PiggyCode: a MAC layer network coding scheme to improve TCP performance over wireless networks

Luca Scalia∗, Fabio Soldo†, Mario Gerla‡

∗ Dipartimento di Ingegneria Elettrica, Elettronica e delle Telecomunicazioni, Università degli Studi di Palermo
† Dipartimento di Elettronica, Politecnico di Torino
‡ Computer Science Department, University of California, Los Angeles
E-mail: luca.scalia@tti.unipa.it, soldo@tlc.polito.it, gerla@cs.ucla.edu

Abstract—In this paper we propose PiggyCode, a network-coding based scheme specifically designed to enhance TCP performance over IEEE 802.11 multi-hop wireless networks. The root of this approach is a network coding module operating between the Network and the MAC layer. Each node running PiggyCode encodes, whenever it is possible, TCP-DATA and TCP-ACK packets belonging to the same information flow. The coding approach is conceptually analogous to piggyback the TCP-ACK packet within the TCP-DATA packet, with the substantial difference that, by performing network coding operations, the actual packet size remains unchanged. The proposed scheme is simple and effective. It leverages the benefits of network coding in the wireless environment, to jointly reduce the overall number of transmissions on the channel and speed up the delivery process of TCP-ACK packets, thus achieving significant improvements in terms of TCP performance.

I. INTRODUCTION

The idea to apply network coding as a transmission paradigm was originally proposed by Ahlswede et al. in [1]. Basically, the key concept is to allow intermediate nodes to encode, before transmitting, the information they received through their input links. In this sense, network coding can be viewed as an extension of the traditional routing paradigm store-and-forward: with network coding, nodes store, code, and forward packets. By applying it as a relaying paradigm, it was shown in [1] that, in multicast traffic scenarios, it is possible to achieve the maximum admissible throughput of the network (also known as broadcast capacity) whereas traditional routing solutions, generally, do not allow it. In fact, differently from the case of unicast and broadcast, where the classical maxflow/mincut theorem allows to achieve a precise solution to the maximum achievable rate, for a given network scenario, in the case of multicast sessions, the problem becomes much harder and only an upper bound can be substantially provided1. Network coding, from a theoretical point of view, unifies the results for the maximum achievable rate in unicast, multicast and broadcast sessions. In fact, it can be proved that, by enhancing intermediate nodes with network coding capabilities, a source node $s$ can transmit to a set of receiver nodes, $T$, at a rate equal to the broadcast capacity, that is $\min_i(\text{mincut}(s, t_i))$, regardless the actual cardinality of the set $T$ (see [4]).

Passing from theory to practice, the concrete deployment of network coding in real networks is made possible by considering each packet $P_i$, $i = 1, ..., K$, stored in a generic node $N$, as a vector over some finite field $F_{2^q}$. More in detail, assume that each packet consists of $b$ bits. We can interpret a subset of $q$ consecutive bits as a symbol in $F_{2^q}$. Thus, the entire packet can be represented as a vector of size $L = \left\lceil \frac{b}{q} \right\rceil$, over $F_{2^q}$.

When a relaying node $N$ has a packet to transmit, before performing the actual forwarding operations, it generates a new packet by means of a linear combination of the $K$ packets $P_1, P_2, ..., P_K$ it stores in its buffer. Referring to the random network coding scheme proposed in [5], a set of coefficients $m_i \in \{1, 2, ..., K\}$, is randomly drawn out of the finite field $F_{2^q}$. The vector $m = (m_1, m_2, ..., m_K)$ formed by these coefficients is called encoding vector and the new packet $P'$ generated as a linear combination of the stored $P_1, P_2, ..., P_K$ packets, namely $P' = \sum_{i=1}^{K} m_i P_i$, is the encoded packet to be sent over the output link. Note that all the algebraic operations are performed as operations in $F_{2^q}$, thus ensuring that the new packet, $P'$, can be represented as a vector over $F_{2^q}$.

In the decoding phase, any receiving node that retrieves at least $K$ independent versions of the encoded packet $P'$, that is, $K$ linearly independent encoding vectors, is able to recover each of the initial $P_1, P_2, ..., P_K$ packets. Obviously, the larger the field in which the random coefficients are picked up, the higher the probability to collect, at the receiver side, linearly independent encoding vectors. However, by increasing the field size, the coding and decoding procedures become more complicated and computationally expensive.

II. APPLYING NETWORK CODING IN WIRELESS NETWORKS: DISCUSSION AND RELATED WORK

Since its apparition, the popularity of network coding has grown considerably and a lot of research attention has focused

1Given a source node $s$ and a set of receiver nodes $T = \{t_1, t_2, ..., t_N\}$, in the unicast scenario ($N=1$), the maxflow/mincut theorem asserts that $\text{maxflow}(s,T) = \min_i(\text{mincut}(s, t_i))$ [2]. Similarly, in the broadcast scenario, $\text{maxflow}(s,T) = \min_i(\text{mincut}(s, t_i))$ [3]. In the multicast case, with conventional routing the maxflow/mincut theorem gives us only an upper bound to the maximum rate achievable at source: $\text{maxflow}(s,T) \leq \min_i(\text{mincut}(s, t_i))$

2If the packets stored at a node have different sizes, it is sufficient to append the necessary number of 0 bits to fit the size $b = \max_i b_i$, where $b_i$ is the number of bits of packet $P_i$
on it. Most of these research works have inevitably considered multicast applications in order to exploit the advantages deriving from it. Nonetheless, the possibility of deploying network coding in wireless scenarios has several unexpected, but encouraging, implications that make it also suitable for unicast scenarios. In fact, since the nature of the wireless medium is intrinsically broadcast, every node is potentially able to hear the transmissions not directly intended to it\(^3\).

Leveraging this “overhearing” capability we can enable a sort of collaboration among nodes that definitely facilitates the information delivery process. An interesting discussion on the advantages of using network coding in the wireless scenario can be found in [6]. Here the authors observe that wireless specific phenomena, such as frequent retransmissions and nodes mobility, can further improve network coding performance (i.e., by considering a new encoding process for each retransmission). However, even in [6], only the case of multicast content distribution is accounted.

Recently, in [8], [9], [10], it has been considered the possibility of deploying network coding solutions for unicast traffic scenarios. In [9] the authors provide a theoretical analysis of the unicast information distribution. The first practical approaches to network coding for multiple unicast data flows are proposed in [10] and [8]. Specifically, in [8] Katabi et al. introduce COPE, a practical network coding scheme that operates above the MAC layer and performs packet coding operations according to an “opportunistic” mechanism. Basically, whenever a node has to deliver packets to the nodes within its communication range, it opportunistically combines (that is, “xor’s”) those packets that are more likely to be decoded by its neighbors, based on the estimation of their buffers. With this approach authors are able to achieve significant throughput gains, especially in the case of UDP traffic. However in case of TCP traffic with hidden terminals the average throughput gain remains low (about 2-3%). A similar idea, is also presented in [10]. In this research work, authors propose a network coding scheme to perform information exchange between two different nodes in a line topology.

The network coding scheme we propose here partially back-ups on these works as pure coding and decoding procedures (bitwise xor). Nonetheless it takes a completely different point of view in terms of the forwarding procedure and the selection of the packets to transmit. To fully exploit the peculiarities of both TCP, and network coding in wireless networks, we design our scheme along the guidelines summarized below. The first is a well known network coding technique, the other two are part of our contribution.

- implementation of a 2.5 network coding layer to perform simple coding/decoding operations (xor)
- introduction of a cross-layer interaction between the routing and the network coding layer, able to dynamically re-route TCP-ACK packets on the same path of TCP-DATA packets.
- TCP-DATA and TCP-ACK coding. Contrarily to other network coding schemes (e.g. [8]) we code together only packets of different size. As we detail later, this choice has several beneficial effects on TCP performance.

Our scheme does not require neither the estimation of the neighbors’ buffer nor the use of control messages to allow protocol’s operations. Moreover, to the best of our knowledge, this is the first attempt to improve TCP performance by exploiting a network coding-based scheme explicitly designed to this goal.

III. NETWORK SCENARIO

PiggyCode is a packet forwarding scheme rooted on the theory of network coding. Just like other network coding schemes already proposed for the wireless environment, PiggyCode exploits the intrinsic broadcast nature of the wireless medium.

In what follows, we assume that all nodes have “overhearing” capabilities, that is, they are able to listen to transmissions not explicitly directed to them. Each node is also supposed to have a buffer in which it temporarily stores transmitted packets. To illustrate the protocol operations, we consider as a reference scenario, the ad-hoc wireless network reported in Fig. 1. In this specific example, nodes are placed according to a grid topology. Their transmission range is configured such that each of them can directly communicate exclusively with its one-hop neighbors (adjacent nodes), while the carrier sense range is configured such to detect the transmissions of one-hop neighbors and nodes placed in the same column of the grid. For instance, referring to Fig. 1-(a), node \(A\) can communicate only with nodes \(X, Y\) and \(F_A\), and can detect the transmissions of node \(Z\).

We consider a single TCP connection established from node \(A\) to node \(B\). This requires lesser notations while conveying the general idea. Nonetheless our conjectures can be extended to more general topologies and traffic scenarios. Referring to Fig. 1, nodes that do not take part to the TCP data exchange are indicated with black circles, whereas active nodes which transmit either TCP-DATA or TCP-ACK packets are indicated with white circles. As shown in the figure, in general, TCP-DATA and TCP-ACK packets are routed to the respective destinations according to possibly different paths. Moreover, nodes forwarding TCP-DATA and TCP-ACK packets may contend for the same wireless resource.

IV. PIGGYCODE PROTOCOL DESCRIPTION

The objective of PiggyCode is to improve TCP performance in wireless networks and reduce the overall usage of the medium by opportunealy coding in a single packet TCP-DATA and TCP-ACK packets belonging to the same flow. This allows to limit the total number of competing packets with the positive effect of drastically reducing the overall network interference.

To achieve this goal, PiggyCode interacts with the routing protocol in order to re-route TCP-ACK packets on the same path used by TCP-DATA packets. At each TCP-ACK re-routing, due to the superposition of TCP-DATA and TCP-ACK packets, new network coding opportunities arise, and thus,
Fig. 1. Network topology and TCP-ACK re-routing procedure: TCP-DATA and TCP-ACK packets route before (a) and after PiggyCode operations (b).

Fig. 2. Architecture of a node supporting PiggyCode.

Further packets transmission can be saved. This mechanism greatly increases coding opportunities at the intermediate nodes, thus reducing the overall congestion (and then interference) of the network, without introducing neither additional delays nor any overhead traffic. The coding approach is conceptually similar (but not analogous) to conventional piggybacking. In fact, PiggyCode is meant to be similar to piggybacking in that it allows to transport acknowledgment packets together with data packets. Nevertheless it is not analogous to traditional piggybacking since, thanks to opportune coding techniques at the network nodes (that is, xoring), (i) the actual ack packet results physically transferred together with the data packet, while maintaining the original data packet size, (ii) the acknowledgement information travels in the opposite direction of the data flow, and, (iii) this kind of piggyback applies in the case of a single TCP data flow. Major details can be found in the following subsection.

A. PiggyCode Packet Header

To better comprehend PiggyCode protocol operations let us first introduce the packet header structure adopted. Basically, the PiggyCode packet header is made up of the following fields:

- TCP-TYPE (1 bit): includes the information bit used to discriminate between TCP-DATA packets and TCP-ACK packets.
- CODED (1 bit): includes the information bit used to discriminate between coded and non-coded packets.
- PKT-ID-VECTOR (64 bits): the vector that contains the unique identifiers of the packets that have been coded together.

We note that the size of this field is in general variable and it is a multiple of 32 bits. Basically, it depends on how many packets we want to code together. In our case, since we always perform encoding operations on 2 packets, it is equal to 64 bits.

B. Node Architecture

The reference node architecture is reported in Fig. 2. As shown in this figure, we add a network coding layer between the Network and the MAC layer.

The network coding layer embeds a coding and a decoding block. The coding block first performs the coding operations between the packets received from the upper layer and the ones already enqueued in the outgoing buffer. Note that, since the new packet will be encoded with a packet already placed in the interface queue, it saves a considerable amount of time for its transmission. Basically, it transparently jumps ahead in the outgoing queue without affecting the other packets waiting time. However, before encoding the packet, the coding block first saves a temporary copy of the packet itself in the decoding buffer, in order to allow the decoding process.

The decoding block is demanded to decode the encoded packets it received through its network interface. It interacts with the sniffer module in order to get coded packets not directly sent to the node and with the decoding buffer to exploit the stored copies of clear packets in the decoding process.

C. Coding and Decoding Procedures

In what follows, we describe the operations related to the coding and decoding operations performed by the PiggyCode protocol. Consider the network scenario in Fig. 1 and a TCP connection established from node $A$ to node $B$.

When node $B$ sends a TCP-ACK packet to its next hop node, $F_B$, all nodes falling in node $B$’s transmission range, thanks to their overhearing capabilities, are able to listen the TCP-ACK packet.

Among them, a new forwarder for the TCP-ACK packet is locally chosen according to the following rule: the node that is in charge of relaying TCP-DATA packets (node $F_2$ in Fig. 1-(a)) becomes responsible for forwarding the received TCP-ACK packet. The other nodes, node $F_D$ included, simply discard it. Note that, if two nodes involved in TCP-DATA forwarding overhear the same TCP-ACK packet, only the one that transmitted the last TCP-DATA packet will process it, thus avoiding the forwarding of the same duplicate packet. With this design choice, we let TCP-ACK packets follow the same path of the TCP-DATA packets (see Fig. 1-(b)).

4This information bit may be acquired, in a cross-layer fashion, at the IP level, by opportuneely setting the “options” field of the IP packet header.
After receiving the TCP-ACK packet, node $F_2$ starts the coding process. First it stores a temporary copy of the TCP-ACK in the decoding buffer. Then it tries to encode it with the TCP-DATA packet, available in the outgoing buffer, that will be transmitted first. If the coding operation is successful, a new encoded packet is created and the PiggyCode header is appended to it. Precisely, the CODED field is set to 1 and the unique identifiers of the encoded TCP-DATA and TCP-ACK are inserted in the PKT-ID-VECTOR field. Otherwise, the TCP-ACK packet is simply inserted in the transmitting buffer. The packet is then passed to the MAC layer for transmission. Note that since the TCP-DATA and TCP-ACK packets have different sizes, the TCP-ACK packet is appended as many 0 bits as needed to fit the TCP-DATA data size. Similarly, when a TCP-DATA packet is received by the coding layer, it tries to encode it with the “first” TCP-ACK packet available in the outgoing buffer Fig. 2.

Suppose now that a node receives a coded packet. Basically, the first operations it performs are the extraction of the PiggyCode packet header and the check of the PKT-ID-VECTOR field, in order to get the original identifiers of the encoded packets. These identifiers are used as a search key to retrieve the original copy of one of the two encoded packets that the node has previously stored in its decoding buffer. Once the non-coded version of one of these two packets is found, it is xored with the received coded packet. The result of this operation is the retrieval of the original content of the second encoded packet. Note that, since all nodes directly involved in the TCP forwarding procedure own a clear copy of their transmitted packets, the probability they have to decode an overheard coded packet is equal to 1. In fact, the coded packet is always made up of packets of different types, and, one of the them, is always a packet that the receiving node has previously sent. This is a crucial point, since it differentiates our network coding approach from the others, in which the decoding probability at the intermediates nodes is always less than 1, due to factors such as node density [6] or heuristic adopted to estimate neighbors’ buffer [8].

Finally, we also want to remark that none of the coding/decoding operations causes re-ordering issues at the intended destination nodes. In fact, in PiggyCode, only packets of different types are actually coded together, and then TCP-DATA packets will not “overtake” other TCP-DATA packets. Obviously, the same considerations still apply to TCP-ACK packets. This is another important aspect that further differentiates PiggyCode from the other network coding proposed for the wireless environment.

V. PERFORMANCE EVALUATION

In this section we report the results obtained by comparing the TCP performance achieved by using the proposed PiggyCode protocol and the legacy IEEE 802.11 MAC layer. Since we wanted to purely evaluate the effects of our network coding scheme on a generic TCP connection, we used the “full TCP” version present in the NS-2 framework. From our point of view, the use of different and possibly better TCP versions was not relevant, since we were interested in evaluating the benefits of the proposed protocol with respect to the legacy IEEE 802.11 MAC. As anticipated before, as simulation platform we used the NS-2 network simulator, version 2.29, in which we implemented all the protocol features we described in previous sections. As network topology, we considered the one reported in Fig. 1, with the only difference that node $A$ and node $B$, respectively the source and destination pairs of the TCP connection, are now separated by 2 intermediate hops.

For our evaluations we adopted the AODV ad-hoc routing protocol. We considered 100 seconds long simulation runs, in which, unless differently indicated, the IEEE 802.11 basic-rate and data-rate of the nodes were respectively set to 1 and 2 Mbps.

A. Throughput Analysis

As a preliminary overview of the PiggyCode performance, in table I we report the different TCP throughput behaviors respectively obtained at 1, 2, 5.5 and 11 Mbps when an FTP file transfer is activated between nodes $A$ and $B$. As we can see from the table, PiggyCode allows relevant throughput gains in comparison to standard 802.11 MAC. However, the effectiveness of the proposed protocol results particularly appreciable as available bandwidth of the network increases. We can intuitively explain this behavior as follows. When the overall link capacity is limited, although TCP tries to inject as much traffic as it can, the number of packets traveling across the network results quite low. Then, only few packets can be actually encoded (low coding opportunities) at the intermediate nodes. This implies that PiggyCode gains result strongly limited. As soon as the link capacity increases, the

<table>
<thead>
<tr>
<th>Throughput (Mbps)</th>
<th>PiggyCode</th>
<th>Standard TCP</th>
<th>Gain (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Mbps</td>
<td>0.359060</td>
<td>0.344509</td>
<td>+4.2%</td>
</tr>
<tr>
<td>2 Mbps</td>
<td>0.654373</td>
<td>0.608734</td>
<td>+7.4%</td>
</tr>
<tr>
<td>5.5 Mbps</td>
<td>1.326786</td>
<td>1.187959</td>
<td>+11.6%</td>
</tr>
<tr>
<td>11 Mbps</td>
<td>1.905243</td>
<td>1.6309/93</td>
<td>+16.8%</td>
</tr>
</tbody>
</table>

Fig. 3. Throughput performance in the case of PiggyCode and standard IEEE 802.11 MAC.
TCP sender tends to fill it by injecting more and more packets into the network thus increasing the overall coding probability. This implies that the number of saved transmissions increase and, consequently, a further portion of the link capacity results now available to the TCP sender. To provide a better understanding of the PiggyCode performance we used, as a test scenario, a rate-controlled traffic source as performed in [11]. The rationale behind this unusual and, obviously, unpractical choice derived from the fact that we wanted to evaluate the performance of our scheme as a function of the offered load in scenarios in which the typical TCP aggressiveness in saturating the link bandwidth do not occur. For these experiments, we maintained the link capacity fixed to 2 Mbps. In Fig. 3, we plot the throughput behavior in the case standard TCP traffic over legacy 802.11 and PiggyCode. As we can see from the figure, there is a threshold above which the TCP over PiggyCode definitively outperforms the common TCP over 802.11 MAC. Before this threshold, PiggyCode throughput performance result almost the same of adopting the standard 802.11 MAC. In fact, in this case, the number of coding opportunities at each node results strongly penalized by the low source rate.

Fig. 4 confirms this intuition. For the same network scenario, we plot the number of forwarded TCP-ACKs as a function of the offered load. As we can notice from the figure, starting from the threshold individuated before, by using TCP over PiggyCode the number of forwarded TCP-ACKs drops dramatically, resulting less than 1/5 of the number of forwarded TCP-ACKs obtained in the case standard TCP is used over legacy 802.11 MAC. This partially justifies the performance improvement we observed in Fig. 3. In fact, another important aspect that is worth to mention is that, by reducing the number of TCP-ACK transmissions, the MAC-level not only saves the MAC frame transmission related to each TCP-ACK, but also the correspondent MAC-level acknowledgement required by the 802.11 DCF two-way handshake rules.

Thus, at the MAC layer, a double advantage is achieved in terms of overall number transmissions. Definitively, this approach results beneficial to the overall network performance. In fact, coding the TCP-DATA and TCP-ACK packets in a single transmission, significantly reduces the congestion of the portion of the network which was previously interested by the TCP-ACK, and thus the overall interference of the network. This is an important point, since nodes adopting some rate adaption algorithm (for example the ARF algorithm) would definitively beneficiate of the improved link capacity, thus increasing their transmission rate.

B. Impact of the Buffer

To complete our performance analysis, it is worth to analyze the impact of the buffer size on the achieved throughput. As it is known, the buffer size physically limits the amount of storable packets at the nodes. In the case of PiggyCode, this may cause a significant reduction in the number of coding opportunities. Fig. 5 reports PiggyCode throughput performance in terms of the buffer size at each node, when a FTP file transfer is activated between nodes A and B. From the figure, we notice that, when the buffer size is able to contain only a limited number of packets, PiggyCode gains result strongly penalized because of the lack of an adequate number of coding opportunities. Conversely, as the buffer capacity increases, the number of coding opportunities gets higher and higher and then the TCP throughput rapidly increases too. However, as soon as the buffer size becomes too high and goes over the capability of the TCP sender to fill it entirely, its impact on the performance becomes irrelevant and thus the throughput remains constant.

C. Delay Performance: TCP-ACK Acceleration

The use PiggyCode has another beneficial effect since the TCP-ACK delivery process is significantly accelerated.
made valiant effort to propose various methods to make TCP survive in such a challenging environments. However, the large majority of these solutions operate and apply modifications at the TCP itself. Our scheme propose a different, and complementary, approach. In fact, PiggyCode operates at a different layer and independently from the TCP version used. It does not directly modify the TCP policies, and, for this reason, it can be deployed with the standard TCP as well as all "enhanced TCP versions. PiggyCode interacts with the routing in order to maximize the network coding opportunities. Moreover, coding together TCP-ACK and TCP-DATA packets significantly contributes to reduce the overall usage of the network and, specifically, improve TCP performance in the wireless networks. PiggyCode has been shown to achieve very interesting performance gains on the generic grid topology we presented, reaching a gain up to 16% in terms of throughput, and reducing by a factor up to 10 the average time required to deliver TCP-ACK packets with respect to legacy MAC IEEE 802.11. This represents an encouraging step towards a complete understanding of ad hoc designed network coding-based schemes.

We are currently working on PiggyCode extensions able to consider the presence different competing traffic flows. This is the scenario we actually aim at, where we want to definitively test the effectiveness of the proposed coding approach.

VI. CONCLUSIONS

TCP was originally designed for wired networks, where buffer overflows represent the main cause for packet losses. However, in multihop ad hoc wireless networks, several other inherent factors contribute to the TCP performance deterioration. These includes, unpredictable channel errors, medium access contention complicated by hidden/exposed terminal problems, and frequent route breakage. Many researchers have

---

**REFERENCES**


