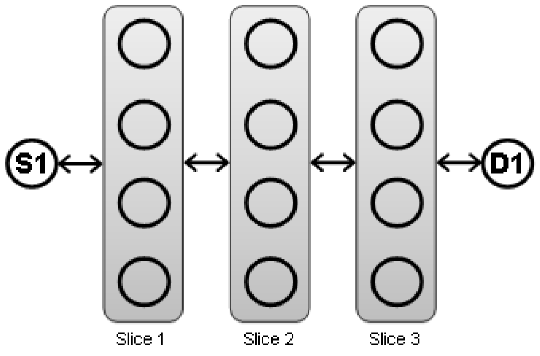


Fig. 34 Slice Example

Since topology changes dynamically, each relay maintains a "hop-count" window of 10 samples for each flow. Every time the sampling window is filled, a relay resets its hop-count to the minimum number of hop-count number in the current sampling window. A similar approach is also used so that relays could learn the current shortest path length from the flow source to flow destination. Based on the hop-count and the shortest path length, a relay could determine itself a non-helpful node and suppress all of its forwarding until it is within a reasonable range to the flow destination.

Cross-Slice Loss and Redundancy Estimate

Based on the above slice assignment algorithm, relays then actively estimate the loss rate between their own slice (*slice*^{i}) and the adjacent downstream slice (*slice*$^{i+1}$), where the superscript *i* denotes the hop-count calculated as previously described. Each relay *j* annotates in the packet header the total number of packets it has sent in the current generation, which is denoted by S_j. The relays in the downstream slice then stamp in the header N^i_{sent}, which is the sum of all S_j. Also, another field, N^{i+1}_{recv}, is added to the header to denote the total number of packets a particular node in *slice*$^{i+1}$ has received. Both N^i_{sent} and N^{i+1}_{recv} will be overheard by adjacent upstream relays. These additional counters are all of two bytes length and thus to avoid overflow, they are reset when every new generation arrives. Fig. 35 below shows an example of the above procedure where node 1, 2, and 3 are the upstream nodes in *slice*^{i} and we assume only one downstream node in *slice*$^{i+1}$ for simplification.

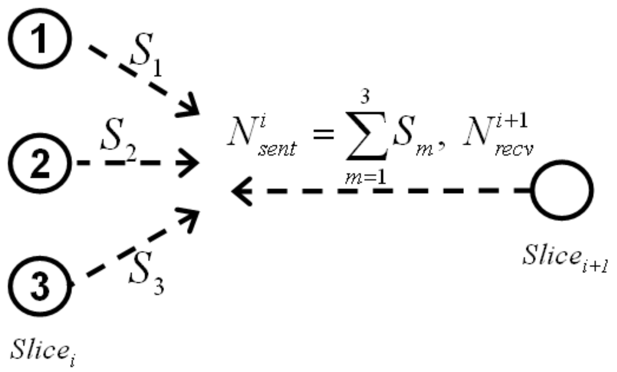


Fig. 35 Piggybacked Counters

Based on the overheard counters N^i_{sent} and N^{i+1}_{recv} from downstream nodes, a relay in the *slice*^{i} then computes an instantaneous loss rate from *slice*^{i} to *slice*$^{i+1}$ as follows:

$$P_{si} = 1 - N^{i+1}_{recv} / N^i_{sent}. \tag{21}$$

The relay then takes an average for every ten P_{si} samples, which is denoted by $\overline{P_{si}}$. Since the loss rate may vary significantly over time, a smoothed loss rate is then calculated by taking the exponential moving average of the instantaneous link loss rate as follows:

$$\widehat{P_{si}}' = \widehat{P_{si}} + \alpha \times \left(\overline{P_{si}} - \widehat{P_{si}} \right), \tag{22}$$

where α is a proper smoothing factor (set to 1/6 in our simulation experiments). The cross-slice redundancy from *slice*^{i} to *slice*$^{i+1}$ is then set as follows:

$$R_i = K + \frac{1}{1 - \widehat{P_{si}}'}, \tag{23}$$

where *K* is the base redundancy (set to 0.8 in our experiments). The value 0.8 is chosen based on our simulation studies in which it introduces the minimum additional redundancy to avoid under-estimate.

Intra-Slice Redundancy Distribution

The redundancy distribution algorithm is based on the heuristic that if a relay learns that its neighboring nodes are making good progress, it should reduce its share to avoid overstressing the network.

The procedure works as follows. Upon receiving an innovative packet from upstream node X, a relay will initiate a re-encoding event. Namely, it creates a number of new re-encoded packets that include the ID = X in the header. When relays in the upstream slice overhear this packet, they increment the count of the trigger node. Upon a new generation arrival, based on the track of triggering nodes in the previous generation, a relay *j* then calculates the percentage of packets that are triggered by the relay itself or an unknown node, which is denoted by $share_{in}^{j}$. Note that we use an aggressive approach here to consider all re-encoding events triggered by unknown nodes as if triggered by the relay itself. This is because overhearing has a high loss rate in MANETs and in case a relay misses piggybacked feedback, it has to assume no other nodes will compensate for such a loss.

Fig. 36 below shows an example of the above $share_{in}^{j}$ calculation procedure, where we assume generation size is 4 and there is a redundant packet generated when *DoF* = 3. In this example, the downstream node informs all upstream nodes that *DoF* 1 to 3 are all triggered by node 2 and only *DoF* = 4 is triggered by node 3. Therefore, nodes 1, 2, and 3 set their $share_{in}^{j}$ to 0, 3/4, and 1/4 accordingly. Assuming node 1 loses the feedback information for *DoF* = 1 and receives only the others, node 1 will then consider it is the triggering node of *DoF* = 1 and then set its $share_{in}^{j}$ to 1/4 instead.

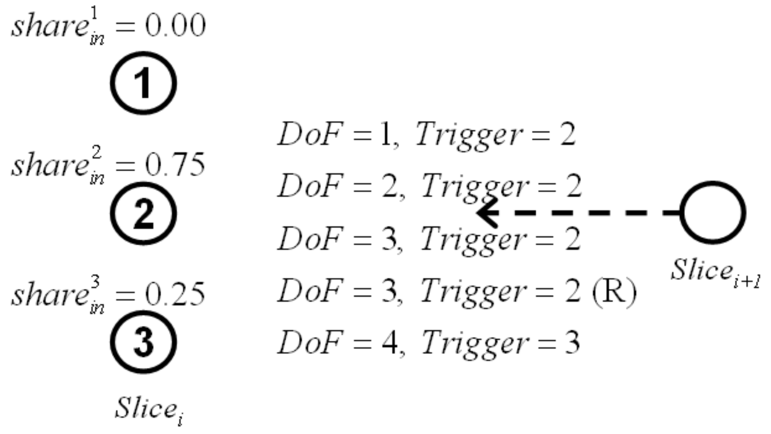


Fig. 36 $share_{in}^{j}$ **Example**

Based on the estimated $share_{in}^{j}$, a local redundancy for node *j* in slice *i* is then updated as:

$$R_i^j = R_i \times share_{in}^{j}. \tag{24}$$

Again, the local redundancy is per flow, which means relay *j* now uses redundancy R_i^j only for the corresponding flow. With this estimate, if a node learns that all of the triggers are by some other neighbors, it will effectively suppress itself in the next generation.

Impact of Inaccurate Estimates

As the adaptation is designed for MANET scenarios, inaccurate estimates are inevitable. Our coding scheme tries to avoid under-estimation of loss rates and relies on TCP to recover from over-estimates. In the current design, the base redundancy K and the aggressive estimate of redundancy share introduce extra redundancy to protect against under-estimate. Over-estimates lead to congestion and more losses, not recoverable by RLC. This triggers the TCP congestion control mechanism, which in turn reduces the sending window. As the TCP sending rate slows down, the contention is alleviated and thus the redundancy falls back to the affordable range. In addition, our adaptation treats congestion/collision losses and random errors identically. Thus, flows might all encounter more losses when congestion happens, which eventually exceeds the working range of redundancy and triggers TCP congestion window reduction, similar to over-estimates. It is possible to further shorten this feedback cycle by introducing loss discrimination algorithms to CodeMP.

4.2.2 Ack Piggy Coding

To alleviate TCP DATA-ACK self-interference, we adopt a network layer XOR-based coding that is extended from our previous ComboCoding design. The idea of such an XOR-based inter-flow coding is similar to COPE [7], but with modifications to address special types of bi-directional flows—TCP DATA and ACKs.

Similar to ComboCoding, the Ack Piggy coding module employs an inter-flow coding buffer at the network layer. All TCP DATA packets will be buffered in the coding module at each relay for a given **Piggy coding timer** *T*. If an ACK arrives before timeout, the DATA will be XOR'ed with the ACK and the inter-flow coded packet will be broadcasted. If no ACK arrives in *T*, the DATA will be re-encoded based on the algorithm given in the previous section and sent to the next-hop via broadcast. If DATA and ACK packets are not of the same length, padding zeros are appended to the shorter packet.

In addition to our previous design, there are still changes needed to accommodate multipath scenarios. In the single path scenario, it has been shown by L. Scalia et al. that XORing DATA and ACK within a TCP session guarantees decodability [20]. Even if the DATA flow is intra-flow random linear coded (RLC), on single path, this decodability property still holds. However, in multi-path routing, decodability is no longer guaranteed since only those nodes that have overheard the corresponding RLC coded packets can decode the XOR packet. As a result, if a packet is first RLC coded and then XOR coded with another RLC coded packet, the multipath redundancy is reduced.

Due to this reason, in our multipath coding scheme, all ACK flows are not RLC coded (i.e., generation size of ACK flows will be set to one). With this modification, the decodability of (RLC coded) DATA packets will be higher since all nodes that have heard the ACK can decode the XOR packet. Nevertheless, since XOR mixing of DATA and ACK precludes intra-flow RLC mixing, and since the latter mixing is the one that provides redundancy, ACK piggybacking leads to some reduction in multipath redundancy and thus impacts robustness.

Fig. 37 below shows an example of the ACK Piggy coding implementation. In this example, we assume A1 has been heard and buffered at nodes 5, 6, and 7. At time t_1, node 2 first sends a DATA packet C_1, and then at time t2, node 6 sends ACK A_1. At time t_3 Node 4 then XOR C_1 and A_1 and broadcasts the XOR packet. At time t_4, node 2 can decode $C_1 \oplus A_1$ and continue propagating A_1. At the same time instant t_4, assume nodes 1 and 3 cannot hear node 2 but receive $C_1 \oplus A_1$. They will not be able to decode and must discard the XOR packet. On the other hand, since ACKs are sent uncoded, it is practical to assume nodes 5 and 7 could receive A_1 from either node 6 or any other relays. Therefore, it is highly likely that nodes 5, 6, and 7 are able to decode $C_1 \oplus A_1$ and continue propagating C_1. Also note that in the same example, if node 2 does not receive $C_1 \oplus A_1$ due to any reason (either jamming or even it just moves away), no nodes will be able to recover A_1. This design choice reduces the multipath redundancy to the ACK flows, which have been found through simulations that it has caused instability to TCP.

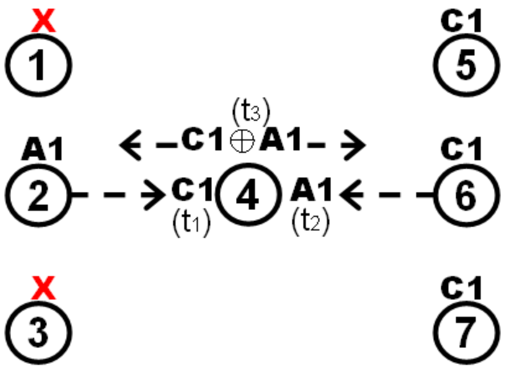


Fig. 37 Ack Piggy Coding Example

In the proposed scheme above, we have simplified our system design for this early study of multipath coding in MANETs. We are aware of a number of design enhancements, which we discuss briefly here, and which we intend to investigate in future work. First, our current design XOR an incoming ACK with only the oldest DATA packet in the buffer. It is possible to further improve the multipath redundancy for ACK flows by allowing ACKs to be XOR'ed with all or with a number of the buffered DATA packets. For example, in Fig. 37, if by the time node 4 receives A_1, it buffers not only C_1 from node 2 but also C_1' from node 1 and C_1" from node 3, node 4 can instead generate 3 different XOR'ed packets $C_1 \oplus A_1$, $C_1' \oplus A_1$, and $C_1'' \oplus A_1$. Such a modification would allow A_1 to be potentially decoded and propagated by nodes 1, 2, and 3.

In addition, our design buffers TCP DATA and waits for TCP ACK, which so far is an engineering decision since we found this works better in our simulations. It is also possible that one could buffer TCP ACK and wait for TCP DATA, although it might need additional tuning. Further, we use a fixed timer of 5 ms in all our simulations. We have found from our simulations that 5ms is not always the best timer for all cases. More analysis and experiments are required to fine tune the Piggy ACK strategy.

4.2.3 Impact of Packet Reordering

One of the major challenges for TCP over multiple concurrent paths is to packet reordering. Simulation shows that in a naïve multipath broadcast scheme without network coding, data packets frequently arrive out-of-order. Out-of-order deliveries lead to dup-acks, which are treated as congestion indications and thus cause the sender to unnecessarily reduce its congestion window. In CodeMP, the out of sequencing due to multiple paths is prevented by Network Coding. In fact, the NC middleware at the destination reassembles packets within each generation and delivers them in sequence to the destination. Duplicate ACKs can be created by multiple paths on their way back to sender. However, ACKs are stamped with the generation number. If they are recognized by the source to belong to the same generation, they are dropped. In CodeMP, however, there is a possibility that a new generation arrives before the proceeding generations are not fully decoded, in which case a duplicate ACK is generated and it is properly interpreted as a congestion indicator (as it belongs to a new generation). Recall that the window increases linearly in CodeMP, spanning multiple generations. In case of heavy congestion, generations fail to be timely assembled. Then, a timeout causes a window reduction, and so does a triple dup-ACK. Our simulation traces show that dup-ACKs are extremely rare so that one can say that practically network coding resolves the TCP sequencing problem over multiple paths. The larger the generation size, the rarer the out-of-ordering but the higher the coding overhead and the end to end delay. From simulation, we have found experimentally that a generation size of 8 is the best compromise in most situations.

4.3 Simulation Results

CodeMP was tested on QualNet 4.5. We evaluate our proposed coding scheme in both static and MANET scenarios. For static topologies, we compare the multipath coding scheme with non-coded single-path TCP and a single-path TCP with ComboCoding. For MANET scenarios, we compare the multipath coding scheme with TCP running on a popular MANET routing protocol, OLSR [44]. In both static and MANET scenarios, we also compare our scheme with a multipath broadcasting approach, in which all relays forward a received packet once; a "unit generation size" random linear coding scheme is used to eliminate packets that have been forwarded by the same relay.

In all simulations, unless otherwise specified, nodes are 150 meters apart, and the Physical and MAC layer protocols are standard 802.11g, with **RTS/CTS disabled**, even in the MAC unicast mode. The channel bit-rate is 54Mbps. In all multipath runs, nodes communicate using pure-broadcast as we rely on multipath opportunistic forwarding. For baseline single-path simulations, nodes communicate using pseudo-broadcast as in [7]. The transport layer protocol is TCP-NewReno and the application traffic is a Generic-FTP, which backlogs the TCP port with continuously generated 1500-byte packets. The generation size of the random linear coding (intra-flow coding) module is set to 8 packets in all simulations.

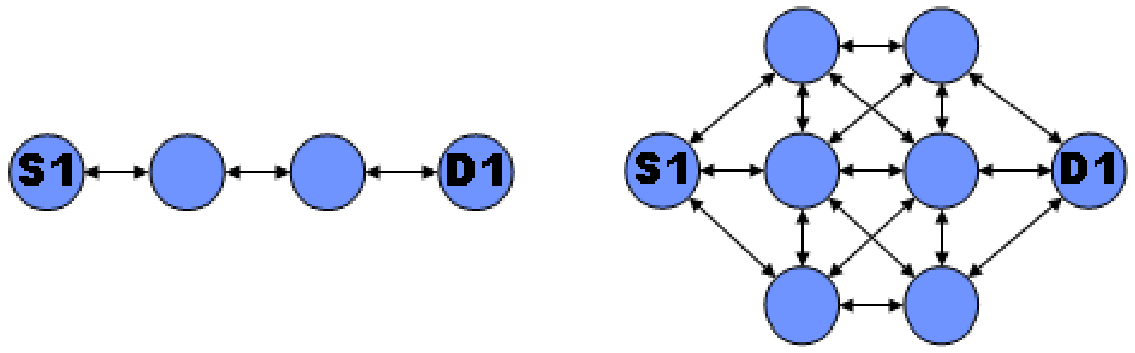
The ACK Piggy coding timer is set to 5ms, i.e., TCP DATA packets will be buffered in the Ack Piggy coding module for up to 5ms. In all simulation sets, variable link loss rates of up to 40% are introduced in order to simulate a challenged environment subject to random interference and jamming. Also, unless otherwise noted, all simulations last for 110 seconds and the FTP application starts at 20 second.

4.3.1 Static Scenarios with Time-Varying Jamming

In the first set of simulations, we perform a series of static topology runs to validate the effectiveness of the proposed coding scheme and to collect baseline measurements. In this set of simulations, all single path topology assumes known routes and thus uses pseudo-broadcast; all multipath (grid) topology assumes no topology knowledge and uses pure-broadcast.

Single-Session Scenarios

We first perform several single TCP session simulations over a 3-hop topology as shown in Fig. 38. Two configurations are used for the single path topology: no coding and ComboCoding. Three configurations are used for the multipath (grid) topology: multipath broadcasting, multipath coding, and multipath coding with ACK Piggy coding. The multipath broadcasting is done by setting the generation size to 1 with a fixed redundancy of 1.00 at every node.



(a) Single Path (b) Grid

Fig. 38 Single Session Static Topology

Fig. 39 plots the instantaneous TCP goodput over time. The application starts sending packets at time 20 seconds, and the packet error rate (PER) for all links is 0%, 40%, and 20% during the time intervals of 20~50 seconds, 50~80 seconds, and 80~110 seconds respectively.

Table 6 below summarizes the average TCP goodput for each period. The first observation is that with zero errors, single path without coding performs the best. In a perfect scenario with given route and no random errors it is well known that coding is ineffective. This fact is confirmed by Fig. 39 results. CodeMP is designed for mobility and random errors, it cannot dynamically adapt to perfect link conditions. A future extension of this work will address the ability to monitor link loss and disable coding, i.e., setting generation size = 1 and redundancy = 0, when the loss estimate drops below a threshold, say 5%.

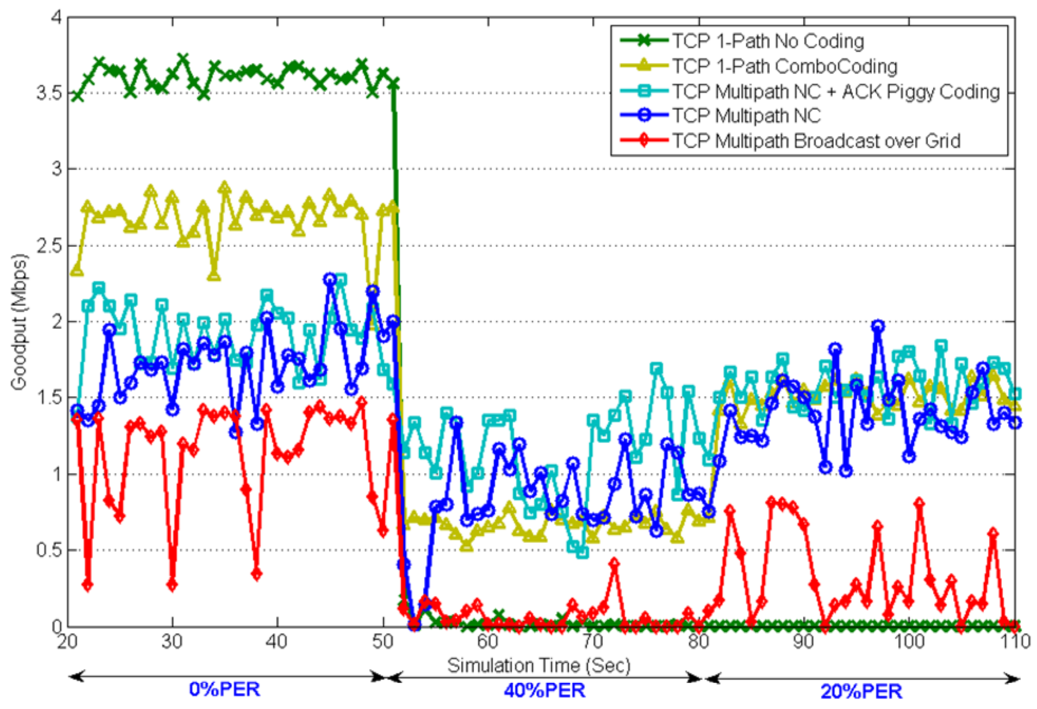


Fig. 39 Single Session Instantaneous TCP Goodput

Table 6 CodeMP Single Path Average TCP Goodput (Mbps)

Config	0% PER	40% PER	20% PER
1-Path No Coding	3.61	0.02	0.00
1-Path ComboCoding	2.66	0.67	1.51
Multipath NC + Piggy Coding	1.92	1.16	1.56
Multipath NC	1.71	0.87	1.38
Multipath Broadcast over Grid	1.13	0.10	0.31

We next find that as random losses occur, TCP without coding fails to work. Also, single path TCP + ComboCoding works better than multipath TCP + NC when links have no random errors, but with 40% random errors, multipath NC outperforms single path ComboCoding since it exploits multipath redundancy. For multipath runs, we found that multipath NC with ACK Piggy coding achieves 10% to 30% higher goodput than without Ack Piggy coding. Besides, TCP + multipath broadcast performs the worst, which is mainly due to the out-of-order data packets.

Fig. 40 shows a typical snapshot of the received TCP data sequence number and the corresponding timestamp for three selected schemes: 1-path ComboCoding, CodeMP (without ACK Piggy coding), and multipath broadcast. Note that since each scheme has different progress, to show threes lines in a single snapshot, we use different offsets Y_0 for each of them as noted in the legend; in other words the real sequence number should be $Y + Y_0$. In Fig. 40, each data point represents a TCP sequence number received by the TCP destination along with its corresponding receiving timestamp. If all data packets arrive in order, it should show a monotonically increasing line. Each drop in Fig. 40 implies an out-of-order data deliver, which will then trigger a dup-ACK. We find that for both 1-Path ComboCoding and CodeMP, sequence numbers grow stably over time. In contrast, in multipath broadcast, they often arrive out of order. This is because the single-path ComboCoding has virtually no chance to create out-of-order packets, and CodeMP, mitigates reordering by virtue of network coding generations. Although we still see from our packet trace that CodeMP has out-of-order data packets, it happens rarely and does not cause noticeable impact. Also note that Fig. 40 omits single-path no coding and CodeMP with Ack Piggy coding for a cleaner presentation. However, the single-path no coding demonstrates a similar pattern as single-path ComboCoding. Moreover, both CodeMP with and without Ack Piggy coding show a similar trend. Hence, we conclude that it is packet reordering that causes the dramatic goodput degradation of multipath broadcast.

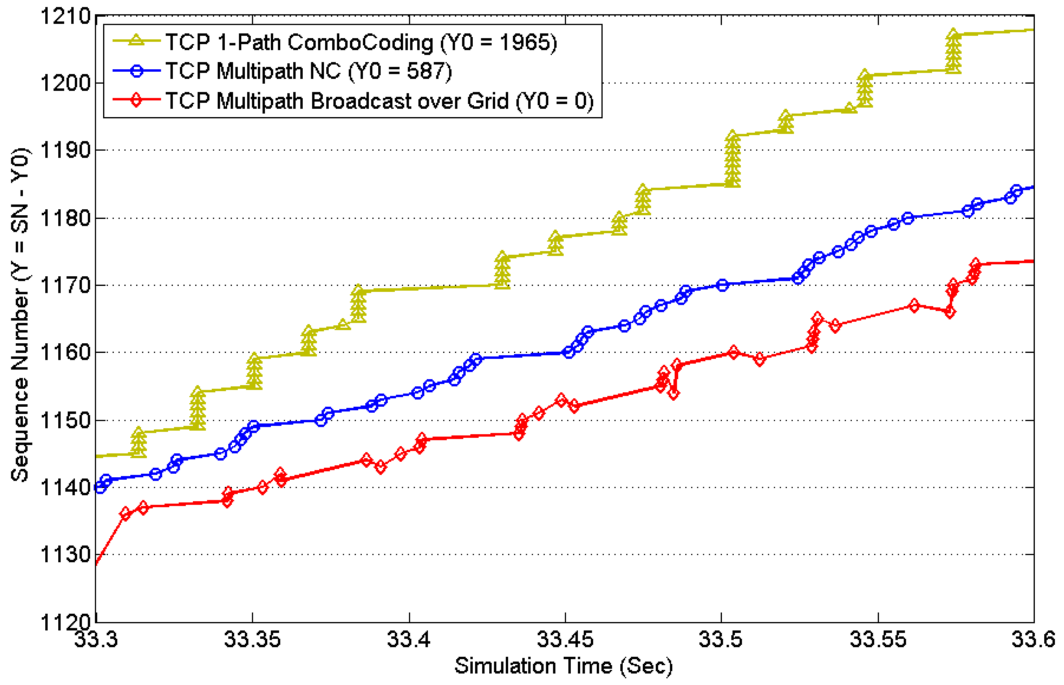


Fig. 40 Received TCP Data Sequence Number over Time

Fig. 41 below shows the normalized transmission overhead for each scheme. The normalized transmission overhead is defined as:

$$\frac{\sum_{i=1}^{N} S_{tx_i}}{D \times N}, \tag{25}$$

where N is the total number of nodes, D is the total number of DATA packets received by the TCP destination, and S_{tx_i} is the number of MAC frames physically transmitted by node i. The number of MAC frames transmitted is obtained from the simulation statistics reported by QualNet. Eq. (25) can be interpreted as the average number of MAC frames needed per node per successful TCP DATA packet delivery.

We find that the overhead in single path experiments is higher than in multipath ones. This is because for single path, unicast MAC is used, which produces relatively inefficient link-layer retransmission. In the multipath cases, broadcast MAC is used. Redundant RLC packets compensate for losses, causing less overhead than MAC retransmission. We also found that in this static single session simulation, ACK Piggy coding does not significantly reduce the overhead since the TCP DATA-ACK self-interference does not present as a major problem in this simple multipath configuration.

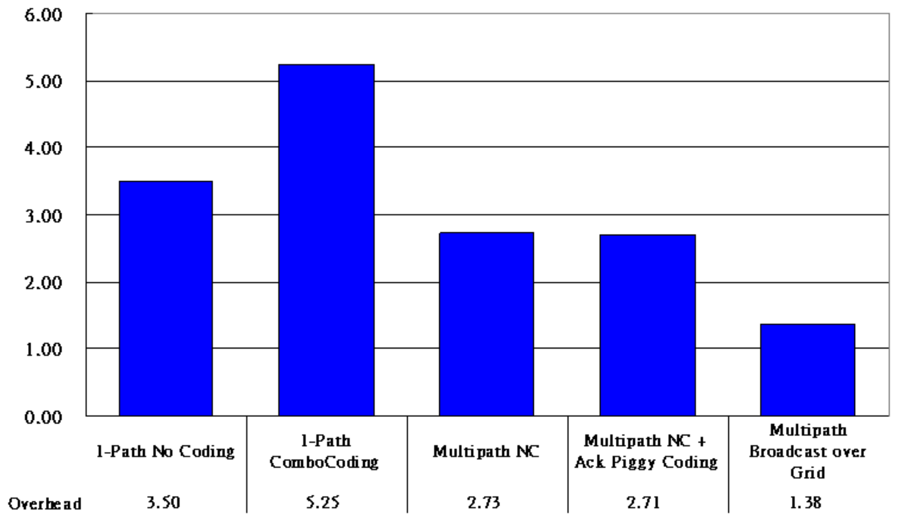
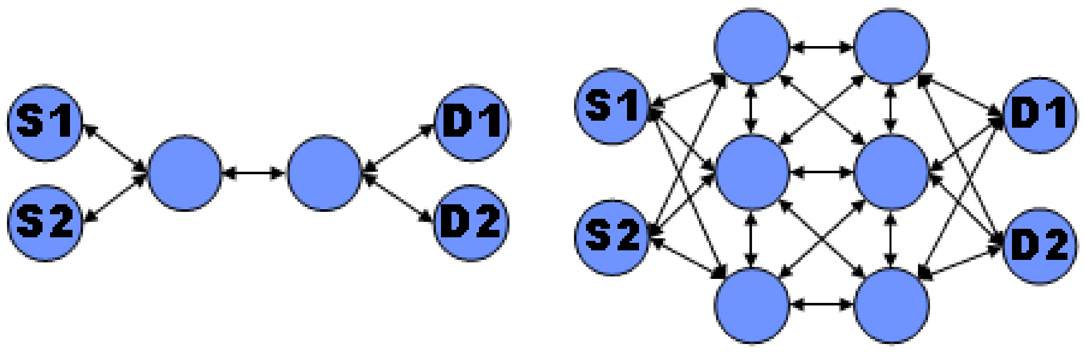


Fig. 41 Single Session Normalized Transmission Overhead

2-Session Scenarios

The next set of experiments reconfigures the topologies to accommodate two TCP sessions as shown in Fig. 42.



(a) Single Path (b) Grid

Fig. 42 2-Session Static Topology

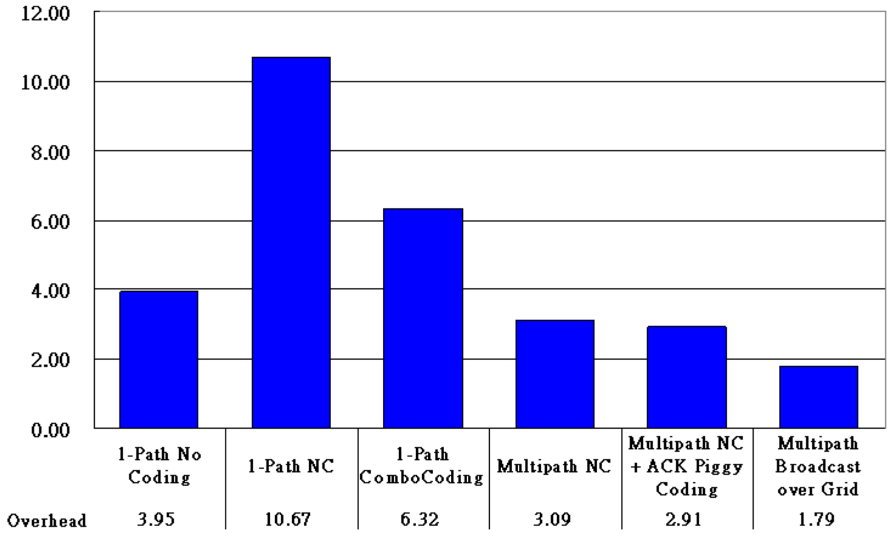
Table 7 summarizes the TCP goodput per session for each setting in each loss rate period. Note that we also consider the simple single-path adaptive NC without Ack Piggy coding in this run, which is literally ComboCoding with zero Ack Piggy buffer timer. The results show a similar trend except that the single path ComboCoding works the best. We also notice that Ack Piggy coding helps more in single path than in multipath.

We focus on two important performance measures of the 2-session experiments: aggregate throughput and fairness. In terms of aggregate throughput, we note that (for the Piggy ACK versions—both single- and multi-path) the aggregate throughput in the 2-session case is only marginally lower (by at most 10%) than the single session case. Moreover, the two sessions share the resources fairly. In contrast, the no coding versions exhibit extreme unfairness, especially for 20% loss rate, where complete capture by one flow is observed. This brings up another important benefit of Network Coding especially when combined with Piggy ACKs, namely the ability to maintain fair sharing among TCP sessions, even in extreme loss situations.

Table 7 CodeMP 2-Session Average TCP Goodput (Kbps)

Config	0% PER	40% PER	20% PER
1-Path No Coding	1087.90	65.12	0.00
	2113.12	63.08	159.74
1-Path NC	749.57	226.10	473.91
	756.53	324.40	504.22
1-Path ComboCoding	1196.03	283.85	736.05
	1375.03	410.42	661.50
Multipath NC	1163.67	553.37	864.26
	647.58	537.40	660.28
Multipath NC + Piggy Coding	932.25	371.92	780.70
	883.92	577.13	684.44
Multipath Broadcast over Grid	959.69	131.07	618.50
	700.83	74.96	2.05

Fig. 43 shows the normalized transmission overhead for each setting. Similar to the single session runs, single path schemes have higher overhead than multipath schemes. Under 2 sessions, ACK Piggy coding also helps reduce overhead in the multipath since it reduces the number of transmissions and hence mitigates interference.



	1-Path No Coding	1-Path NC	1-Path ComboCoding	Multipath NC	Multipath NC + ACK Piggy Coding	Multipath Broadcast over Grid
Overhead	3.95	10.67	6.32	3.09	2.91	1.79

Fig. 43 2-Session Normalized Transmission Overhead

4.3.2 MANET Scenarios—Single-Session Corridor Mobility

We next evaluate the performance of CodeMP under mobility scenarios. We first consider the corridor model given in Fig. 44. In the corridor model, we configure three groups that are equally located between a static source and a static destination. We place four nodes in each group. Each node moves within its corridor using random 'way-point' model with a pause time of 10 seconds. We vary the maximum moving speed for separate runs and set the minimum speed to 10 m/s slower than the maximum speed. Since this is a dynamic topology where routes cannot be pre-configured, a dynamic MANET routing algorithm is required. For the single path runs, we select the state-of-the-art MANET routing protocol, OLSR [44]. We use all the default values specified in RFC 3626 for OLSR control timers. For multipath, we use the multipath broadcast scheme as a naïve multipath without redundancy.

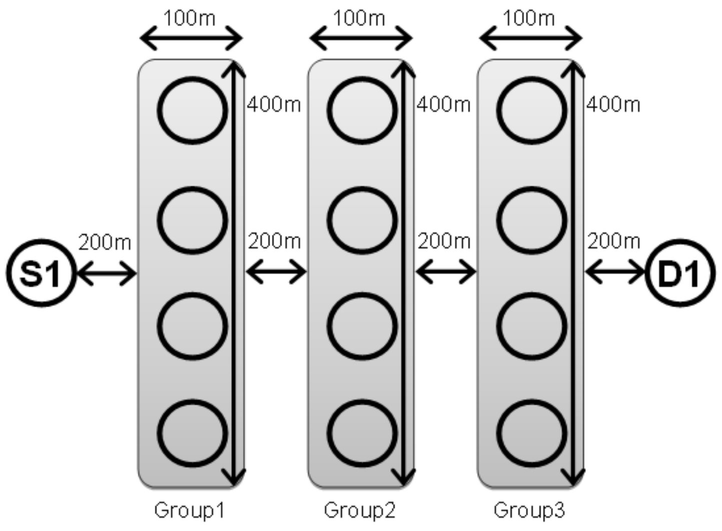


Fig. 44 MANET Corridor Model

Fig. 45 below shows the average TCP goodput vs. maximum moving speed graph. As expected, the goodput degrades for all cases as the moving speed increases. OLSR under high mobility degrades very rapidly, as expected. From the packet trace, OLSR actually shows a bimodal behavior. When OLSR can maintain a valid route, it delivers an instantaneous goodput as high as 7Mbps. As soon as the route breaks, OLSR drops to zero goodput and takes several seconds to recover from the route change. As nodes move faster, OLSR needs longer recovery times resulting in lower goodput. Multipath broadcast, likewise, is not as efficient as multipath NC. However, it is still more robust to mobility than OLSR. We next notice that in this structured topology, DATA-ACK self-interference is a major issue and thus ACK Piggy coding improves the performance significantly. With ACK Piggy coding, CodeMP improves goodput by 30% in low mobility and by 100% in high mobility.

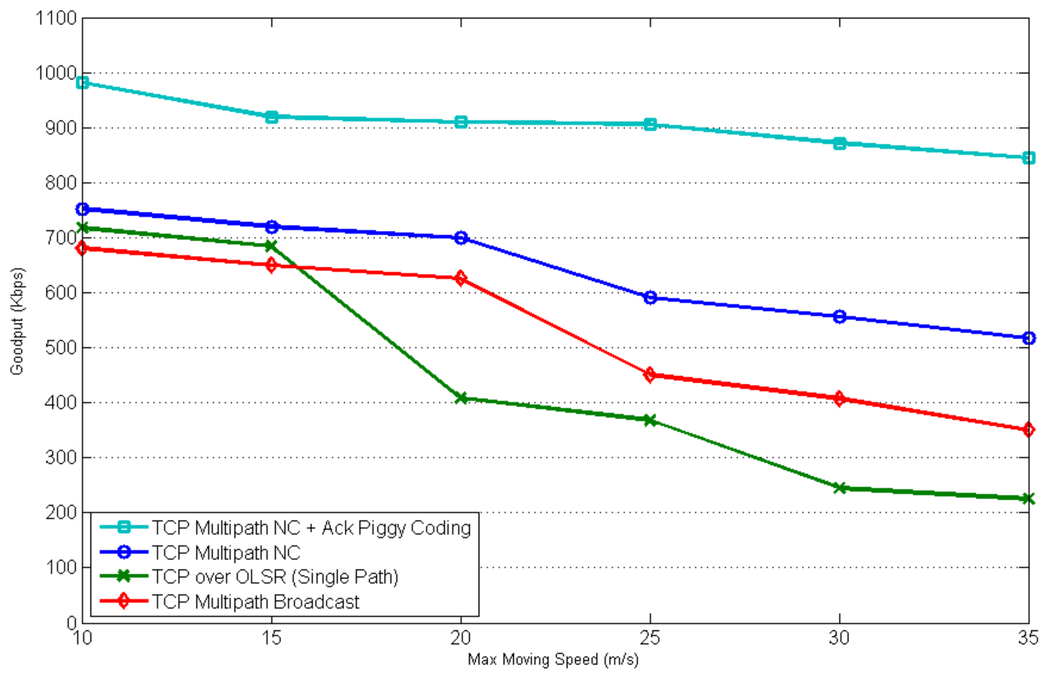


Fig. 45 Corridor Mobility Goodput vs. Max. Moving Speed

4.3.3 MANET Scenarios—Single-Session Global Mobility

The previous experiments have shown that CodeMP provides efficient and robust communications over "structured" scenarios with local mobility. We next move to a more global mobility model, as shown in Fig. 46. In this model, 20 nodes move in a random 'way-point' model with 10-second pause time. TCP source and destination nodes are fixed as in the corridor mobility. All other nodes move with maximum speed varying between 10 and 35 m/s. Minimum speed is 10 m/s slower than maximum speed as in the corridor setting. We again compare CodeMP with the two non-coded schemes, OLSR and multipath broadcast.

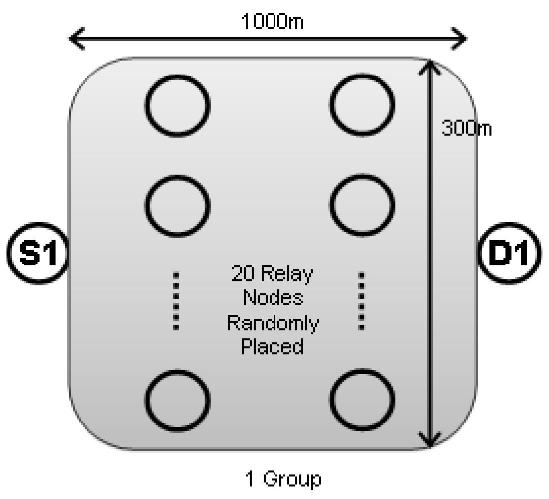


Fig. 46 MANET Single-Group Model

Fig. 47 below shows the average TCP goodput vs. maximum moving speed chart. We first notice that in this more mobile scenario, OLSR fails to work when the maximum speed reaches 35 m/s. Even with around 25 m/s maximum speed, OLSR has dropped to below 100 Kbps. The packet trace reveals that OLSR still works in bimodal manner, where instantaneous goodput jumps from 0Mbps to 7Mbps frequently and rapidly, with varying recovery time. We next observe that in this scenario, as nodes move with more freedom, TCP DATA-ACK self-interference is no longer the performance bottleneck and thus ACK Piggy coding does not improve much. From the packet trace, we notice that the main challenge in this case are the frequent route breaks and thus the inability of ACK flows to exploit multipath redundancy limits the improvements introduced by ACK Piggy coding.

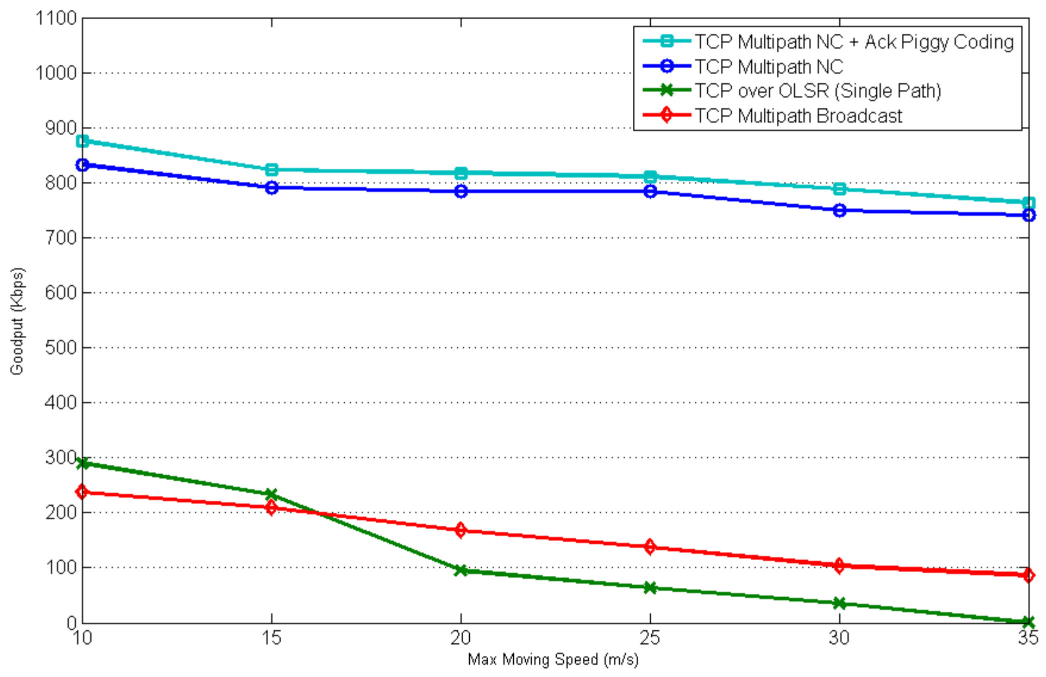


Fig. 47 Single-Session Goodput vs. Max Moving Speed

4.3.4 MANET Scenario—2-Session; Global Mobility; Jamming

In the last set of experiments, we run an extreme scenario, in which nodes are moving as in the global mobility model (Fig. 46) with a maximum speed of 25 m/s. We introduce a second TCP session side by side with the first one. As before, a time-varying random error rate is induced by jammers as follows: 0%, 40%, and 20% during the time intervals of 20~50 seconds, 50~80 seconds, and 80~110 seconds respectively. The same set of coding schemes as in the 1-session global mobility experiments are used, namely, OLSR, multipath broadcast, and CodeMP, with and without Piggy ACKs. From these experiments, we plan to determine whether the proposed scheme maintains efficient and robust communications even in this extreme case, while guaranteeing fair share among co-existing sessions with acceptable transmission overhead.

Table 8 below summarizes the average TCP goodput per session for each setting in each loss rate period. We first find that when there is no random error, OLSR successfully delivered 3Mbps for around 25% to 50% of the time resulting a goodput as high as 1.4Mbps. However, OLSR fails to provide reasonable fairness and thus one session continuously occupies more bandwidth. Further, OLSR again fails when loss rates increase and is unable to recover. We next notice that multipath broadcast is neither efficient nor fair. Multipath broadcast also fails under high random error rates. For multipath NC cases, both with and without Ack Piggy coding perform almost the same, while with Ack Piggy coding, it is slightly fairer. Table 9 below summarizes the Jain's fairness index of each setting in each period [45]. Overall, adaptive multipath coding with Ack Piggy coding is the fairest scheme.

Table 8 CodeMP Single-Group Two-Session Average Goodput (Kbps)

Config	0% PER	40% PER	20% PER
OLSR (Single Path)	445.64	0.00	0.00
	1435.24	0.00	115.05
Multipath Broadcast	126.16	29.90	116.33
	97.08	31.54	57.75
Multipath NC	532.89	374.37	458.75
	483.33	440.72	503.40
Multipath NC + Piggy Coding	575.90	430.49	482.10
	491.52	404.68	519.78

Table 9 CodeMP Two-Session Scenario Jain's Fairness Index

Config	0% PER	40% PER	20% PER
OLSR (Single Path)	0.783	0.000	0.500
Multipath Broadcast	0.983	0.999	0.898
Multipath NC	0.998	0.993	0.998
Multipath NC + Piggy Coding	0.994	0.999	0.999

Fig. 48 below shows the normalized transmission overhead (as defined in eq. (25)) for each setting. As shown in Fig. 48, OLSR has the least overhead among all schemes since it utilizes only one path at a time. As opposed to OLSR, multipath broadcast uses all paths at all times resulting in the highest overhead. Among the CodeMP schemes, ACK Piggy coding reduces the overhead by 35%. In this extreme test with global mobility, random errors, and multiple TCP sessions, CodeMP with ACK Piggy coding achieves much better overall goodput than OLSR, with much better fairness and only 38% more transmission overhead, which is the least overhead of all multipath cases.

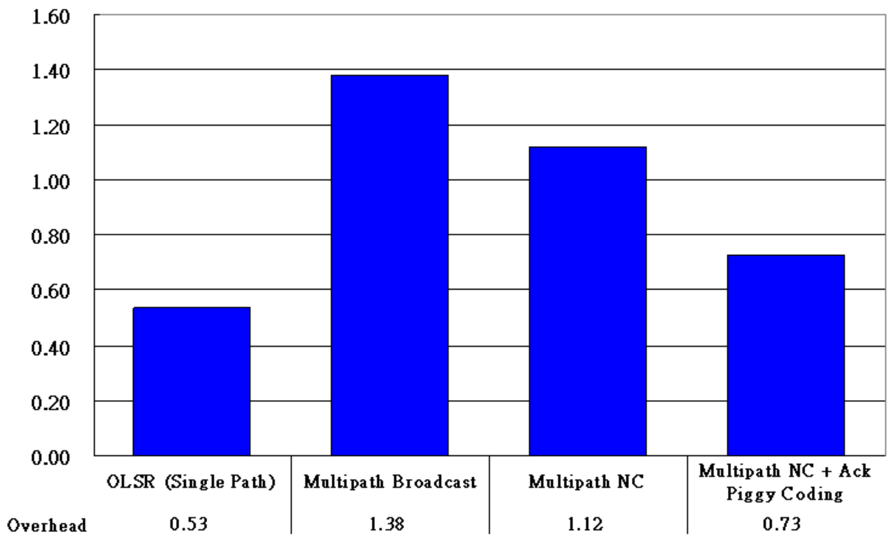


Fig. 48 Normalized Transmission Overhead

CHAPTER 5

Testbed Implementation for Pipeline Coding

5.1 Wireless Multi-hop Ad-hoc Testbed Experiments

The proposed pipeline coding scheme was implemented in Click modular router [37] as a packet-level coding element inside Linux kernel. Fig. 49 shows the configuration of our experiment topology. The source coding redundancy at the streaming server is 2.0 for both batch coding and Pipeline Coding. The generation size is set to 8 for both coding schemes. The bit-rate of the source stream is 192 kbps. Table 10 summarizes the format of the streaming file we tested. Similar to the simulation, we artificially introduce random drops at the receiving sides to emulate a highly lossy scenario.

Table 10 Summary of Streaming Format (Wireless Ad-hoc Testbed)

Video Compression	MPEG2-PS
Bit Rate	VBR with Peak 192Kbps
Resolution	240x180
Frame Rate	24 FPS
Audio Compression	MPEG1-LayerII (MP2)
Audio Bit Rate	64Kbps (CBR)
Sampling Rate	32KHz

Fig. 50(a) shows the packet delivery ratio of one of the clients for the simple multipath without coding, batch coding, and Pipeline Coding cases. As we expect from the simulation experiments, batch coding does not significantly outperform multipath without coding. Pipeline coding does much better than the two previous schemes and constantly delivers more than 98% of the packets. Fig. 50(b) gives a similar result obtained from previous simulation that Pipeline Coding consistently reduces the delay by one magnitude. Fig. 50(c) shows the video quality in PSNR (Peak Signal-toNoise Ratio) for these three cases, which shows a significant improvement when using Pipeline Coding.

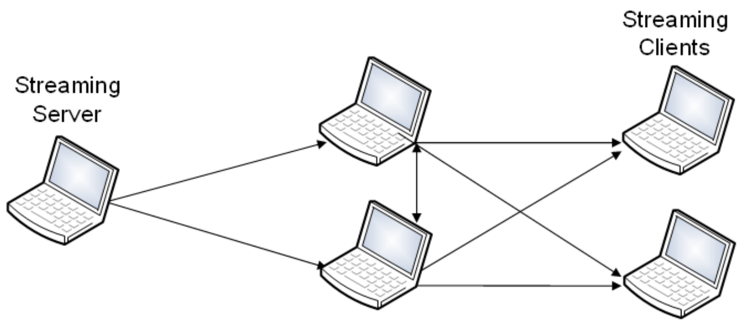
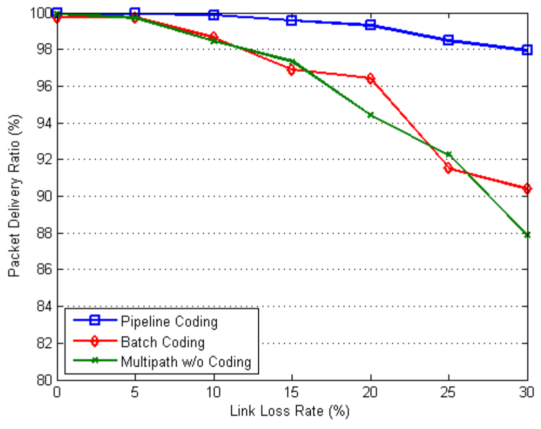
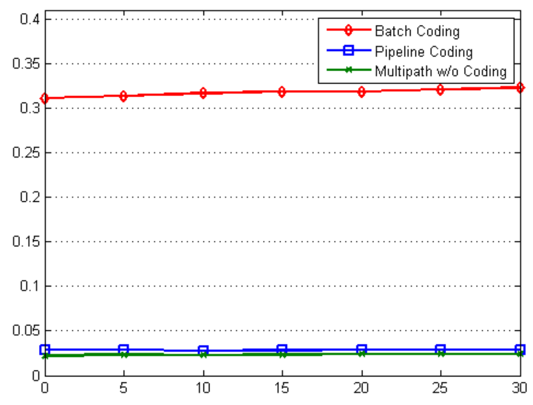


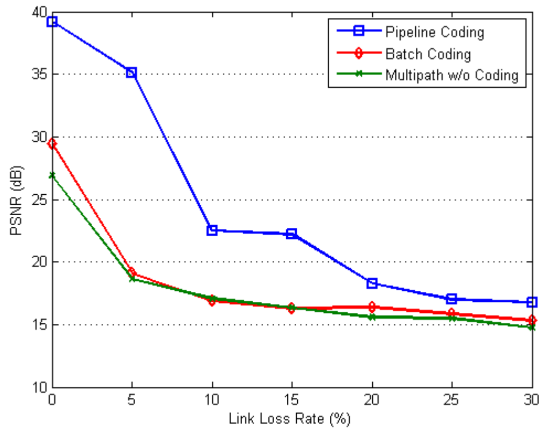
Fig. 49 Testbed Experiment Topology



(a) Packet Delivery Ratio



(b) Delay



(c) PSNR

Fig. 50 Testbed Experiment Results

5.2 Video Streaming in VANET over WiMAX and WiFi

We further extend our Click implementation as a vehicular router element and test Pipeline Coding in our Campus VANET Testbed (C-VeT). The experiment consists of a video stream from a vehicle to infrastructure over heterogeneous wireless channels (WIMAX and WiFi). The quality of the channel is impaired by mobility and external interference. Through the testbed experiment, we show that Network Coding can improve video quality by introducing redundancy when needed.

Fig. 51 below shows the configuration of our experiment topology. As a part of our C-VeT platform, we have a WiMAX base station (BS) installed on the roof of Boelter Hall, where a WiFi access point (AP) is also installed. We also have another WiFi AP installed on the roof of parking structure 6. Both WiFi APs connect to the same CS department subnet and the WiMAX BS connects to another CS department subnet. Fig. 52 illustrates the experiment design. A vehicular router is installed on a campus vehicle. On the vehicular router, a streaming server is running on top of Pipeline Coding module. The Pipeline Coding module encodes all streaming data packets and duplicates each coded packet over both WiFi and WiMAX interfaces. The WiFi interface is configured to support layer-2 roaming without restarting the network layer coding module.

As we have two concurrent interfaces, the source coding redundancy in this experiment at the streaming server is reduced from 2.0 to 1.5. The generation size is set to 8 for Pipeline Coding module. Also in this experiment, unicast is used for both WiFi and WiMAX, which provides us a higher channel bandwidth, so the bit-rate of the source stream is increased to 450 kbps. Table 11 summarizes the format of the streaming file we tested.

Table 11 Summary of Streaming Format (C-VeT Experiments)

Video Compression	MPEG2-PS
Bit Rate	VBR with Peak 450Kbps
Resolution	320x240
Frame Rate	29.97 FPS
Audio Compression	MPEG1-LayerII (MP2)
Audio Bit Rate	64Kbps (CBR)
Sampling Rate	32KHz
Packet Size	1316 Bytes

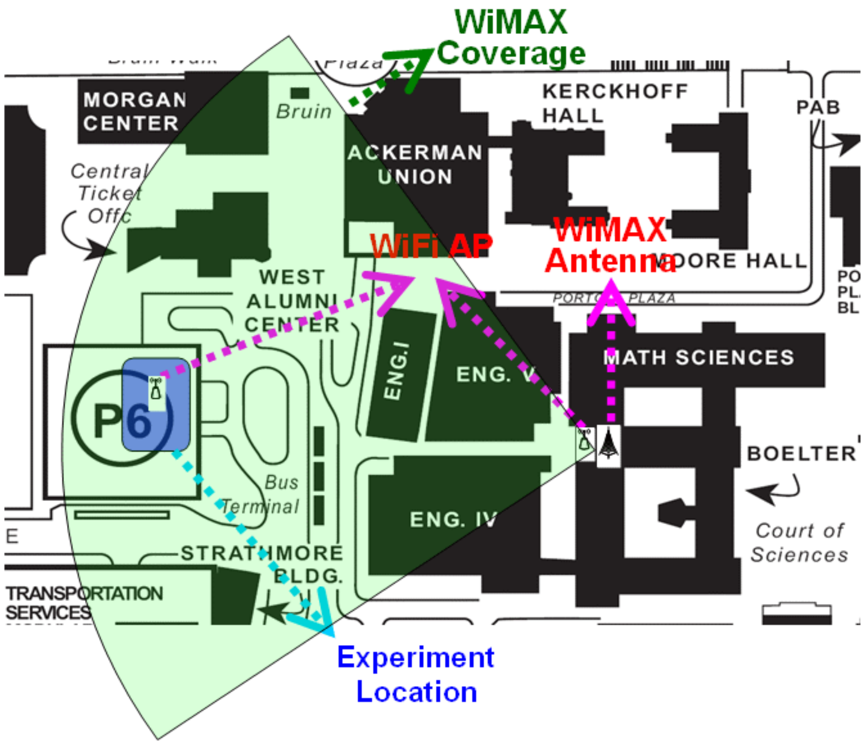


Fig. 51 C-VeT Experiment Topology

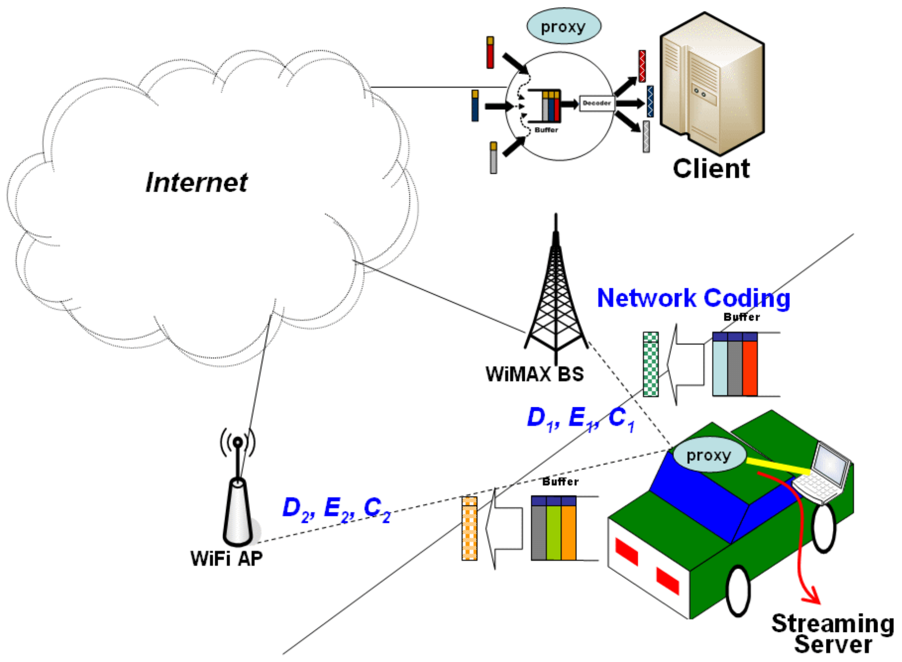
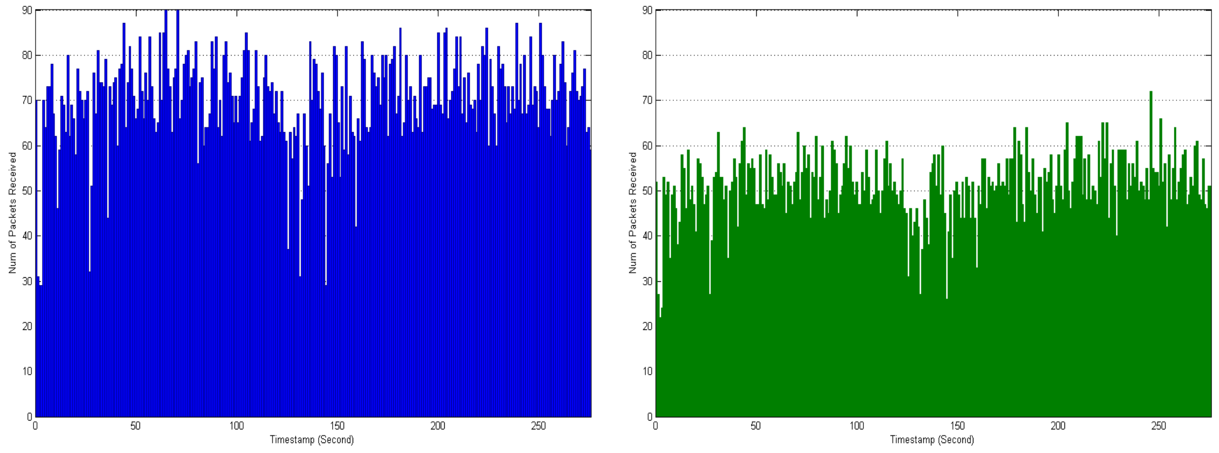


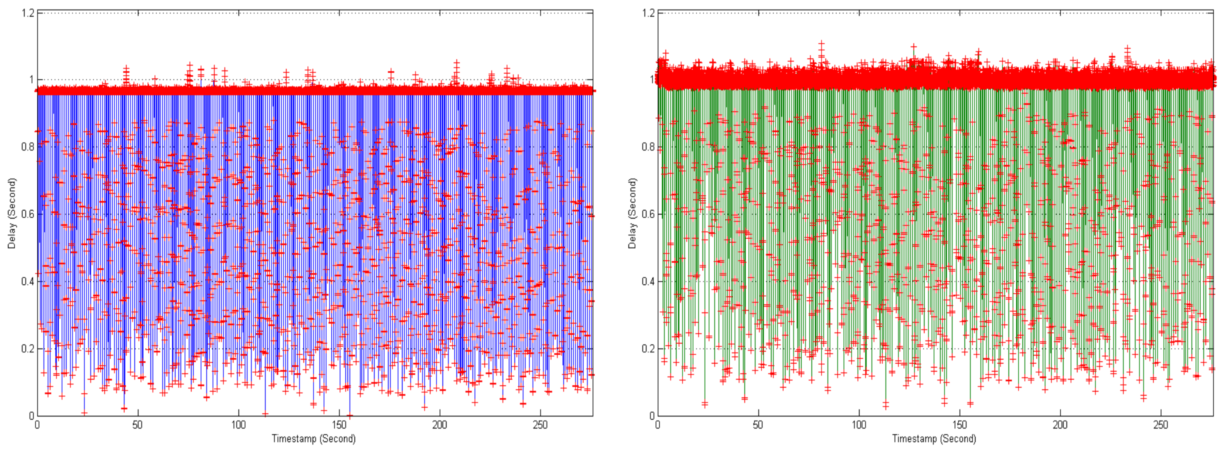
Fig. 52 Experiment Illustration

We first perform a static experiment to collect baseline measurements. In this experiment, the vehicle is parked on the roof of parking structure 6, which is under the coverage of both WiFi and WiMAX. The video stream lasts for about five minutes, which generates a total of 13,614 packets. With a redundancy of 1.5, each of the WiFi and WiMAX interfaces transmits 20,376 coded packets. Fig. 53 shows the number of packets received per second for both WiFi and WiMAX. We first notice that, although unicast MAC is used on both WiFi and WiMAX, not 100% of packets are delivered. For WiFi, only 95.7% of the coded packets are received; for WiMAX, only 69.9% of the coded packets are received. Fig. 54 plots the end-to-end delay of each received coded packet. The average end-to-end delay for WiFi is 909 milliseconds and for WiMAX is 945 milliseconds. With a difference of 30~40 milliseconds in delay, we find that when the vehicle is under WiFi coverage, most of the innovative packets are delivered via the WiFi link. In this experiment, 99.8% of the decoded data packets are received from WiFi.



(a) WiFi (b) WiMAX

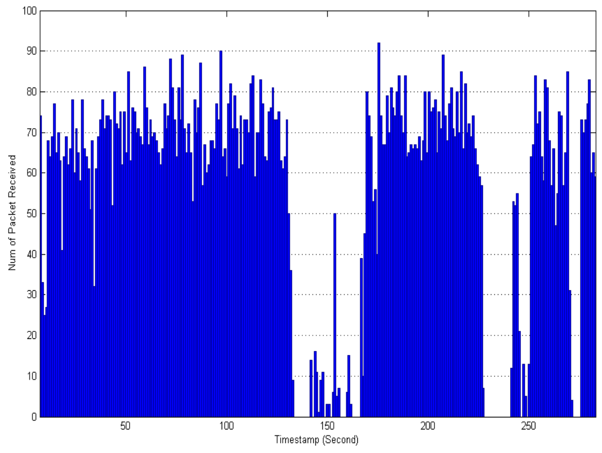
Fig. 53 Number of Packets Received per Second (Static)

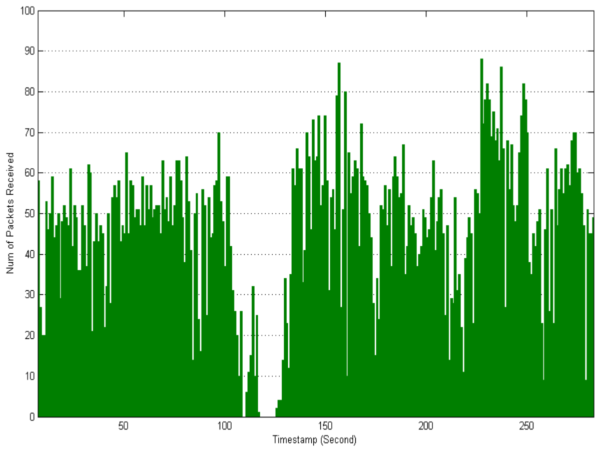


(a) WiFi (b) WiMAX

Fig. 54 End-to-End Delay of Each Received Packet (Static)

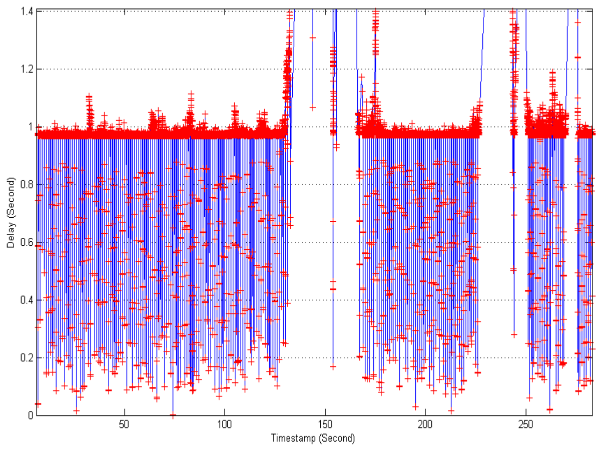
After characterizing WiFi and WiMAX links, mobility is introduced to our next experiment. In this experiment, the vehicle moves around the roof of the parking structure 6, where both WiFi and WiMAX become intermittent due to mobility and obstacles blocking the line-of-sight from the vehicle to antennas. The video stream again lasts for about five minutes, which generates a total of 13,614 packets. Similar to the previous experiment, with a redundancy of 1.5, each of the WiFi and WiMAX interfaces transmits 20,376 coded packets. Fig. 55 shows the number of packets received per second for both WiFi and WiMAX. We observe that both WiFi and WiMAX fail to maintain connectivity throughout the entire 5-min experiment period. However, when WiFi fails, WiMAX seamlessly takes over. In this experiment, WiFi delivers only 57% of the packets and WiMAX delivers only 72% of the packets. However, our trace shows that Pipeline Coding modules still successfully decodes 97% of the data packets (13,614 in total), and the recorded video clip shows only few frames are skipped or dropped. Fig. 56 shows the end-to-end delay of each received coded packet. We notice that when WiFi is under coverage, most of the delays are still 30~40 milliseconds lower than WiMAX. However, when WiFi losses connectivity, due to the random backoff, its delay increases dramatically. In contrast, WiMAX delay is bounded since WiMAX uses slotted MAC that has no random backoff. Through this experiment, we conclude that in this heterogeneous VANET scenario, Pipeline Coding helps the switching in between WiFi and WiMAX links and makes the streaming quality better.

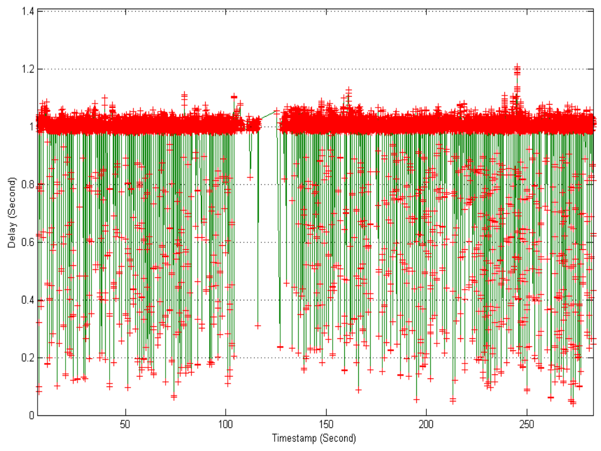




(a) WiFi (b) WiMAX

Fig. 55 Number of Packets Received per Second (Mobile)





(a) WiFi (b) WiMAX

Fig. 56 End-to-End Delay of Each Received Packet (Mobile)

CHAPTER 6

Summary

In this thesis, we aim to close the gap between network coding protocols and applications. Most of the previous work on network coding does not consider application constraints, such as delay for streaming applications and compatibility for TCP applications. We propose a low-latency coding scheme, Pipeline Coding, which is designed for multicast stream distribution in high loss rate scenarios. For TCP applications, we propose a combined inter- and intra-flow coding, ComboCoding, which provides a robust communication paradigm in a static error-prone network. For TCP over error-prone, intermittent MANETs, a multipath network coding, CodeMP, is proposed. In addition, a VANET testbed for Pipeline Coding is implemented. We summarize our studies as follows.

6.1 Pipeline Coding for Multicast Streams

Most of the previous network coding implementations have been based on batch network coding, where all blocks in the same batch are mixed together. Batch coding requires that the entire batch is received before decoding at destination. Thus, it introduces high decoding delays that impact the stream reception quality. In our first study, we propose Pipeline Coding, which is a low-latency coding scheme ideal for delay constrained applications such as video streaming in a lossy wireless multihop network. In Pipeline Coding, instead of waiting for the entire batch (i.e., generation), packet are encoded and decoded incrementally. Consequently, pipeline network coding yields several benefits: (1) reduced decoding delay, (2) further improved throughput, (3) transparency to higher layers (UDP, TCP, or other applications), (4) no special hardware support and (5) easier implementation. We provide a first-order analysis of our proposed Pipeline Coding scheme and validate the performance improvement via intensive simulation experiments. Through these studies, we have shown that compared to batch coding, Pipeline Coding achieves higher throughput while maintaining a low end-to-end delay.

6.2 ComboCoding for TCP over Multihop Wireless Networks

TCP over wireless networks is challenging due to random losses and DATA-ACK interference. Random linear coding schemes have been proposed to improve TCP robustness against extreme random losses, but a critical issue still remains of DATA-ACK interference. To address this problem, we use inter-flow coding between DATA and ACK at potential intersecting nodes to mitigate self-interference. In addition, a pipelined random linear coding scheme with adaptive redundancy is introduced to overcome high loss rates over unreliable links. The resulting coding scheme, ComboCoding, combines inter-flow and intra-flow coding to provide robust, fair TCP communication in multi-flow disruptive wireless networks while maintains total transparency to TCP. By exploiting the benefits of both types of coding, ComboCoding reduces the interference between DATA and ACKs within a TCP session and also exhibits robustness to high link loss rates. The simulation results show that in a 3-hop string topology, ComboCoding successfully achieves 2 Mbps throughput with 30% per link packet loss rate, while TCP-NewReno with no coding delivers only 200Kbps. The adaptive ComboCoding was tested on a longer string topology with time-varying link loss, and simulation results show that it outperforms all other coding schemes and quickly adapts to changes in link quality. We also demonstrated that under multi-flow scenarios, in the situation where conventional TCP collapses, ComboCoding manages to stay fair and stable.

6.3 CodeMP: Network Coded Multipath for TCP in Disruptive MANETs

TCP over Mobile Ad-hoc Networks (MANETs) is challenging due to frequent route breaks, high random errors, and DATA-ACK interference. Network coded multipath approaches have been shown in several previous studies to be an effective transmission paradigms in disruptive networks. However, most of the previous studies either have no adaptive redundancy control or rely on theoretical models that require knowledge at all relays of the entire network state. In this study, we propose a network coded multipath scheme for conventional TCP—CodeMP—that adapts to frequent link changes in MANET and requires no explicit control messages. The proposed scheme, CodeMP, consists of three main components: (1) random linear coding, (2) multipath routing, and (3) ACK Piggy coding. The RLC and multipath routing schemes adapt to dynamic scenarios and work closely with each other to provide robust and efficient multipath redundancy. The ACK Piggy coding helps reduce intra-session interference and greatly enhances fairness among concurrent sessions. The proposed CodeMP scheme is implemented at the network layer, totally transparent to TCP or other transport layer protocols, and without explicit control messages or cross layer optimization. Simulation results show that the proposed scheme adapts well to environment changes (mobility, time-varying jamming) and to different number of co-existing sessions. It consistently provides efficient, robust, and fair communications. In an extreme situation where nodes are moving as fast as 25 m/s, with varying packet error rate culminating at 40% and two co-existing TCP sessions, CodeMP can still deliver at least 700Kbps aggregate goodput with a fairness index of 0.99, with merely 38% more overhead than OLSR (which delivers zero goodput in presence of errors). This early study of a practical design of multipath network coding has achieved an important goal, mainly to demonstrate that conventional TCP Reno, with the help of Network Coding, can be made to work with acceptable performance even in extreme conditions where most TCP variants have failed.

6.4 Testbed Experiments for Pipeline Coding

Conventional network coding approaches are mostly based on a block-based coding scheme. The block-based coding scheme (batch coding scheme) introduces noticeable coding delay and thus results in discarding generations that have insufficient coded packets received. The proposed pipeline network coding relaxes the limitation of encoding and decoding based on a whole block of data. We implement our pipeline network coding scheme on Click modular router. Using the Pipeline Coding router, we conduct a number of testbed experiments using a simply 2-hop wireless ad-hoc network. The experiment results show a significant delay reduction as well as a remarkable packet delivery ratio improvement for multicast applications. Also, the streaming quality is improved by up to 10dB in PSNR value (peak signal-to-noise ratio).

The implemented Pipeline Coding module is also ported to vehicular routers on our Campus VANET Testbed (C-VeT). Using C-VeT platform, we evaluate the effectiveness of Pipeline Coding, where vehicles upload streaming files to infrastructure using heterogeneous WiFi and WiMAX links concurrently. Our testbed experiments show that without explicit scheduling of coded packets, Pipeline Coding improves both the packet delivery ratio and video quality by providing smooth switch in between WiFi and WiMAX links. We also show that our Pipeline Coding implementation is easy to deploy to a VANET testbed and is capable of running more future experiments.

6.5 Future Work

Several areas of future work are suggested by this study. First, in our coding scheme designs, we have strived to maintain the transparency to TCP layers and thus have not considered the explicit interaction between TCP and the network coding layer. It is important to study the interactions between TCP protocols and the proposed coding schemes. In particular, in order to be robust to jamming attack, our current design reacts to both random errors and congestion losses by increasing the coding redundancy. In the case of pure congestion without jamming, added redundancy will exacerbate congestion level and might impact the stability of TCP control mechanisms. A potential improvement is to exploit loss discrimination algorithms so that the coding layer reacts to only random errors and possibly jamming attacks. However, it is still an open issue to distinguish jamming attacks from congestion.

The other future study is to optimize the design of multipath coding for PiggyCoded packets. In our current design, unlike TCP DATA flows, TCP ACK flows are all sent without random linear coding. This effectively reduces the spatial redundancy of ACK flows to a single path when the ACKs are XOR coded. We have discussed a number of alternatives such as XOR a single ACK with more than one RLC coded DATA packets. To enable multipath for PiggyCoded ACKs, it will be important to determine how much spatial redundancy should be introduced and how to adaptively control such a redundancy factor.

We are also interested in extending our VANET testbed implementation to a more intelligent coding scheme. In our current implementation, Pipeline Coding module duplicates every coded packet to both WiFi and WiMAX links. From the experiment results, we have observed that when both interfaces are within coverage, most of the coded packets sent to WiMAX links are not innovative when received at the destination. Hence, it is important to reduce the number of redundant packets on the higher-latency WiMAX link when the lower-latency WiFi link is in operation.

Lastly, as our study mostly aims to propose practical design and implementation that is deployable to a realistic environment, a significant amount of effort is spent on addressing practical issues. The analysis is, however, equally important to the practice so that the performance gain from each segment of the procedure can be analytically validated. For example, analytical models provide more insights on how much benefit we separately gain from multiple paths, Pipeline Coding, and PiggyCode. To identify the gains from each part is helpful to provide a guideline on how to choose suitable combinations under different circumstances.

References

[1] J. F. Kurose and K. W. Ross, *Computer Network: A Top-Down Approach 5/e*, Addison Wesley Publishing Co., Inc., Boston, MA, 2010.

[2] R. Ahlswede, N. Cai, S.-Y. R. Li, R. W. Yeung, "Network information flow," *IEEE Trans. on Information Theory*, vol. 46, no. 4, pp. 1204-1216, July 2000.

[3] R. Koetter and M. Medard, "An algebraic approach to network coding," *IEEE/ACM Trans. on Networking*, vol. 11, no. 5, pp. 782-795, 2003.

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

[5] S.-Y. R. Li, R. W. Yeung, N. Cai, "Linear network coding," *IEEE Trans. on Info. Theory*, vol. 49, no. 2, pp. 371-381, Feb. 2003.

[6] S. Chachulski, M. Jennings, S. Katti, D. Katabi, "Trading Structure for Randomness in Wireless Opportunistic Routing," in *Proc. of ACM SIGCOMM 2007.*

[7] S. Katti, H. Rahul, W. Hu, D. Katabi, M. Medard, J. Crowcroft, "XORs in the Air: Practical Wireless Network Coding," *IEEE/ACM Trans. on Networking*, vol. 16, no. 3, June 2008.

[8] T. Ho, M. Medard, J. Shi, M. Effros, D. Karger, "On randomized network coding," in *Allerton*, 2003.

[9] D. S. Lun, M. Medard, R. Koetter, "Efficient Operation of Wireless Packet Networks Using Network Coding," in *Proc. of Internation Workshop on Convergent Technologies (IWCT)*, 2005.

[10] J.-S. Park, M. Gerla, D. S. Lun, Yu. Yi, M. Medard, "CodeCast: A Network Coding based Ad hoc Multicast Protocol," *IEEE Wireless Communications*, October 2006.

[11] J. K. Sundararajan, D. Shah, M. Medard, M. Mitzenmacher, J. Barros, "Network Coding meets TCP," in *Proc. of IEEE INFOCOM 2009*.

[12] S. Y. Oh, M. Gerla, "Robust MANET Routing using Adaptive Path Redundancy and Coding," in *Proc of THE FIRST International Conference on COMmunication Systems and NETworkS (COMSNETS)*, January 2009.

[13] C.-C. Chen, C.-N. Lien, U. Lee, S. Y. Oh, "CodeCast: Network Coding Based Multicast in MANETs," in *Demos of the 10th International Workshop on Mobile Computing Systems and Applications (HotMobile 2009)*, Feb. 2009.

[14] L. Rizzo, "Effective Erasure Codes for Reliable Computer Communication Protocols," in *Proc. Of ACM SIGCOMM*, vol. 27, no. 2, pp. 24-36, Apr. 1997.

[15] J. W. Byers, M. Luby, M. Mitzenmacher, A. Rege, "A digital fountain approach to reliable distribution of bulk data," in *Proc. of SIGCOMM 1998*.

[16] D. MacKay, "Fountain Codes," *IEE Proceedings on Communications*, vol. 152, no. 6, pp. 1062-1068, Dec. 2005.

[17] C.-C. Chen, S. Y. Oh, P. Tao, M. Gerla, M. Y. Sanadidi, "Pipeline Network Coding for Multicast Streams (Invited Paper)," in *Proc. of the 5th International Conference on Mobile Computing and Ubiquitous Networking (ICMU 2010)*, April 2010.

[18] P. Tao, C.-C. Chen, S. Y. Oh, M. Gerla, M. Y. Sanadidi, "Demo Abstract: Pipeline Network Coding for Multicast Streams," in *Demos of IEEE INFOCOM 2010*, March 2010.

[19] Y. Huang, M. Ghaderi, D. Towsley, W. Gong, "TCP Performance in Coded Wireless Mesh Networks," in *Proc. of IEEE SECON 2008*.

[20] L. Scalia, F. Soldo, M. Gerla, "PiggyCode: a MAC layer network coding scheme to improve TCP performance over wireless networks," in *Proc. of IEEE GLOBECOM 2007*.

[21] P. S. David and A. Kumar, "Network coding for TCP throughput enhancement over a multi-hop wireless network," in *Proc of IEEE COMSWARE 2008*.

[22] A. Shokrollahi, "Raptor Codes," *IEEE Trans. On Networking*, vol. 14, pp. 2551-2567, Jun. 2006.

[23] A. Fujimura, S. Y. Oh, M. Gerla, "Network coding vs. erasure coding: Reliable multicast in ad hoc networks," in *Proc of MilCom 2008*.

[24] C.-C. Chen, C. Chen, S. Y. Oh, M. Gerla, M. Y. Sanadidi, "ComboCoding: Combined Intra/Inter-Flow Network Coding for Wireless Multihop TCP," submitted to *ICNP 2010*.

[25] C.-C. Chen, C. Chen, S. Y. Oh, M. Gerla, M. Y. Sanadidi, "ComboCoding: Combined Intra/Inter-Flow Network Coding for TCP over Multihop Lossy Wireless Netowkrs," submitted to *ACITA 2010*.

[26] M. Handley, S. Floyd, J. Padhye, J. Widmer, "*TCP Friendly Rate Control (TFRC)*," RFC 3448, January 2003.

[27] G.-S. Ahn, A.T. Campbell, A. Veres and L.-H. Sun, "SWAN: Service Differentiation in Stateless Wireless Ad Hoc Networks," in *Proc. Of IEEE INFOCOM 2002*, June 2002.

[28] Vicente E. Mujica V., Dorgham Sisalem , Radu Popescu-Zeletin and Adam Wolisz "TCP-Friendly Congestion Control over Wireless Networks," in *Proc. of European Wireless*, Ferburary 2004,.

[29] Ijaz Haider Naqvi , Tanguy Perennou , "A DCCP Congestion Control Mechanism for Wired- cum-Wireless Environments", in *Proc. of the IEEE Wireless Communications and Networking Conference*, March 2007.

[30] K. Xu, S. Bae, S. Lee, M. Gerla, "TCP Behavior across Multihop Wireless Networks and the Wired Internet," in *Proc. of ACM WoWMoM 2002*, October 2002.

[31] J. Li, C. Blake, D. Couto, H. Lee, and R. Morris, "Capacity of Ad Hoc Wireless Networks," in Proc. of ACM SIGMETRICS 2000, June 2000.

[32] J. Chen, M. Gerla, Y. Z. Lee, M. Y. Sanadidi, "TCP with Delayed Ack for Wireless Networks," *Ad Hoc Networks*, vol. 6, pp. 1098-1116, 2008.

[33] Scalable Networs Inc. QualNet. http://www.scalble-networks.com.

[34] K. Xu, M. Gerla, S. Bae, "Effectiveness of RTS/CTS Handshake in IEEE 802.11 Based Ad-Hoc Networks," *Ad Hoc Networks*, vol. 1, pp. 107-123, 2003.

[35] J. Padhey, V. Firoiu, D. Towsley, J. Kurose, "Modeling TCP Throughput: A Simple Model and its Empirical Validation," in *Proc. of ACM SIGCOMM 1998*.

[36] D. S. J. De Couto, D. Aguayo, J. Bicket, R. Morris, "A high-throughput path metric for multi-hop wireless routing," in *Proc. of MOBICOM, 2003*.

[37] Eddie Kohler, Robert Morris, Benjie Chen, John Jannotti, and M. Frans Kaashoek, "The Click modular router," *ACM Transactions on Computer Systems 18(3)*, pp. 263-297, August 2000.

[38] S. Deb, M. Medard, and C. Chout, “Algebraic Gossip: A Network Coding Approach to Optimal Multiple Rumor Mongering,” in *Proc. Allerton’04*, Allerton, IL, Sep. 2004.

[39] X. Zhang, B. Li, "Optimized multipath network coding in lossy wireless networks," in *IEEE Journal on Selected Areas in Communications*, vol. 27, no. 5, pp. 622-634, 2009.

[40] Y. Lin, B. Li, B. Liang, "CodeOR: Opportunistic routing in wireless mesh networks with segmented network coding," in *Proc. of IEEE ICNP 2008*.

[41] X. Zhang, B. Li, "Dice: a game theoretic framework for wireless multipath network coding," in *Proc. of MobiHoc* 2008.

[42] A. Tiwari, A. Ganguli, A. Sampath, D. S. Anderson, B.-H. Shen, N. Krishnamurthi, J. Yadegar, M. Gerla, D. Krzysiak, "Mobility Aware Routing for the Airborne Network backbone," in *Proc. of IEEE MILCOM 2008*.

[43] H. Seferoglu, A. Markopoulou, and K. K. Ramakrishnan, “I2NC: Intra- and Inter-Session Network Coding for Unicast Flows in Wireless Networks,” in *Proc. of IEEE INFOCOM 2011*.

[44] T. Clausen, P. Jacquet, Optimized Link State Routing Protocol (OLSR), IETF RFC 3626, Oct. 2003.

[45] R. Jain, D.M. Chiu, and W. Hawe, "A Quantitative Measure of Fairness and Discrimination for Resource Allocation in Shared Systems," *DEC Research Report TR-301*, 1984.

[46] Kaixin Xu, Mario Gerla, Lantao Qi, and Yantai Shu, "Enhancing TCP Fairness in Ad Hoc Wireless Networks Using Neighborhood RED", in *Proc. of MobiCom '03*, Sept. 2003.