Mixer: Efficient Many-to-All Broadcast in Dynamic Wireless Mesh Networks

Carsten Herrmann
Networked Embedded Systems Group
TU Dresden, Germany
carsten.herrmann@tu-dresden.de

Fabian Mager
Networked Embedded Systems Group
TU Dresden, Germany
fabian.mager@tu-dresden.de

Marco Zimmerling
Networked Embedded Systems Group
TU Dresden, Germany
marco.zimmerling@tu-dresden.de

ABSTRACT

Many-to-all communication is a prerequisite for many applications and network services, including distributed control and data replication. However, current solutions do not meet the scalability and latency requirements of emerging applications. This paper presents Mixer, a many-to-all broadcast primitive for dynamic wireless mesh networks. Mixer integrates random linear network coding (RLNC) with synchronous transmissions and approaches the order-optimal scaling in the number of messages to be exchanged. To achieve an efficient operation in real networks, we design Mixer in response to the theory of RLNC and the characteristics of physical-layer capture. Our experiments demonstrate, for example, that Mixer outperforms the state of the art by up to 3.8× and provides a reliability greater than 99.99 % even at a node moving speed of 60 km/h.

CCS CONCEPTS

• **Networks → Network protocol design**; **Link-layer protocols**; **Cyber-physical networks**; *Network dynamics*; • **Computer systems organization → Embedded and cyber-physical systems**; *Dependable and fault-tolerant systems and networks*;

KEYWORDS

Wireless mesh networks, Many-to-all broadcast, Random linear network coding (RLNC), Synchronous transmissions, Capture effect, Cyber-Physical Systems (CPS), Industrial Internet of Things (IIoT)

ACM Reference Format:
Carsten Herrmann, Fabian Mager, and Marco Zimmerling. 2018. Mixer: Efficient Many-to-All Broadcast in Dynamic Wireless Mesh Networks. In *The 16th ACM Conference on Embedded Networked Sensor Systems (SenSys '18), November 4–7, 2018, Shenzhen, China.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3274783.3274849

1 INTRODUCTION

Many-to-all broadcast is the process of disseminating information from one, multiple, or all nodes to every node in a network. It is a universal communication primitive as it can serve any possible traffic pattern (point-to-point, one-to-many, all-to-all, etc.), and it is fundamental for a growing number of applications and network services involving multiple sources and multiple destinations.

SenSys '18, November 4–7, 2018, Shenzhen, China

ACM ISBN 978-1-4503-5952-8/18/11.
https://doi.org/10.1145/3274783.3274849

For example, closing distributed feedback loops in cyber-physical systems relies on wireless communication among sensors (sources), actuators (destinations), and controllers (acting as sources and destinations). To enable coordination in autonomous systems, such as collaborative agents [35], robotic materials [9], and swarming drones [56], each node needs to collect information (*e.g.*, location) from every other node and disseminate its own information to all others. Indeed, a certain class of closed-loop control problems is only tractable if each node can make decisions with knowledge of the full system state [5], requiring many-to-all communication.

The same need arises in support of programming abstractions [52] and fault-tolerance mechanisms [60], for example, when some application logic is replicated across multiple distributed devices and nodes need to report every message to all replicas [20]. The initial distribution of messages across sources depends on the application and can also change dynamically at runtime. For instance, in drone-assisted disaster response, all nodes need to regularly exchange one message with all others to prevent collisions or to keep a desired flight formation [7], while sometimes one node may have multiple messages to disseminate, such as an image taken with an on-board camera informing a group of human rescuers on the ground [25].

To support these emerging applications, a many-to-all broadcast primitive needs to meet the following key requirements:

- *Fast and reliable:* To reduce the impact on application performance and to keep up with the dynamics of physical processes, many-to-all communication must be fast (*i.e.*, support end-to-end communication delays and intervals of a few hundred milliseconds [2] or less) and also highly reliable [62].
- *Support for dynamic mesh topologies:* Rotating, flying, or otherwise mobile entities cause significant network dynamics, while multi-hop communication and mesh topologies are either beneficial or a necessity for the application scenario [4, 27, 45, 57].
- *Support for adequate message sizes:* Many applications feature payloads that are tens of bytes in size or larger [25, 45].
- *Energy efficient:* The employed devices are often battery-powered [2, 54] or harvest energy from the environment [9]. Moreover, size and weight constraints call for small batteries, low-power radios, and resource-limited microcontrollers [25, 54].

Existing many-to-all solutions fall short of these requirements. Approaches based on routing, such as WirelessHART, ISA100.11a, and RPL on top of TSCH (6TiSCH), which exchange messages via an explicitly built and maintained structure, target different scenarios with static nodes and packet intervals of several seconds [16]. Using them for many-to-all broadcast may require many-to-one upward routing followed by one-to-all downward routing, which suffers from scalability, efficiency, and reliability issues [29]. Furthermore,

since these solutions rely on a known and stable network topology, they may fail in distributed or uncoordinated settings [23] and perform poorly in the presence of mobile devices [19] or other network dynamics [61]. Some recent proposals based on synchronous transmissions overcome this problem by decoupling the protocol logic from the time-varying network topology. For example, Chaos works well for all-to-all exchange of small payloads (*e.g.*, one byte per node) as required for network-wide consensus [3] and data aggregation [37], but performs inefficiently for payloads larger than a few bytes [47]. A series of network-wide Glossy floods [21] is then a better option; however, the required bandwidth and overall latency increase rapidly with the number of messages to be exchanged.

Contribution and road-map. This paper presents Mixer, a new many-to-all broadcast primitive for dynamic wireless mesh networks. Mixer supports the full spectrum from one-to-all to all-to-all communication, and significantly outperforms prior many-to-all solutions in latency, goodput, and radio-on time while providing nearly perfect reliability despite significant network dynamics.

The key ideas behind Mixer are as follows: (1) Rather than performing M sequential floods to disseminate M messages, Mixer *overlays* all M floods by letting nodes mix packets using *random linear network coding (RLNC)*. This way, Mixer disseminates all M messages at once and approaches the theoretically optimal scaling as M increases. (2) Mixer combines RLNC with *synchronous transmissions*. While RLNC aims to maximize the utility of individual packets, synchronous transmissions aim to maximize spatial reuse.

To exploit the synergy of both concepts for efficient many-to-all communication in real wireless networks, we must tackle a number of challenges as outlined in §2. Our design of Mixer, described in §3, addresses these challenges and yields significant improvements compared to a straightforward combination of the two concepts.

We prototype Mixer on the TelosB (see §4), which has a 802.15.4 radio and a 16-bit MSP430 microcontroller, to allow for a fair comparison with the state of the art on public testbeds. We also port compute-intensive parts of Mixer to modern 32-bit ARM Cortex-M0+/M4 platforms to project the performance gains with more processing power and faster physical layers, such as 802.11.

We evaluate Mixer in §5 using experiments on two testbeds with up to 94 nodes, on dynamic networks with failing devices and a mobile node attached to a car driving 20–60 km/h, and through microbenchmarks on four different platforms. We find that Mixer is up to 3.8× faster and more efficient than fine-tuned sequential Glossy floods and provides a reliability greater than 99.99 % even in the presence of node mobility. For example, Mixer achieves a goodput of up to 53.7 kbit/s and needs less than 300 ms to exchange 27 60-byte messages in an all-to-all fashion on FlockLab [42]. Our microbenchmarks indicate that the same scenario would take less than 10 ms when running Mixer on faster CPUs and physical layers.

In summary, this paper contributes the following:

- Mixer, a many-to-all broadcast primitive that approaches the order-optimal scaling in the number of messages in real dynamic wireless mesh networks.
- A design that combines RLNC with synchronous transmissions and thereby enables Mixer to perform efficiently in practice, while being highly reliable and resilient to network dynamics.
- An open-source implementation and experiments demonstrating several-fold performance gains over the state of the art.

2 OVERVIEW

This section introduces relevant concepts and provides an overview of Mixer's operating principle, scope, and key design challenges.

2.1 Basic Operation and Terminology

The principle behind Mixer's operation is best explained by an analogy with flooding. Assume a set of M messages is to be exchanged between N nodes. Using sequential flooding, this takes $O(M \cdot T)$, where T is the time needed to flood a single message. Although protocols like Glossy [21] achieve the theoretically minimum T in practice, the scaling with factor T becomes problematic as M grows. Mixer improves the scaling to $O(M + T)$ by considering all M messages together: Rather than performing M floods in sequence, Mixer *overlays* the M floods and simultaneously disseminates all messages. To this end, nodes mix packets using RLNC and transmit random linear combinations of previously received packets.

Mathematically speaking, each Mixer node maintains a system of linear equations given in (1).

$$\underbrace{\begin{pmatrix} p_1 \\ \vdots \\ p_M \end{pmatrix}}_{\text{coded payloads}} = \underbrace{\begin{pmatrix} c_{11} & \cdots & c_{1M} \\ \vdots & \ddots & \vdots \\ c_{M1} & \cdots & c_{MM} \end{pmatrix}}_{\text{coding matrix}} \cdot \underbrace{\begin{pmatrix} m_1 \\ \vdots \\ m_M \end{pmatrix}}_{\text{messages}} \tag{1}$$

The set of messages $m_1, \ldots, m_M$ forms a *generation* of size M. Mixer nodes exchange linear combinations of these messages, that is, the ith packet's *payload* equals $p_i = \sum_k c_{ik} m_k$, with $c_i = (c_{i1}, \ldots, c_{iM})$ the *coding vector* of packet i. A sender transmits c_i together with p_i in the same packet. A receiver extracts c_i and p_i from the packet and maintains the *coding matrix* C. When C reaches *full rank* (*i.e.*, a node has collected M linearly independent packets), then (1) has a unique solution and the node can decode all messages by solving the system of linear equations. Sending nodes build packets by adding up a random subset of already collected rows (p_i and c_i) of (1), which is possible irrespective of C's rank. All computations are performed over the finite field GF(2). As a consequence, the size of c_i is M bits (one bit per c_{ik}).

Fig. 1 shows a trace from a 2-hop 802.15.4 network as $N = 5$ nodes exchange $M = 5$ messages in an all-to-all fashion using Mixer; that is, initially each node has its own (one) message, and the goal is that every node acquires the (four) messages from the other nodes. The operation of Mixer, called *round*, proceeds in a series of adjacent *slots*. Node 1, the *initiator*, starts the round by transmitting (Tx) its message in slot 1, which is received (Rx) by nodes 2, 3, and 5. Since the packet contains a message that they did not know before, the rank of their coding matrix C increases from 1 to 2. As a result of this, nodes 2, 3, and 5 may now start to mix packets using RLNC and transmit linear combinations of multiple messages. Note that the use of RLNC allows the nodes to pick the coefficients of c_i randomly without any knowledge of the current network topology.

Every time a node receives an *innovative* packet—one that is linearly independent from all previously received packets—the rank of the coding matrix C increases. Once C has full rank, all messages can be decoded using, for example, Gaussian elimination. In the example of Fig. 1, node 2 is the first to reach full rank in slot 9.

We also see in Fig. 1 that often multiple nodes transmit in the same slot. This happens for the first time in slot 3; however, both

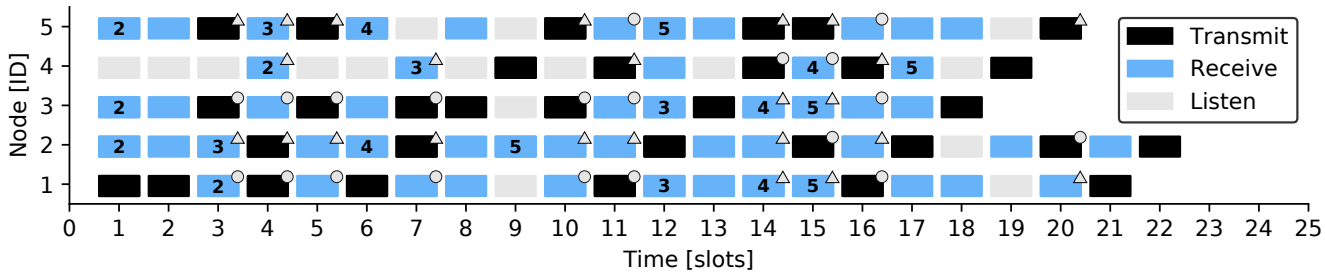


Figure 1: Real trace of Mixer in a 2-hop 802.15.4 network with 5 nodes exchanging 5 messages in an all-to-all fashion. *Numbers indicate when the rank of the coding matrix C at each node increases and are equal to the current rank. Each node can receive at most one packet per slot. Symbols in the upper corner mark which node received what packet in case there are multiple transmitters in the given slot.*

node 1 and node 2 receive despite the collision. While traditional solutions try to *avoid* collisions using carrier sensing, handshaking, or scheduling, Mixer and several other recent proposals (*e.g.*, [21, 58]) aim to *take advantage* of collisions to improve spatial reuse. Since nodes in Mixer typically transmit different packets (as they mix messages randomly), a common receiver can successfully receive one of the packets only due to the *capture effect* [39, 43].

The capture effect occurs if certain signal power and timing conditions are met. For instance, using 802.15.4 radios operating in the 2.4 GHz band with OQPSK modulation, the SINR at the receiver must exceed 3 dB and the packet with the strongest signal must arrive no later than 128 μs after the first packet [65]. Although the exact SINR and timing conditions are highly dependent on the concrete physical layer, the capture effect has already been exploited in many popular wireless technologies, including Bluethooth Low Energy (BLE) [59], 802.11 [38], and 802.15.4 [43].

Scope. Although this paper focuses on low-power embedded systems and 802.15.4, in principle Mixer works on any physical layer that features the capture effect. Moreover, unlike analog network coding [31, 53], Mixer does not require any changes to existing physical layers and hence runs on commodity low-power devices.

Similar to Glossy [21] and other recent works [14, 15, 37], Mixer is a communication primitive that conceptually sits between the physical layer and a higher-layer protocol. This higher-layer protocol is reponsible for informing all N nodes about the (dynamically changing) initial distribution of M messages to nodes before each Mixer round. Alternatively, Mixer can also be used with a statically configured initial distribution. Determining such distributions and building generations of messages is beyond the scope of this paper, but existing techniques can be used [26, 41] and coupled with existing higher-layer protocols (*e.g.*, LWB [19] and A^2 [3]) that readily support the network-wide scheduling of Mixer rounds.

2.2 Design Challenges

Theoretical results suggest that RLNC-based gossip protocols like Mixer perform optimally in static and dynamic networks [12, 24]. Specifically, it has been shown that the number of slots needed to disseminate all M messages has order-optimal scaling $O(M + T)$ [24]. This result is based on specific connectivity measures of the (time-varying) network graph, and the constant factors hidden by the O-notation heavily depend on these properties. In Mixer, the connectivity of the wireless network is tightly coupled to the extent to which the capture effect can be exploited and changes from one slot to the next even if nodes are stationary. Our overarching goal in designing Mixer is to combine RLNC and synchronous transmissions such that the constants hidden in $O(M + T)$ are as small as possible. This entails addressing four main challenges:

C1: *When should a node send or listen?* A capture threshold of 3 dB is quite small, so there is a good chance to benefit from capture in practice. However, because of the very same condition, the probability of capture drops rapidly as the number of synchronous transmitters increases [43]. How can a node locally decide whether it should send or listen in a slot, maximizing spatial reuse without destroying capture?

C2: *What should a node transmit?* To achieve low latency (*i.e.*, small number of slots), we must devise a policy that allows senders to easily build packets that are likely innovative for their neighbors even if the nodes are mobile (*i.e.*, neighborhood changes quickly).

C3: *How to ensure synchronous transmissions in the absence of a global clock?* To meet the timing condition of capture, nodes must communicate in a globally slotted fashion (see Fig. 1). This is difficult because nodes spend varying amounts of time on processing in each slot (*e.g.*, when building the next transmit packet), which impairs synchronization in the face of different clock drifts across nodes.

C4: *How to achieve an efficient runtime operation?* RLNC improves the utilization of the wireless medium (*i.e.*, helps reduce the number of slots), but requires nodes to store and process the coding vector and payload bytes of the packets. Limited memory and compute power may hinder harnessing these benefits (*e.g.*, by blowing up the length of the slots), so we need to design efficient coding and decoding strategies, which are preferably running in parallel to radio activities whenever possible. Moreover, nodes need a way to locally figure out whether they are still helpful for the dissemination process or can turn off their radio to save energy.

2.3 Phases within a Mixer Round

Before describing the design of Mixer in detail, we illustrate its effectiveness in addressing challenges **C1** and **C2**—the *when* and *what* to transmit—via a comparison with a naïve application of RLNC, where every node sends packets at random with a fixed transmit probability p_t and builds packets by randomly summing up already collected rows. Fig. 3 shows the average rank increase across 27 nodes on the FlockLab testbed [42] for the random approach (including our mechanisms to address challenges **C3** and **C4**) and when all design features of Mixer are enabled. Despite the fact that Mixer reduces the number of slots required to reach full rank (27

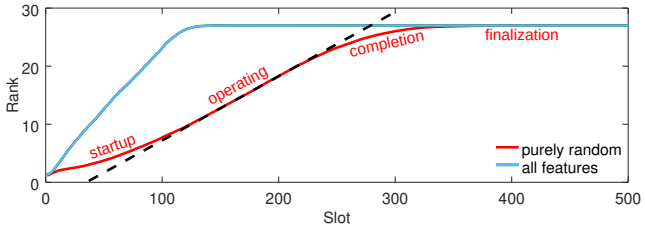


Figure 3: Average rank increase in a one-to-all scenario on FlockLab with different feature sets ($M = 27$ messages).

in this scenario) from 320 to 120, we can distinguish four distinct phases, which are most apparent in the purely random approach.

In the middle we see a phase with almost linear behavior, emphasized by the dashed line. We refer to this as the *operating phase* since the rate of average rank increase is high and steady. Before comes a *startup phase* with less but increasing rank growth. After the operating phase follows a time with decreasing rank improvements. We call this the *completion phase* as it eventually leads to all nodes reaching full rank. Last, there is a *finalization phase* where nodes still communicate but no longer depend on incoming packets.

The poor performance of the random approach, especially during the startup and completion phases, induces questions on the reasons for the observed behavior and potential improvements. Consequently, these questions drive the discussion in §3. As visible in Fig. 3, our design of Mixer improves significantly on the purely random approach, effectively addressing challenges **C1**–**C4**.

3 DESIGN

This section describes the architecture of Mixer and discusses all major design elements. In §3.1 we introduce the core architecture of Mixer, which consists of a transport layer and a processing layer. Since the latter contains the most characteristic features of Mixer, we present it first in §3.2. It combines two core mechanisms (§3.2.1 and §3.2.2) with a number of phase-related features (§3.2.3 to §3.2.5), systematically addressing challenges **C1**, **C2**, and (partially) **C4**. In §3.3 we present important mechanisms of the transport layer targeting challenges **C3** and **C4**. We conclude with a short list of other design properties facilitating **C4** in §3.4.

3.1 Core Architecture

To significantly exploit the capture effect, the design of Mixer incorporates two key ingredients: well-synchronized, time-slotted communication to meet the capture window and a lightweight but effective mechanism to steer the set of active transmitters per time slot. From a single node's perspective, these two design elements are responsible to decide *when* to transmit (and when to receive). Their counterpart is composed of a number of mechanisms influencing *what* to transmit with the particular goal to improve the efficiency of RLNC in the given scenario. Most of the components interact with each other; Mixer represents the entirety of all design elements, translated to an appropriate software architecture.

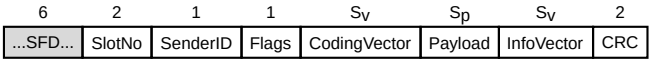


Figure 4: Mixer packet format. *Parts in gray are defined by IEEE 802.15.4, CRC is generated by the radio. Sizes in bytes.*

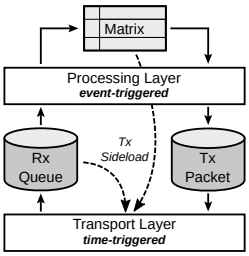
The timing conditions of capture suggest the encapsulation of the low-level packet transport functionality in a self-contained component that provides reliable synchronous packet exchange. Mixer follows this approach with a two-layer architecture composed of a time-triggered *transport layer* and an event-triggered *processing layer* (Fig. 2a). The layers are decoupled by receive and transmit queues, allowing a high degree of parallelized activities on both layers. Fig. 2b shows the main activities on each layer within the slots. Each slot has the same fixed length T_s, which accounts for the air time T_a of one packet and processing time T_p. By default, the transport layer is in receive mode. Transmit decisions are made by the processing layer and passed to the transport layer on demand, which executes them in the next slot.

Fig. 4 shows the packet format. Note that the payload field contains a linear combination of messages, so its size S_p is equal to the message size. The packet size, instead, also accounts for all other fields, such as the size of the coding vector S_v.

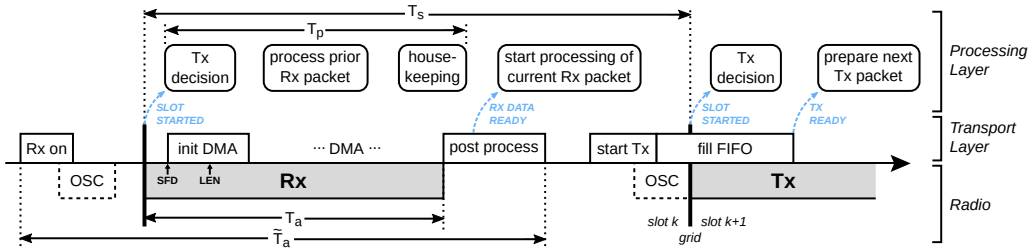
3.2 Processing Layer

In the following subsections we detail the individual design mechanisms of Mixer. The transport layer is considered in §3.3; for the moment we assume it to be working such that we can take slots as the base unit of the communication grid. We start from the baseline of the purely random approach introduced in §2.3. The discussion is driven by the question on how to improve on the observed results, particularly with respect to the identified phases (Fig. 3).

3.2.1 Semi-coordinated Transmissions. Since real-world and in particular dynamic networks have varying local node densities, a fixed transmit probability p_t performs unsatisfactory (Fig. 3). An adequate policy should adapt p_t to the local densities, striving for the goal to maximize the number of received packets per slot within the network. To reach this objective, the number and selection of transmitters in each slot should be well balanced—high enough and



(a) Core architecture.



(b) Radio, transport layer, and processing layer activities within each slot.

Figure 2: Software architecture and interactions between design components in each Mixer node.

Algorithm 1 Semi-coordinated Transmission (core alg.)

```
d = 1 + num_neighbors                    ▷ local density (from history)
owner = (slot_no + 1) mod N              ▷ assign owner to next slot
if owner = my_node_id then               ▷ if my slot: transmit
    p_t = 1
else if owner is neighbor then           ▷ if foreign slot: receive
    p_t = 0
else if Tx in current slot then          ▷ do not transmit twice in a row
    p_t = 0
else                                     ▷ if shared slot:
    p_t = 1 / (d + 1)                    ▷ transmit with probability
end if                                   ▷ (+1 accounts for an unknown neighbor)
```

spatially distributed to reach as many receivers as possible, but still low enough to allow the capture effect to occur.

For this purpose, each Mixer node maintains a list of received SenderIDs (Fig. 4) within the last H slots. Using this sliding-window history information, which is discarded at the end of a round, nodes monitor their current neighborhood to drive an *adaptive transmit policy* as shown in Algorithm 1. This policy updates the transmit probability p_t of a node depending on the estimated local node density d. Furthermore it incorporates a kind of local round-robin scheduling on a selected subset of slots by assigning an owner to each slot. Owners use their slots to transmit for sure while neighbors respect this behavior.

The level of determinism induced by this policy increases with node density because a higher density means that a larger portion out of N consecutive slots is owned by some node within a neighborhood (in return, the number of shared slots is lower). Hence, in high-density regions of the network nodes use stronger coordination than in sparsely populated areas. In this way, we effectively reduce the uncertainty in the expected number of transmitting nodes per slot and decouple the capture probability from the node density (see **C1**). This behavior also implies that Mixer does not purely depend on the capture effect; for example, in a one-hop network, Mixer would tend toward a fully coordinated operation, where nodes transmit one after another in dedicated slots.

3.2.2 Explicit Innovation Forwarding. A sender assembles a packet using RLNC: It adds every row from its matrix to the Tx packet with probability $p_a = 1/2$. In Mixer we add several features somewhat restricting the randomness to improve average performance. The most basic one rests upon the assumption that an innovative packet is also innovative for a node's neighborhood. Thus, a node adds every innovative packet immediately to the prepared Tx packet so the innovation gets relayed with the next transmission for sure.

To thoroughly justify this behavior, we distinguish two cases: If the innovative packet arrives from *outside* the common neighborhood (cluster), the above assumption is clearly reasonable. Otherwise, if the packet is sent from *inside* the cluster, there is still a chance that it is innovative for some neighbor(s). On the other hand, including it has no disadvantage for the other neighbors: Since the packet is innovative for the current node, it is linearly independent from its Tx packet (*i.e.*, including it corresponds to adding an additional matrix row). Note that this behavior influences only the content of the next Tx packet (*what*), not the transmit policy (*when*).

3.2.3 Improving Startup: Adapted Coordination. The reasons for a slow startup phase (Fig. 3) are best understood with a one-to-all scenario in mind. In this case, all messages reside at the initiator at the beginning of a round. After it started the round, the awoken neighbors are not able to add any innovation to the packets they send as they simply do not have any. Innovation can only be added by the initiator if it decides to transmit and if the (randomly built) packets include some. Clearly, the situation will improve (a) if the initiator ensures that it sends something innovative, and (b) if there is a mechanism that prioritizes the transmissions of the initiator at the beginning of a round. Again, (a) and (b) address the *when* and *what* to transmit.

Improving the number of innovative packets during the startup phase is easy: Since every message is initially available at exactly one node, its *originator*, this node knows that a packet will be innovative for all other nodes if it incorporates the message for the first time. Hence, if a node initially has x messages, it can easily generate x innovative packets by simply transmitting each of its x messages once. Mixer nodes do exactly this before starting to build packets at random.

Prioritizing the transmission of innovative packets requires adaptations of the transmit policy since there is no connection to packet contents so far. Mixer bridges this gap by assigning the owner role of slot k to the originator of message k during the first M slots. Thus, in case of a perfect wake-up order of all nodes, every slot $k \leq M$ is used to transmit message k by its originator, generating a fast-growing coding potential. As a side effect, this rule circumvents the question on how the standard transmit policy performs as long as the history information is very incomplete.

However, using this strategy, we have to cope with two issues: First, if the originator of message k is not awake in slot k, the slot is unused. This can lead to corner case situations in which nothing happens for a long time. Mixer avoids this problem by including the shared slot rule (cf. Alg. 1) also during the startup phase, but with $p_t = 1/\min(k, 16)$ which is independent of d and stimulates a fast wake-up of all nodes. Second, in one-to-all scenarios the initiator would transmit in every of the M startup slots and hence would have no chance to discover its neighbors. Further, if its neighbors stay silent the whole time, nodes farther away would not wake up. Mixer avoids this problem with the help of a flag (used only during startup): If a node sending in slot k is also the originator of message $k+1$, it marks this in the packet header's flags field and stays silent in slot $k+1$. Nodes receiving the packet in slot k then know that the owner of slot $k+1$ will not use its slot. Thus, they transmit with $p_t = 1$ in slot $k+1$ to push packets into their "back country."

3.2.4 Improving Completion: Active Requests. In the completion phase we see a significant slowdown in the average rank increase with the purely random approach (Fig. 3) due to the well-known coupon collector's problem [10, 17]. It is present here because a node cannot include messages that are outside the row space of its current matrix. Since there is a high probability that the missing pieces are also missing at a considerable number of neighbors, it is difficult to resolve the situation efficiently without any feedback on the missing pieces. Mixer nodes address this problem by progressively providing such feedback in the form of *active requests* and by adapting their transmit policy in response to these requests.

Recall that a node requires rank M to recover all messages, that is, M linearly independent rows in the coding matrix C in (1). Mixer

nodes keep C in row echelon form. Therefore, rows can be identified with their pivot elements (the main diagonal of C): If c_{ii} is zero, then row i is missing. Below, we describe how Mixer deals with missing rows. Mixer also includes mechanisms to deal with missing columns, which are conceptually similar, but we do not discuss them here due to space limitations. For the same reason, we skip minor details and instead focus on the main concepts.

The transition from the operating to the completion phase is floating and encompassed by a simple rule which provokes an increasing number of active requests when a node's rank tends toward full rank. Every request is communicated in the form of a flag and corresponding markers in the InfoVector field (Fig. 4) of a packet that is anyway being sent; thus, requests do not consume extra packets or slots. If a node receives active requests, it has to decide how to react, which again translates into the questions of *what* and *when* to transmit. Before answering these two questions, we discuss how a node stores requests. Storing them is wise since it may be possible to help multiple nodes with one response packet.

In case of a request, InfoVector contains a bit field wherein each set bit marks one missing row. Nodes could store every received request separately, but this may consume a considerable amount of memory and slow down processing. Instead, Mixer nodes maintain an *any-mask* and an *all-mask*. If a request arrives, they `OR` the bit field to the any-mask and `AND` it to the all-mask. Thus, the all-mask contains bits that have been set in *all* incoming requests, while the any-mask contains bits that have been set in *at least one* request. This way, nodes get an idea of which rows might be more helpful than others. In addition to storing requests, nodes monitor the traffic and try to discover if pending requests got serviced. If not, they drop the stored requests after three slots so they do not affect the communication for a long time.

In case of pending requests, a node decides *what* to transmit as follows: It selects a requested row index from the all-mask or, if the all-mask is empty, from the any-mask. Then it tries to build a packet whose coding vector contains no non-zero elements to the left of the selected index. If this is possible, the packet is definitely innovative for the requesting node. In this case, a node is a *helper*. Otherwise, if a node cannot build such packet, it is a *non-helper* and it does not matter what it sends. Instead, it should consider *not* to send, which brings us to the question on *when* to transmit.

Extending the transmit policy with rules for request handling is nontrivial. We pursue three goals: (a) Potential helpers should send preferred while non-helpers should decrease their transmit probability p_t. (b) NodeID-based owner roles should be more and more dissolved towards the end of a round in favor of helper/non-helper roles. (c) Phases without pending requests should be unaffected. With these goals in mind, choosing p_t breaks down into three tasks:

1) Identify own role: helper or non-helper.
2) Estimate role of all n neighbors respectively the number of helpers n_+ and non-helpers $n_- = n - n_+$.
3) Adapt p_t based on n_+, n_-, and original owner role.

Step 1 is explained above. For step 2 we need some heuristic since it is impossible for a node to determine the required information precisely. In fact, we do not even know if all neighbors process the same request masks, though this appears as a reasonable assumption at least for the majority of neighbors. A simple heuristic could estimate n_+ as a fixed percentage of n. To get more precise estimates, we design a more advanced variant which uses an additional prerequisite: Since the InfoVector field is sent with every packet anyway, the nodes always utilize it to transmit their current row state. Receiving nodes store this information in their history such that every node has a reasonably up-to-date information on the row state of its neighbors. Using this information, a Mixer node is able to estimate n_+ (and hence n_-) more precisely.

Table 1: Transmit probabilities with requests pending.

	Own Slot	Foreign Slot	Shared Slot
p_t Helper	$\tilde{r}/n_+ + (1-\tilde{r})$	$\tilde{r}/n_+$	$1/n_+$
p_t Non-Helper	$\tilde{r}/(\mathrm{e}n_-) + (1-\tilde{r})$	$\tilde{r}/(\mathrm{e}n_-)$	$1/(\mathrm{e}n_-)$

The adaptation of p_t in step 3 is based on the following reasoning. In expectation, one helper should send a response packet. Thus, the transmit probability for helpers $p_{\mathrm{t}+} = 1/n_+$. The transmit probability for non-helpers $p_{\mathrm{t}-}$ should match the probability that no helper sends a packet, that is,

$$p_{\mathrm{t}-} = \frac{\left(1-\frac{1}{n_+}\right)^{n_+}}{n-n_+} \xrightarrow{n_+\to\infty} \frac{\mathrm{e}^{-1}}{n-n_+} = \frac{1}{\mathrm{e}\,n_-} \tag{2}$$

The numerator in (2) converges fast, so it is sufficient to use the simplified term as approximation. Since we do not want to break the original NodeID-based owner roles abruptly, each node lets p_t slide based on its relative rank $\tilde{r} = r/M$, leading to the transmit probabilities listed in Table 1.

3.2.5 Improving Finalization: Smart Shutdown. From a single node's perspective, the main communication task is accomplished when reaching full rank. The remaining slots are needless for that node, but may be useful to support unfinished neighbors. Though, at some point in time all neighbors have full rank and there is no reason to stay active; the node could turn off to save energy. However, we need a prudent signaling mechanism to establish an efficient but cautious shutdown of the nodes.

If a node has full rank, it sets a flag in every packet it sends. Receiving nodes mark the full rank status in their history. A node can shutdown if all its neighbors reached full rank. It informs its neighbors by sending a packet with a shutdown flag set before it turns off. The shutdown flag allows a neighbor to immediately remove that node from its history, leading to an immediate adaptation of its transmit policy. Without the flag, the update would be delayed until the vanished node falls out of the history window H.

Since the packet with the shutdown flag is sent only once, it could easily get lost at *some* neighbors, but it is received with high probability by *at least one* neighbor. We address this issue with two mechanisms: First, on receiving a packet with the full-rank flag set, the history window applied to the sending node gets shortened (we use $H/3$ in §5) based on the expectation that the node may shutdown in the near future. Second, the full-rank status of all nodes is actively propagated through the network with the help of full-rank nodes. Since those do not have the need to provide request information, they use the InfoVector field to mark finished nodes instead. With this mechanism it is very unlikely that a full-rank hint gets lost. Although this does not replace lost shutdown flags, it enables finished nodes to shutdown themselves instead, which eventually leads to the same result.

3.3 Transport Layer

3.3.1 Establishing the Slot Grid. On the packet transport layer we have to solve the task of establishing a reliable slot grid as a prerequisite for synchronous transmissions. Since system parameters are known in advance, it is possible to select a fixed nominal slot length T_s. Unfortunately, real-world hardware suffers from clock frequency offsets, so we need some mechanism that compensates for such effects and avoids that grid points drift apart among nodes. As a natural solution, we implement a phase-locked loop (PLL).

The reference signal of the PLL is built from timestamps taken on the reception of start-of-frame delimiter (SFD) fields, which is part of the synchronization header of each packet (see Fig. 4). This requires that senders start transmissions with appropriate temporal accuracy, which is typically achieved via meticulous timer polling. The phase difference between the reference signal and the local slot grid is low-pass filtered and fed into a proportional-integral (PI) controller that computes a correction term for the next grid point and a start offset for transmissions (the latter counteracts a potential cumulative drift induced by time-of-flight delays). All filter coefficients and gain parameters are chosen empirically based on simulations and results from testbed experiments.

In principle it is possible to consider only SFD timestamps that stem from *specific* neighbors (*e.g.*, predefined ones or those with minimum hop distance to the initiator). Our experiments suggest that taking reference values from arbitrary nodes is sufficient and works most of the time. If it sporadically fails, a node recognizes this situation and resynchronizes itself as part of a fallback mechanism.

We want to emphasize that the slot grid is only needed during a round (*i.e.*, when Mixer is active). Mixer does not require to keep the slot grid between rounds. In fact, each Mixer node except the initiator assumes to be out of sync at the beginning of a round. To lock onto the slot grid, a node activates its receiver continuously until it receives the first valid frame. The maximum length of this initial listen phase is only limited by a timeout for the whole round. This timeout is chosen by the user and can be set to a large value (*e.g.*, in the range of seconds or even infinity), effectively decoupling the timing requirements of Mixer from the rest of the system. At the end of a round, each node returns a reference time, which can be used (*e.g.*, by a higher-level protocol) to schedule the next round.

3.3.2 Updating Packets via On-the-fly Sideload. Immediately forwarding innovation as described in §3.2.2 seems to be easy from a conceptual point of view, but it appears ambitious when looking for an efficient implementation. The reason is a hard time constraint which becomes clear when considering the transition from a receive to a transmit slot (Fig. 2b). To achieve an optimal performance, the slot length T_s should be as small as possible (bounded by T_a plus a minimal overhead for pre- and postprocessing). On the other hand, updating the transmit packet takes time (process received packet, determine if innovative, if so: add) and has to take place (theoretically) between end of Rx and start of Tx.

With Mixer, we introduce a feature that allows to solve this problem in an elegant and efficient way. First, if a node transmits a packet, it always starts the radio *before* filling the radio chip's transmit FIFO. This is possible because the transmitter needs to generate a number of synchronization symbols (despite some device specific tasks, denoted OSC in Fig. 2b) before it sends the actual data, which provides some time for the program. As a result, a processing task can change the packet content until right before the slot starts (irrespective of the packet size). Second, our low-level transmit routines allow to add additional data to the packet content *while* the packet is moved to the FIFO. In other words, they allow to *sideload a second payload into the transmit data stream on-the-fly*. This second payload is incorporated into the transmit data during very short, anyway required waiting periods in the transfer loop and hence incur no extra CPU load—it literally *comes for free*.

With the help of the sideload feature, it is easy to solve the immediate update problem: Every time a packet is received, the Rx routine marks it as the current sideload, so it gets added to the next Tx packet. If it is innovative, the Tx packet carries the innovation. If not, it does not hurt the Tx packet (by neutralizing its coding vector to zero) with high probability. If this happens anyway, the transmission is aborted before it becomes "visible in the air."

Besides innovation forwarding, the sideload feature also simplifies interlocking critical activities between the transport layer and the processing layer. Overall, it proves to be a very useful tool.

3.4 Efficient Runtime Operation

Besides the features described above, we facilitate an efficient runtime operation of Mixer by:

- computing over finite field GF(2), which enables an efficient implementation on standard hardware and needs just one bit per message in the coding vectors;
- keeping the coding matrix in row echelon form, which limits the amount of memory needed for storing packets, reveals useful information for free (*e.g.*, the rank of the matrix), and spreads computational load across multiple slots;
- parallelizing radio and CPU activities, which boosts performance by allowing to reduce the slot length to the minimum.

4 IMPLEMENTATION

We prototype Mixer on TelosB devices running at 4 MHz CPU clock rate. Our implementation[1] comprises about 7900 lines of C code, where 3500 lines account for the hardware abstraction layer.The compiled program has a footprint of 21 KB in flash and 300 bytes in RAM (w/o stack, Rx/Tx queues, matrix, history, and request masks). With payload size S_p, coding vector size $S_v = \lceil M/8 \rceil$, and packet size $S = 12 + 2 \cdot S_v + S_p$ (see Fig. 4), the amount of RAM needed for the variable-sized elements can be approximated[2] as follows:

$$m\,[\text{bytes}] = \underbrace{5 \cdot S}_{\text{queues}} + \underbrace{M \cdot (S_v + S_p + 2)}_{\text{matrix}} + \underbrace{(N+9) \cdot S_v}_{\text{request masks}} + \underbrace{4 \cdot N}_{\text{history}} \quad (3)$$

Using Mixer requires choosing the slot length T_s based on the payload size S_p and the generation size M. To achieve good performance, T_s should be small. However, as visible in Fig. 2b, T_s is lower-bounded by the minimum time needed for low-level packet transport $\tilde{T}_a$ and by the time taken by the processing layer T_p. We profiled our code to derive formulas for both bounds that make it easy to find a reasonable value for the slot length $T_s \geq \max(\tilde{T}_a, T_p)$.

[1] The latest source code of Mixer is available to the public under a BSD license at `https://mixer.nes-lab.org`.

[2] Eqn. (3) is simplified, in particular it ignores padding bytes introduced for alignment purposes and some small internal data elements like flags.

The low-level packet transport time can be expressed as

$$\tilde{T}_a[\mu s] = (440 + 32 \cdot S) \cdot 1.037 \quad (4)$$

which accounts for the packet air time (*i.e.*, 32 μs per byte in IEEE 802.15.4 networks) and a static overhead for basic buffer handling and RF oscillator calibration. The multiplier in (4) matches internal tolerance settings; the chosen values compensate for clock drift of up to 1000 ppm. Further, the processing time reads as

$$T_p[\mu s] = 600 + (26 + 0.155 \cdot (S_v + S_p)) \cdot M + 1.8 \cdot S \quad (5)$$

A value determined using (5) ensures that the processing layer can handle the stream of packets *on average*; temporary overload is compensated by the Rx queue (see Fig. 2).[3]

Fig. 5 plots (4) and (5) against the generation size M for a payload size of 60 bytes. We see that the crossover point is for $M = 75$ messages. For smaller M, Mixer effectively processes packets at line rate, as the slot length is bounded by the packet air time and therefore by the bitrate of the physical layer.

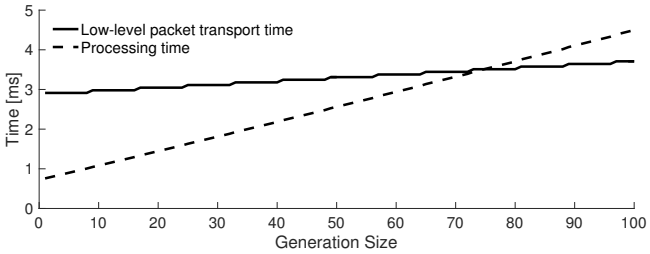


Figure 5: Low-level packet transport time $\tilde{T}_a$ and processing time T_p depending on the number of messages M (*i.e.*, generation size) for a payload size of 60 bytes.

5 EVALUATION

Our evaluation answers the following questions:

- How does Mixer's performance compare to the state of the art for different number of messages, message sizes, and initial message distributions (§5.1)?
- How robust is Mixer to network dynamics caused by node failures (§5.2) and node mobility (§5.3)?
- To what extent does Mixer benefit from faster processors and/or faster physical layers (§5.4)?

5.1 Performance

We compare Mixer against the state-of-the-art many-to-all solution based on sequential flooding, called S-Glossy. To implement S-Glossy, we use the original Glossy code [18], which achieves the minimum latency for flooding a single message in 802.15.4 networks [21]. Note that S-Glossy is equivalent to a data-only round in LWB [19]. It has already been shown [19] that LWB greatly outperforms routing-based many-to-many solutions [51], rendering a comparison against such schemes obsolete. Similarly, results from our work-in-progress reports show that an earlier version of Mixer outperforms Chaos for messages larger than a few bytes [46, 47]. Other RLNC-based many-to-all approaches like [11, 12, 22] provide only theoretical or simulation results and are not applicable to practical wireless mesh networks because the assumed communication models do not fit. See our discussion in §7 for more details.

Testbeds. We run experiments on two testbeds. On FlockLab, we use 27 TelosB nodes sparsely deployed across one floor in an office building [42]. Indriya features 94 devices densely deployed across three floors [13]. Nodes transmit at maximum power (0 dBm) on channel 26, yielding a network diameter of 4 and 8 hops on FlockLab and Indriya, respectively. Both testbeds experience interference from devices affecting the 2.4 GHz band, such as microwaves ovens and Wi-Fi, Bluetooth, and BLE devices. Each run lasts 30–60 min.

Metrics. *Reliability* is the percentage of delivered messages (*i.e.*, received *and* decoded in case of Mixer). *Latency* is the time from the beginning of a round until all messages are delivered (*i.e.*, this includes the time needed for decoding in Mixer). *Goodput* is the number of delivered message bits per unit of time. *Radio-on time*, typically used as a proxy for energy efficiency, denotes the accumulated time the radio is on during a round. We report averages over all nodes and rounds during a run as well as 25th/75th percentiles.

Parameters. We fine-tune S-Glossy based on several test runs so it achieves the shortest possible latency at a reliability above 99.9 %. We set the slot length in Mixer as detailed in §4, and set the round length conservatively. The size of the history window is $3 \cdot N$. Since we always measured a reliability of 100 % with Mixer in the following experiments, we do not report this metric.

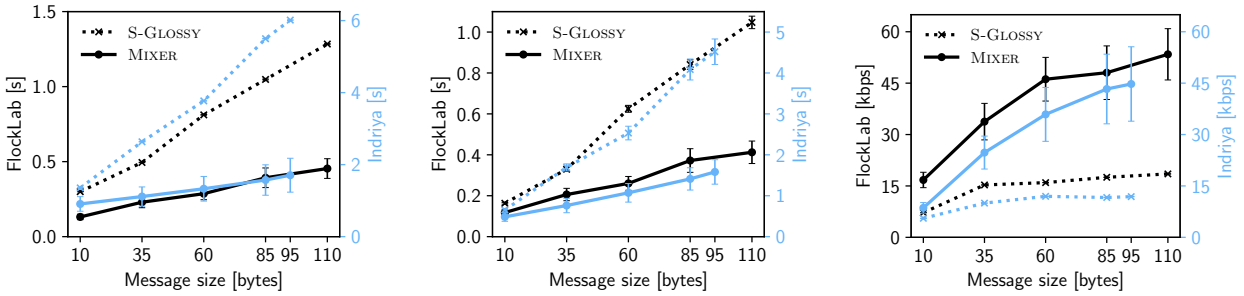
5.1.1 Impact of Message Size. To evaluate the impact of the message size, we run tests in which each node initially has exactly one message (all-to-all). We consider message sizes of 10, 35, 60, 85, and 95/110 bytes. Note that 95 and 110 bytes are very close to the largest message sizes that fit into an 802.15.4 packet given the overhead of Mixer's header information for 27 and 94 messages on FlockLab and Indriya, respectively.[4]

Results. Fig. 6 plots performance of Mixer and S-Glossy against message size. We see that Mixer outperforms S-Glossy across all metrics and settings, by 2.3–2.8× on FlockLab and by 1.4–3.8× on Indriya. The main reason is that Mixer needs significantly fewer slots than S-Glossy (*e.g.*, 95 vs. 260 on FlockLab): The combination of RLNC and synchronous transmissions is more efficient in terms of communication. The range of performance improvements on Indriya is wider than on FlockLab as (*i*) with $M = 94$ messages the slot length is determined by the processing time T_p, which limits the improvement to 1.4× for small messages, and (*ii*) the larger network diameter allows for higher spatial reuse, pushing the improvement up to 3.8× for large messages.

Looking at each individual metric, we find that latency, shown in Fig. 6a, increases linearly with message size because the slot length of both primitives increases linearly, too. On FlockLab, Mixer's latency is 131–453 ms, a range enabling feedback control in industrial automation [2]. Thanks to Mixer's smart shutdown feature, radio-on time (Fig. 6b) follows the same linear trend. Both primitives reach their highest goodput (Fig. 6c) with 110-byte messages on FlockLab: 18.5 kbit/s for S-Glossy and 53.7 kbit/s for Mixer.

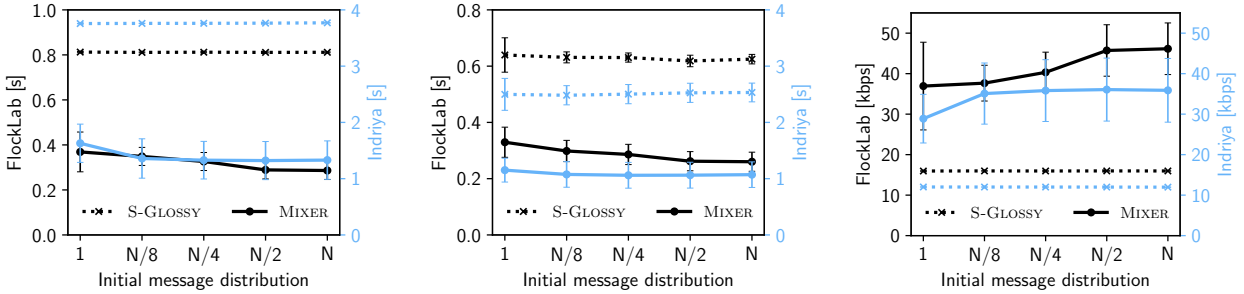
[3] Rearranging (5) gives $T_p \approx 0.02M^2 + 27M + 0.16S_pM + 1.8S_p + 620$, which reveals a quadratic dependency on the generation size M. However, for realistic values of M the linear term clearly dominates due to the coefficients. This is mainly because processing happens in machine words, not bits.

[4] The limited RAM on the TelosB would prevent us from running all-to-all experiments with messages larger than 35 bytes on Indriya. Thus, for these message sizes on Indriya, nodes do not store the transmitted full-size payloads (*i.e.*, only the coding vectors are stored), and instead perform cycle-accurate computations on a fake payload.



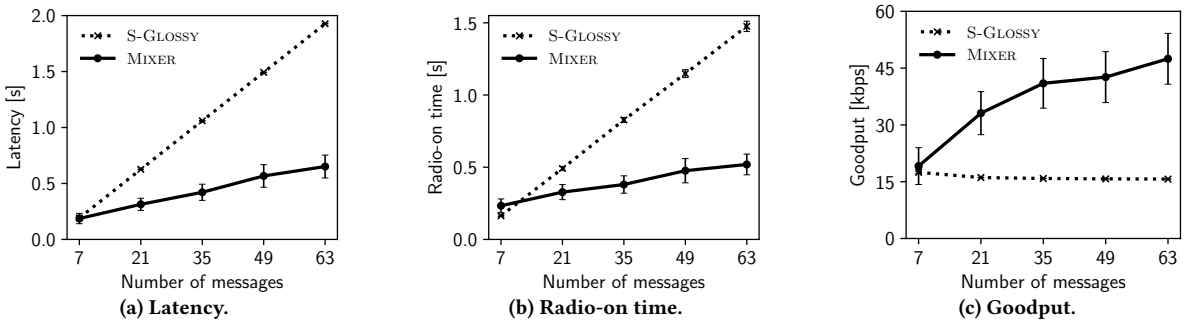
(a) Latency. (b) Radio-on time. (c) Goodput.

Figure 6: Performance in an all-to-all scenario on FlockLab and Indriya for different message sizes.



(a) Latency. (b) Radio-on time. (c) Goodput.

Figure 7: Performance on FlockLab and Indriya for different initial message distributions.



(a) Latency. (b) Radio-on time. (c) Goodput.

Figure 8: Performance on FlockLab for different number of messages equally distributed to seven source nodes.

5.1.2 Impact of Initial Message Distribution. Theoretical analysis shows that starting with a well-mixed distribution, where initially all messages are equally spread over the network, can boost performance compared to the case where all messages reside at the same node [24]. To evaluate this aspect, we fix the message size (60 bytes) and the number of messages M (27 on FlockLab, 94 on Indriya), and vary the fraction of the N nodes that initially holds the M messages: N (all-to-all), $N/2$, $N/4$, $N/8$, and 1 (one-to-all). Messages are equally distributed across the respective source nodes.

Results. Our results in Fig. 7 confirm that Mixer benefits from a well-mixed initial message distribution, while performance with S-Glossy is unaffected. The effect becomes noticeable when messages are pooled at $N/4$ or fewer nodes, yet the performance loss is at most 22 % compared to the all-to-all case.We attribute this behavior especially to Mixer's improved startup phase (§3.2.3), which ensures fast-growing coding potential even in one-to-all scenarios.

5.1.3 Impact of Number of Messages (Generation Size). In a final set of performance experiments, we investigate the impact of the number of messages M to be exchanged in a round (generation size). We use again a message size of 60 bytes and equally distribute 7, 21, 35, 49, and 63 messages to seven source nodes on FlockLab.[5]

Results. Looking at Fig. 8, we see that Mixer only has a performance advantage over S-Glossy if there are at least a handful of messages to be exchanged. Otherwise, the coding potential is too small and fine-tuned sequential Glossy floods perform better. Nevertheless, Mixer's performance advantage grows quickly with the number of messages: Mixer is 2× faster and more efficient than S-Glossy for 21 messages, and already 3× better for 63 messages.

Interestingly, we find in every *all-to-all* experiment that Mixer needs on average about $3 \cdot M$ slots to deliver M messages despite vastly different payload sizes, network diameters, and node densi-

[5]We only show results from FlockLab because the number of active nodes on Indriya changed significantly during our experiments so that the results would not be comparable to those in Figs. 6 and 7.

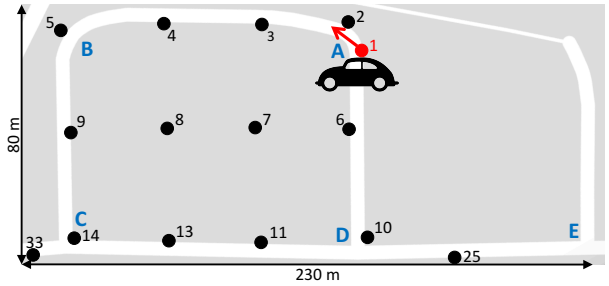


Figure 9: Setup of outdoor experiment with a mobile node mounted on a car that drives at a speed of 20–60 km/h.

ties on the two testbeds. This gives an idea of the constants hidden by the O-notation in the order-optimal scaling $O(M + T)$ of RLNC-based gossip for our Mixer implementation. Indeed, for small generation sizes M, the number of slots T needed to disseminate one message dominates, whereas M dominates for large generation sizes. T is determined by the diameter of the network.

5.2 Network Dynamics: Node Failures

In practice, nodes can suddenly drop out due to disconnection or failure. While the absence of a node for several rounds is handled by a higher-layer protocol, Mixer must cope with situations where nodes disappear shortly before or during a round.

To study this aspect, we run experiments on FlockLab in which we let a given number of nodes simultaneously fail. We consider the failure of 1, 2, and 5 nodes and vary the slot in which the failure occurs from 1 to 50. For each combination we perform at least 100 rounds. Nodes exchange 10-byte messages in an all-to-all fashion, and non-failing nodes log after each round how many of the 27 messages they can successfully decode.

Results. Fig. 10 plots the probability that a non-failing node decodes all messages against the slot in which the failure occurs. We see that a failure before or at the very beginning of a round prevents the non-failing nodes from decoding all messages. This is because the nodes fail before they can transmit for the first time. Thus, a failure in later slots increases the probability that the non-failing nodes can decode all messages. The increase shifts to the right with more failing nodes as nodes also need to transmit linearly independent packets. Beyond a certain point (*e.g.*, slot 20 for 1 failing node), the probability to decode all messages is close to 100 %.

Note that, barring packet losses due to other reasons, a node *for sure* decodes all messages that initially resided at non-failing nodes. For instance, in our runs each non-failing node always decodes at least 26, 25, and 22 messages for 1, 2, and 5 failing nodes, respectively.

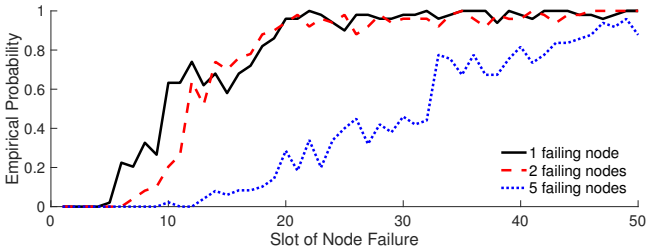


Figure 10: Probability that nodes decode all messages against the slot in which a set of nodes concurrently fails.

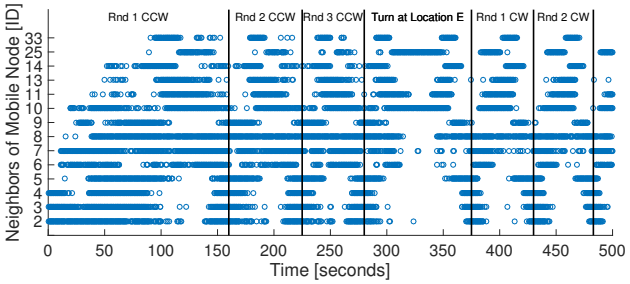


Figure 11: 1-hop neighbors of mobile node as the car drives rounds counterclockwise (CCW) and clockwise (CW).

We can therefore conclude that Mixer is highly robust to node failures, providing guaranteed service to non-failing nodes while salvaging messages of failing nodes with high probability.

5.3 Network Dynamics: Node Mobility

Emerging applications increasingly rely on nodes attached to mobile entities [35, 56]. We investigate Mixer's resilience against the resulting network dynamics in an outdoor experiment.

Scenario. We deploy 14 battery-powered TelosB nodes on cardboard boxes in a 80 m ×230 m area as illustrated in Fig. 9. Another node is mounted on a car and attached over USB to a laptop. Nodes transmit with 0 dBm on channel 26, using Mixer to periodically exchange 15 28-byte messages in an all-to-all fashion. The messages contain performance counters and the IDs of all nodes from which the nodes have *directly* received a packet during the previous Mixer round (*i.e.*, their 1-hop neighbors). We set the slot length to 2 ms according to the guidelines in §4. Nodes initiate a round every 500 ms, while the round length is 150 slots. At the end of each round, the node on the car uses the remaining 200 ms to log the messages it received over USB before the next round begins.

We first measure for 10 min with the car standing next to location A (see Fig. 9). Then we measure for 10 min while performing different maneuvers with the car. We repeatedly pass locations A→B→C→D, then make a turn at location E, and repeatedly pass locations D→C→B→A. We drive with a speed of 20-40 km/h while going in circles, and hit 60 km/h between locations E and C. Such speeds are typical of state-of-the-art mini and micro drones [8].

Results. Fig. 11 shows the 1-hop neighbors of the node on the driving car over time. We can see that the mobile node cannot directly communicate with all other nodes: It has different sets of neighbors depending on its location. Indeed, we recognize a recurring pattern that allows us to infer how often the car drove around the circle. For example, before completing the fourth round counterclockwise (CCW), the car drove up to location E, leaving the mobile node with only two neighors and increasing the network diameter to at least three hops. Then, the car made a turn and continued to drive the circle clockwise (CW) three more times.

Despite heavy network dynamics due to the high speed of movement, we measure the same performance compared to when the car was standing. As visible from Table 2, Mixer consistently provides high reliability >99.99 % and low latency ≤98.4 ms. For shorter messages (*e.g.*, a few bytes carrying GPS data), one could further reduce the slot length to 1 ms. As a result, latency in this scenario would reduce to about 50 ms, which is sufficient for drone swarm coordination requiring all-to-all communication every 100 ms [7, 56]. In

Table 2: Mixer's performance with and without mobility.

Performance Metric	With Mobility	Without Mobility
Reliability [%]	>99.99	>99.99
Latency [ms]	96.2	98.4

summary, the results show that Mixer is highly robust to network dynamics and satisfies the demands of emerging applications.

5.4 Potential of Faster CPUs and PHYs

Mixer can benefit from a more powerful processor (CPU): Using the same physical layer (PHY), a faster CPU allows Mixer to process packets at line rate for a wider range of payload sizes and number of messages. For example, 2× faster processing shifts the crossover point in Fig. 5 from 75 to 215 messages. Conversely, to fully exploit a faster PHY, the processing speed should increase as well.

CPU cores. We perform microbenchmarks on the TelosB and three different 32-bit ARM cores: Cortex-M0+ running at 32 MHz (labeled L0), Cortex-M4 running at 80 MHz (L4), and Cortex-M4 running at 180 MHz (F4). Despite higher clock speeds, L4 and F4 offer extended instruction sets and richer hardware capabilities than the L0.

Processing speedup. To measure the processing speedup, we port Mixer's packet processing routines to the ARM cores. We profile the time needed to process received packets (building transmit packets involves similar operations, merely with lower variance). To obtain real-world execution times for the same input on all cores, we run tests on FlockLab with 60-byte messages for different generation sizes ($M = \{7, 35, 63\}$) and trace the sequence of received packets at each node. We feed the collected traces into the four cores to get a total of 40,000 execution time measurements per core.

The speedups, shown in Fig. 12, are 7–10×, 28–40×, and 62–106× for the L0, L4, and F4, respectively. The speedup depends on the generation size M, because Mixer processes payloads in batches and delayed from coding vectors. This optimization becomes more important with higher average batch size and thus with more messages M. We use a highly optimized implementation of this approach on the TelosB, and expect that similar ARM-specific code optimizations reduce the differences in speedups for different M.

Projected benefits. We use (4) and (5) to project the impact of a faster CPU and/or a faster PHY on Mixer's performance. Specifically, we multiply $\tilde{T}_a$ with the speedup in PHY bitrate and T_p with the processing speedup over the TelosB. Using the speedups from Fig. 12, we assume that all processing activites in Mixer benefit as much as the processing of received packets. In this way, we can check, for example, whether a CPU core is under-, over-, or well-dimensioned for a given PHY bitrate.

As an example, consider an all-to-all scenario with 60-byte messages on FlockLab. We know from §5.1 that Mixer needs about 95 slots with 802.15.4, and studies suggest that capture (and hence

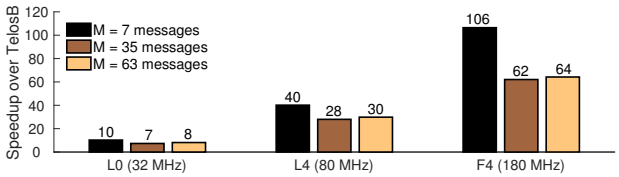


Figure 12: Speedup of ARM cores over TelosB in processing received packets for different generation sizes.

Table 3: Projected latency of Mixer on FlockLab (all-to-all, 60-byte messages) for different PHY bitrates and CPU cores. *Latencies in italics are CPU-bound (i.e., core is underdimensioned).*

PHY Bitrate [Mbit/s]	TelosB [ms]	L0 [ms]	L4 [ms]	F4 [ms]
0.25	295.6	295.6	295.6	295.6
1	*161.8*	73.9	73.9	73.9
11	*161.8*	*23.1*	6.7	6.7
54	*161.8*	*23.1*	*5.8*	*2.6*

Mixer) works comparable or better with other PHYs [48]. Table 3 lists projected average latency in milliseconds for 16 PHY/CPU combinations, using the *minimum* speedups for each core from Fig. 12. We see, for example, that the L0 is sufficient to fully leverage a PHY bitrate of 1 Mbit/s used by BLE, while the L4 or the F4 is needed to match 11 or 54 Mbit/s of 802.11 variants. The latter combination (F4, 54 Mbit/s) would reduce latency by 100× compared to the TelosB.

6 DISCUSSION

Larger finite fields. Mixer currently uses GF(2), which keeps the coding vectors small and allows for a straightforward and efficient implementation on standard hardware. Instead, the network coding literature favors larger finite fields, such as GF(2^8) [22], to increase the chances that a received packet is innovative. We studied the impact of larger finite fields on Mixer's performance in simulation and found that the gains are smaller than one may expect: Using GF(2^2) reduces the average number of slots by about 10 % compared with GF(2), but GF($2^3 \ldots 2^8$) do not provide further improvements. We attribute this to the fact that the spreading of innovation in an area (as promoted by larger finite fields) is upper-bounded by the influx of messages into that area. It is also questionable whether the fewer slots with larger finite fields can indeed translate into shorter latencies in practice as the computational load and the size of the coding vectors would increase by several orders of magnitude.

Robustness to interference. Mixer achieves nearly perfect reliability in almost all our experiments conducted under typical wireless interference in office buildings. Nevertheless, it would be possible to borrow standard techniques such as frequency hopping from other technologies (*e.g.*, Bluetooth) to make Mixer even more robust to interference. To this end, the slot number may serve as an index into a pseudo-random sequence of channel frequencies that is known to all nodes. One may also increase the slot length T_s to tolerate interference bursts (at the cost of higher latency), or adapt T_s in a pseudo-random fashion to evade systematic jamming.

Setting the length of a round. Mixer provides a parameter that specifies the number of slots constituting a round. Together with the slot length T_s this parameter defines the nominal length of a round and makes the running time of Mixer predictable, in addition to the timeout mentioned in §3.3.1. However, one limitation of Mixer is that the number of slots required until all nodes have reached full rank is difficult to predict. While some applications allow for using a conservative estimate, this approach may be problematic for applications with critical time constraints. Theoretical works have looked at the worst-case number of slots for different network and communication models [11, 22, 24, 50]. It would be worthwhile to adapt these models to Mixer (*e.g.*, by incorporating existing capture models [33, 64]) to determine safe bounds on the length of a round.

Our experiments suggest that useful predictions are within reach: We observe in all all-to-all runs that Mixer needs on average about $3 \cdot M$ slots despite different network topologies and payload sizes.

7 RELATED WORK

Theoretical foundations. Ahlswede et al. introduced network coding, showing that it achieves the multicast capacity of wireline networks [1]. It was later found that these bounds can be achieved using linear codes, and that encoding and decoding can be done in polynomial time [34, 40]. This also holds if nodes pick random coefficients [28]. These works form the theoretical foundation of RLNC, which we combine in Mixer with the following technique.

Synchronous transmissions. Mixer exploits simultaneous transmissions from multiple senders. SourceSync is the first system that demonstrates the benefits of multiple senders transmitting the *same* packet in real 802.11 networks [58]. Glossy uses this concept for fast and reliable flooding in multi-hop 802.15.4 networks [21]. These protocols rely on accurate symbol-level synchronization to benefit from sender diversity. In Mixer, instead, nodes transmit *different* packets, which relaxes the required synchronization to the length of the preamble to possibly receive one of the transmitted packets due to the capture effect [38, 39, 65]. The capture effect has been used for collision resolution [63], network flooding [43], aggregation [37], and agreement [3]. Instead, Mixer exploits the capture effect for efficient many-to-all broadcasting of sizable messages.

Practical wireless network coding. Network coding has been extensively studied in wireless and sensor networks; however, the vast majority of works focuses on theoretical gains or evaluates new protocol designs only in simulation (see [55] for a recent survey), thereby ignoring many practical issues that complicate or even prevent a real implementation.

COPE [32] and MORE [6] are the first implementations of network coding for multiple unicast flows and a single multicast flow in 802.11 networks. Pacifier [36] achieves a higher multicast throughput than MORE. These works target stationary networks and specific traffic patterns, which allows them to leverage long-lived network and routing state for coding and packet forwarding. This, however, makes them unfit for dynamic networks and concurrent multicast flows, both of which Mixer readily supports. Moreover, they target PC-class devices with plenty of compute power, memory, wireless bandwidth, and energy. Mixer can cope with stringent constraints on any of these resources, allowing low-power wireless systems to benefit from network coding without putting restrictions on the traffic pattern.

Splash [14] and Pando [15] integrate pipelined flooding with XOR and fountain coding for one-to-all data dissemination in 802.15.4 networks. While these solutions run on resource-constrained devices, only the source encodes packets; all other nodes forward the encoded packets and decode, which simplifies design and implementation. They also assume stationary networks and support only a single source node. Instead, Mixer supports dynamic networks, any number of sources, and efficiently performs forwarding, en-/recoding, and decoding at every node in the network. As a result, Mixer performs comparable or better to these specialized protocols, and yet supports a much broader range of scenarios.

Many-to-all broadcasting. The unstructured spreading of messages in Mixer is reminiscent of gossip [30]. Deb and Médard showed that combining gossip with RLNC for the dissemination of multiple messages outperforms any non-coding approach (in terms of needed slots) in a specific communication scenario [11, 12]. More precisely, they consider a *random phone call model* where the underlying network graph is complete. Later works study variants of this approach theoretically [50] and in simulation [22], primarily on static networks. However, the underlying network model does not fit wireless mesh networks because (a) the random phone call model implies that all links work independently (*i.e.*, there is no interference); (b) it is assumed that a node is able to receive multiple packets simultaneously or can perfectly avoid collisions. Further, the results rely on assumptions regarding the initial message distribution and the field size used for network coding. In particular, [12, 50] consider cases where the field size q has been chosen such that $q \geq M$ while the simulations in [22] use GF(2^8). With packet size constraints as in 802.15.4, such field sizes can lead to significant limitations and performance degradation as discussed in §6.

Recently, it was shown that RLNC-based gossip achieves the optimal scaling $O(M + T)$ also in dynamic networks for any initial message distribution and field size [24]. Furthermore, [24] provides results for a *broadcast model* that fits much better to the inherent nature of wireless networks. However, the analysis is purely theoretical and still assumes that nodes are able to receive multiple packets simultaneously. As for dynamic wireless mesh networks, Mixer is the first design that translates the projected benefits from theory into practice by combining RLNC with synchronous transmissions.

Concurrently to our work, Mohammad and Chan [49] proposed Codecast, combining LT codes [44] with synchronous transmissions. LT codes can be interpreted as a special variant of RLNC. However, they lack the recoding capabilities of generic RLNC: In Codecast, a node is not able to recode arbitrary payloads, it can only (re-)encode previously decoded messages. Thus, the coding potential at the nodes grows slower than it does with Mixer, eventually leading to longer rounds. Indeed, the results in [49] suggest that Mixer outperforms Codecast by up to 3× on FlockLab. Further, the design of Codecast suffers from severe scalability issues for more than $M = 30$ messages.

8 CONCLUSIONS

We have presented Mixer, an efficient, versatile, and reliable many-to-all broadcast primitive for wireless mesh networks. Mixer supports the full spectrum from one-to-all to all-to-all communication, from small to large messages, and from static to dynamic networks. Unlike prior practical many-to-all solutions that route or flood messages Mixer exploits the synergy of RLNC and synchronous transmissions for simultaneously disseminating all messages. Our implementation of Mixer is up to 3.8× faster and more efficient than the state of the art, while providing nearly 100 % reliability. Thus, Mixer empowers emerging wireless systems and enables applications that seemed out of reach so far.

ACKNOWLEDGMENTS

We thank our reviewers for their valuable feedback as well as Johannes Neumann for his contribution to the early development versions of Mixer. This work was supported by the German Research Foundation (DFG) within the Cluster of Excellence cfaed (EXC 1056) and SPP 1914 project EcoCPS (grant ZI 1635/1-1).

REFERENCES

[1] R. Ahlswede, N. Cai, S. R. Li, and R. W. Yeung. 2000. Network Information Flow. *IEEE Transactions on Information Theory* 46, 4 (2000).

[2] J. Åkerberg, M. Gidlund, and M. Björkman. 2011. Future Research Challenges in Wireless Sensor and Actuator Networks Targeting Industrial Automation. In *Proceedings of the 9th IEEE International Conference on Industrial Informatics (INDIN)*.

[3] B. Al Nahas, S. Duquennoy, and O. Landsiedel. 2017. Network-wide Consensus Utilizing the Capture Effect in Low-power Wireless Networks. In *Proceedings of the 15th ACM Conference on Embedded Networked Sensor Systems (SenSys)*.

[4] I. Bekmezci, O. K. Sahingoz, and S. đTemel. 2013. Flying Ad-Hoc Networks (FANETs): A survey. *Ad Hoc Networks* 11, 3 (2013).

[5] V. D. Blondel and J. N. Tsitsiklis. 2000. A Survey of Computational Complexity Results in Systems and Control. *Automatica* 36, 9 (2000).

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

[7] T. H. Chung, M. R. Clement, M. A. Day, K. D. Jones, D. Davis, and M. Jones. 2016. Live-Fly, Large-Scale Field Experimentation for Large Numbers of Fixed-Wing UAVs. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*.

[8] J. R. Clapper, J. J. Young, J. E. Cartwright, and J. G. Grimes. 2007. *Unmanned Systems Roadmap 2007-2032.* Technical Report. US Department of Defense. https://www.globalsecurity.org/intell/library/reports/2007/dod-unmanned-systems-roadmap_2007-2032.pdf

[9] N. Correll, P. Dutta, R. Han, and K. Pister. 2017. New Directions: Wireless Robotic Materials. In *Proceedings of the 15th ACM Conference on Embedded Networked Sensor Systems (SenSys)*.

[10] B. Dawkins. 1991. Siobhan's Problem: The Coupon Collector Revisited. *The American Statistician* 45, 1 (1991).

[11] S. Deb and M. Médard. 2004. Algebraic Gossip: A Network Coding Approach to Optimal Multiple Rumor Mongering. In *Proceedings of the 42nd Allerton Conference on Communication, Control, and Computing*.

[12] S. Deb, M. Médard, and C. Choute. 2006. Algebraic Gossip: A Network Coding Approach to Optimal Multiple Rumor Mongering. *IEEE Transactions on Information Theory* 52, 6 (2006).

[13] M. Doddavenkatappa, M. C. Chan, and A. L. Ananda. 2011. Indriya: A Low-Cost, 3D Wireless Sensor Network Testbed. In *Proceedings of the ICST Conference on Testbeds and Research Infrastructures for the Development of Networks and Communities (TridentCom)*.

[14] M. Doddavenkatappa, M. C. Chan, and B. Leong. 2013. Splash: Fast Data Dissemination with Constructive Interference in Wireless Sensor Networks. In *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*.

[15] W. Du, J. C. Liando, H. Zhang, and M. Li. 2015. When Pipelines Meet Fountain: Fast Data Dissemination in Wireless Sensor Networks. In *Proceedings of the 13th ACM Conference on Embedded Networked Sensor Systems (SenSys)*.

[16] S. Duquennoy, B. Al Nahas, O. Landsiedel, and T. Watteyne. 2015. Orchestra: Robust Mesh Networks Through Autonomously Scheduled TSCH. In *Proceedings of the 13th ACM Conference on Embedded Networked Sensor Systems (SenSys)*.

[17] W. Feller. 1968. *An Introduction to Probability Theory and Its Applications* (3rd ed.). Wiley.

[18] F. Ferrari. 2011. Original Glossy Implementation for the TelosB Platform. https://sourceforge.net/p/contikiprojects/code/HEAD/tree/ethz.ch/glossy/

[19] F. Ferrari, M. Zimmerling, L. Mottola, and L. Thiele. 2012. Low-power Wireless Bus. In *Proceedings of the 10th ACM Conference on Embedded Networked Sensor Systems (SenSys)*.

[20] F. Ferrari, M. Zimmerling, L. Mottola, and L. Thiele. 2013. Virtual Synchrony Guarantees for Cyber-physical Systems. In *Proceedings of the 32nd IEEE International Symposium on Reliable Distributed Systems (SRDS)*.

[21] F. Ferrari, M. Zimmerling, L. Thiele, and O. Saukh. 2011. Efficient Network Flooding and Time Synchronization with Glossy. In *Proceedings of the 10th ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN)*.

[22] C. Fragouli, J. Widmer, and J. Le Boudec. 2008. Efficient Broadcasting Using Network Coding. *IEEE/ACM Transactions on Networking* 16, 2 (2008).

[23] B. Haeupler. 2011. Analyzing Network Coding Gossip Made Easy. In *Proceedings of the 43rd Annual ACM Symposium on Theory of Computing (STOC)*.

[24] B. Haeupler. 2016. Analyzing Network Coding (Gossip) Made Easy. *Journal of the ACM* 63, 3 (2016).

[25] S. Hayat, E. Yanmaz, and R. Muzaffar. 2016. Survey on Unmanned Aerial Vehicle Networks for Civil Applications: A Communications Viewpoint. *IEEE Communications Surveys & Tutorials* 18, 4 (2016).

[26] J. Heide, M. V. Pedersen, F. H. P. Fitzek, and M. Médard. 2011. On Code Parameters and Coding Vector Representation for Practical RLNC. In *Proceedings of the IEEE International Conference on Communications (ICC)*.

[27] H. Hellwagner and C. Bettstetter. 2016. Networking research challenges in multi-UAV systems. https://bettstetter.com/uav-networking-challenges/

[28] T. Ho, M. Médard, R. Koetter, D. R. Karger, M. Effros, J. Shi, and B. Leong. 2006. A Random Linear Network Coding Approach to Multicast. *IEEE Transactions on Information Theory* 52, 10 (2006).

[29] T. Istomin, C. Kiraly, and G. P. Picco. 2015. Is RPL Ready for Actuation? A Comparative Evaluation in a Smart City Scenario. In *Proceedings of the 12th European Conference on Wireless Sensor Networks (EWSN)*.

[30] R. Karp, C. Schindelhauer, S. Shenker, and B. Vocking. 2000. Randomized Rumor Spreading. In *Proceedings of the 41st Annual Symposium on Foundations of Computer Science (FOCS)*.

[31] S. Katti, S. Gollakota, and D. Katabi. 2007. Embracing Wireless Interference: Analog Network Coding. In *Proceedings of ACM SIGCOMM*.

[32] S. Katti, H. Rahul, W. Hu, D. Katabi, M. Médard, and J. Crowcroft. 2006. XORs in the Air: Practical Wireless Network Coding. In *Proceedings of ACM SIGCOMM*.

[33] A. Kochut, A. Vasan, A. U. Shankar, and A. Agrawala. 2004. Sniffing out the correct Physical Layer Capture model in 802.11b. In *Proceedings of the 12th IEEE International Conference on Network Protocols (ICNP)*.

[34] R. Koetter and M. Medard. 2003. An Algebraic Approach to Network Coding. *IEEE/ACM Transactions on Networking* 11, 5 (2003).

[35] L. Kong, X. Chen, X. Liu, Q. Xiang, Y. Gao, N. B. Baruch, and G. Chen. 2017. AdaSharing: Adaptive Data Sharing in Collaborative Robots. *IEEE Transactions on Industrial Electronics* 64, 12 (2017).

[36] D. Koutsonikolas, Y. C. Hu, and C. Wang. 2012. Pacifier: High-throughput, Reliable Multicast Without "Crying Babies" in Wireless Mesh Networks. *IEEE/ACM Transactions on Networking* 20, 5 (2012).

[37] O. Landsiedel, F. Ferrari, and M. Zimmerling. 2013. Chaos: Versatile and Efficient All-to-all Data Sharing and In-network Processing at Scale. In *Proceedings of th 11th ACM Conference on Embedded Networked Sensor Systems (SenSys)*.

[38] J. Lee, W. Kim, S.-J. Lee, D. Jo, J. Ryu, T. Kwon, and Y. Choi. 2007. An Experimental Study on the Capture Effect in 802.11a Networks. In *Proceedings of the 2nd ACM International Workshop on Wireless Network Testbeds, Experimental Evaluation and Characterization (WinTECH)*.

[39] K. Leentvaar and J. Flint. 1976. The Capture Effect in FM Receivers. *IEEE Transactions on Communications* 24, 5 (1976).

[40] S. R. Li, R. W. Yeung, and N. Cai. 2003. Linear Network Coding. *IEEE Transactions on Information Theory* 49, 2 (2003).

[41] Y. Li, E. Soljanin, and P. Spasojevic. 2011. Effects of the Generation Size and Overlap on Throughput and Complexity in Randomized Linear Network Coding. *IEEE Transactions on Information Theory* 57, 2 (2011).

[42] R. Lim, F. Ferrari, M. Zimmerling, C. Walser, P. Sommer, and J. Beutel. 2013. FlockLab: A Testbed for Distributed, Synchronized Tracing and Profiling of Wireless Embedded Systems. In *Proceedings of the 12th ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN)*.

[43] J. Lu and K. Whitehouse. 2009. Flash Flooding: Exploiting the Capture Effect for Rapid Flooding in Wireless Sensor Networks. In *Proceedings of IEEE INFOCOM*.

[44] Michael Luby. 2002. LT Codes. In *Proceedings of the 43rd Symposium on Foundations of Computer Science (FOCS)*.

[45] M. Luvisotto, Z. Pang, and D. Dzung. 2016. Ultra High Performance Wireless Control for Critical Applications: Challenges and Directions. *IEEE Transactions on Industrial Informatics* 13, 3 (2016).

[46] F. Mager, C. Herrmann, and M. Zimmerling. 2017. One for All, All for One: Toward Efficient Many-to-Many Broadcast in Dynamic Wireless Networks. In *Proceedings of the 4th ACM Workshop on Hot Topics in Wireless (HotWireless)*.

[47] F. Mager, J. Neumann, C. Herrmann, M. Zimmerling, and F. H. P. Fitzek. 2016. All-to-all Communication in Multi-hop Wireless Networks with Mixer: Poster Abstract. In *Proceedings of the 14th ACM Conference on Embedded Network Sensor Systems (SenSys)*.

[48] J. Manweiler, N. Santhapuri, S. Sen, R. R. Choudhury, S. Nelakuditi, and K. Munagala. 2009. Order Matters: Transmission Reordering in Wireless Networks. In *Proceedings of the ACM Annual International Conference on Mobile Computing and Networking (MobiCom)*.

[49] M. Mohammad and M. C. Chan. 2018. Codecast: Supporting Data Driven In-network Processing for Low-power Wireless Sensor Networks. In *Proceedings of the 17th ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN)*.

[50] D. Mosk-Aoyama and D. Shah. 2006. Information Dissemination via Network Coding. In *Proceedings of the IEEE International Symposium on Information Theory (ISIT)*.

[51] L. Mottola and G. P. Picco. 2011. MUSTER: Adaptive Energy-Aware Multisink Routing in Wireless Sensor Networks. *IEEE Transactions on Mobile Computing* 10, 12 (2011).

[52] L. Mottola and G. P. Picco. 2011. Programming Wireless Sensor Networks: Fundamental Concepts and State of the Art. *Comput. Surveys* 43, 3 (2011).

[53] B. Nazer and M. Gastpar. 2011. Compute-and-Forward: Harnessing Interference Through Structured Codes. *IEEE Transactions on Information Theory* 57, 10 (2011).

[54] RCR Wirelessss News. 2016. Amazon's ambitions will push delivery drones' battery lives to the limit and maybe beyond. http://www.rcrwireless.com/20160802/europe/five-challenges-drones-tag28

[55] P. Ostovari, J. Wu, and A. Khreishah. 2014. *Network Coding Techniques for Wireless and Sensor Networks.* Springer.

[56] J. A. Preiss, W. Honig, G. S. Sukhatme, and N. Ayanian. 2017. Crazyswarm: A Large Nano-Quadcopter Swarm. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*.
[57] Qualcomm. 2017. The path to 5G: Cellular Vehicle-to-Everything (C-V2X). https://www.qualcomm.com/documents/path-5g-cellular-vehicle-everything-c-v2x
[58] H. Rahul, H. Hassanieh, and D. Katabi. 2010. SourceSync: A Distributed Wireless Architecture for Exploiting Sender Diversity. In *Proceedings of ACM SIGCOMM*.
[59] C. Roest. 2015. *Enabling the Chaos Networking Primitive on Bluetooth LE.* Master's thesis. TU Delft.
[60] F. B. Schneider. 1990. Implementing Fault-tolerant Services Using the State Machine Approach: A Tutorial. *Comput. Surveys* 22, 4 (1990).
[61] M. Schuß, C. A. Boano, M. Weber, and K. Römer. 2017. A Competition to Push the Dependability of Low-Power Wireless Protocols to the Edge. In *Proceedings of the International Conference on Embedded Wireless Systems and Networks (EWSN)*.
[62] B. Sinopoli, L. Schenato, M. Franceschetti, K. Poolla, M. I. Jordan, and S. S. Sastry. 2004. Kalman Filtering With Intermittent Observations. *IEEE Transactions on Automatic Control* 49, 9 (2004).
[63] K. Whitehouse, A. Woo, F. Jiang, J. Polastre, and D. Culler. 2005. Exploiting the Capture Effect for Collision Detection and Recovery. In *Proceedings of the 2nd IEEE Workshop on Embedded Networked Sensors (EmNets)*.
[64] M. Wilhelm, V. Lenders, and J. B. Schmitt. 2014. On the Reception of Concurrent Transmissions in Wireless Sensor Networks. *IEEE Transactions on Wireless Communications* 13, 12 (2014).
[65] D. Yuan and M. Hollick. 2013. Let's Talk Together: Understanding Concurrent Transmission in Wireless Sensor Networks. In *Proceedings of the 38th IEEE Conference on Local Computer Networks (LCN)*.