# Flow Synchronization for Network Coding

Thorsten Biermann, Martin Dräxler, and Holger Karl
University of Paderborn, Germany
{thorsten.biermann, martin.draexler, holger.karl}@uni-paderborn.de

***Abstract*— Network Coding (NC) is a means to improve network performance in various ways. Most evaluations so far were done with simplified assumptions about the application scenario, namely equal data rates and packet sizes for traffic to be encoded. Traffic in real networks, however, does not have this property. Hence, as deterministic and random NC require these properties, flows have to be synchronized prior to encoding to guarantee these properties and to be able to benefit from NC in real networks.**

**In this paper, we present a set of algorithms that synchronize arbitrary flows in wired and wireless scenarios for joint encoding later on. These algorithms are based on fragmentation and Active Queue Management (AQM) techniques. To demonstrate the benefits of our approach, we developed an encoder and decoder for deterministic XOR NC that uses this synchronization technique.**

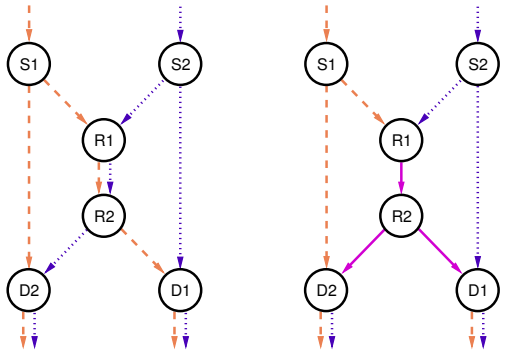
**Simulation results show that with our synchronization techniques, NC, even in scenarios with bursty, self-similar traffic where NC could not have been deployed so far, increases throughput and lowers packet loss and variance of end-to-end delay compared to plain forwarding.**

***Index Terms*— network coding, deterministic, random, inter-session, flow, packet, synchronization**

## I. Introduction

Network Coding (NC) is a technique where data packets are not just forwarded on the path from the source to the destination but are also mixed with packets of other flows [1]. Various techniques that implement this idea have been developed and evaluated in recent years [2]. These evaluations were mainly done in theory and pointed out the high potential of NC to increase the transmission quality in many different scenarios. These analyses are mainly based on highly simplified assumptions about the traffic that is encoded. Characteristics of real traffic, like different data or packet rates, differ packet sizes, or burstiness of the data flows, are not considered. Dealing with these issues is, however, a mandatory issue to benefit from NC techniques in the real world.

In this paper, we focus on deterministic, linear NC where packets of (usually) two unicast or multicast flows are jointly encoded. Unlike plain forwarding, linear NC fully utilizes the maximum flow from a source to the destinations in multicast transmission [3]. A simple form of linear NC is calculating the exclusive OR (XOR) of each packet pair. This transformation is simple to implement and can be applied in a variety of topologies [4]; the probably most famous of them is the *butterfly topology* (Figure 1). Although we will focus on this topology in the course of this paper, our techniques can simply be applied in other topologies suitable for linear NC [4]. They all have in common that two plain flows are multicast at a certain node and that these plain flows are jointly encoded at a node that receives both of them. Finally, the encoded flow is decoded again where both the encoded and one of the two plain flows are received.




(a) NC is not active. Both flows share the same bottleneck R1→R2.

(b) NC is active. The high bottleneck load on R1→R2 is diminished.

Figure 1. Basic data flow in the butterfly topology. Two multicast flows (dashed/dotted) are deterministically encoded at node R1, thus saving one time slot on the link R1→R2.

Data flows that traverse the NC topology, like the dashed and dotted flows in Figure 1, can be of different types. They could be single Transmission Control Protocol (TCP) streams, label-switched paths, or multiple smaller flows that jointly traverse a part of the network over the same links and, hence, can be aggregated and treated as a single flow. The small flows even do not need to be of the same type. The only requirement is the common path through the network topology. In the rest of the paper, the term *flow* is used as a synonym for any of these concrete types.

Two flows that arrive at the bottleneck router R1 usually have different properties. I.e., their packet sizes are not equal and their data rates differ as well. Without additional effort, packets of these flows can only be fed into the encoder as they arrive (Figure 2).

Deterministic and random NC schemes require equally-sized packet pairs as input. Furthermore, encoders cannot handle flows with differing data rates because this means that packets of the low-rate flow are missing for encoding packets of the high-rate flow.

The research leading to these results has received funding from the European Community's Seventh Framework Programme (FP7/2007-2013) under grant agreement n° 216041.

Manuscript received April 30, 2009; revised June 14, 2009; accepted September 03, 2009.


doi:10.4304/jcm.4.11.873-884

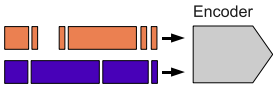



Figure 2. Packets of both input flows arrive at different rates and have different packet lengths at the encoder. As deterministic and random NC require packet pairs of equal size, effective encoding is impossible.

To solve these problems, we introduce the notion of *flow synchronization*. A flow synchronization unit transforms two (or more) input data flows into corresponding flows with equal packet size and data rate. This is achieved by combining buffering, fragmentation, and queue management techniques.

In this paper, we are going to answer the question whether flow synchronization can enable NC techniques to deal with real-world data flows. To achieve this, we make the following contributions:

- We introduce a new way of identifying packets. Instead of using sequence numbers, our scheme is based on hashing. The main advantage is a lower overhead and a simpler decoder implementation compared to conventional sequence numbering. This scheme will be discussed in Section III-A.
- We designed algorithms for synchronizing, encoding, and decoding arbitrary packet flows. These algorithms introduce an adjustable trade-off between latency and coding gain. They work independently of the underlying network topology and transmission technique (wired/wireless) and can be integrated at any position in the protocol stack. More information will be given in Section III-B.
- We evaluated the proposed mechanisms under realistic traffic scenarios, including self-similar, long-range-dependent flows, representing flows with high burstiness. This property is especially challenging because bursts result in high, temporary data rate differences that complicate successful encoding. The results in Section IV show that NC can still be beneficial in such real-world scenarios.

The problem of finding suitable topologies and flows that benefit from NC is not this paper's focus. We always assume that two flows are present in a butterfly topology that allow NC. Mechanisms for finding such scenarios will be discussed in Section II.

## II. Related work

The problem of jointly encoding multiple flows, which we focus on in this paper, is also known as *inter-session NC*. It can be divided into the following basic sub problems: finding suitable topologies/flows for encoding and handling real-world traffic for encoding.

Linear NC can only be applied in network topologies that fulfill certain requirements, i.e., where data flows pass nodes which are interconnected according to certain rules. The most famous topology that allows this kind of NC is the butterfly topology [1]. It has been shown that the butterfly can be generalized to support pair-wise linear NC in many more scenarios [4] and that such topologies can be found in a distributed manner [5]. Furthermore, relay networks that have multicast capability in the downstream, like wireless meshes or Passive Optical Networks (PONs), can be mapped to the butterfly topology [6]. This is illustrated in Figure 3.

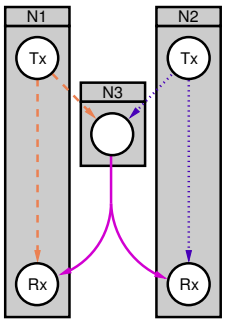



Figure 3. Node N3 acts as relay for N1 and N2. System components (Tx, Rx) within N1, N2, and N3 are regarded as communicating nodes in a butterfly. This way, the coding scheme becomes attractive even for unicast flows as the transfer from Tx to Rx practically comes for free.

When applying and evaluating NC schemes in real networks, components, like encoder and decoder, have to deal with the characteristics of real network traffic. This issue has mostly been neglected in previous studies as they were mainly focused on theoretic analysis or simulations that assumed known flow patterns or even just equal rate flows. Our work ties in at this point and presents techniques for handling arbitrary flows without any a priori knowledge.

A system that deals with real TCP and User Datagram Protocol (UDP) traffic is COPE [7]. COPE is an additional layer that adds linear NC support to wireless networks to increase throughput. This layer is situated between the Internet Protocol (IP) and Medium Access Control (MAC) layer and, like our approach, does not depend on assumptions about certain traffic patterns.

The packet coding algorithm of COPE does not delay packets. I.e., if the medium is idle and there are packets available for sending, they are sent out immediately even if there are no packets of other flows for joint encoding. As we also assume background traffic that cannot be encoded, our synchronization scheme introduces an adjustable trade-off between additional waiting time and achieved coding gain. I.e., if desired, the system can be configured to wait for a certain time before sending out packets to increase the coding benefit.

Incoming packets are only filed into two categories: small and large packets. Packets of each category are only encoded with packets of the same category. This scheme is suboptimal as packets are encoded that have different sizes, i.e., coding is inefficient. In contrast, our flow synchronization scheme always produces packet pairs of equal length which allows a maximum benefit from coding.

Although COPE tries to avoid packet reordering, it cannot be avoided due to its packet selection algorithm.

Therefore, COPE contains a module to ensure in-order packet delivery for TCP packets at the receiver to avoid unnecessary retransmissions due to the reordering process. Our flow synchronization does not introduce packet reordering at all and, hence, seamlessly integrates with any transport protocol.

## III. System Architecture

Our NC architecture consists of four functional components: *packet identification*, *synchronization*, *encoding*, and *decoding*. These tasks are assigned to different nodes in the network. Responsibilities within the butterfly are shown in Figure 4.

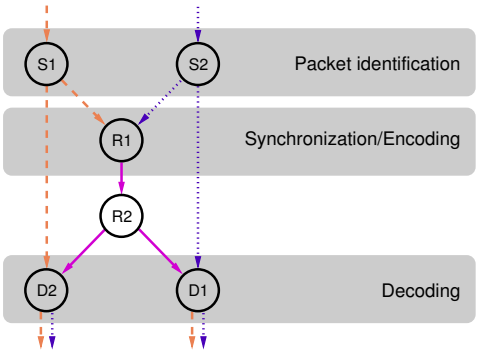



Figure 4. Functional components of the NC architecture and their assignment to nodes within the butterfly.

The different components are discussed in detail in the following subsections.

### A. Packet identification

To be able to correctly decode an encoded packet, the decoder must exactly know which packets from the two flows have been used for encoding, else it cannot pick the right uncoded packet for decoding. The conventional way to achieve this unique packet identification is to augment packets with sequence numbers. Existing sequence numbers within the packets (e.g., from TCP) can usually not be reused as many end-to-end flows might be aggregated and, hence, the sequence numbers cannot uniquely identify a packet.

Adding the additional sequence numbers on top of each packet must be done at S1 and S2 in the butterfly topology as this is the latest point where the plain data flows split up to the encoder and decoder nodes (Figure 4). This technique, however, requires that the *data flows* to which the packets belong are already uniquely identifiable within the network – a feature which is not always given, e.g., when using IP.

For NC it is actually not important to know from which flow a packet originates that has been used for encoding. The only important property for correct decoding is the packet's content. E.g., if there are two identical packets $a$ and $b$ (within a single or multiple flows), and $a$ is used for joint encoding with a packet $c$ as $a \oplus c$, then $b$ can be used for decoding $c$ as well. This insight leads to an alternative to sequence numbering for identifying packets in the context of NC: *Hashing*. Instead of adding sequence numbers at S1/S2 that are used to identify packets at the encoder (R1) and the decoders (D1/D2), a hash digest of the packets' content is calculated at R1 and D1/D2 to identify the packets. This approach has the following advantages compared to the sequence numbering method:

- Functions for identifying data flows and adding sequence numbers to packets at S1/S2 are not required as the packets' content itself is used for identification. S1 and S2 just need multicast capabilities.
- The decoder implementation is simplified. Distinguishing between different flows, e.g., by maintaining different packet buffers, is not required anymore as just the packet content matters.
- Encryption techniques, like Internet Protocol Security (IPsec), can be used together with NC. As the packet and flow identification via certain header fields in the packets to be encoded are not required anymore, encrypting the packets does not matter.

Both sequence numbering and hashing require to carefully choose the length of the included identifier (sequence number/hash). Unnecessarily long identifiers increase the overhead and too short identifiers cause identifier collisions in form of sequence number wrap-arounds and hash collisions. I.e., packets cannot be uniquely identified and, hence, cannot be correctly decoded. These corrupted packets will be discarded by the upper layers' Cyclic Redundancy Check (CRC).

The choice of suitable identifier lengths and the parameters that influence their overhead are obviously important for implementing encoders and decoders. They are evaluated in the following subsections.

*1) Sequence numbers:* When flow information is available in the environment where NC is used, e.g., in label-switched networks, sequence numbering is a simple method for identifying packets within the flows [8]. To keep the overhead caused by this numbering as low as possible, the length of the sequence numbers has to be chosen carefully.

For NC, sequence numbers need to uniquely identify all packets from the point in time $t_\mathrm{E}$ when they are encoded until the point in time $t_\mathrm{D}$ when the resulting encoded packet is not further required for any decoding process. Hence, there must not be any collision among sequence numbers during the time interval $t_\mathrm{C} = t_\mathrm{D} - t_\mathrm{E}$. This interval depends on the duration required for encoding and decoding, the waiting times in all queues, as well as on the propagation time of the encoded packet from the encoding to the decoding node. The number of available sequence numbers must be large enough to identify all packets of a flow during the interval $t_\mathrm{C}$.

We choose sequence numbers from a pool of size $S$, i.e, the bit length of the binary representation is $\log_2(S)$. To calculate $S$, parameters about the data flows to be encoded are required, namely, the peak data rate $R$ and the minimum packet size $P$. Then, $S$ can be calculated according to Equation (1).

$$S(t_C, R, P) = t_C \cdot \frac{R}{P} \quad (1)$$

*2) Hashing:* In scenarios where flows cannot be uniquely identified or where encoder/decoder implementations have to be simple, sequence numbering is not an option. Here, hashing can be used as a powerful alternative. A similar problem as for the sequence number length has to be solved for hashing, too: How to choose the length of a hash which identifies a certain packet?

We assume that the used hash function generates hash values that are uniformly distributed within the set of all possible hash values. The size of this set is $H$, i.e., the binary representation of the hashes has size $\log_2(H)$.

To determine the smallest hash length that avoids collisions, we first calculate the smallest number of hash values $n(p, H)$ such that the expected probability of finding a collision among these $n$ hash values is at least $p$. The resulting Equation (2) is basically a generalization of the famous *birthday problem* [9].

$$n(p, H) = \sqrt{2 \cdot H \cdot \ln\left(\frac{1}{1-p}\right)} \quad (2)$$

As a hash is calculated for each packet within a flow, we are interested in the time $t_C$ during which a hash collision for a flow with data rate $R$ and packet size $P$ occurs with probability less than $p$. This time can be calculated according to Equation (3).

$$t_C = n(p, H) \cdot \frac{P}{R} \quad (3)$$

To directly calculate the required hash length for a set of given parameters describing the scenario ($t_C$, $R$, $P$, and $p$), we solve Equation (3) for $H$ and derive Equation (4). $H$ is the required number of available hash values such that, when hashing each packet of a flow with data rate $R$ and packet size $P$, a hash collision within the time interval $t_C$ occurs with probability at most $p$.

$$H(t_C, R, P, p) = \frac{R^2 \cdot t_C^2}{2 \cdot P^2 \cdot \ln\left(\frac{1}{1-p}\right)} \quad (4)$$

Note that large values for $p$ introduce a packet error floor for the encoded flows. The reason for this is the increased hash collision probability which causes corrupted packets after decoding, which will be dropped.

To give an impression of the required hash and sequence number lengths, Figure 5 shows a plot of $H$ and $S$ depending on $t_C$ for some typical flow data rates. The time to collision is evaluated in the interval $0\,\text{s} < t_C \leq 1\,\text{s}$, which covers most wired and wireless network technologies. The packet size $P = 438\,\text{byte}$ is set to the average packet size in the Internet [10], and for hashing, a collision probability of $p = 10^{-6}$ is tolerated.

The plot shows that for a given data rate $R$ the required hashes need to be approximately 3.5 times longer than

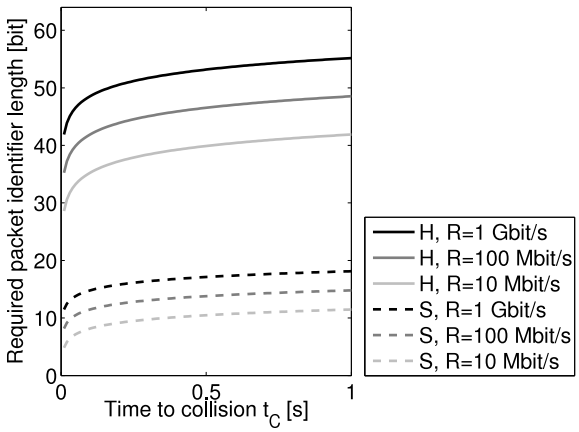



Figure 5. Required packet identifier length for hashing (H) and sequence numbering (S) depending on the required time to collision $t_C$ and data rate $R$; $P = 438\,\text{byte}$, $p = 10^{-6}$.

the required sequence numbers at the same $t_C$. Although this looks like a clear advantage in favor of the sequence numbers, this is usually not the case. As the numbers do not include any information about the flow a packet belongs to, but this information is required for the sequence numbering approach, the flow identification has to be added additionally at the encoder. This is required independently of the fact whether the plain packets already contain a flow identification as this information will not be available at the decoder due to the encoding.

In case of a TCP flow, a 4-tuple consisting of source/destination IP addresses and ports identifies a flow and would require additional $96\,\text{bit}$ on top of the sequence number. Compared to the required hash length of about $55\,\text{bit}$ for an average data rate of $1\,\text{Gbit/s}$ and $1\,\text{s}$ time to collision with a probability of $10^{-6}$ (Figure 5), hashing clearly outperforms the traditional sequence numbering. Figure 6 illustrates this for two TCP packets.

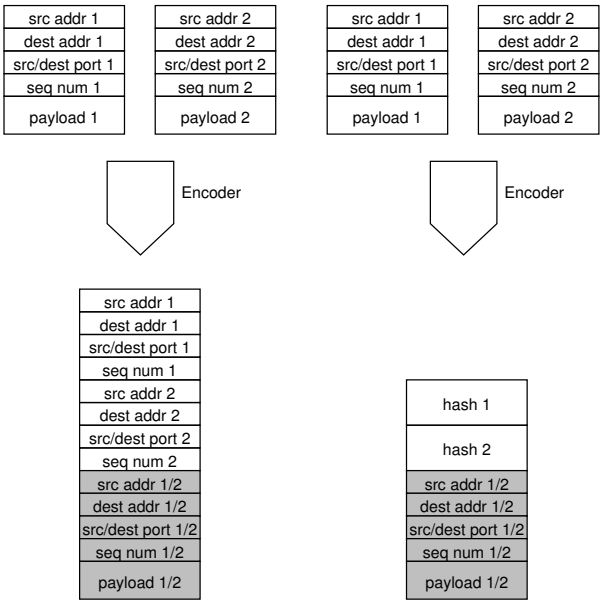
The resulting advantage of hashing is shown in Figure 7. This plot shows how the expected time to collision $t_C$ depends on the full identifier length for hashing and sequence numbering when two TCP flows are jointly encoded. The full identifier includes packet identifiers (sequence numbers or hash values) for both techniques and an additional flow identification for the sequence numbering, consisting of the source/destination IP addresses and ports ($96\,\text{bit}$ in total).

Implementing the proposed hashing scheme requires hash functions that support arbitrary digest lengths to adapt the packet identifier to the flow properties. Examples for families of such hash functions are HAIFA [11] or LAKE [12].

### B. Synchronization

For simplicity, we only discuss the synchronization of two flows in the following. Our methods easily extend to more than two flows.

*1) Overview:* To synchronize two flows, two separate tasks have to be fulfilled: *packet rate synchronization* and




(a) Sequence numbering (b) Hashing

Figure 6. Comparison of packet identification for two TCP packets using sequence numbering and hashing. Gray fields are encoded and, hence, cannot be used for identification. Note that TCP/IP header fields that are not relevant for packet identification have been left out.

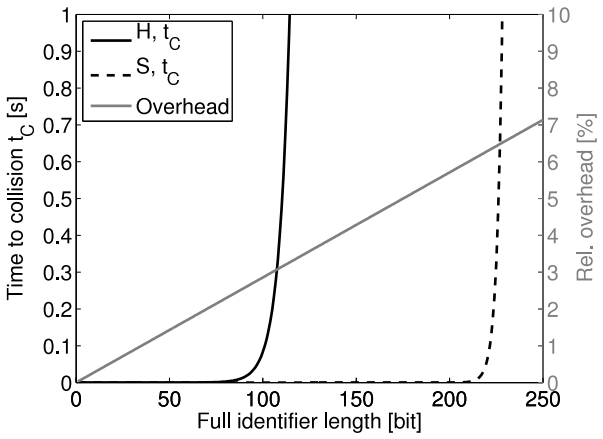



Figure 7. Time to collision $t_C$ depending on used identifier length for two TCP flows being jointly encoded. The identifier includes the flow and packet identification for sequence numbering (S) and, as flow identification is not required anymore, just the packet identification for hashing (H). $R = 1\,\text{Gbit/s}$, $P = 438\,\text{byte}$, $p = 10^{-6}$.

*packet size synchronization*. Packet rate synchronization enables the NC system to encode two flows with different (instantaneous or mean) packet or data rates. During the packet size synchronization, packets are fragmented or aggregated to create packets of the same size. Fragmentation is necessary at this point as the additional header carrying the packet identifiers might increase the encoded packet's size beyond the outgoing interface's Maximum Transfer Unit (MTU).

All packets that are produced by this synchronization process are fed into the encoder. Figure 8 shows how these components interact.

For two flows that are going to be synchronized, two

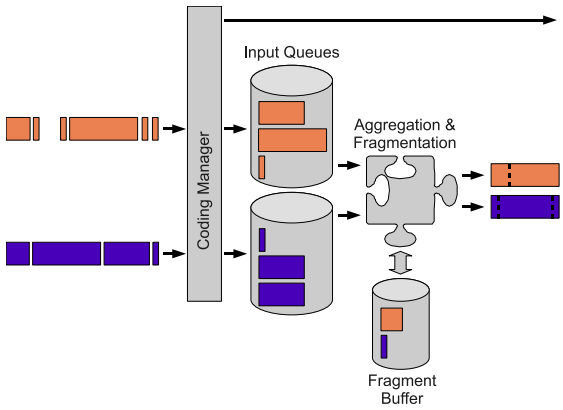



Figure 8. Components of the flow synchronizer. If the coding manager decides that coding has to be applied, e.g., based on the output link utilization, incoming packets are stored in the appropriate input queue before they are passed to the aggregation/fragmentation unit. This unit produces packet pairs of equal size.

corresponding input queues are instantiated. These queues store all incoming packets of their associated flows. There are two possibilities for a packet to be dequeued again: both input queues' fill levels have reached the coding byte length $l_B$ or its maximum Time To Live (TTL) within the queue $t_{TTL}$ has elapsed. These two parameters determine the behavior and the output of the synchronizer.

Parameter $l_B$ defines the desired output packet size after synchronization. This size should be chosen according to (1) the additional overhead introduced by the encoder, e.g., for packet identification as discussed in the previous section, and (2) the MTU of the output network interface. The optimal value to avoid fragmentation at the output interface would be the interface's MTU minus the overhead added by the encoder.

The second parameter $t_{TTL}$ is required to support encoding flows with different mean data rates. Furthermore, it avoids high packet delays in situations where packets from the partner flow are missing and the required fill level $l_B$ is not reached. In this case, available packets in the queues are dequeued by the aggregation/fragmentation unit for immediate processing.

There will be situations where coding is not required or performs even worse than plain forwarding. E.g., XORing two flows even if the output link is operated at very low utilization usually does not make sense. In such situations, the coding manager (Figure 8) can decide to bypass the whole synchronization and coding process.

Another special handling for an arriving packet is required when the packet's appropriate input queue is full. As packet loss before the bottleneck, i.e., caused by the encoding process, should be avoided in any case, packets should be sent out uncoded in this situation instead of dropping them before the encoder. There are multiple options to achieve this. The simplest one that avoids packet reordering is to dequeue head packets of the queue such that the arriving packet fits into the queue; the dequeued packets bypass the encoding process and are forwarded uncoded.

*2) Packet processing in detail:* Whenever a packet arrives at the synchronization/encoding node, the function ENCODERRECEIVEPACKET is executed with the received packet as parameter. This function is shown in Algorithm 1.

**Algorithm 1** ENCODERRECEIVEPACKET($pkt$)

```
fid ← EXTRACTFLOWID(pkt)
q ← GETINPUTQUEUE(fid)
// simple coding manager
if LENGTH(q) + LENGTH(pkt) > CAPACITY(q) then
   FORWARD(pkt)
else
   ENQUEUE(q, pkt)
   STARTTTL(pkt, t_TTL)
end if
// check queue fill levels
doCoding ← TRUE
for each input queue q do
   if LENGTH(q) ≥ l_B then
      doCoding ← doCoding & TRUE
   else
      doCoding ← doCoding & FALSE
   end if
end for
if doCoding then
   // trigger synchronization and encoding
   PREPARECODING()
end if
```

The first steps in ENCODERRECEIVEPACKET are to determine the flow identifier and the corresponding input queue for the incoming packet. Now, depending on the queue length, the packet is either forwarded uncoded or it is enqueued for later encoding (cp. *Coding Manager* in Figure 8). Thereafter, the input queue lengths are checked. If all lengths are larger than the required fill level for encoding $l_B$, the encoding process is triggered by calling ENCODERPREPARECODING. This method is also invoked in case the timeout $t_{TTL}$ has elapsed for any packet in the queues.

The method ENCODERPREPARECODING performs the actual flow synchronization and passes two packets of equal size to the encoder. This is done by choosing "data chunks" of both input flows from the fragment buffer and from the input queues. The selection of chunks is done in the following order until the total size of all chosen parts is exactly $l_B$:

1) Complete fragments (from fragment buffer)
2) Fragment of fragment (from fragment buffer)
3) Complete packet (from input queue)
4) Fragment of packet (from input queue)

This order has the advantage that packets are not delayed unnecessarily and packets are only fragmented if required.

ENCODERPREPARECODING is shown in detail in Algorithm 2. Note that the pseudo code assumes that all queues and buffers are filled. Handling of special cases is omitted.

**Algorithm 2** ENCODERPREPARECODING()

```
for each input queue q do
   v ← {} // vector for selected data chunks
   l_v ← 0 // byte length of v
   f ← GETFRAGMENTQUEUE(q) // fragment buffer
   // step 1: check for complete fragments
   while l_v + LENGTH(HEAD(f)) ≤ l_B do
      frag ← DEQUEUE(f)
      APPEND(v, frag)
      l_v ← l_v + LENGTH(frag)
   end while
   // step 2: check for fragment of fragment
   frag ← DEQUEUE(f)
   l_rem ← l_B − l_v // remaining byte length
   frag1 ← CUT(frag, 0, l_rem)
   frag2 ← CUT(frag, l_rem, LENGTH(frag) − l_rem)
   APPEND(v, frag1)
   ENQUEUEFRONT(f, frag2)
   // step 3: check for complete packets
   while l_v + LENGTH(HEAD(q)) ≤ l_B do
      pkt ← DEQUEUE(q)
      STOPTTL(pkt)
      APPEND(v, pkt)
      l_v ← l_v + LENGTH(pkt)
   end while
   // step 4: check for fragment of packet
   pkt ← DEQUEUE(q)
   STOPTTL(pkt)
   l_rem ← l_B − l_v // remaining byte length
   frag1 ← CUT(frag, 0, l_rem)
   frag2 ← CUT(frag, l_rem, LENGTH(pkt) − l_rem)
   APPEND(v, frag1)
   ENQUEUE(f, frag2)
   // pass collected chunks to encoder
   SENDTOENCODER(v)
end for
```

### C. Encoding

The actual encoder module is straightforward. It receives packets from the synchronizer and encodes these packets, i.e., calculates their XOR value in our case. This operation is illustrated in Figure 9.

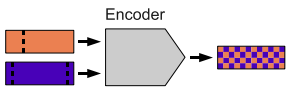



Figure 9. Two input packets are jointly encoded. The resulting packet contains the XORed content and the (hash) identifiers of the packets used for encoding.

In case the processing of the input queues at the synchronizer has been triggered by an elapsed TTL, both packets to be encoded can have different sizes as all packets in the queues have simply been flushed. The encoder handles such different packets by filling the shorter packet with zeros prior to encoding.

In addition to the encoded payload, an additional header is added to the output packet that contains information about the packets and fragments used for encoding. This header will be discussed in detail in Section III-E.

### D. Decoding

The decoder module requires slightly more functionality than the encoder as it must be able to undo the fragmentation introduced by the synchronizer.

To deliver both flows in the original form, both the uncoded and coded flow have to be buffered. The resulting data flow is illustrated in Figure 10.

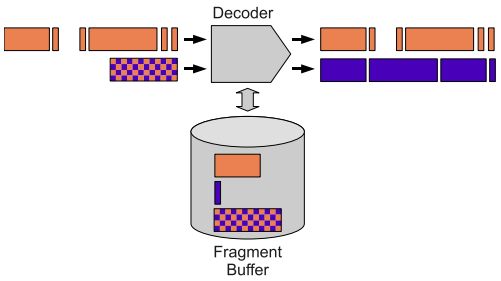



Figure 10. Incoming encoded packets and plain packets required for decoding are buffered and decoded whenever possible. The buffer also contains decoded fragments that cannot be fully assembled at this time.

Each arriving packet is inspected by the decoder to determine whether this packet is required for the decoding process or not. If not, i.e., it is not encoded and is not a packet from a flow used for encoding, it is forwarded to the upper layer. Otherwise, the packet is stored in the fragment buffer. Uncoded packets that *have* been used for encoding are not only stored in the buffer but are also immediately forwarded to the upper layer.

The decoder checks for decoding possibilities when a new fragment is added to the buffer. If there are enough fragments to decode an encoded packet, the actual decoding operation is triggered and the fragments which will not be required anymore for further decoding are removed from the buffer. Furthermore, if the result of a decoding operation just delivers a fragment of a plain packet, this fragment is also stored in the fragment buffer until the whole decoded packet can be delivered. The basic decoding process is given in Algorithm 3.

**Algorithm 3** DECODERRECEIVEPACKET($pkt$)

```
STOREINBUFFER(pkt)
// try to decode packets
for each encoded pkt_E in buffer do
    DECODEPARTIAL(pkt_E)
    // decoded fragments will be stored in buffer, too
end for
// try to assemble decoded fragments in buffer
ASSEMBLEFRAGMENTS()
```

There might be situations where encoded packets cannot be decoded completely because some plain packets required for decoding (*key packets*) are already available and some are not. In this case, decoding just the part that already *can* be decoded at this time might be beneficial to reduce delay. We call this *partial decoding* as depicted in Algorithm 4.

**Algorithm 4** DECODEPARTIAL($pkt_E$)

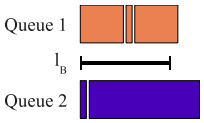
```
for each packet pkt_O encoded in pkt_E do
    // check for required key packets
    pkt_key[ ] ← GETKEYPACKETSFROMBUFFER(pkt_O)
    if all key packets are present then
        // do actual decoding
        pkt_D ← DECODE(pkt_O, pkt_key[ ])
        if pkt_D is a fragment then
            STOREINBUFFER(pkt_D)
            // try to assemble decoded fragments in buffer
            ASSEMBLEFRAGMENTS()
        else
            SENDTOUPPERLAYER(pkt_D)
        end if
    end if
end for
```

### E. A packet life example

To illustrate the interaction of the proposed mechanisms, this section steps through the life of some packets that pass the synchronization and encoding process.

All figures in this section show a graphical and textual representation of either the encoder node's input queue or the resulting encoded packets, respectively. Additionally, the configured coding byte length $l_B$ is drawn next to the graphical representations.

At the beginning, all input queues and the fragment buffer are empty. Thereafter, five packets arrive at the encoding node; three of them belonging to the first flow and two to the second flow to be encoded. All of them are enqueued to the appropriate input queues. This situation in depicted in Figure 11; it happens before any TTL timeout expires. The queues grow from left to right.

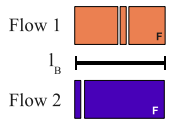



| *flow id* | *size* | *hash* |
|---|---|---|
| 1 | 572 | 925d7518 |
| 1 | 40 | 382e6ace |
| 1 | 572 | 511e91fb |
| 2 | 40 | fe3f401c |
| 2 | 1496 | f4763667 |

Figure 11. Input queues after first packet arrival.

Both queue fill levels exceed the required length of $l_B$ to start encoding. Thus, the synchronization process is triggered to be able to produce an encoded packet. The resulting synchronized packets are shown in Figure 12.

The textual representation of the encoded packet contains two offsets. The first, $offset_E$, denotes the end of a packet/fragment within the encoded packet. It allows the decoder to exactly identify the position of each encoded packet for decoding. The second, $offset_O$, has similar semantics but describes from which part of an original packet a given fragment was taken. Furthermore, $offset_O$ plays an important role at the decoder: A value of 0 signals that the packet has not been fragmented and can be decoded immediately. If the value is negative, it marks a fragment as the last one of the corresponding original packet. Hence, the decoder is able to determine for a received fragment whether it has already received all remaining fragments of the original packet or not.




| *flow id* 1 | | |
|---|---|---|
| $offset_E$ | *hash* | $offset_O$ |
| 572 | 925d7518 | 0 |
| 612 | 382e6ace | 0 |
| 1096 | 511e91fb | 484 |
| *flow id* 2 | | |
| $offset_E$ | *hash* | $offset_O$ |
| 40 | fe3f401c | 0 |
| 1096 | f4763667 | 1056 |

Figure 12. Synchronized packets that will be used for the first encoded packet. The information contained in the table is also added to the encoded packet's header to permit later decoding.

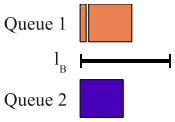
Note that, thanks to the partial decoding algorithm, the decoder for flow 2 will be able to decode packet fe3f401c by only using the key packet 925d7518 of flow 1.

In addition to sending out the encoded packet, the encoder stores the remaining fragments of the packets that have been cut in the fragment buffer. The buffer's content is given in Figure 13.


| *flow id* | *size* | *hash* |
|---|---|---|
| 1 | 88 | 511e91fb |
| 2 | 440 | f4763667 |

Figure 13. Contents of fragment buffer after synchronization.

Now, assume that the packet arrival rates for both flows decrease and the encoder only receives one additional packet for each flow before $t_{\mathrm{TTL}}$ expires. The resulting input queues are shown in Figure 14.

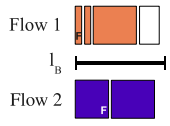



| *flow id* | *size* | *hash* |
|---|---|---|
| 1 | 40 | 7ff13585 |
| 1 | 572 | c2b7dae3 |
| 2 | 572 | 3b6f16a7 |

Figure 14. Input queues after second packet arrival.

As $t_{\mathrm{TTL}}$ has expired, the encoder is forced to trigger the synchronization and coding process although $l_{\mathrm{B}}$ is not reached yet to avoid further delay. Figure 15 depicts the resulting synchronized packets that are encoded.

Note that both fragments 511e91fb and f4763667 are marked with a negative $offset_O$ so the decoder recognises that it has received all fragments for both original packets.




| *flow id* 1 | | |
|---|---|---|
| $offset_E$ | *hash* | $offset_O$ |
| 88 | 511e91fb | -572 |
| 128 | 7ff13585 | 0 |
| 700 | 3b6f16a7 | 0 |
| 1012 | *zero fill* | |
| *flow id* 2 | | |
| $offset_E$ | *hash* | $offset_O$ |
| 440 | f4763667 | -1496 |
| 1012 | 3b6f16a7 | 0 |

Figure 15. Synchronized packets that will be used for the second encoded packet. The missing data for flow 1 is filled with zeros.

## IV. Evaluation

The proposed mechanisms have been evaluated by simulation. The system model and the observed results are discussed in the following subsections.

### A. System model

To evaluate the proposed flow synchronization mechanisms, we implemented our algorithms in a simulator. This simulator is based on OMNeT++ 4.0 [13], a discrete event simulation system written in C++. In addition, we use the INET framework [14], version INETMANET-20080920B, which provides ready-to-use implementations of the IEEE 802.3 Ethernet link layer. All modifications described in the following are done on top of this Ethernet link layer.

*1) Topology:* We use a simple butterfly topology to transport two multicast flows. It consists of seven nodes and is depicted in Figure 16. The data sources have been implemented within a single node (A) for simplification; nodes F and G are the data sinks. All remaining nodes (B, C, D, E) are routers.

The links between nodes are identically parametrized. They are error-free, have a capacity of 10 MBit/s, and add a packet delay of 10 ms.

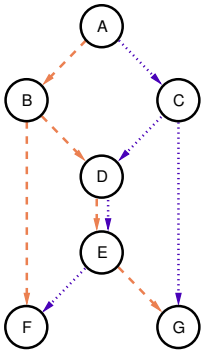



Figure 16. Simulated butterfly topology. Node A sends two multicast flows to the destinations F and G. Encoding will be done at D.

*2) Nodes:* Each node in the scenario contains an Ethernet Link Layer (LL), consisting of an IEEE 802.3 MAC and Link Layer Control (LLC) sublayer. The output queues between the link layers and the Network Interface Cards (NICs) are all drop-tail queues and have a capacity of 4380 byte. This capacity corresponds to 10 packets of the mean packet size, using the traffic model described later on.

Depending on the node type (source, sink, or router), there are several "applications" on top of the LL that perform the desired actions. The source node (A) contains three traffic generators. One for the foreground traffic that is encoded at router D later on and one additional generator for each link's background traffic. Background traffic is always of Self-Similar/Long-Range-Dependent (SSLRD) type, whereas for the foreground traffic different traffic types are evaluated. Both foreground generators are configured to send data at the same mean data rate.

The sink nodes (F, G) instantiate a single decoder application which decodes encoded packets and delivers uncoded packets. This decoder does not reorder arriving packets nor does it check for correctness of received packets. These tasks are delegated to a higher-layer protocol.

Routers (B, C, D, E) contain a routing application that performs routing and coding decisions. The decision how an incoming packet is treated depends on the packet's flow identifier. In this simulation, the identifier is just an integer, contained in each packet. Figure 17 shows a sample routing table of Router D.

```
# XOR-coded transmission
1 000000000600,2,code_xor
2 000000000600,2,code_xor
```

Figure 17. Sample routing table for router D. Flows 1 and 2 are jointly encoded using an XOR encoder. Resulting packets are sent with destination MAC address 00:00:00:00:06:00 (router E) via interface 2.

Like the source nodes, each router contains a background traffic generator for each of its interfaces to represent local traffic.

To get implementation-independent results, the time required for data operations, like encoding or decoding, is not taken into account during the simulation.

*3) Packet preprocessing:* To get an impression about the power of the proposed flow synchronization technique we evaluate two alternative packet processing techniques: *Simple NC* and *Packet Length Matching (PLM) NC*.

*a) Simple NC:* Packets of the data flows to be encoded are not synchronized. Resulting packet pairs (heads of the input queues) are encoded regardless of their lengths. Eventually, length differences are padded with zeros.

*b) PLM NC:* Packets of the data flows to be encoded are not synchronized either. Packet pairs, however, are chosen from the input queues such that their length difference is minimized. Just like flow synchronization, PLM NC uses a TTL timer for packets in input queues to avoid starvation.

*4) Traffic model:* The traffic generators we used in our simulations have basically two parameters: the packet inter-arrival time and the packet size. In the Internet, one can usually find just three characteristic packet sizes – 40 bytes (60 %), 572 bytes (20 %), and 1496 bytes (20 %) [10]. This trimodal distribution is caused by the MTUs of popular transport technologies and TCP acknowledgments.

For the inter-packet times we evaluated three different models: Constant Bit Rate (CBR), Exponential (EXP), and Self-Similar/Long-Range-Dependent (SSLRD) traffic. CBR or EXP traffic is trivial to implement but represents only a minority of data flows in the current Internet, like Voice over IP (VoIP) or video streaming. SSLRD traffic is bursty and models, e.g., aggregated TCP flows. Generating SSLRD traffic is a bit more complex. We used the Fractional Gaussian Noise (FGN) method [15] with exponentially distributed inter-packet times within the batches, Hurst parameter $H = 0.7$, and variance $V = 75$. These parameters are chosen according to [15]. The batch length is set to the duration of sending 100 packets at the desired mean data rate.

The main parameter related to the generated traffic is the *relative pre-coding bottleneck load*. This factor defines the ratio of the overall load passing the bottleneck link to the bottleneck link capacity. Any coding mechanism is disregarded for the calculation, i.e., load reduction on the bottleneck due to coding is not included.

The second parameter concerning the traffic is the fraction of overall traffic that is actually codeable. This parameter heavily determines the achievable gain of NC.

*5) Metrics:* Depending on the application that runs on top of the network, various figures of merit are of interest. They can be categorized into two groups: end-to-end metrics and intermediate metrics. For the end-to-end metrics we only monitor one of the diagonal occurrences, e.g., from A to F or from A to G, as both of them deliver the same results due to symmetry of the topology. We evaluate the following end-to-end metrics during our simulation:

*a) PSR:* The Packet Success Rate (PSR) is the fraction of packets sent by the source (A) that arrives at the destination (F or G).

*b) BSR:* The Byte Success Rate (BSR) is the fraction of bytes sent by the source (A) that arrives at the destination (F or G). It is required in addition to the PSR as the traffic model produces packets of different size. I.e., many small packets can produce a high PSR although the throughput in terms of data volume is low.

*c) EED:* The End-to-End Delay (EED) is the time that elapses from the point in time when a packet is sent by the application at the source (A) until it arrives at the sink at the destination (F or G).

*d) Variance of EED:* The variance of the EED values as described above.

In addition to these metrics, we evaluated the following non-end-to-end metrics to gain further insights into the synchronization and coding mechanisms:

*e) Fragments per plain packet:* This is the average number of fragments a plain packet is split into by the flow synchronization process, i.e., the number of encoded packets a plain packet is spread over.

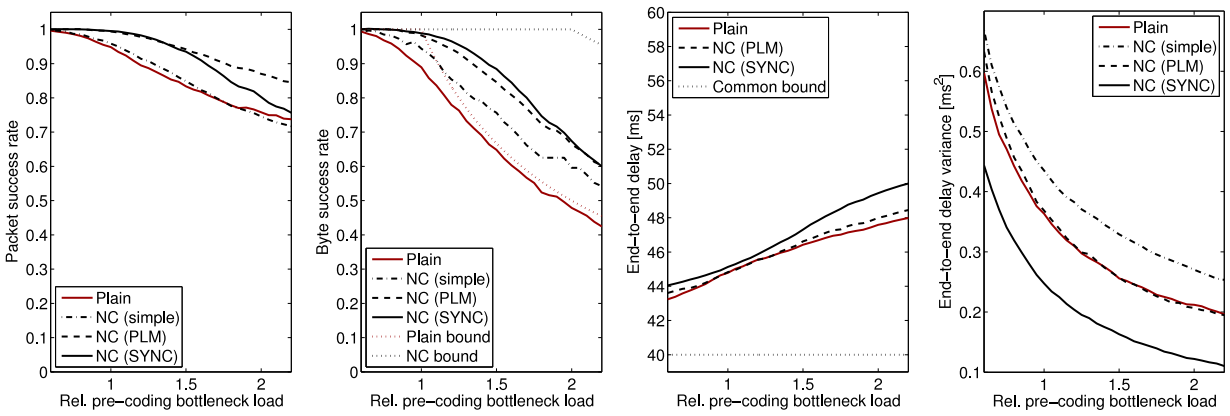
*f) Plain packets per coded packet:* This is the average number of plain packets or fragments of plain packets that are contained in a single encoded packet. Besides the fragmentation behavior, this metric also gives an idea about the additional overhead by the encoding process as packet identifiers for each contained packet are required.

### B. Results

Confidence intervals with a confidence level of 95 % have been calculated for all plots that are shown in the following subsections. They are omitted in the plots due to their small size and for the sake of clarity.

*1) Performance comparison:* To get a first overview of the performance gains that can be achieved with NC using our flow synchronization techniques, Figure 18 shows the comparison of PSR, BSR, EED, and EED variance for plain forwarding, NC with Packet Length Matching (PLM), and NC flow synchronization (SYNC).

The plots clearly show that when data flows are not identical in terms of packet size and rate NC without




(a) PSR (b) BSR (c) EED (d) EED variance

Figure 18. Comparison of plain forwarding, simple NC, NC with PLM, and NC with flow synchronization (FSYNC). For BSR and EED, theoretical bounds are shown additionally. EED for simple NC is omitted due to its high variance. SSLRD traffic, $C = 0.95$, $t_{\mathrm{TTL}} = 1$ ms, $l_{\mathrm{B}} = 1296$ byte.

flow synchronization is no real alternative to plain packet forwarding. Especially the EED varies heavily throughout the evaluated load range and has a mean of 58 ms. In contrast, when using the proposed flow synchronization technique, NC outperforms plain forwarding in terms of PSR, BSR, EED, and EED variance throughout nearly the whole evaluated load range.

The behavior of the NC scheme with PLM differs for the evaluated metrics. While it is even better in terms of PSR compared to flow synchronization, which is caused by the missing packet aggregation before encoding, the EED and its variance is comparable to forwarding. Regarding the BSR, PLM performs slightly worse than flow synchronization. Note that this difference raises for other synchronization parameters, as shown in the next section.

The BSR and EED plots also contain theoretical bounds for NC and plain forwarding. The minimum possible EED is the sum of the links' propagation delays; the maximum possible BSRs have been calculated based on the bottleneck link capacity. Whereas the BSR simulation results of plain forwarding are just below the theoretical optimum, all NC results are clearly below the possible optimum. This is caused by the additional overhead introduced at the encoder and ineffectiveness of the coding schemes.

*2) Influence of synchronization parameters:* The behavior of the flow synchronization process and, hence, the trade-off between the achieved throughput and latency, can be tuned via the two parameters $t_{\mathrm{TTL}}$ and $l_{\mathrm{B}}$. The influence of these synchronization parameters on the three main metrics PSR, BSR, and EED is illustrated in Figure 19.

The results confirm the expected trade-off between throughput (PSR, BSR) and latency (EED). High throughput also causes the EED to raise and is achieved for high values of $t_{\mathrm{TTL}}$ and $l_{\mathrm{B}}$. A large $t_{\mathrm{TTL}}$ causes packets to be collected over a long period of time at the encoder. Hence, $l_{\mathrm{B}}$ is usually the limiting factor that triggers the synchronization and encoding process. Low values for $t_{\mathrm{TTL}}$ cause this process to be triggered before $l_{\mathrm{b}}$ is reached and, hence, cause the encoding to be more inefficient as zero-filling is required.

The second parameter $l_{\mathrm{B}}$ also influences the achieved throughput. The reason is that large values reduce fragmentation during the synchronization process. This, in turn, reduces packet dependencies for successfully decoding an encoded packet as its probability of being spread over multiple packets is reduced.

In sum, high throughput comes at the price of a high packet delay due to buffering as both a high $t_{\mathrm{TTL}}$ and a high $l_{\mathrm{b}}$ cause packets to be buffered for a longer time at the encoder node.

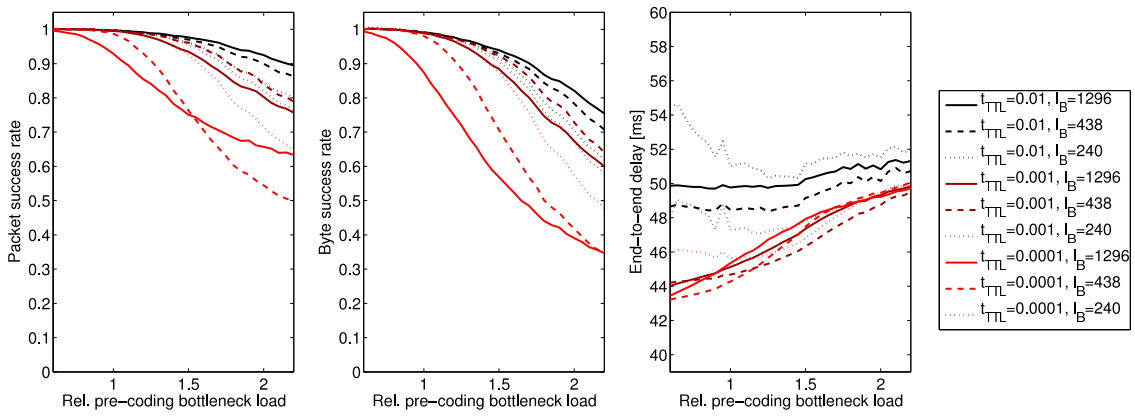
An interesting effect occurs in Figure 19(c). Large values for $t_{\mathrm{TTL}}$ cause the EED variance (and the size of the confidence intervals) to noticeably raise. This is caused by the long duration over which packets are collected at the encoder. All packets being jointly encoded arrive at the same time at the decoder, i.e., bursts are created. This is even worse at low values for $l_{\mathrm{b}}$. Here, the high fragmentation ratio enforces this effect.

*3) Influence of traffic properties:* Another interesting property of NC is the influence of traffic characteristics towards the achieved NC performance. Figure 20 shows the measured values for the PSR, BSR, and EED while varying the traffic type (SSLRD, EXP, CBR) and the fraction of codeable foreground traffic $C$.

According to the expectations, the higher the ratio $C$ of codeable traffic, the higher the benefits of NC. Both metrics PSR and] BSR show a similar qualitative behavior. Concerning the traffic type, NC performs slightly better when confronted with CBR traffic. The measurements for SSLRD and EXP traffic are closely together where SSLRD can be handled slightly better.

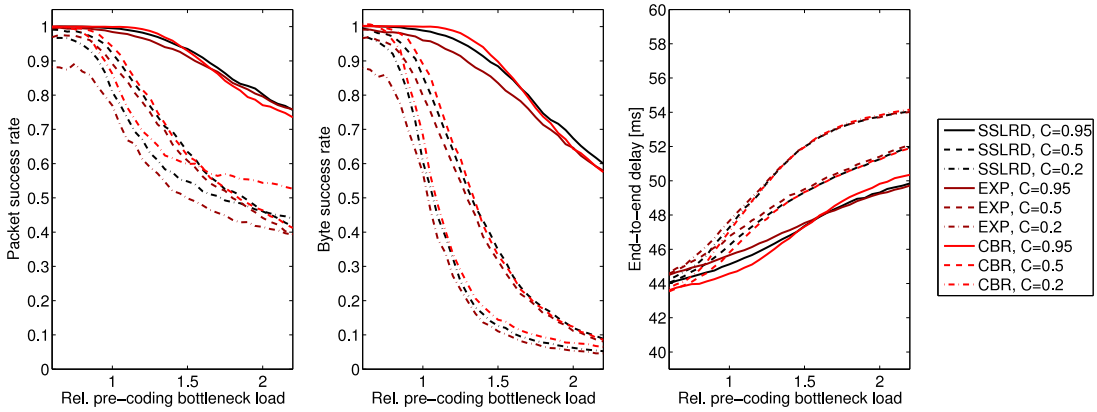
For the EED, the differences between the three traffic types are even smaller than for the success rates. The behavior depending on $C$ can be explained by the higher packet rate for large values of $C$. This, again, results in shorter waiting times for packets in the input queues until the required amount of data is available ($l_{\mathrm{B}}$ is reached).

*4) Packet fragmentation/aggregation:* Our flow synchronization technique applies fragmentation to deal with




(a) PSR (b) BSR (c) EED

Figure 19. Influence of synchronization parameters $t_{\mathrm{TTL}}$ and $l_{\mathrm{B}}$ on the three main metrics PSR, BSR, and EED; SSLRD traffic, $C = 0.95$.




(a) PSR (b) BSR (c) EED

Figure 20. Influence of traffic type (CBR, EXP, SSLRD) and fraction of codeable traffic $C$ on PSR, BSR, and EED; $t_{\mathrm{TTL}} = 1\,\mathrm{ms}$, $l_{\mathrm{B}} = 1296\,\mathrm{byte}$.

arbitrary data flows and diverse transmission technologies. Fragmentation, however, makes packets more vulnerable to packet loss as multiple sub-packets have to be delivered to successfully deliver the original packet. Hence, it is desirable to keep the rate of fragmented packets low.

We measured the average number of fragments that are created from one plain packet during the flow synchronization process. The results are depicted in Figure 21(a).

The plot shows that fragmentation occurs for less than 1 % of the 40 bytes packets and for less than 10 % of the 572 bytes packets. In contrast to this small amount, packets of size 1496 bytes are always split into two fragments due to the simulated MTU of 1496 bytes.

On the other hand, the flow synchronization algorithm aggregates packets to increase NC benefits (cp. Figure 19). The average number of plain packets from both flows that are jointly encoded depending on the synchronization parameters is depicted in Figure 21(b).

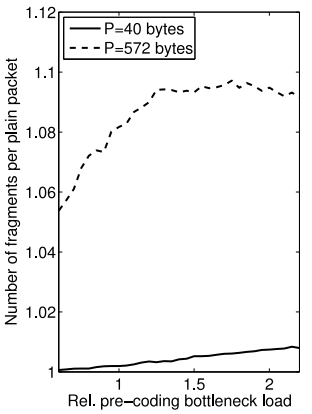
For small values of $t_{\mathrm{TTL}}$ and $l_{\mathrm{B}}$ the number of plain packets per encoded packet is nearly independent of the bottleneck load as already few packets are sufficient to trigger the encoding process. Large values of the two parameters cause the number of packets per encoded packets to grow. Furthermore, the number grows with the amount of bottleneck load as more packets arrive within the time interval $t_{\mathrm{TTL}}$.

## V. Conclusions

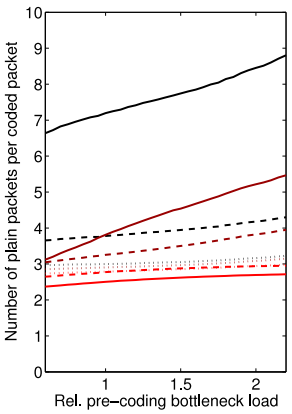
Based on these results, we can conclude that (1) flow synchronization is necessary to benefit from random and deterministic NC in real networks and (2) our approach, presented in this paper, fulfills this synchronization task as expected. We have overcome the restriction that NC benefits can only be achieved for flows with equal and constant bit rates and equal packet sizes.

The evaluation shows that the performance of deterministic, linear NC in terms of PSR, BSR, and EED decreases only insignificantly for EXP and SSLRD traffic with varying rates compared to simple CBR traffic. This makes NC attractive for many new application scenarios, like applying NC in core networks to increase resilience, that were impossible to realize before due to the strict input traffic requirements.

Furthermore, compared to plain forwarding, NC with our flow synchronization clearly performs better in terms of PSR, BSR, and EED variance, while showing comparable EED behavior. There is a trade-off between latency




(a) Average no. of fragments resulting from packets of size $P$. At $P = 1496$ bytes, the fragment count is always 2. $t_{\mathrm{TTL}} = 1$ ms, $l_{\mathrm{B}} = 1296$ byte.




(b) Average no. of plain packets or fragments of plain packets per coded packet. The colors and line styles are identical to those in Figure 19.

Figure 21. Behavior of packet fragmentation and packet aggregation in the evaluated load interval; SSLRD traffic, $C = 0.95$.

and throughput that can be adjusted via the synchronization parameters. This adjustability enables to adapt the NC behavior to the needs of encoded flows.

In the future, we will extend our synchronization scheme to support adjusting the level of packet aggregation. This feature avoids loosing small packets, like TCP acknowledgments, under high load (cp. Figure 18) and avoids retransmission of large data packets caused by lost control packets. In this context, we will also evaluate the influence of NC towards today's transport protocols, like TCP, and their congestion avoidance mechanisms.

## REFERENCES

[1] R. Ahlswede, N. Cai, S.-Y. R. Li, and R. W. Yeung, "Network information flow," *Information Theory, IEEE Transactions on*, vol. 46, pp. 1204–1216, 2000.

[2] C. Fragouli, J.-Y. L. Boudec, and J. Widmer, "Network coding: An instant primer," *SIGCOMM Comput. Commun. Rev.*, vol. 36, no. 1, pp. 63–68, 2006.

[3] S.-Y. R. Li, R. W. Yeung, and N. Cai, "Linear network coding," *Information Theory, IEEE Transactions on*, vol. 49, no. 2, pp. 371–381, Feb. 2003.

[4] C.-C. Wang and N. B. Shroff, "Beyond the butterfly – a graph-theoretic characterization of the feasibility of network coding with two simple unicast sessions," in *Proc. IEEE International Symposium on Information Theory*, June 2007.

[5] ——, "Intersession network coding for two simple multicast sessions," in *45th Allerton Conference on Communication, Control and Computing*, 2007.

[6] T. Biermann, Z. A. Polgar, and H. Karl, "Cooperation and coding framework," in *Proc. International Workshop on the Network of the Future (Future-Net)*, June 2009.

[7] S. Katti, H. Rahul, W. Hu, D. Katabi, M. Médard, and J. Crowcroft, "XORs in the air: Practical wireless network coding," *Networking, IEEE/ACM Transactions on*, vol. 16, no. 3, pp. 497–510, June 2008.

[8] T. Biermann, A. Schwabe, and H. Karl, "WIP: Creating butterflies in the core – a network coding extension for MPLS/RSVP-TE," in *Proc. IFIP/TC6 Networking 2009*, May 2009.

[9] E. H. McKinney, "Generalized birthday problem," *American Mathematical Monthly*, vol. 73, pp. 385–387, 1966.

[10] C. Fraleigh, S. Moon, B. Lyles, C. Cotton, M. Khan, D. Moll, R. Rockell, T. Seely, and S. Diot, "Packet-level traffic measurements from the Sprint IP backbone," *Network, IEEE*, vol. 17, no. 6, pp. 6–16, Nov. 2003.

[11] E. Biham and O. Dunkelman, "A framework for iterative hash functions – HAIFA," in *Second NIST Cryptographic Hash Workshop*, 2006.

[12] J.-P. Aumasson, W. Meier, and R. C. Phan, "The hash function family LAKE," in *Fast Software Encryption: 15th International Workshop, FSE*. Springer-Verlag, Feb. 2008, pp. 36–53.

[13] A. Varga *et al.*, "OMNeT++ discrete event simulation system." [Online]. Available: http://www.omnetpp.org/

[14] INET Community, "INET framework for OMNeT++." [Online]. Available: http://inet.omnetpp.org/

[15] V. Paxson, "Fast, approximate synthesis of fractional gaussian noise for generating self-similar network traffic," *SIGCOMM Comput. Commun. Rev.*, vol. 27, no. 5, pp. 5–18, 1997.

**Thorsten Biermann** is currently a Ph.D. candidate at the Research Group Computer Networks of the University of Paderborn, Paderborn, Germany. He received his Diploma and BS degrees in computer science from the University of Paderborn, in 2008 and 2006, respectively.

His research interests include Future Internet technologies with focus on practical network coding and wireless cooperation techniques. Currently, he is working in the European Union research project 4WARD.

**Martin Dräxler** received his BS degree in computer science from the University of Paderborn, Paderborn, Germany, in 2009.

Currently, he is a student assistant at the Research Group Computer Networks, University of Paderborn.

**Holger Karl** studied computer science in Karlsruhe, Germany, and at the University of Massachusetts, Amherst, USA, and received his Ph.D. degree from the Humboldt University of Berlin, Berlin, Germany, in 1999. He was an Assistant Professor at the Technical University Berlin, Berlin, Germany.

He is a Full Professor of computer science at the University of Paderborn, Paderborn, Germany and head of the Research Group Computer Networks, which concentrates on Internet and wireless communication. His main research interests are architectural questions for future mobile communication systems, cross-layer optimization, and wireless sensor networks.