INVITED PAPER

# Network Coding Meets TCP: Theory and Implementation

*This paper explains how network coding can be applied in a manner compatible with the transmission control protocol (TCP).*

By Jay Kumar Sundararajan, Devavrat Shah, Muriel Médard, *Fellow IEEE*, Szymon Jakubczak, Michael Mitzenmacher, and João Barros

**ABSTRACT** | The theory of network coding promises significant benefits in network performance, especially in lossy networks and in multicast and multipath scenarios. To realize these benefits in practice, we need to understand how coding across packets interacts with the acknowledgment (ACK)-based flow control mechanism that forms a central part of today's Internet protocols such as transmission control protocol (TCP). Current approaches such as rateless codes and batch-based coding are not compatible with TCP's retransmission and sliding-window mechanisms. In this paper, we propose a new mechanism called TCP/NC that incorporates network coding into TCP with only minor changes to the protocol stack, thereby allowing incremental deployment. In our scheme, the source transmits random linear combinations of packets currently in the congestion window. At the heart of our scheme is a new interpretation of ACKs—the sink acknowledges every degree of freedom (i.e., a linear combination that reveals one unit of new information) even if it does not reveal an original packet immediately. Thus, our new TCP ACK rule takes into account the network coding operations in the lower layer and enables a TCP-compatible sliding-window approach to network coding. Coding essentially masks losses from the congestion control algorithm and allows TCP/NC to react smoothly to losses, resulting in a novel and effective approach for congestion control over lossy networks such as wireless networks. An important feature of our solution is that it allows intermediate nodes to perform re-encoding of packets, which is known to provide significant throughput gains in lossy networks and multicast scenarios. Simulations show that our scheme, with or without re-encoding inside the network, achieves much higher throughput compared to TCP over lossy wireless links. We present a real-world implementation of this protocol that addresses the practical aspects of incorporating network coding and decoding with TCP's window management mechanism. We work with TCP-Reno, which is a widespread and practical variant of TCP. Our implementation significantly advances the goal of designing a deployable, general, TCP-compatible protocol that provides the benefits of network coding.

**KEYWORDS** | Congestion control; network coding; TCP; wireless networks

## I. INTRODUCTION

The concept of coding across data has been put to extensive use in today's communication systems at the link level, due to practical coding schemes that are known to achieve data rates very close to the fundamental limit, or capacity, of the additive white Gaussian noise channel [1]. Although the fundamental limits for many multiuser information theory problems have yet to be established, it is well known that there are significant benefits to coding beyond the link level.

For example, consider multicasting over a network of broadcast-mode links in wireless systems. Due to the broadcast nature of the medium, a transmitted packet is

Manuscript received November 17, 2009; revised March 1, 2010; accepted October 16, 2010. Date of publication January 13, 2011; date of current version February 18, 2011. This work was presented in part at the 28th IEEE Conference on Computer Communications, Rio de Janeiro, Brazil, April 2009. The work was supported by the National Science Foundation (NSF) under Grants CNS-0627021, CNS-0721491, and CCF-0915922, subcontract #18870740-37362-C issued by Stanford University and supported by DARPA, subcontracts # 060786 and # 069145 issued by BAE Systems and supported by DARPA and SPAWARSYSCEN under Contracts N66001-06-C-2020 and N66001-08-C-2013, respectively, subcontract # S0176938 issued by the University of California Santa Cruz, supported by the United States Army under Award W911NF-05-1-0246, and the DARPA Grant HR0011-08-1-0008.

**J. Sundararajan** was with the Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, MA 02139 USA. He is now with Qualcomm Inc., San Diego, CA 92121 USA (e-mail: sjaykumar@gmail.com).

**D. Shah**, **M. Médard**, and **S. Jakubczak** are with the Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, MA 02139 USA (e-mail: devavrat@mit.edu; medard@mit.edu; szym@mit.edu).

**M. Mitzenmacher** is with the School of Engineering and Applied Sciences, Harvard University, Cambridge, MA 02138 USA (e-mail: michaelm@eecs.harvard.edu).

**J. Barros** is with the Instituto de Telecomunicações, Departamento de Engenharia Electrotécnica e de Computadores, Faculdade de Engenharia da Universidade do Porto, 4200-465 Porto, Portugal (e-mail: jbarros@fe.up.pt).

Digital Object Identifier: 10.1109/JPROC.2010.2093850

0018-9219/$26.00 ©2011 IEEE

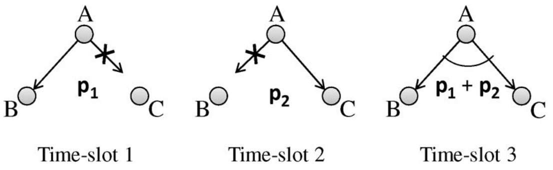


**Fig. 1.** ***Coding over a broadcast-mode link.***

likely to be received by several nodes. If one of the nodes experienced a bad channel state and thereby lost the packet, then a simple retransmission strategy may not be the best option, since the retransmission is useless from the viewpoint of the other receivers that have already received the packet. In Fig. 1, node A broadcasts 2 packets to nodes B and C. In the first time-slot, only node B receives packet $\mathbf{p_1}$ and in the second slot, only node C receives packet $\mathbf{p_2}$. At this point, if instead of retransmitting $\mathbf{p_1}$ or $\mathbf{p_2}$, node A is allowed to mix the information and send a single packet containing the bitwise XOR of $\mathbf{p_1}$ and $\mathbf{p_2}$, then both B and C receive their missing packet in just one additional time slot. This example shows that if we allow coding across packets, it is possible to convey new information simultaneously to all connected receivers.

Another scenario where coding across packets can make a significant difference is in certain network topologies where multiple flows have to traverse a bottleneck link. The now standard example is the butterfly network from [2], which is shown in Fig. 2. Here, node A wants to multicast a stream of packets to nodes F and G. Assume the links are error free with a capacity of one packet per slot. If all nodes are only allowed to forward packets, then node D can forward either the packet from B $(\mathbf{p_1})$ or the one from C $(\mathbf{p_2})$. It can be seen that alternating between these options gives a multicast throughput of 1.5 packets per slot.

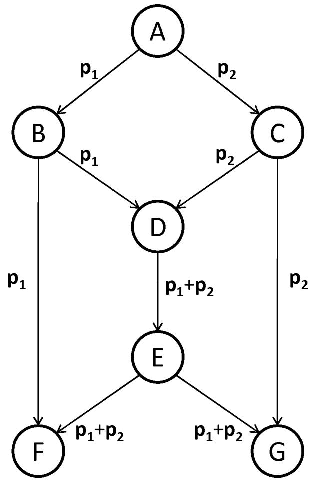


**Fig. 2.** ***The butterfly network of [2].***

However, if node D sends a bitwise XOR of $\mathbf{p_1}$ and $\mathbf{p_2}$ as shown in the figure, then it is possible to satisfy both receivers simultaneously, resulting in a multicast throughput of two packets per time slot. This is the highest possible, since it meets the min-cut bound for each receiver.

Through the butterfly network example, Ahlswede *et al.* [2] introduced the field of network coding. With network coding, a node inside the network, instead of simply forwarding the incoming packets onto outgoing links, is now allowed to send a coded version of the incoming packets.

Although both the examples above use a bitwise XOR code, the coding operation could be much more general. For instance, we could view groups of bits as elements of a finite field, and a packet as a vector over this field. Coding could then correspond to performing linear combinations of these vectors, with coefficients chosen from the field of operation. In order to decode, the receiver will have to collect as many linear combinations as the number of packets that were mixed in, and then solve the resulting system of linear equations by Gauss–Jordan elimination.

Network coding achieves the min-cut bound for multicast in any network as long as all the multicast sessions have the same destination set [2], [3]. Li *et al.* [4] showed that linear coding suffices for this purpose. An algebraic framework for network coding was proposed by Koetter and Médard in [3]. Ho *et al.* [5] presented a random linear network coding approach for this problem that is easy to implement and does not compromise on throughput. The problem of multicast using network coding with a cost criterion has been studied, and distributed algorithms have been proposed to solve this problem [6], [7]. Network coding also readily extends to networks with broadcast-mode links or lossy links [8]–[10]. Jiang *et al.* [11] highlights the need for coding for the case of multicast traffic, even if feedback is present. In all these situations, coding is indispensable from a throughput perspective.

Besides improving throughput, network coding can also be used to simplify network management. The work by Bhadra and Shakkottai [12] proposed a scheme for large multihop networks, where intermediate nodes in the network have no queues. Only the source and destination nodes maintain buffers to store packets. The packet losses that occur due to the absence of buffers inside the network are compensated for by random linear coding across packets at the source.

Network coding has emerged as an important potential approach to the operation of communication networks, especially wireless networks. The major benefit of network coding stems from its ability to *mix* data, across time and across flows. This makes data transmission over lossy wireless networks robust and effective. There has been a rapid growth in the theory and potential applications of network coding. These developments have been summarized in several survey papers and books such as [13].

However, extending coding technologies to the network setting in a practical way has been a challenging task. Indeed, the most common way to implement a multicast connection today is to initiate multiple unicast connections, one for each receiver, even though coding can theoretically dramatically improve multicast performance. To a large extent, this theory has not yet been implemented in practical systems.

## II. BRINGING NETWORK CODING TO PRACTICE

Despite the potential of network coding, we still seem far from seeing widespread implementation of network coding across networks. We believe a major reason for this is the incremental deployment problem. It is not clear how to naturally add network coding to existing systems, and to understand ahead of time the actual effects of network coding in the wild. There have been several important advances in bridging the gap between theory and practice in this space. The distributed random linear coding idea, introduced by Ho *et al.* [14], is a significant step towards a robust implementation. The work by Chou *et al.* [15] put forth the idea of embedding the coefficients used in the linear combination in the packet header, and also the notion of grouping packets into batches for coding together. The work by Katti *et al.* [16] used the idea of local opportunistic coding to present a practical implementation of a network coded system for unicast. The use of network coding in combination with opportunistic routing was presented in [17]. Despite these efforts, we believe that incremental deployment remains a hurdle to increased adoption of network coding in practical settings, and we therefore seek a protocol that brings out the benefits of network coding while requiring very little change in the existing protocol stack.

### A. The Incremental Deployment Problem

A common and important feature of today's protocols is the use of feedback in the form of acknowledgments (ACKs). The simplest protocol that makes use of ACKs is the Automatic Repeat reQuest (ARQ) protocol. It uses the idea that the sender can interpret the absence of an ACK to indicate the erasure of the corresponding packet within the network, and in this case, the sender simply retransmits the lost packet. Thus, ARQ ensures reliability. The ARQ scheme can be generalized to situations that have imperfections in the feedback link, in the form of either losses or delay in the ACKs. Bertsekas and Gallager [18] presented a summary of various protocols based on ARQ.

Besides ensuring reliability, the ACK mechanism forms the basis of control algorithms in the network aimed at preventing congestion and ensuring fair use of the network resources. Compared to a point-to-point setting where reliability is the main concern, the network setting leads to several new control problems just to ensure that the network is up and running and that all users get fair access to the resources. These problems are usually tackled using feedback. Therefore, in order to realize the theoretically proven benefits of network coding, we have to find a way to incorporate coding into the existing network protocols, *without disrupting the feedback-based control operations*.

Flow control and congestion control in today's Internet are predominantly based on the transmission control protocol (TCP), which works using the idea of a sliding transmission window of packets, whose size is controlled based on feedback [19], [20]. The TCP paradigm has clearly proven successful. We therefore see a need to find a sliding-window approach for network coding that is as similar as possible to TCP, that makes use of ACKs for flow and congestion control. (This problem was initially proposed in [21].)

### B. Current Approaches Are Not TCP Compatible

Current approaches that use coding across packets are not readily compatible with TCP's retransmission and sliding-window mechanism. The digital fountain codes [22]–[24] constitute a well-known solution to the problem of packet transmission over lossy links. From a batch of $k$ packets, the sender generates a stream of random linear combinations in such a way that the receiver can, with high probability, decode the batch once it receives *any* set of slightly more than $k$ linear combinations. Fountain codes have a low complexity and do not use feedback, except to signal successful decoding of the block. In contrast to fountain codes that are typically applied end-to-end, the random linear network coding solution of [5] and [9] allows an intermediate node to easily re-encode the packets and generate new linear combinations without having to decode the original packets.

An important problem with both these approaches is that although they are rateless, the encoding operation is typically performed on a batch of packets. Several other works also focus on such a batch-based solution [15], [17], [25], [26]. With a batch-based approach, there is no guarantee that the receiver will be able to extract and pass on to higher layers any of the original packets until the entire batch has been received and decoded. Therefore, packets are acknowledged only at the end of a batch. This leads to a decoding delay that interferes with TCP's own retransmission mechanism for correcting losses. TCP would either timeout, or learn a very large value of round- trip time (RTT), causing low throughput. Thus, TCP cannot readily run on a batch-based rateless coding module.

Lacan and Lochin [27] proposed an on-the-fly coding scheme with ACKs, but there again, the packets are acknowledged only upon decoding. Chen *et al.* [28] proposed distributed rate control algorithms for network coding in a utility maximization framework, and pointed out its similarity to TCP. However, to implement such algorithms in practice, we need to create a clean interface between network coding and TCP. Thus, none of these works allows an ACK-based sliding-window network coding approach that is compatible with TCP. This is the problem we address in our current work.

### C. Our Solution

In this paper, we show how to incorporate network coding into TCP, allowing its use with minimal changes to the protocol stack, and in such a way that incremental deployment is possible.

The main idea behind TCP is to use ACKs of newly received packets as they arrive *in correct sequence order* in order to guarantee reliable transport and also as a feedback signal for the congestion control loop. This mechanism requires some modification for systems using network coding. The key difference to be dealt with is that under network coding the receiver does not obtain original packets of the message, but linear combinations of the packets that are then decoded to obtain the original message once enough such combinations have arrived. Hence, the notion of an ordered sequence of packets as used by TCP is missing, and further, a linear combination may bring in new information to a receiver even though it may not reveal an original packet immediately. The current ACK mechanism does not allow the receiver to acknowledge a packet before it has been decoded. For network coding, we need a modification of the standard TCP mechanism that acknowledges every unit of information received. A new unit of information corresponds mathematically to a *degree of freedom*; essentially, once $n$ degrees of freedom have been obtained, a message that would have required $n$ unencoded packets can be decoded. We present a mechanism that performs the functions of TCP, namely reliable transport and congestion control, based on acknowledging every degree of freedom received, whether or not it reveals a new packet.

Our solution, known as TCP/NC, introduces a new network coding layer between the transport layer and the network layer of the protocol stack. Thus, we recycle the congestion control principle of TCP, namely that the number of packets involved in transmissions cannot exceed the number of ACKs received by more than the congestion window size. However, we introduce two main changes. First, whenever the source is allowed to transmit, it sends a random linear combination of all packets in the congestion window. Second, the receiver acknowledges degrees of freedom and not original packets. (This idea was previously introduced in [29] in the context of a single-hop erasure broadcast link.) An appropriate interpretation of the degree of freedom allows us to order the receiver degrees of freedom in a manner consistent with the packet order of the source. This lets us utilize the standard TCP protocol with the minimal change. Since the receiver does not have to wait to decode a packet, but can send a TCP ACK for every degree of freedom received, the problems of using batchwise ACKs is eliminated.

We use the TCP-Vegas protocol in the initial description, as it is more compatible with our modifications. In a later part of the paper, we also demonstrate the compatibility of our protocol with the more commonly used TCP-Reno. We do not consider bidirectional TCP in this work.

It is important to note that the introduction of the new network coding layer does not cause any change in the interface to TCP, as seen by an application. Moreover, the interface seen by TCP looking downwards in the protocol stack is also unchanged—the network coding layer accepts regular TCP packets from the TCP sender and delivers regular TCP ACKs back to the sender. Similarly, it delivers regular TCP packets to the receiver and accepts the ACKs generated by the receiver. This means that the basic features of the TCP layer implementation do not need to be changed. Further details about this interface are discussed in Section V-C5.

The first part of the paper explains the details of our new protocol along with its theoretical basis and a queuing analysis in an idealized setting. Following this, we present a real-life implementation of the protocol and discuss the practical issues that need to be addressed. Finally, we analyze the algorithm's performance based on simulations as well as real-world experiments.

## III. PRELIMINARIES

Consider a single source that has a message to transmit. We view the message as being split into a stream of packets $\mathbf{p_1}, \mathbf{p_2}, \ldots$. The $k$th packet in the source message is said to have an *index* $k$. We treat a packet as a vector over a finite field $F_q$ of size $q$, by grouping the bits of the packet into groups of size $\lfloor \log_2 q \rfloor$ bits each. In the system we propose, a node, in addition to forwarding incoming packets, is also allowed to perform linear network coding. This means that the node may transmit a packet obtained by linearly combining the vectors corresponding to the incoming packets, with coefficients chosen from the field $F_q$. For example, it may transmit $\mathbf{q_1} = \alpha\mathbf{p_1} + \beta\mathbf{p_2}$ and $\mathbf{q_2} = \gamma\mathbf{p_1} + \delta\mathbf{p_2}$, where $\alpha, \beta, \gamma, \delta \in F_q$. Assuming the packets have $\ell$ symbols, the encoding process may be written in matrix form as

$$\begin{pmatrix} q_{11} & q_{12} & \cdots & q_{1\ell} \\ q_{21} & q_{22} & \cdots & q_{2\ell} \end{pmatrix} = C \cdot \begin{pmatrix} p_{11} & p_{12} & \cdots & p_{1\ell} \\ p_{21} & p_{22} & \cdots & p_{2\ell} \end{pmatrix}$$

where $C = \begin{pmatrix} \alpha & \beta \\ \gamma & \delta \end{pmatrix}$ is called the coefficient matrix. Note that even if an intermediate node performs re-encoding on these linear combinations, the net effect may still be represented using such a linear relation, with $C$ being replaced by the overall transfer matrix.

Upon receiving the packets $\mathbf{q_1}$ and $\mathbf{q_2}$, the receiver simply needs to invert the matrix $C$ using Gauss–Jordan elimination, and apply the corresponding linear operations on the received packets to obtain the original message packets $\mathbf{p_1}$ and $\mathbf{p_2}$. In matrix form, the decoding process is given by

$$\begin{pmatrix} p_{11} & p_{12} & \cdots & p_{1\ell} \\ p_{21} & p_{22} & \cdots & p_{2\ell} \end{pmatrix} = C^{-1} \cdot \begin{pmatrix} q_{11} & q_{12} & \cdots & q_{1\ell} \\ q_{21} & q_{22} & \cdots & q_{2\ell} \end{pmatrix}.$$

In general, the receiver will need to receive as many linear combinations as the number of original packets involved, in order to be able to decode.

In this setting, we introduce some definitions that will be useful throughout the paper (see [29] for more details).

*Definition 1 (Seeing a Packet):* A node is said to have *seen* a packet $\mathbf{p_k}$ if it has enough information to compute a linear combination of the form $(\mathbf{p_k} + \mathbf{q})$, where $\mathbf{q} = \sum_{\ell > k} \alpha_\ell \mathbf{p}_\ell$, with $\alpha_\ell \in F_q$ for all $\ell > k$. Thus, $\mathbf{q}$ is a linear combination involving packets with indices larger than $k$.

The notion of "seeing" a packet is a natural extension of the notion of "decoding" a packet, or more specifically, receiving a packet in the context of classical TCP. For example, if a packet $\mathbf{p_k}$ is decoded then it is indeed also seen, with $\mathbf{q} = \mathbf{0}$. A node can compute any linear combination whose coefficient vector is in the span of the coefficient vectors of previously received linear combinations. This leads to the following definition.

*Definition 2 (Knowledge of a Node):* The *knowledge of a node* is the set of all linear combinations of original packets that it can compute, based on the information it has received so far. The coefficient vectors of these linear combinations form a vector space called the *knowledge space* of the node.

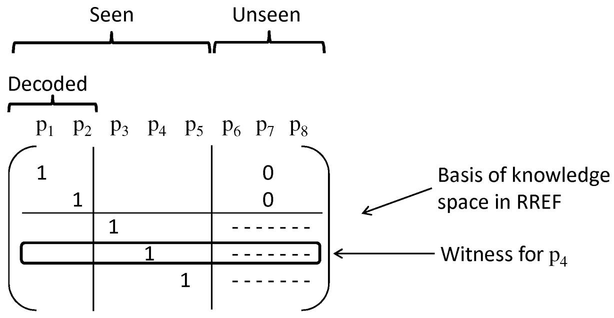
We state a useful proposition without proof (see [29, Corollary 1] for details).

*Proposition 1:* If a node has seen packet $\mathbf{p_k}$, then it knows exactly one linear combination of the form $\mathbf{p_k} + \mathbf{q}$ such that $\mathbf{q}$ is itself a linear combination involving only *unseen* packets.

The above proposition inspires the following definition.

*Definition 3 (Witness):* We call the unique linear combination guaranteed by Proposition 1 the *witness for seeing* $\mathbf{p_k}$.

A compact representation of the knowledge space is the basis matrix. This is a matrix in reduced row echelon form (RREF) such that its rows form a basis of the knowledge space. It is obtained by performing Gauss–Jordan elimination on the coefficient matrix. Fig. 3 explains the notion of a seen packet in terms of the basis matrix. Essentially, the seen packets are the ones that correspond to the pivot columns of the basis matrix. Given a seen packet, the corresponding pivot row gives the coefficient vector for the witness linear combination. An important observation is that *the number of seen packets is always equal to the dimension of the knowledge space*, or the number of degrees of freedom that have been received so far. A newly received linear combination that increases the dimension is said to be *innovative*. We assume throughout the paper that the field size is very large. As a consequence, each reception will be innovative with high probability, and will cause the next unseen packet to be seen (see Lemma 1).



Number of seen packets = Rank of matrix = Dim of knowledge space

**Fig. 3.** *Seen packets and witnesses in terms of the basis matrix.*

*Example:* Suppose a node knows the following linear combinations: $\mathbf{x} = (\mathbf{p_1} + \mathbf{p_2})$ and $\mathbf{y} = (\mathbf{p_1} + \mathbf{p_3})$. Since these are linearly independent, the knowledge space has a dimension of 2. Hence, the number of seen packets must be 2. It is clear that packet $\mathbf{p_1}$ has been seen, since $\mathbf{x}$ satisfies the requirement of Definition 1. Now, the node can compute $\mathbf{z} \triangleq \mathbf{x} - \mathbf{y} = (\mathbf{p_2} - \mathbf{p_3})$. Thus, it has also seen $\mathbf{p_2}$. That means $\mathbf{p_3}$ is unseen. Hence, $\mathbf{y}$ is the witness for $\mathbf{p_1}$, and $\mathbf{z}$ is the witness for $\mathbf{p_2}$.

## IV. THE NEW PROTOCOL

In this section, we present the logical description of our new protocol, followed by a way to implement these ideas with as little disturbance as possible to the existing protocol stack.

### A. Logical Description

The main aim of our algorithm is to mask losses from TCP using random linear coding. We make some important modifications in order to incorporate coding. First, instead of the original packets, we transmit random linear combinations of packets in the congestion window. While such coding helps with erasure correction, it also leads to a problem in acknowledging data. TCP operates with units of packets,[1] which have a well-defined ordering. Thus, the

[1] Actually, TCP operates in terms of bytes. For simplicity of presentation, the present section uses packets of fixed length as the basic unit. All the discussion in this section extends to the case of bytes as well, as explained in Section V.

packet sequence number can be used for acknowledging the received data. The unit in our protocol is a degree of freedom. However, when packets are coded together, there is no clear ordering of the degrees of freedom that can be used for ACKs. Our main contribution is the solution to this problem. The notion of seen packets defines an ordering of the degrees of freedom that is consistent with the packet sequence numbers, and can therefore be used to acknowledge degrees of freedom.

Upon receiving a linear combination, the sink finds out which packet, if any, has been newly seen because of the new arrival and acknowledges that packet. The sink thus pretends to have received the packet even if it cannot be decoded yet. We will show in Section IV-C that at the end this is not a problem because if all the packets in a file have been seen, then they can all be decoded as well.

The idea of transmitting random linear combinations and acknowledging seen packets achieves our goal of masking losses from TCP as follows. As mentioned in Section III, with a large field size, every random linear combination is very likely to cause the next unseen packet to be seen. Hence, even if a transmitted linear combination is lost, the next successful reception of a (possibly) different random linear combination will cause the next unseen packet to be seen and acknowledged. From the TCP sender's perspective, this appears as though the transmitted packet waits in a fictitious queue until the channel stops erasing packets and allows it through. Thus, there will never be any duplicate ACKs. Every ACK will cause the congestion window to advance. In short, *the lossiness of the link is presented to TCP as an additional queuing delay that leads to a larger effective RTT*. The term RTT thus has a new interpretation. It is the effective time the network takes to *reliably* deliver a degree of freedom (including the delay for the coded redundancy, if necessary), followed by the return of the ACK. This is larger than the true network delay it takes for a lossless transmission and the return of the ACK. The more lossy the link is, the larger will be the effective RTT. Presenting TCP with a larger value for RTT may seem counterintuitive as TCP's rate is inversely related to RTT. However, if done correctly, it improves the rate by preventing loss-induced window closing, as it gives the network more time to deliver the data in spite of losses, before TCP times out. Therefore, losses are effectively masked.

The natural question that arises is: How does this affect congestion control? Since we mask losses from the congestion control algorithm, the TCP-Reno style approach to congestion control using packet loss as a congestion indicator is not immediately applicable to this situation. However, the congestion related losses are made to appear as a longer RTT. Therefore, we can use an approach that infers congestion from an increase in RTT. The natural choice is TCP-Vegas. The discussion in this section is presented in terms of TCP-Vegas. The algorithm, however, can be extended to make it compatible with TCP-Reno as well. This is discussed in detail in Section V, where a real-world implementation with TCP-Reno is presented.

TCP-Vegas uses a proactive approach to congestion control by inferring the size of the network buffers even before they start dropping packets. The crux of the algorithm is to estimate the RTT and use this information to find the discrepancy between the expected and actual transmission rate. As congestion arises, buffers start to fill up and the RTT starts to rise, and this is used as the congestion signal. This signal is used to adjust the congestion window and hence the rate. For further details, the reader is referred to [30].

In order to use TCP-Vegas correctly in this setting, we need to ensure that it uses the effective RTT of a degree of freedom, including the fictitious queuing delay. In other words, the RTT should be measured from the point when a packet is first sent out from TCP, to the point when the ACK returns saying that this packet has been seen. This is indeed the case if we simply use the default RTT measurement mechanism of TCP-Vegas. The TCP sender notes down the transmission time of every packet. When an ACK arrives, it is matched to the corresponding transmit timestamp in order to compute the RTT. Thus, no modification is required.

Consider the example shown in Fig. 4. Suppose the congestion window's length is 4. Assume TCP sends four packets to the network coding layer at $t = 0$. All four transmissions are linear combinations of these four packets. The first transmission causes the first packet to be seen. The second and third transmissions are lost, and the fourth transmission causes the second packet to be seen (the discrepancy is because of losses). As far as the RTT estimation is concerned, transmissions 2, 3, and 4 are treated as attempts to convey the second degree of freedom. The RTT for the second packet must include the final

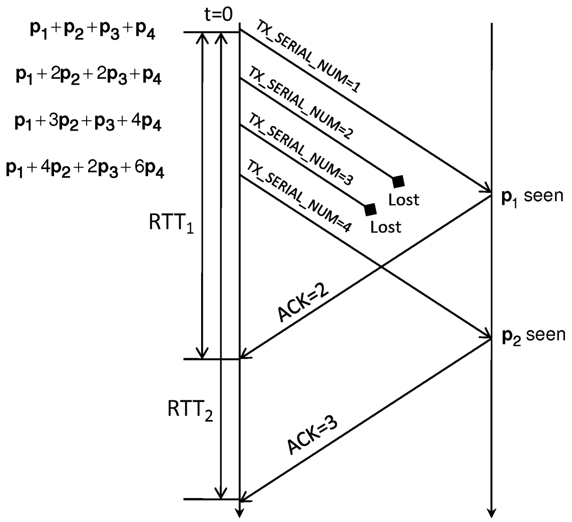


Fig. 4. *Example of coding and ACKs.*

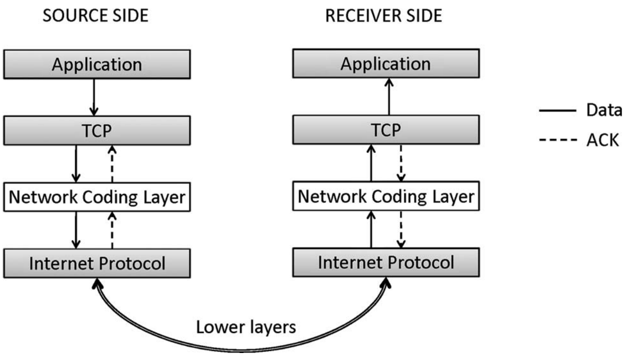


**Fig. 5.** ***New network coding layer in the protocol stack.***

attempt that successfully delivers the second degree of freedom, namely the fourth transmission. In other words, the RTT is the time from $t = 0$ until the time of reception of ACK $= 3$.

### B. Implementation Strategy

The implementation of all these ideas in the existing protocol stack needs to be done in as nonintrusive a manner as possible. We present a solution that embeds the network coding operations in a separate layer below TCP and above IP on the source and receiver side, as shown in Fig. 5. The exact operation of these modules is described next.

The sender module accepts packets from the TCP source and buffers them into an encoding buffer which represents the coding window,[2] until they are acknowledged by the receiver. The sender then generates and sends random linear combinations of the packets in the coding window. The coefficients used in the linear combination are also conveyed in the header.

For every packet that arrives from TCP, $R$ linear combinations are sent to the IP layer on average, where $R$ is the redundancy parameter. The average rate at which linear combinations are sent into the network is thus a constant factor more than the rate at which TCP's congestion window progresses. This is necessary in order to compensate for the loss rate of the channel and to match TCP's sending rate to the rate at which data are actually sent to the receiver. If there is too little redundancy, then the data rate reaching the receiver will not match the sending rate because of the losses. This leads to a situation where the losses are not effectively masked from the TCP layer. Hence, there are frequent timeouts leading to a low throughput. On the other extreme, too much redundancy is also bad, since then the transmission rate becomes limited by the rate of the code itself. Besides, sending too many linear combinations can congest the network. The ideal level of redundancy is to keep $R$ equal to the reciprocal of the probability of successful reception. Thus, in practice, the value of $R$ should be dynamically adjusted by estimating the loss rate, possibly using the RTT estimates.

Upon receiving a linear combination, the receiver module first retrieves the coding coefficients from the header and appends it to the basis matrix of its knowledge space. Then, it performs a Gauss–Jordan elimination to find out which packet is newly seen so that this packet can be acknowledged. The receiver module also maintains a buffer of linear combinations of packets that have not been decoded yet. Upon decoding the packets, the receiver module delivers them to the TCP sink.

The algorithm is specified below using pseudocode. This specification assumes a one-way TCP flow.

*1) Source Side:* The source side algorithm has to respond to two types of events: the arrival of a packet from the source TCP and the arrival of an ACK from the receiver via IP.

```
1)  Set NUM to 0.
2)  Wait state: If any of the following events occurs,
    respond as follows; else, wait.
3)  Packet arrives from TCP sender.
    a)  If the packet is a control packet used for
        connection management, deliver it to the IP
        layer and return to wait state.
    b)  If the packet is not already in the coding
        window, add it to the coding window.
    c)  Set NUM=NUM+$R$ ($R$ = redundancy factor).
    d)  Repeat the following $\lfloor \mathrm{NUM} \rfloor$ times.
        i)   Generate a random linear combination
             of the packets in the coding window.
        ii)  Add the network coding header specifying
             the set of packets in the coding
             window and the coefficients used for the
             random linear combination.
        iii) Deliver the packet to the IP layer.
    e)  Set NUM: = fractional part of NUM.
    f)  Return to the wait state.
4)  ACK arrives from receiver: Remove the acknowledged
    packet from the coding buffer and hand
    over the ACK to the TCP sender.
```

*2) Receiver Side:* On the receiver side, the algorithm again has to respond to two types of events: the arrival of a

[2] Whenever a new packet enters the TCP congestion window, TCP transmits it to the network coding module, which then adds it to the coding window. Thus, the coding window is related to the TCP layer's congestion window but generally not identical to it. For example, the coding window will still hold packets that were transmitted earlier by TCP, but are no longer in the congestion window because of a reduction of the window size by TCP. However, this is not a problem because involving more packets in the linear combination will only increase its chances of being innovative.

packet from the source and the arrival of ACKs from the TCP sink.

1) *Wait state*: If any of the following events occurs, respond as follows; else, wait.
2) *ACK arrives from TCP sink*: If the ACK is a control packet for connection management, deliver it to the IP layer and return to the wait state; else, ignore the ACK.
3) *Packet arrives from source side*.
    a) Remove the network coding header and retrieve the coding vector.
    b) Add the coding vector as a new row to the existing coding coefficient matrix, and perform Gauss–Jordan elimination to update the set of seen packets.
    c) Add the payload to the decoding buffer. Perform the operations corresponding to the Gauss–Jordan elimination, on the buffer contents. If any packet gets decoded in the process, deliver it to the TCP sink and remove it from the buffer.
    d) Generate a new TCP ACK with sequence number equal to that of the oldest unseen packet.

### C. Soundness of the Protocol

We argue that our protocol guarantees reliable transfer of information. In other words, every packet in the packet stream generated by the application at the source will be delivered eventually to the application at the sink. We observe that the ACK mechanism ensures that the coding module at the sender does not remove a packet from the coding window unless it has been acknowledged, i.e., unless it has been seen by the sink. Thus, we only need to argue that if all packets in a file have been seen, then the file can be decoded at the sink.

*Theorem 1:* From a file of $n$ packets, if every packet has been seen, then every packet can also be decoded.

*Proof:* If the sender knows a file of $n$ packets, then the sender's knowledge space is of dimension $n$. Every seen packet corresponds to a new dimension. Hence, if all $n$ packets have been seen, then the receiver's knowledge space is also of dimension $n$, in which case it must be the same as the sender's and all packets can be decoded. ■

In other words, seeing $n$ different packets corresponds to having $n$ linearly independent equations in $n$ unknowns. Hence, the unknowns can be found by solving the system of equations. At this point, the file can be delivered to the TCP sink. In practice, one does not have to necessarily wait until the end of the file to decode all packets. Some of the unknowns can be found even along the way. In particular, whenever the number of equations received catches up with the number of unknowns involved, the unknowns can be found. Now, for every new equation received, the receiver sends an ACK. The congestion control algorithm uses the ACKs to control the injection of new unknowns into the coding window. Thus, the discrepancy between the number of equations and number of unknowns does not tend to grow with time, and therefore will hit zero often based on the channel conditions. As a consequence, the decoding buffer will tend to be stable.

An interesting observation is that the arguments used to show the soundness of our approach are quite general and can be extended to more general scenarios such as random linear coding based multicast over arbitrary topologies.

### D. Queuing Analysis for an Idealized Case

In this section, we focus on an idealized scenario in order to provide a first-order analysis of our new protocol. We aim to explain the key ideas of our protocol with emphasis on the interaction between the coding operation and the feedback. The model used in this section will also serve as a platform that we can build on to incorporate more practical situations.

We abstract out the congestion control aspect of the problem by assuming that the capacity of the system is fixed in time and known at the source, and hence the arrival rate is always maintained below the capacity. We also assume that nodes have infinite capacity buffers to store packets. We focus on a topology that consists of a chain of erasure-prone links in tandem, with perfect end-to-end feedback from the sink directly to the source. In such a system, we investigate the behavior of the queue sizes at various nodes. We show that our scheme stabilizes the queues for all rates below capacity.

*1) System Model:* The network we study in this section is a daisy chain of $N$ nodes, each node being connected to the next one by a packet erasure channel. We assume a slotted time system. The source generates packets according to a Bernoulli process of rate $\lambda$ packets per slot. The point of transmission is at the very beginning of a slot. Just after this point, every node transmits one random linear combination of the packets in its queue. The relation between the transmitted linear combination and the original packet stream is conveyed in the packet header. We ignore this overhead for the analysis in this section. We ignore propagation delay. Thus, the transmission, if not erased by the channel, reaches the next node in the chain almost immediately. However, the node may use the newly received packet only in the next slot's transmission. We assume perfect, delay-free feedback from the sink to the source. In every slot, the sink generates the feedback signal after the instant of reception of the previous node's transmission. The erasure event happens with a probability $(1-\mu_i)$ on the channel connecting node $i$ and $(i+1)$, and is assumed to be independent across different channels and over time.

Thus, the system has a capacity $\min_i \mu_i$ packets per slot. We assume that $\lambda < \min_i \mu_i$, and define the load factor $\rho_i = \lambda/\mu_i$.

*2) Queue Update Mechanism:* Each node transmits a random linear combination of the current contents of its queue and hence, it is important to specify how the queue contents are updated at the different nodes. Queue updates at the source are relatively simple because in every slot, the sink is assumed to send an ACK directly to the source, containing the index of the oldest packet not yet seen by the sink. Upon receiving the ACK, the source simply drops all packets from its queue with an index lower than the sink's request.

Whenever an intermediate node receives an innovative packet, this causes the node to see a previously unseen packet. The node performs a Gauss–Jordan elimination to compute the witness of the newly seen packet, and adds this to the queue. Thus, intermediate nodes store the witnesses of the packets that they have seen. The idea behind the packet drop rule is similar to that at the source—an intermediate node may drop the witnesses of packets up to but excluding what it believes to be the sink's first unseen packet, based on its knowledge of the sink's status at that point of time.

However, the intermediate nodes, in general, may only know an outdated version of the sink's status because we assume that the intermediate nodes do not have direct feedback from the sink (see Fig. 6). Instead, the source has to inform them about the sink's ACK through the same erasure channel used for the regular forward transmission. This feed-forward of the sink's status is modeled as follows. Whenever the channel entering an intermediate node is in the ON state (i.e., no erasure), the node's version of the sink's status is updated to that of the previous node. In practice, the source need not transmit the sink's status explicitly. The intermediate nodes can infer it from the set of packets that have been involved in the linear combination—if a packet is no longer involved, that means the source must have dropped it, implying that the sink must have acknowledged it already.

*Remark 1:* This model and the following analysis also work for the case when not all intermediate nodes are involved in the network coding. If some node simply forwards the incoming packets, then we can incorporate this in the following way. An erasure event on either the link entering this node or the link leaving this node will cause a packet erasure. Hence, these two links can be replaced by a single link whose probability of being ON is simply the product of the ON probabilities of the two links being replaced. Thus, all noncoding nodes can be removed from the model, which brings us back to the same situation as in the above model.

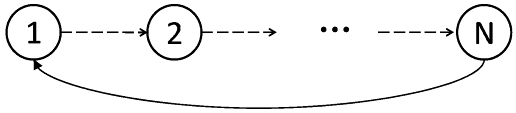


**Fig. 6.** ***Topology: Daisy chain with perfect end-to-end feedback.***

*4) Queuing Analysis:* We now analyze the size of the queues at the nodes under the queuing policy described above. The following theorem shows that if we allow coding at intermediate nodes, then it is possible to achieve the capacity of the network, namely $\min_k \mu_k$. In addition, it also shows that the expected queue size in the heavy-traffic limit $(\lambda \rightarrow \min_k \mu_k)$ has an asymptotically optimal linear scaling in $1/(1-\rho_k)$.

If we only allow forwarding at some of the intermediate nodes, then we can still achieve the capacity of a new network derived by collapsing the links across the noncoding nodes, as described in Remark 1.

*Theorem 2:* As long as $\lambda < \mu_k$ for all $1 \leq k < N$, the queues at all the nodes will be stable. The expected queue size in steady state at node $k$ $(1 \leq k < N)$ is given by

$$\mathbb{E}[Q_k] = \sum_{i=k}^{N-1} \frac{\rho_i(1-\mu_i)}{(1-\rho_i)} + \sum_{i=1}^{k-1} \rho_i.$$

*An implication*: Consider a case where all the $\rho_i$'s are equal to some $\rho$. Then, the above relation implies that in the limit of heavy traffic, i.e., $\rho \rightarrow 1$, the queues are expected to be longer at nodes near the source than near the sink.

*A useful lemma*: The above theorem will be proved after the following lemma. The lemma shows that the random linear coding scheme has the property that every successful reception at a node causes the node to see the next unseen packet with high probability, provided the field is large enough. This fact will prove useful while analyzing the evolution of the queues.

*Lemma 1:* Let $S_A$ and $S_B$ be the set of packets seen by two nodes A and B, respectively. Assume $S_A \setminus S_B$ is nonempty. Suppose A sends a random linear combination of its witnesses of packets in $S_A$ and B receives it successfully. The probability that this transmission causes B to see the oldest packet in $S_A \setminus S_B$ is $(1-1/q)$, where $q$ is the field size.

*Proof:* Let $M_A$ be the RREF basis matrix for A. Then, the coefficient vector of the linear combination sent by A is $\mathbf{t} = \mathbf{u}M_A$, where $\mathbf{u}$ is a vector of length $|S_A|$ whose entries are independent and uniformly distributed over the finite field $F_q$. Let $d^*$ denote the index of the oldest packet in $S_A \setminus S_B$.

Let $M_B$ be the RREF basis matrix for B before the new reception. Suppose $\mathbf{t}$ is successfully received by B. Then, B will append $\mathbf{t}$ as a new row to $M_B$ and perform Gauss–Jordan elimination. The first step involves subtracting from $\mathbf{t}$, suitably scaled versions of the pivot rows such that all entries of $\mathbf{t}$ corresponding to pivot columns of $M_B$ become 0. We need to find the probability that after this step, the leading nonzero entry occurs in column $d^*$, which corresponds to the event that B sees packet $d^*$. Subsequent steps in the Gauss–Jordan elimination will not affect this event. Hence, we focus on the first step.

Let $P_B$ denote the set of indices of pivot columns of $M_B$. In the first step, the entry in column $d^*$ of $\mathbf{t}$ becomes

$$t'(d^*) = t(d^*) - \sum_{i \in P_B, i < d^*} t(i) \cdot M_B(r_B(i), d^*)$$

where $r_B(i)$ is the index of the pivot row corresponding to pivot column $i$ in $M_B$. Now, due to the way RREF is defined, $t(d^*) = u(r_A(d^*))$, where $r_A(i)$ denotes the index of the pivot row corresponding to pivot column $i$ in $M_A$. Thus, $t(d^*)$ is uniformly distributed. Also, for $i < d^*$, $t(i)$ is a function of only those $u(j)$'s such that $j < r_A(d^*)$. Hence, $t(d^*)$ is independent of $t(i)$ for $i < d^*$. From these observations and the above expression for $t'(d^*)$, it follows that for any given $M_A$ and $M_B$, $t'(d^*)$ has a uniform distribution over $F_q$, and the probability that it is not zero is therefore $(1 - (1/q))$. ■

*Computing the expected queue size*: For the queuing analysis, we assume that a successful reception always causes the receiver to see its next unseen packet, as long as the transmitter has already seen it. The above lemma argues that this assumption becomes increasingly valid as the field size increases. In reality, some packets may be seen out of order, resulting in larger queue sizes. However, we believe that this effect is minor and can be neglected for a first-order analysis.

With this assumption in place, the queue update policy described earlier implies that the size of the physical queue at each node is simply the difference between the number of packets the node has seen and the number of packets it believes the sink has seen.

To study the queue size, we define a virtual queue at each node that keeps track of the degrees of freedom backlog between that node and the next one in the chain. The arrival and the departure of the virtual queues are defined as follows. A packet is said to arrive at a node's virtual queue when the node sees the packet for the first time. A packet is said to depart from the virtual queue when the next node in the chain sees the packet for the first time. A consequence of the assumption stated above is that the set of packets seen by a node is always a contiguous set. This allows us to view the virtual queue maintained by a node as though it were a first-in–first-out (FIFO) queue. The size of the virtual queue is simply the difference between the number of packets seen by the node and the number of packets seen by the next node downstream.

We are now ready to prove Theorem 2. For each intermediate node, we study the expected time spent by an arbitrary packet in the physical queue at that node, as this is related to the expected physical queue size at the node, by Little's law.

*Proof of Theorem 2:* Consider the $k$th node, for $1 \leq k < N$. The time a packet spends in this node's queue has two parts.

1) *Time until the packet is seen by the sink*: The virtual queue at a node behaves like a FIFO $Geom/Geom/1$ queue. The Markov chain governing its evolution is identical to that of the virtual queues studied in [29]. Given that node $k$ has just seen the packet in question, the additional time it takes for the next node to see that packet corresponds to the waiting time in the virtual queue at node $k$. For a load factor of $\rho$ and a channel ON probability of $\mu$, the expected waiting time was derived in [29] to be $(1 - \mu)/\mu(1 - \rho)$, using results from [31]. Now, the expected time until the sink sees the packet is the sum of $(N - k)$ such terms, which gives $\sum_{i=k}^{N-1}((1 - \mu_i)/\mu_i(1 - \rho_i))$.
2) *Time until sink's ACK reaches intermediate node*: The ACK informs the source that the sink has seen the packet. This information needs to reach node $k$ by the feed-forward mechanism. The expected time for this information to move from node $i$ to node $i + 1$ is the expected time until the next slot when the channel is ON, which is just $1/\mu_i$ (since the $i$th channel is ON with probability $\mu_i$). Thus, the time it takes for the sink's ACK to reach node $k$ is given by

$$\sum_{i=1}^{k-1} \frac{1}{\mu_i}.$$

The total expected time $T_k$ a packet spends in the queue at the $k$th node $(1 \leq k < N)$ can thus be computed by adding the above two terms. Now, assuming the system is stable (i.e., $\lambda < \min_i \mu_i$), we can use Little's law to derive the expected queue size at the $k$th node, by multiplying $T_k$ by $\lambda$

$$\mathbb{E}[Q_k] = \sum_{i=k}^{N-1} \frac{\rho_i(1 - \mu_i)}{(1 - \rho_i)} + \sum_{i=1}^{k-1} \rho_i.$$

■

## V. REAL-WORLD IMPLEMENTATION

In this section, we discuss some of the practical issues that arise in designing an implementation of the TCP/NC protocol compatible with real TCP/IP stacks. These issues were not considered in the idealized setting discussed up to this point. We present a real-world implementation of TCP/NC and thereby show that it is possible to overcome these issues and implement a TCP-aware network-coding layer that has the property of a clean interface with TCP. In addition, although our initial description used TCP-Vegas, our real-world implementation demonstrates the compatibility of our protocol with the more commonly used TCP variant—TCP-Reno. The rest of this section pertains to TCP-Reno.

### A. Sender Side Module

*1) Forming the Coding Buffer:* The description of the protocol in Section IV assumes a fixed packet length, which allows all coding and decoding operations to be performed symbol-wise on the whole packet. That is, an entire packet serves as the basic unit of data (i.e., as a single unknown), with the implicit understanding that the exact same operation is being performed on every symbol within the packet. The main advantage of this view is that the decoding matrix operations (i.e., Gauss–Jordan elimination) can be performed at the granularity of packets instead of individual symbols. Also, the ACKs are then able to be represented in terms of packet numbers. Finally, the coding vectors then have one coefficient for every packet, not every symbol. Note that the same protocol and analysis of Section IV holds even if we fix the basic unit of data as a symbol instead of a packet. The problem is that the complexity will be very high as the size of the coding matrix will be related to the number of symbols in the coding buffer, which is much more than the number of packets (typically, a symbol is one byte long).

In practice, TCP is a byte-stream oriented protocol in which ACKs are in terms of byte sequence numbers. If all packets are of fixed length, we can still apply the packet-level approach, since we have a clear and consistent map between packet sequence numbers and byte sequence numbers. In reality, however, TCP might generate segments of different sizes. The choice of how many bytes to group into a segment is usually made based on the maximum transmission unit (MTU) of the network, which could vary with time. A more common occurrence is that applications may use the PUSH flag option asking TCP to packetize the currently outstanding bytes into a segment, even if it does not form a segment of the maximum allowed size. In short, it is important to ensure that our protocol works correctly in spite of variable packet sizes.

A closely related problem is that of repacketization. Repacketization, as described in [19, Ch. 21], refers to the situation where a set of bytes that were assigned to two different segments earlier by TCP may later be reassigned to the same segment during retransmission. As a result, the grouping of bytes into packets may not be fixed over time.

Both variable packet lengths and repacketization need to be addressed when implementing the coding protocol. To solve the first problem, if we have packets of different lengths, we could elongate the shorter packets by appending sufficiently many dummy zero symbols until all packets have the same length. This will work correctly as long as the receiver is somehow informed how many zeros were appended to each packet. While transmitting these extra dummy symbols will decrease the throughput, generally this loss will not be significant, as packet lengths are usually consistent.

However, if we have repacketization, then we have another problem, namely it is no longer possible to view a packet as a single unknown. This is because we would not have a one-to-one mapping between packets sequence numbers and byte sequence numbers; the same bytes may now occur in more than one packet. Repacketization appears to destroy the convenience of performing coding and decoding at the packet level.

To counter these problems, we propose the following solution. The coding operation described in Section IV involves the sender storing the packets generated by the TCP source in a *coding buffer*. We preprocess any incoming TCP segment before adding it to the coding buffer as follows.

1) First, any part of the incoming segment that is already in the buffer is removed from the segment.
2) Next, a separate TCP packet is created out of each remaining contiguous part of the segment.
3) The source and destination port information is removed. It will be added later in the network coding header.
4) The packets are appended with sufficiently many dummy zero bytes, to make them as long as the longest packet currently in the buffer.

Every resulting packet is then added to the buffer. This processing ensures that the packets in the buffer will correspond to disjoint and contiguous sets of bytes from the byte stream, thereby restoring the one-to-one correspondence between the packet numbers and the byte sequence numbers. The reason the port information is excluded from the coding is because port information is necessary for the receiver to identify which TCP connection a coded packet corresponds to. Hence, the port information should not be involved in the coding. We refer to the remaining part of the header as the TCP subheader.

Upon decoding the packet, the receiver can identify the dummy symbols using the $Start_i$ and $End_i$ header fields in the network coding header (described below). With these fixes in place, we are ready to use the packet-level algorithm of Section IV. All operations are performed on the packets in the coding buffer. Fig. 7 shows a typical state of the buffer after this preprocessing. The gaps at the end of

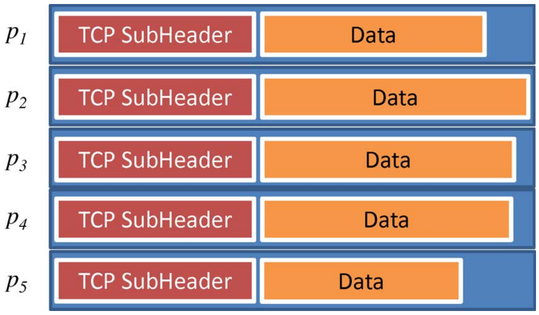


**Fig. 7.** *The coding buffer.*

the packets correspond to the appended zeros. It is important to note that the TCP control packets such as SYN packet and reset packet are allowed to bypass the coding buffer and are directly delivered to the receiver without any coding.

*2) The Coding Header:* A coded packet is created by forming a random linear combination of a subset of the packets in the coding buffer. The coding operations are done over a field of size 256 in our implementation. In this case, a field symbol corresponds to one byte. The header of a coded packet should contain information that the receiver can use to identify what is the linear combination corresponding to the packet. We now discuss the header structure in more detail.

We assume that the network coding header has the structure shown in Fig. 8. The typical sizes (in bytes) of the various fields are written above them. The meaning of the various fields are described next.

- *Source and destination port*: The port information is needed for the receiver to identify the coded packet's session. It must not be included in the coding operation. It is taken out of the TCP header and included in the network coding header.
- *Base*: The TCP byte sequence number of the first byte that has not been acknowledged. The field is used by intermediate nodes or the decoder to decide which packets can be safely dropped from their buffers without affecting reliability.
- $n$: The number of packets involved in the linear combination.
- $Start_i$: The starting byte of the $i$th packet involved in the linear combination.
- $End_i$: The last byte of the $i$th packet involved in the linear combination.
- $\alpha_i$: The coefficient used for the $i$th packet involved in the linear combination.

The $Start_i$ (except $Start_1$) and $End_i$ are expressed relative to the previous packet's $End$ and $Start$, respectively, to save header space. As shown in the figure, this header format will add $5n + 7$ bytes of overhead for the network coding header in addition to the TCP header, where $n$ is the number of packets involved in a linear combination. (Note that the port information is not counted in this overhead, since it has been removed from the TCP header.) We believe it is possible to reduce this overhead by further optimizing the header structure.

*3) The Coding Window:* In the theoretical version of the algorithm, the sender transmits a random linear combination of all packets in the coding buffer. However, as noted above, the size of the header scales with the number of packets involved in the linear combination. Therefore, mixing all packets currently in the buffer will lead to a large coding header.

To solve this problem, we propose mixing only a constant-sized subset of the packets chosen from within the coding buffer. We call this subset the *coding window*. The coding window evolves as follows. The algorithm uses

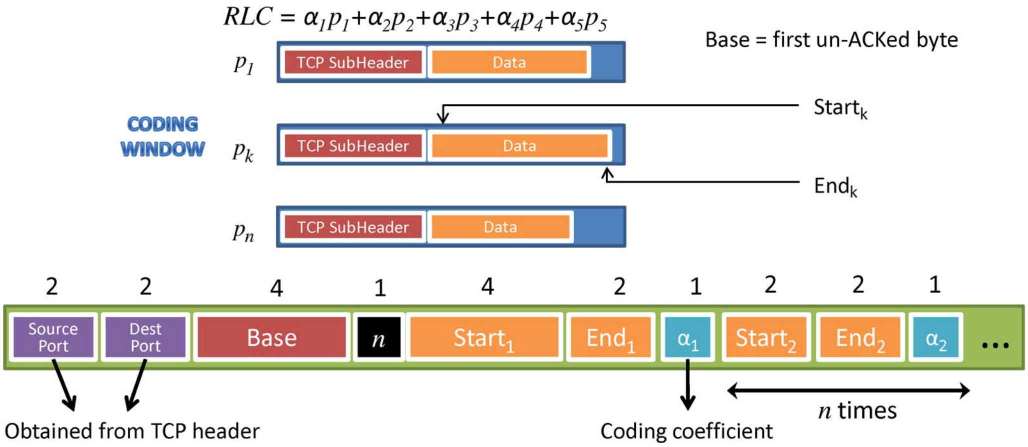


**Fig. 8.** *The network coding header.*

a fixed parameter for the maximum coding window size $W$. The coding window contains the packet that arrived most recently from TCP (which could be a retransmission), and the $(W-1)$ packets before it in sequence number, if possible. However, if some of the $(W-1)$ preceding packets have already been dropped, then the window is allowed to extend beyond the most recently arrived packet until it includes $W$ packets.

Note that this limit on the coding window implies that the code is now restricted in its power to correct erasures and to combat reordering-related issues. The choice of $W$ will thus play an important role in the performance of the scheme. The correct value for $W$ will depend on the length of burst errors that the channel is expected to produce. Other factors to be considered while choosing $W$ are discussed in Section V-C.

4) *Buffer Management:* A packet is removed from the coding buffer if a TCP ACK has arrived requesting a byte beyond the last byte of that packet. If a new TCP segment arrives when the coding buffer is full, then the segment with the newest set of bytes must be dropped. This may not always be the newly arrived segment, for instance, in the case of a TCP retransmission of a previously dropped segment.

## B. Receiver Side Module

The decoder module's operations are outlined below. The main data structure involved is the decoding matrix, which stores the coefficient vectors corresponding to the linear combinations currently in the decoding buffer.

1) *Acknowledgment:* The receiver side module stores the incoming linear combination in the decoding buffer. Then it unwraps the coding header and appends the new coefficient vector to the decoding matrix. Gauss–Jordan elimination is performed and the packet is dropped if it is not innovative (i.e., if it is not linearly independent of previously received linear combinations). After Gauss–Jordan elimination, the oldest unseen packet is identified. Instead of acknowledging the packet number as in Section IV, the decoder acknowledges the last seen packet by *requesting the byte sequence number of the first byte of the first unseen packet*, using a regular TCP ACK. Note that this could happen before the packet is decoded and delivered to the receiver TCP. The port and IP address information for sending this ACK may be obtained from the SYN packet at the beginning of the connection. Any ACKs generated by the receiver TCP are not sent to the sender. They are instead used to update the receive window field that is used in the TCP ACKs generated by the decoder (see subsection below). They are also used to keep track of which bytes have been delivered, for buffer management.

2) *Decoding and Delivery:* The Gauss–Jordan elimination operations are performed not only on the decoding coefficient matrix, but correspondingly also on the coded packets themselves. When a new packet is decoded, any dummy zero symbols that were added by the encoder are pruned using the coding header information. A new TCP packet is created with the newly decoded data and the appropriate TCP header fields and this is then delivered to the receiver TCP.

3) *Buffer Management:* The decoding buffer needs to store packets that have not yet been decoded and delivered to the TCP receiver. Delivery can be confirmed using the receiver TCP's ACKs. In addition, the buffer also needs to store those packets that have been delivered but have not yet been dropped by the encoder from the coding buffer. This is because, such packets may still be involved in incoming linear combinations. The *Base* field in the coding header addresses this issue. *Base* is the oldest byte in the coding buffer. Therefore, the decoder can drop a packet if its last byte is smaller than *Base*, and in addition, has been delivered to and acknowledged by the receiver TCP. Whenever a new linear combination arrives, the value of *Base* is updated from the header, and any packets that can be dropped are dropped.

The buffer management can be understood using Fig. 9. It shows the receiver side windows in a typical situation. In this case, *Base* is less than the last delivered byte. Hence, some delivered packets have not yet been dropped. There could also be a case where *Base* is beyond the last delivered byte, possibly because nothing has been decoded in a while.

4) *Modifying the Receive Window:* The TCP receive window header field is used by the receiver to inform the sender how many bytes it can accept. Since the receiver TCP's ACKs are suppressed, the decoder must copy this information in the ACKs that it sends to the sender. However, to ensure correctness, we may have to modify the value of the TCP receive window based on the decoding buffer size. The last acceptable byte should thus be the minimum of the receiver TCP's last acceptable byte and the last byte that the decoding buffer can accommodate. Note that while calculating the space left in the decoding buffer, we can include the space occupied by data that has

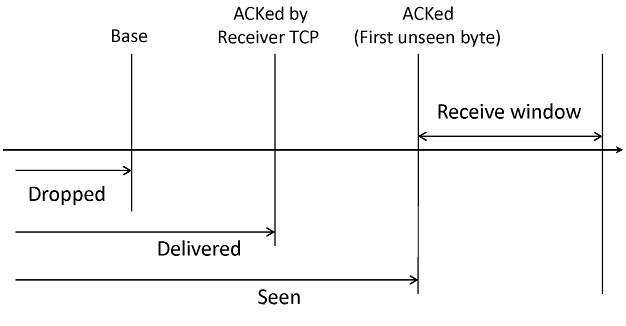


**Fig. 9.** *Receiver side window management.*

already been delivered to the receiver because such data will get dropped when *Base* is updated. If window scaling option is used by TCP, this needs to be noted from the SYN packet, so that the modified value of the receive window can be correctly reported. Ideally, we would like to choose a large enough decoding buffer size so that the decoding buffer would not be the bottleneck and this modification would never be needed.

### C. Discussion of the Practicalities

*1) Redundancy Factor:* The choice of redundancy factor is based on the effective loss probability on the links. For a loss rate of $p_e$, with an infinite window $W$ and using TCP-Vegas, the theoretically optimal value of $R$ is $1/(1-p_e)$. The basic idea is that of the coded packets that are sent into the network, only a fraction $(1-p_e)$ of them are delivered on average. Hence, the value of $R$ must be chosen so that in spite of these losses, the receiver is able to collect linear equations at the same rate as the rate at which the unknown packets are mixed in them by the encoder. As discussed below, in practice, the value of $R$ may depend on the coding window size $W$. As $W$ decreases, the erasure correction capability of the code goes down. Hence, we may need a larger $R$ to compensate and ensure that the losses are still masked from TCP. Another factor that affects the choice of $R$ is the use of TCP-Reno. The TCP-Reno mechanism causes the transmission rate to fluctuate around the link capacity, and this leads to some additional losses over and above the link losses. Therefore, the optimal choice of $R$ may be higher than $1/(1-p_e)$.

*2) Coding Window Size:* There are several considerations to keep in mind while choosing $W$, the coding window size. The main idea behind coding is to mask the losses on the channel from TCP. In other words, we wish to correct losses without relying on the ACKs. Consider a case where $W$ is just 1. Then, this is a simple repetition code. Every packet is repeated $R$ times on average. Now, such a repetition would be useful only for recovering one packet, if it was lost. Instead, if $W$ was say 3, then every linear combination would be useful to recover any of the three packets involved. Ideally, the linear combinations generated should be able to correct the loss of any of the packets that have not yet been acknowledged. For this, we need $W$ to be large. This may be difficult, since a large $W$ would lead to a large coding header. Another penalty of choosing a large value of $W$ is related to the interaction with TCP-Reno. This is discussed in Section V-C3.

The penalty of keeping $W$ small, on the other hand, is that it reduces the error correction capability of the code. For a loss probability of 10%, the theoretical value of $R$ is around 1.1. However, this assumes that all linear combinations are useful to correct any packet's loss. The restriction on $W$ means that a coded packet can be used only for recovering those $W$ packets that have been mixed to form that coded packet. In particular, if there is a contiguous burst of losses that result in a situation where the receiver has received no linear combination involving a particular original packet, then that packet will show up as a loss to TCP. This could happen even if the value of $R$ is chosen according to the theoretical value. To compensate, we may have to choose a larger $R$.

The connection between $W$, $R$, and the losses that are visible to TCP can be visualized as follows. Imagine a process in which whenever the receiver receives an innovative linear combination, one imaginary token is generated, and whenever the sender slides the coding window forward by one packet, one token is used up. If the sender slides the coding window forward when there are no tokens left, then this leads to a packet loss that will be visible to TCP. The reason is, when this happens, the decoder will not be able to see the very next unseen packet in order. Instead, it will skip one packet in the sequence. This will make the decoder generate duplicate ACKs requesting that lost (i.e., unseen) packet, thereby causing the sender to notice the loss.

In this process, $W$ corresponds to the initial number of tokens available at the sender. Thus, when the difference between the number of redundant packets (linear equations) received and the number of original packets (unknowns) involved in the coding up to that point is less than $W$, the losses will be masked from TCP. However, if this difference exceeds $W$, the losses will no longer be masked. The theoretically optimal value of $W$ is not known. However, we expect that the value should be a function of the loss probability of the link. For the experiment, we chose values of $W$ based on trial and error. Further research is needed in the future to fully understand the tradeoffs involved in the choice of $R$ and $W$.

*3) Working With TCP-Reno:* By adding enough redundancy, the coding operation essentially converts the lossiness of the channel into an extension of the RTT. This is why our initial discussion in Section IV proposed the use of the idea with TCP-Vegas, since TCP-Vegas controls the congestion window in a smoother manner using RTT, compared to the more abrupt loss-based variations of TCP-Reno. However, the coding mechanism is also compatible with TCP-Reno. The choice of $W$ plays an important role in ensuring this compatibility. The choice of $W$ controls the power of the underlying code, and hence determines when losses are visible to TCP. As explained above, losses will be masked from TCP as long as the number of received equations is no more than $W$ short of the number of unknowns involved in them. For compatibility with Reno, we need to make sure that whenever the sending rate exceeds the link capacity, the resulting queue drops are visible to TCP as losses. A very large value of $W$ is likely to mask even these congestion losses, thereby temporarily giving TCP a large estimate of capacity. This will eventually lead to a timeout, and will affect throughput. The value of $W$ should therefore be large enough to mask the link losses and small enough to allow TCP to see the queue drops due to congestion.

*4) Computational Overhead:* It is important to implement the encoding and decoding operations efficiently, since any time spent in these operations will affect the RTT perceived by TCP. The finite field operations over GF(256) have been optimized using the approach of [32], which proposes the use of logarithms to multiply elements. Over GF(256), each symbol is one byte long. Addition in GF(256) can be implemented easily as a bitwise XOR of the two bytes.

The main computational overhead on the encoder side is the formation of the random linear combinations of the buffered packets. The management of the buffer also requires some computation, but this is small compared to the random linear coding, since the coding has to be done on every byte of the packets. Typically, packets have a length $L$ of around 1500 bytes. For every linear combination that is created, the coding operation involves $LW$ multiplications and $L(W-1)$ additions over $GF(256)$, where $W$ is the coding window size. Note that this has to be done $R$ times on average for every packet generated by TCP. Since the coded packets are newly created, allocating memory for them could also take time.

On the decoder side, the main operation is the Gauss–Jordan elimination. To identify whether an incoming linear combination is innovative, we need to perform Gauss–Jordan elimination only on the decoding matrix, and not on the coded packet. If it is innovative, then we perform the row transformation operations of Gauss–Jordan elimination on the coded packet as well. This requires $O(LW)$ multiplications and additions to zero out the pivot columns in the newly added row. The complexity of the next step of zeroing out the newly formed pivot column in the existing rows of the decoding matrix varies depending on the current size and structure of the matrix. Upon decoding a new packet, it needs to be packaged as a TCP packet and delivered to the receiver. Since this requires allocating space for a new packet, this could also be expensive in terms of time.

As we will see in Section VI-C, the benefits brought by the erasure correction begin to outweigh the overhead of the computation and coding header for loss rates of about 3%. This could be improved further by more efficient implementation of the encoding and decoding operations.

*5) Interface With TCP:* An important point to note is that *the introduction of the new network coding layer does not require any change in the basic features of TCP*. As described above, the network coding layer accepts TCP packets from the sender TCP and in return delivers regular TCP ACKs back to the sender TCP. On the receiver side, the decoder delivers regular TCP packets to the receiver TCP and accepts regular TCP ACKs. Therefore, neither the TCP sender nor the TCP receiver sees any difference looking downwards in the protocol stack. The main change introduced by the protocol is that the TCP packets from the sender are transformed by the encoder by the network coding process. This transformation is removed by the decoder, making it invisible to the TCP receiver. On the return path, the TCP receiver's ACKs are suppressed, and instead the decoder generates regular TCP ACKs that are delivered to the sender. This interface allows the possibility that regular TCP sender and receiver end hosts can communicate through a wireless network even if they are located beyond the wireless hosts.

While the basic features of the TCP protocol see no change, other special features of TCP that make use of the ACKs in ways other than to report the next required byte sequence number will need to be handled carefully. For instance, implementing the timestamp option in the presence of network coding across packets may require some thought. With TCP/NC, the receiver may send an ACK for a packet even before it is decoded. Thus, the receiver may not have access to the timestamp of the packet when it sends the ACK. Similarly, the TCP checksum field has to be dealt with carefully. Since a TCP packet is acknowledged even before it is decoded, its checksum cannot be tested before acknowledging. One solution is to implement a separate checksum at the network coding layer to detect errors. In the same way, the various other TCP options that are available have to be implemented with care to ensure that they are not affected by the premature ACKs.

## VI. PERFORMANCE RESULTS

In this section, we present simulation results and experimental results aimed at establishing the fairness properties and the throughput benefits of our new protocol. The simulations are based on TCP-Vegas. The experimental results use the TCP-Reno based implementation described in Section V.

### A. Fairness of the Protocol

First, we study the fairness property of our algorithm through simulations.

*1) Simulation Setup:* The protocol described above is simulated using the network simulator (ns-2) [33]. The topology for the simulations is a tandem network consisting of four hops (hence, five nodes), shown in Fig. 10. The source and sink nodes are at opposite ends of the chain. Two FTP applications want to communicate from the source to the sink. There is no limit on the file size. They emit packets continuously until the end of the simulation. They either use TCP without coding or TCP with network coding (denoted TCP/NC). In this simulation, intermediate nodes do not re-encode packets. All the links have a bandwidth of 1 Mb/s, and a propagation delay of 100 ms.

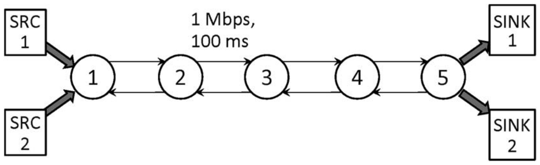


**Fig. 10.** ***Simulation topology.***

The buffer size on the links is set at 200. The TCP receive window size is set at 100 packets, and the packet size is 1000 bytes. The Vegas parameters are chosen to be $\alpha = 28$, $\beta = 30$, $\gamma = 2$ (see [30] for details of Vegas).

*2) Fairness and Compatibility—Simulation Results:* By fairness, we mean that if two similar flows compete for the same link, they must receive an approximately equal share of the link bandwidth. In addition, this must not depend on the order in which the flows join the network. As mentioned earlier, we use TCP-Vegas for the simulations. The fairness of TCP-Vegas is a well-studied problem. It is known that depending on the values chosen for the $\alpha$ and $\beta$ parameters, TCP-Vegas could be unfair to an existing connection when a new connection enters the bottleneck link [34], [35]. Several solutions have been presented to this problem in the literature (for example, see [36] and references therein). In our simulations, we first pick values of $\alpha$ and $\beta$ that allow fair sharing of bandwidth when two TCP flows without our modification compete with each other, in order to evaluate the effect of our modification on fairness. With the same $\alpha$ and $\beta$, we consider two cases:

Case 1: the situation where a network coded TCP flow competes with another flow running TCP without coding;

Case 2: the situation where two coded TCP flows compete with each other.

In both cases, the loss rate is set to 0% and the redundancy parameter is set to 1 for a fair comparison. In the first case, the TCP flow starts first at $t = 0.5$ s and the TCP/NC flow starts at 1000 s. The system is simulated for 2000 s. The current throughput is calculated at intervals of 2.5 s. The evolution of the throughput over time is shown in Fig. 11. The figure shows that the effect of introducing the coding layer does not affect fairness. We see that after the second flow starts, the bandwidth gets redistributed fairly.

For case 2, the simulation is repeated with the same starting times, but this time both flows are TCP/NC flows. The plot for this case is essentially identical to Fig. 11 (and hence is not shown here) because in the absence of losses, TCP/NC behaves identically to TCP if we ignore the effects of field size. Thus, coding can coexist with TCP in the absence of losses, without affecting fairness.

### B. Effectiveness of the Protocol

We now show that the new protocol indeed achieves a high throughput, especially in the presence of losses. We first describe simulation results comparing the protocol's performance with that of TCP in Section VI-B1.

*1) Throughput of the New Protocol—Simulation Results:* The simulation setup is identical to that used in the fairness simulations (see Section VI-A1).

We first study the effect of the redundancy parameter on the throughput of TCP/NC for a fixed loss rate of 5%. By loss rate, we mean the probability of a packet getting lost on each link. Both packets in the forward direction as well as ACKs in the reverse direction are subject to these losses. No re-encoding is allowed at the intermediate nodes. Hence, the overall probability of packet loss across four hops is given by $1-(1-0.05)^4$, which is roughly 19%. Hence, the capacity is roughly 0.81 Mb/s, which when split fairly gives 0.405 Mb/s per flow. The simulation time is 10 000 s.

We allow two TCP/NC flows to compete on this network, both starting at 0.5 s. Their redundancy parameter is varied between 1 and 1.5. The theoretically optimum value is approximately $1/(1-0.19) \simeq 1.23$. Fig. 12 shows the plot of the throughput for the two flows, as a function of the redundancy parameter $R$. It is clear from the plot that $R$ plays an important role in TCP/NC. We can see that the throughput peaks around $R = 1.25$. The peak throughput achieved is 0.397 Mb/s, which is indeed close to the capacity that we calculated above. In the same situation, when two TCP flows compete for the network, the two flows see a throughput of 0.0062 and 0.0072 Mb/s, respectively. Thus, with the correct choice of $R$, the throughput for the flows in the TCP/NC case is very high compared to the TCP case. In fact, even with $R = 1$, TCP/NC achieves about 0.011 Mb/s for each flow improving on TCP by almost a factor of 2.

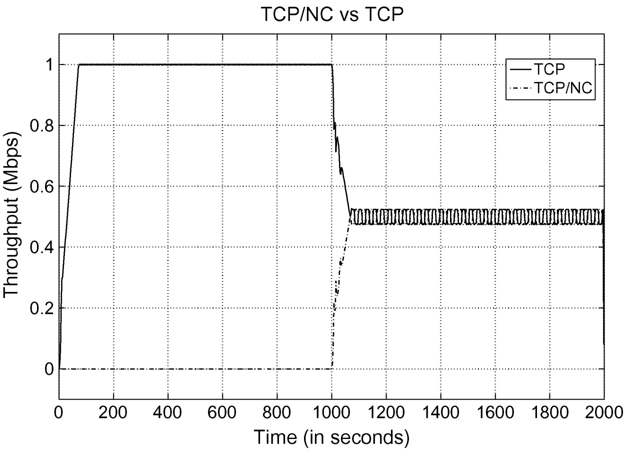


**Fig. 11.** ***Fairness and compatibility—one TCP/NC and one TCP flow.***

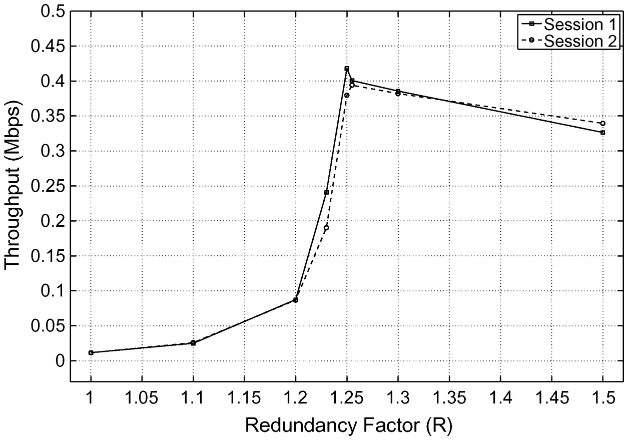
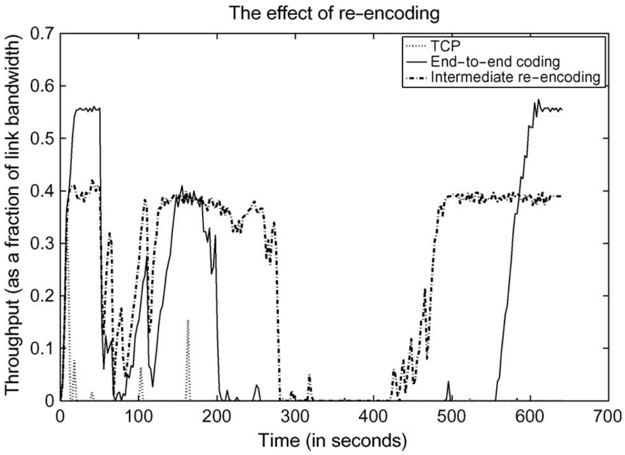


**Fig. 12.** ***Throughput versus redundancy for TCP/NC.***



| TCP | End-to-end coding | Re-encoding at node 3 only |
|---|---|---|
| 0.0042 Mbps | 0.1420 Mbps | 0.2448 Mbps |

**Fig. 14.** ***Throughput with and without intermediate node re-encoding.***

Next, we study the variation of throughput with loss rate for both TCP and TCP/NC. The simulation parameters are all the same as above. The loss rate of all links is kept at the same value, and this is varied from 0% to 20%. We compare two scenarios: two TCP flows competing with each other and two TCP/NC flows competing with each other. For the TCP/NC case, we set the redundancy parameter at the optimum value corresponding to each loss rate. Fig. 13 shows that TCP's throughput falls rapidly as losses increase. However, TCP/NC is very robust to losses and reaches a throughput that is close to capacity. (If $p$ is the loss rate on each link, then the capacity is $(1-p)^4$, which must then be split equally.)

Fig. 14 shows the instantaneous throughput in a 642-s-long simulation of a tandem network with three hops (i.e., four nodes), where erasure probabilities vary with time in some specified manner. The third hop is, on average, the most erasure-prone link. The plots are shown for traditional TCP, TCP/NC with coding only at the source, and TCP/NC with re-encoding at node 3 (just before the worst link). The operation of the re-encoding node is very similar to that of the source—it collects incoming linear combinations in a buffer, and transmits, on average, $R_{\mathrm{int}}$ random linear combinations of the buffer contents for every incoming packet. The $R$ of the sender is set at 1.8, and the $R_{\mathrm{int}}$ of node 3 is set at 1.5 for the case when it re-encodes. The average throughput is shown in the table. A considerable improvement is seen due to the coding, which is further enhanced by allowing intermediate node re-encoding. This plot thus shows that our scheme is also suited to systems with coding inside the network.

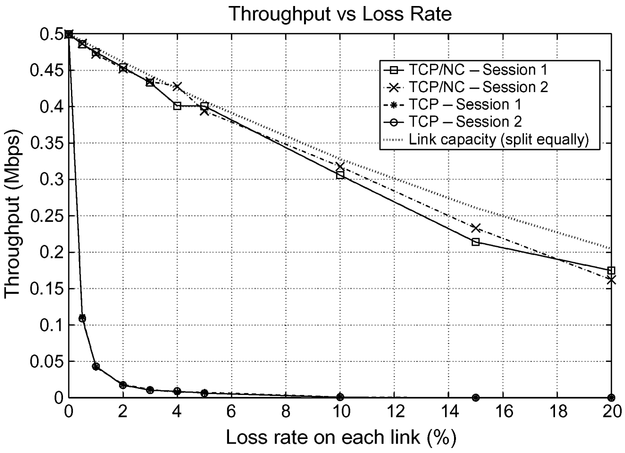


**Fig. 13.** ***Throughput versus loss rate for TCP and TCP/NC.***

*Remark 2:* These simulations are meant to be a preliminary study of our algorithm's performance. Specifically, the following points must be noted.

- Link layer retransmission is not considered for either TCP or TCP/NC. If allowed, this could improve the performance of TCP. However, as mentioned earlier, the retransmission approach does not extend to more general multipath routing solutions, whereas coding is better suited to such scenarios.
- The throughput values in the simulation results do not account for the overhead associated with the network coding headers. The main overhead is in conveying the coding coefficients and the contents of the coding window. If the source and the sink share a pseudorandom number generator, then the coding coefficients can be conveyed succinctly by sending the current state of the generator. However, if intermediate nodes perform re-encoding of packets, then this idea may not work unless the intermediate nodes also use the same

pseudorandom number generator. Also, the coding window contents can be conveyed in an incremental manner to reduce the overhead.

- The loss in throughput due to the finiteness of the field has not been modeled in the simulations. A small field might cause received linear combinations to be noninnovative, or might cause packets to be seen out of order, resulting in duplicate ACKs. However, the probability that such problems persist for a long time falls rapidly with the field size. We believe that for practical choices of field size, these issues will only cause transient effects that will not have a significant impact on performance. These effects remain to be quantified exactly.
- Finally, the decoding delay associated with the network coding operation has not been studied. We intend to focus on this aspect in experiments in the future. A thorough experimental evaluation of all these aspects of the algorithm, on a more general topology, is part of future work.

### C. Experimental Results

We test the protocol on a TCP-Reno flow running over a single-hop wireless link. The transmitter and the receiver are Linux machines equipped with a wireless antenna. The experiment is performed over 802.11a with a bit rate of 6 Mb/s and a maximum of 5 link layer retransmission attempts. RTS-CTS is disabled.

Our implementation uses the Click modular router [37]. In order to control the parameters of the setup, we use the predefined elements of Click. Since the two machines are physically close to each other, there are very few losses on the wireless link. Instead, we artificially induce packet losses using the *RandomSample* element. Note that these packet losses are introduced before the wireless link. Hence, they will not be recovered by the link layer retransmissions, and have to be corrected by the layer above IP. The round-trip delay is empirically observed to be in the range of a few tens of milliseconds. The encoder and decoder queue sizes are set to 100 packets, and the size of the bottleneck queue just in front of the wireless link is set to five packets. In our setup, the loss inducing element is placed before the bottleneck queue.

The quantity measured during the experiment is the goodput over a 20-s-long TCP session. The goodput is measured using *iperf* [38]. Each point in the plots shown is averaged over four or more iterations of such sessions, depending on the variability. Occasionally, when the iteration does not terminate and the connection times out, the corresponding iteration is neglected in the average, for both TCP and TCP/NC. This happens around 2% of the time, and is observed to be because of an unusually long burst of losses in the forward or return path. In the comparison, neither TCP nor TCP/NC uses selective ACKs. TCP uses delayed ACKs. However, we have not implemented delayed ACKs in TCP/NC at this point.

Fig. 16 shows the variation of the goodput with the redundancy factor $R$ for a loss rate of 10%, with a fixed coding window size of $W = 3$. The theoretically optimal value of $R$ for this loss rate is close to 1.11 (1/0.9 to be exact). However, from the experiment, we find that the best goodput is achieved for an $R$ of around 1.25. The discrepancy is possibly because of the type of coding scheme employed. Our coding scheme transmits a linear combination of only the $W$ most recent arrivals, in order to save packet header space. This restriction reduces the strength of the code for the same value of $R$. In general, the value of $R$ and $W$ must be chosen carefully to get the best benefit of the coding operation. As mentioned earlier, another reason for the discrepancy is the use of TCP-Reno.

Fig. 17 plots the variation of goodput with the size of the coding window size $W$. The loss rate for this plot is 5%, with the redundancy factor fixed at 1.06. We see that the best coding window size is 2. Note that a coding window size of $W = 1$ corresponds to a repetition code that simply transmits every packet 1.06 times on average. In comparison, a simple sliding-window code with $W = 2$ brings a substantial gain in throughput by making the added redundancy more useful. However, going beyond 2 reduces the goodput because a large value of $W$ can mislead TCP by masking too many losses, which prevents TCP from reacting to congestion in a timely manner and leads to timeouts. We find that the best value of $W$ for our setup is usually 2 for a loss rate up to around 5%, and is 3 for higher loss rates up to 25%. Besides the loss rate, the value of $W$ could also depend on other factors such as the RTT of the path.

Fig. 15 shows the goodput as a function of the packet loss rate. For each loss rate, the values of $R$ and $W$ have been chosen by trial and error to be the one that maximizes the goodput. We see that in the lossless case, TCP performs better than TCP/NC. This could be because of the

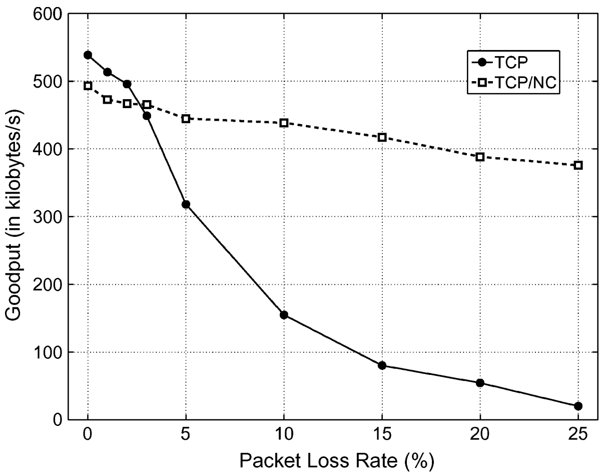


**Fig. 15.** ***Goodput versus loss rate.***

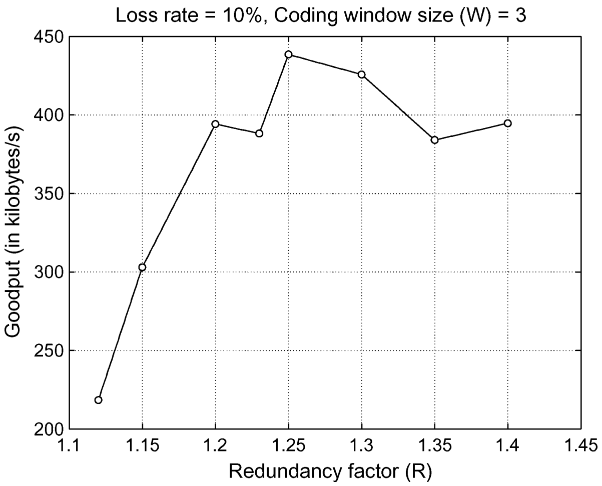


**Fig. 16.** ***Goodput versus redundancy factor for a 10% loss rate and*** $W = 3$.

computational overhead that is introduced by the coding and decoding operations, and also the coding header overhead. However, as the loss rate increases, the benefits of coding begin to outweigh the overhead. The goodput of TCP/NC is therefore higher than TCP. Coding allows losses to be masked from TCP, and hence the fall in goodput is more gradual with coding than without. The performance can be improved further by improving the efficiency of the computation.

## VII. IMPLICATIONS FOR WIRELESS NETWORKING

In considering the potential benefits of our TCP-compatible network coding solution, we focus on the area of wireless links. We now explain the implications of this new protocol for improving throughput in wireless networks.

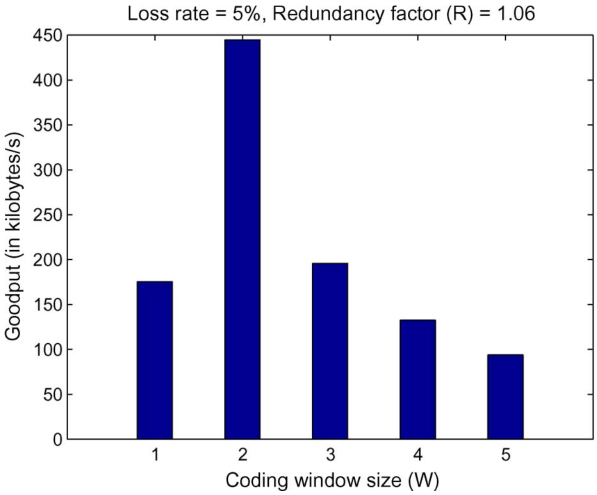


**Fig. 17.** ***Goodput versus coding window size for a 5% loss rate and*** $R = 1.06$.

TCP was originally developed for wired networks and was designed to interpret each packet loss as a congestion signal. Since wired networks have very little packet loss on the links and the predominant source of loss is buffer overflow due to congestion, TCP's approach works well. In contrast, wireless networks are characterized by packet loss on the link and intermittent connectivity due to fading. It is well known that TCP is not well suited for such lossy links. The primary reason is that it wrongly assumes the cause of link losses to be congestion, and reduces its transmission rate unnecessarily, leading to low throughput.

Adapting TCP for wireless scenarios is a very well-studied problem (see [39] and references therein for a survey). The general approach has been to mask losses from TCP using link layer retransmission [40]. However, it has been noted in the literature [41], [42] that the interaction between link layer retransmission and TCP's retransmission can be complicated and that performance may suffer due to independent retransmission protocols at different layers. More importantly, if we want to exploit the broadcast nature of the wireless medium, link layer retransmission may not be the best approach.

### A. Intermediate Node Re-Encoding

Our scheme does not rely on the link layer for recovering losses. Instead, we use an erasure correction scheme based on random linear codes across packets. Coding across packets is a natural way to handle losses. The interface of TCP with network coding that we propose in this paper can be viewed as a generalization of previous work combining TCP with forward erasure correction (FEC) schemes [43]. As opposed to fountain codes and FEC that are typically used for end-to-end coding, our protocol also allows intermediate nodes in the network to perform re-encoding of data. It is thus more general than end-to-end erasure correction over a single path.

Intermediate node re-encoding is an important feature. If nodes are allowed to re-encode data, then we can obtain significant benefits in throughput in multipath and multicast scenarios, and also in a single path unicast scenario with multiple lossy hops. Besides, it gives us the flexibility to add redundancy for erasure correction only where necessary, i.e., before the lossy link. An end-to-end coding approach would congest other parts of the network where the redundancy is not needed.

It is important to note that our scheme respects the end-to-end philosophy of TCP—it would work even if coding operations were performed only at the end hosts. Having said that, if some nodes inside the network also perform network coding, our solution naturally generalizes to such scenarios as well. The queuing analysis in Section IV-D considers such a situation.

### B. Opportunistic Routing and TCP

There has been a growing interest in approaches that make active use of the intrinsic broadcast nature of the wireless medium. In the technique known as opportunistic routing [44], a node broadcasts its packet, and if one of its neighbors receives the packet, that node will forward the packet downstream, thereby obtaining a diversity benefit. If more than one of the neighbors receive the packet, they will have to coordinate and decide who will forward the packet.

The MORE protocol [17] proposed the use of intraflow network coding in combination with opportunistic routing. The random linear mixing (coding) of incoming packets at a node before forwarding them downstream was shown to reduce the coordination overhead associated with opportunistic routing. Another advantage is that the coding operation can be easily tuned to add redundancy to the packet stream to combat erasures. Such schemes can potentially achieve capacity for a multicast connection [5].

However, if we allow a TCP flow to run over an opportunistic routing based system like ExOR [44] or MORE, two issues arise: batching and reordering. Typical implementations use batches of packets instead of sliding windows. ExOR uses batching to reduce the coordination overhead, but as mentioned in [44], this interacts poorly with TCP's window mechanism. MORE uses batching to perform the coding operation. As discussed earlier, if the receiver acknowledges packets only when an entire batch has been successfully decoded, then the decoding delay will interfere with TCP. Since TCP performance heavily relies on the timely return of ACKs, such a delay in the ACKs would affect the RTT calculation and thereby reduce the throughput.

The second issue with opportunistic routing is that it could lead to reordering of packets, since different packets could take different paths to the destination. Reordering is known to interact badly with TCP, as it can cause duplicate ACKs, and TCP interprets duplicate ACKs as a sign of congestion.

Our work addresses both these issues. Since the receiver does not have to wait to decode a packet, but can send a TCP ACK for every degree of freedom received, the batching problem is solved.

As for the reordering issue, we have shown (Lemma 1) that in our scheme, if the linear combination happens over a large enough finite field, then any incoming random linear combination will, with high probability, generate a TCP ACK for the very next unacknowledged packet in order. This is because the random combinations do not have any inherent ordering. The argument holds true even when multiple paths deliver the random linear combinations. Hence, the use of random linear coding with the ACK of degrees of freedom can potentially *address the TCP reordering problem for multipath opportunistic routing schemes.*

Our interface enhancing TCP with network coding yields a new approach to implementing TCP over wireless networks, and it is here where the benefits of our solution are most dramatic.

## VIII. CONCLUSION AND FUTURE WORK

In this work, we propose a new approach to congestion control on lossy links based on the idea of random linear network coding. We introduce a new ACK mechanism that plays a key role in incorporating coding into the sliding-window mechanism of TCP. From an implementation perspective, we introduce a new network coding layer between the transport and network layers on both the source and receiver sides. This means, our changes can be easily deployed in an existing system. Our simulations show that the proposed changes lead to large throughput gains over TCP in lossy links, even with coding only at the source. We demonstrate the practicality of our proposal by implementing it in a real-world experimental setup with TCP-Reno. Significant gains in goodput are seen in the experiments.

We view this work as a first step in taking the theory of network coding to practice. The ideas proposed in this paper give rise to several open questions for future research.

*1) Extensions to Multipath and Multicast:* The scheme has implications for running TCP over wireless networks, in particular, in the context of lossy multipath opportunistic routing scenarios. It is also of interest to extend this approach to other settings such as network-coding-based multipath-TCP for point-to-point connections, as well as network-coding-based multicast connections over a general network. The goal is to present the application layer with the familiar TCP interface while still exploiting the multipath or multicast capabilities of the network. We believe that the proposed ideas and the implementation will lead to the practical realization of this goal and will bring out the theoretically promised benefits of network coding in such scenarios. The idea of coding across packets, combined with our new ACK mechanism, will allow a single TCP state machine to manage the use of several paths. However, further work is needed to ensure that the different characteristics of the paths to the receiver (in case of multipath) or to multiple receivers (in case of multicast) are taken into account correctly by the congestion control algorithm.

*2) Re-Encoding Packets at Intermediate Nodes:* A salient feature of our proposal is that it is simultaneously compatible with the case where only end hosts perform coding (thereby preserving the end-to-end philosophy of TCP), and the case where intermediate nodes perform network coding. Theory suggests that a lot can be gained by allowing intermediate nodes to code as well. Our scheme naturally generalizes to such situations. The ability to code inside the network is important for multicast connections. Even for a point-to-point connection, the ability to re-encode at an intermediate node offers the flexibility of adding redundancy where it is needed, i.e., just before the lossy link. The practical aspects of implementing re-encoding need to be studied further.

*3) Automatic Tuning of TCP/NC Parameters:* More work is needed in the future for fully understanding the role played by the various parameters of the new protocol, such as the redundancy factor $R$ and the coding window size $W$. To achieve high throughputs in a fair manner, the values of $R$ and $W$ have to be carefully adapted based on the characteristics of the underlying link. Ideally, the choice of these parameters should be automated. For instance, the correct values could be learned dynamically based on measurement of the link characteristics such as the link loss rate, bandwidth, and delay. In addition, the parameters have to be extended to cover the case of multipath and multicast scenarios as well. ■

## Acknowledgment

The authors would like to thank Prof. D. Katabi for several useful discussions. They would also like to thank M. Vutukuru and R. Hariharan for their advice and help with the implementation.

## REFERENCES

[1] S.-Y. Chung, G. D. Forney, T. Richardson, and R. Urbanke, "On the design of low-density parity-check codes within 0.0045 dB of the Shannon limit," *IEEE Commun. Lett.*, vol. 5, no. 2, pp. 58–60, Feb. 2001.

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

[3] R. Koetter and M. Médard, "An algebraic approach to network coding," *IEEE/ACM Trans. Netw.*, vol. 11, no. 5, pp. 782–795, Oct. 2003.

[4] S.-Y. Li, R. Yeung, and N. Cai, "Linear network coding," *IEEE Trans. Inf. Theory*, vol. 49, no. 2, pp. 371–381, Feb. 2003.

[5] T. Ho, M. Médard, R. Koetter, D. Karger, M. Effros, J. Shi, and B. Leong, "A random linear network coding approach to multicast," *IEEE Trans. Inf. Theory*, vol. 52, no. 10, pp. 4413–4430, Oct. 2006.

[6] Y. Xi and E. M. Yeh, "Distributed algorithms for minimum cost multicast with network coding," in *Proc. Allerton Annu. Conf. Commun. Control Comput.*, pp. 2073–2082, 2005.

[7] D. Lun, N. Ratnakar, R. Koetter, M. Médard, E. Ahmed, and H. Lee, "Achieving minimum-cost multicast: A decentralized approach based on network coding," in *Proc. IEEE Int. Conf. Comput. Commun.*, Mar. 2005, vol. 3, pp. 1607–1617.

[8] A. F. Dana, R. Gowaikar, R. Palanki, B. Hassibi, and M. Effros, "Capacity of wireless erasure networks," *IEEE Trans. Inf. Theory*, vol. 52, no. 3, pp. 789–804, Mar. 2006.

[9] D. S. Lun, M. Médard, R. Koetter, and M. Effros, "On coding for reliable communication over packet networks," *Phys. Commun.*, vol. 1, no. 1, pp. 3–20, 2008.

[10] D. S. Lun, "Efficient operation of coded packet networks," Ph.D. dissertation, Dept. Electr. Eng. Comput. Sci., Massachusetts Inst. Technol., Cambridge, MA, Jun. 2006.

[11] C. Jiang, B. Smith, B. Hassibi, and S. Vishwanath, "Multicast in wireless erasure networks with feedback," in *Proc. IEEE Symp. Comput. Commun.*, Jul. 2008, pp. 562–565.

[12] S. Bhadra and S. Shakkottai, "Looking at large networks: Coding vs. queueing," in *Proc. IEEE Int. Conf. Comput. Commun.*, Apr. 2006, DOI: 10.1109/INFOCOM.2006.266.

[13] D. S. Lun and T. Ho, *Network Coding: An Introduction*. Cambridge, U.K.: Cambridge Univ. Press, 2008.

[14] T. Ho, "Networking from a network coding perspective," Ph.D. dissertation, Dept. Electr. Eng. Comput. Sci., Massachusetts Inst. Technol., Cambridge, MA, May 2004.

[15] P. A. Chou, Y. Wu, and K. Jain, "Practical network coding," in *Proc. Allerton Conf. Commun. Control Comput.*, pp. 40–49, 2003.

[16] S. Katti, H. Rahul, W. Hu, D. Katabi, M. Medard, and J. Crowcroft, "XORs in the air: Practical wireless network coding," *IEEE/ACM Trans. Netw.*, vol. 16, no. 3, pp. 497–510, Jun. 2008.

[17] S. Chachulski, M. Jennings, S. Katti, and D. Katabi, "Trading structure for randomness in wireless opportunistic routing," in *Proc. ACM SIGCOMM*, pp. 169–180, Aug. 2007.

[18] D. Bertsekas and R. G. Gallager, *Data Networks*, 2nd ed. Englewood Cliffs, NJ: Prentice-Hall, 1991.

[19] W. R. Stevens, *TCP/IP Illustrated, Volume 1: The Protocols*. Reading, MA: Addison-Wesley, 1994.

[20] G. R. Wright and W. R. Stevens, *TCP/IP Illustrated, Volume 2: The Implementation*. Reading, MA: Addison-Wesley, 1994.

[21] C. Fragouli, D. S. Lun, M. Médard, and P. Pakzad, "On feedback for network coding," in *Proc. Conf. Inf. Sci. Syst.*, Mar. 2007, pp. 248–252.

[22] M. Luby, "LT codes," in *Proc. IEEE Symp. Found. Comput. Sci.*, Nov. 2002, pp. 271–282.

[23] A. Shokrollahi, "Raptor codes," in *Proc. IEEE Int. Symp. Inf. Theory*, p. 36, Jul. 2004.

[24] J. W. Byers, M. Luby, and M. Mitzenmacher, "A digital fountain approach to asynchronous reliable multicast," *IEEE J. Sel. Areas Commun.*, vol. 20, no. 8, pp. 1528–1540, Oct. 2002.

[25] B. Shrader and A. Ephremides, "On the queueing delay of a multicast erasure channel," in *Proc. IEEE Inf. Theory Workshop*, Oct. 2006, pp. 423–427.

[26] B. Shrader and A. Ephremides, "A queueing model for random linear coding," in *Proc. IEEE Military Commun. Conf.*, Oct. 2007, DOI: 10.1109/MILCOM.2007.4454981.

[27] J. Lacan and E. Lochin, "On-the-fly coding to enable full reliability without retransmission," ISAE, LAAS-CNRS, France, Tech. Rep., 2008. [Online]. Available: http://arxiv.org/pdf/0809.4576

[28] L. Clien, T. Ho, S. Low, M. Chiang, and J. Doyle, "Optimization based rate control for multicast with network coding," in *Proc. IEEE Int. Conf. Comput. Commun.*, May 2007, pp. 1163–1171.

[29] J. K. Sundararajan, D. Shah, and M. Médard, "ARQ for network coding," in *Proc. IEEE Int. Symp. Inf. Theory*, Jul. 2008, pp. 1651–1655.

[30] L. S. Brakmo, S. W. O'Malley, and L. L. Peterson, "TCP Vegas: New techniques for congestion detection and avoidance," in *Proc. SIGCOMM Symp.*, pp. 24–35, Aug. 1994.

[31] J. J. Hunter, *Mathematical Techniques of Applied Probability, Vol. 2, Discrete Time Models: Techniques and Applications*. New York: Academic, 1983.

[32] N. R. Wagner, *The Laws of Cryptography With Java Code*. [Online]. Available: http://www.cs.utsa.edu/~wagner/lawsbookcolor/laws.pdf

[33] Network simulator (ns-2). [Online]. Available: http://www.isi.edu/nsnam/ns/

[34] U. Hengartner, J. Bolliger, and T. Gross, "TCP Vegas revisited," in *Proc. Int. Conf. Comput. Commun.*, pp. 1546–1555, 2000.

[35] C. Boutremans and J.-Y. Le Boudec, "A note on the fairness of TCP Vegas," in *Proc. Broadband Commun.*, pp. 163–170, 2000.

[36] J. Mo, R. La, V. Anantharam, and J. Walrand, "Analysis and comparison of TCP Reno and Vegas," in *Proc. Int. Conf. Comput. Commun.*, pp. 1556–1563, 1999.

[37] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek, "The click modular router," *ACM Trans. Comput. Syst.*, vol. 18, no. 3, pp. 263–297, Aug. 2000.

[38] NLANR Distributed Applications Support Team, *Iperf—The TCP/UDP Bandwidth Measurement Tool*. [Online]. Available: http://dast.nlanr.net/Projects/Iperf/

[39] S. Rangwala, A. Jindal, K.-Y. Jang, K. Psounis, and R. Govindan, "Understanding congestion control in multi-hop wireless mesh networks," in *Proc. ACM/IEEE Int. Conf. Mobile Comput. Netw.*, pp. 291–302, 2008.

[40] S. Paul, E. Ayanoglu, T. F. L. Porta, K.-W. H. Chen, K. E. Sabnani, and R. D. Gitlin, "An asymmetric protocol for digital cellular communications," in *Proc. Int. Conf. Comput. Commun.*, pp. 1053–1062, 1995.

[41] A. DeSimone, M. C. Chuah, and O.-C. Yue, "Throughput performance of transport-layer protocols over wireless LANs," in *Proc. IEEE Global Telecommun. Conf.*, 1993, vol. 1, pp. 542–549.

[42] H. Balakrishnan, S. Seshan, and R. H. Katz, "Improving reliable transport and handoff performance in cellular wireless networks," *ACM Wireless Netw.*, vol. 1, no. 4, pp. 469–481, Dec. 1995.

[43] F. Brockners, "The case for FEC-fueled TCP-like congestion control," in *Kommunikation in Verteilten Systemen*, R. Steinmetz, Ed. Berlin, Germany: Springer-Verlag, 1999, pp. 250–263.

[44] S. Biswas and R. Morris, "ExOR: Opportunistic multi-hop routing for wireless networks," in *Proc. ACM SIGCOMM*, 2005, pp. 133–144.

## ABOUT THE AUTHORS

**Jay Kumar Sundararajan** received the B.Tech. degree in electrical engineering from the Indian Institute of Technology, Madras, India, in 2003 and the S.M. and Ph.D. degrees in electrical engineering and computer science from the Massachusetts Institute of Technology, Cambridge, in 2005 and 2009, respectively.

He is currently with Qualcomm Inc., San Diego, CA. His research interests include network coding, information theory and network algorithms, and protocols.

Dr. Sundararajan is a co-winner of the Marconi Young Scholar Award in 2008.

**Devavrat Shah** received the B.Tech. degree in computer science and engineering from the Indian Institute of Technology, Bombay, India, in 1999 (with the honor of the President of India Gold Medal) and the Ph.D. degree from the Computer Science Department, Stanford University, Stanford, CA, in October 2004.

He is currently a Jamieson Career Development Associate Professor at the Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge. He is a member of the Laboratory of Information and Decision Systems (LIDS). His research focus is on theory of large complex networks, which includes network algorithms, stochastic networks, network information theory, and large-scale statistical inference.

Dr. Shah was co-awarded the best paper awards at the 2004 IEEE International Conference on Computer Communications and the 2009 ACM SIGMETRICS/Performance; and supervised best student paper awards at the 2008 Neural Information Processing Systems and the 2009 ACM SIGMETRICS/Performance. He received the 2005 George B. Dantzig best dissertation award from the INFORMS. He received the first ACM SIGMETRICS Rising Star Award (2008) for his work on network scheduling algorithms. He was recently awarded the 2010 Erlang Prize from INFORMS which is given to a young researcher for outstanding contributions to applied probability. He is currently an Associate Editor of *Operations Research.*

**Muriel Médard** (Fellow, IEEE) received the B.S. degrees in electrical engineering and computer science and in mathematics in 1989, the B.S. degree in humanities in 1990, the M.S. degree in electrical engineering in 1991, and the Sc.D. degree in electrical engineering in 1995, all from the Massachusetts Institute of Technology (MIT), Cambridge.

She is a Professor at the Electrical Engineering and Computer Science Department, MIT, Cambridge. She was previously an Assistant Professor at the Electrical and Computer Engineering Department and a member of the Coordinated Science Laboratory at the University of Illinois Urbana-Champaign, Urbana. From 1995 to 1998, she was a Staff Member at MIT Lincoln Laboratory in the Optical Communications and the Advanced Networking Groups. Her research interests are in the areas of network coding and reliable communications, particularly for optical and wireless networks.

Prof. Médard has served as an Associate Editor for the Optical Communications and Networking Series of the IEEE Journal on Selected Areas in Communications, as an Associate Editor in Communications for the IEEE Transactions on Information Theory, and as an Associate Editor for the *OSA Journal of Optical Networking*. She has served as a Guest Editor for the IEEE Journal of Lightwave Technology, the Joint Special Issue of the IEEE Transactions on Information Theory and the IEEE/ACM Transactions on Networking and Information Theory and the IEEE Transactions on Information Forensic and Security: Special Issue on Statistical Methods for Network Security and Forensics. She serves as an Associate Editor for the IEEE/OSA Journal of Lightwave Technology. She is a member of the Board of Governors of the IEEE Information Theory Society and its first Vice-President Elect. She has served as Technical Program Committee Co-Chair of ISIT, WiOpt, and CONEXT. She was awarded the 2009 Communication Society and Information Theory Society Joint Paper Award, the 2009 William R. Bennett Prize in the Field of Communications Networking, the 2002 IEEE Leon K. Kirchmayer Prize Paper Award, the Best Paper Award at the Fourth International Workshop on the Design of Reliable Communication Networks DRCN (2003). She received a NSF Career Award in 2001 and was co-winner of the 2004 Harold E. Edgerton Faculty Achievement Award, established in 1982 to honor junior faculty members "for distinction in research, teaching and service to the MIT community." In 2007 she was named a Gilbreth Lecturer by the National Academy of Engineering.

**Szymon Jakubczak** received the M.Sc. degree in computer science from Warsaw University of Technology, Warsaw, Poland, in 2005 and the S.M. degree in electrical engineering and computer science from the Massachusetts Institute of Technology (MIT), Cambridge, in 2007, where he is currently working towards the Ph.D. degree in electrical engineering and computer science.

His research interests include routing and transport in wireless networks, cross-channel design, and software radios.

**Michael Mitzenmacher** received the B.A. degree in mathematics and computer science (*summa cum laude*) from Harvard University, Cambridge, MA, in 1991. After studying mathematics for a year in Cambridge, U.K., on the Churchill Scholarship, he received the Ph.D. degree in computer science from the University of California Berkeley, Berkeley, in 1996.

He then worked at Digital Systems Research Center until joining the Harvard University faculty in 1999. Currently, he is a Professor of Computer Science at the School of Engineering and Applied Sciences, Harvard University. He has authored or coauthored over 150 conference and journal publications on a variety of topics, including algorithms for the Internet, efficient hash-based data structures, erasure and error-correcting codes, power laws, and compression. His textbook on randomized algorithms and probabilistic techniques in computer science was published in 2005 by Cambridge University Press.

Dr. Mitzenmacher was a corecipient of the 2002 IEEE Information Theory Society Best Paper Award and a winner of the 2009 ACM SIGCOMM Test of Time Award (for his work on low-density parity-check codes).

**João Barros** received the undergraduate education in electrical and computer engineering from the Universidade do Porto (UP), Porto, Portugal and Universitaet Karlsruhe, Karlsruhe, Germany and the Ph.D. degree in electrical engineering and information technology from the Technische Universitaet Muenchen (TUM), Muenchen, Germany, in 2004.

He is an Associate Professor at the Department of Electrical and Computer Engineering, School of Engineering, University of Porto, Porto, Portugal, the Head of the Porto Delegation of the Instituto de Telecomunicações, and a Visiting Professor with the Massachusetts Institute of Technology (MIT), Cambridge. In February 2009, he was appointed National Director of the CMU-Portugal Program, a five-year international partnership between Carnegie Mellon University, Pittsburgh, PA, and 12 Portuguese Universities and Research Institutions, with a total budget of 56 M Euros. The focus of his research lies in the general areas of information theory, communication networks, and data security, with special emphasis on network coding, physical-layer security, and vehicular sensor networks.

Dr. Barros received the Best Teaching Award from the Bavarian State Ministry of Sciences, Research and the Arts, as well as scholarships from several institutions, including the Fulbright Commission, the Japanese Foreign Ministry, and the Luso-American Foundation. He served as Secretary of the Board of Governors of the IEEE Information Theory Society between 2006 and 2009. His service also included co-chairing the 2008 IEEE Information Theory Workshop, Porto, Portugal, and participating in many Technical Program Committees of IEEE Conferences. He is a national delegate to the European Commission for the area of security. He has been a Principal Investigator and Co-Principal Investigator of numerous national and international research projects.