Symbol-level Network Coding for Wireless Mesh Networks

Sachin Katti, Dina Katabi, Hari Balakrishnan, and Muriel Medard
Massachusetts Institute of Technology

ABSTRACT

This paper describes MIXIT, a system that improves the throughput of wireless mesh networks. MIXIT exploits a basic property of mesh networks: even when no node receives a packet correctly, any given bit is likely to be received by some node correctly. Instead of insisting on forwarding only correct packets, MIXIT routers use physical layer hints to make their best guess about which bits in a corrupted packet are likely to be correct and forward them to the destination. Even though this approach inevitably lets erroneous bits through, we find that it can achieve high throughput without compromising end-to-end reliability.

The core component of MIXIT is a novel network code that operates on small groups of bits, called symbols. It allows the nodes to opportunistically route groups of bits to their destination with low overhead. MIXIT's network code also incorporates an end-to-end error correction component that the destination uses to correct any errors that might seep through. We have implemented MIXIT on a software radio platform running the Zigbee radio protocol. Our experiments on a 25-node indoor testbed show that MIXIT has a throughput gain of $2.8\times$ over MORE, a state-of-the-art opportunistic routing scheme, and about $3.9\times$ over traditional routing using the ETX metric.

Categories and Subject Descriptors
C.2.2 [**Computer Systems Organization**]: Computer-Communications Networks

General Terms
Algorithms, Design, Performance, Theory

1 Introduction

This paper presents MIXIT a system that significantly improves the throughput of a wireless mesh network compared to the best current approaches. In both traditional routing protocols as well as more recent opportunistic approaches [1, 2], each intermediate node forwards a packet only if it has no errors. In contrast, MIXIT takes a much looser approach: a forwarding node does not attempt to recover from any errors, or even bother to apply an error detection code (like a CRC).

Somewhat surprisingly, relaxing the requirement that a node only forward correct data improves throughput. The main reason for this improvement is a unique property of wireless mesh networks: *Even when no node receives a packet correctly, any given bit is likely to be received correctly by* some *node.*

In MIXIT, the network and the lower layers collaborate to improve throughput by taking advantage of this observation. Rather than just send up a sequence of bits, the PHY annotates each bit with SoftPHY hints [8] that reflect the PHY's confidence in its demodulation and decoding. The MIXIT link layer passes up frames to the network layer with these annotations, but does not try to recover erroneous frames or low-confidence bits using link-layer retransmissions. Instead, the network layer uses the SoftPHY hints to filter out the groups of bits with low confidence in a packet, and then it performs opportunistic routing on groups of high confidence bits. Hence, insisting on receiving fully correct packets on each link is unnecessary.

The core component of the MIXIT protocol is a new network code that allows each link to operate at a considerably high bit-error rate compared to the status quo without compromising end-to-end reliability. Unlike previous work, the network code operates at the granularity of *symbols* [1] rather than packets: each router forwards (using radio broadcast) random linear combinations of the high-confidence symbols belonging to different packets. Thus, a MIXIT router forwards symbols that are likely to be correct, tries to avoid forwarding symbols that are likely to be corrupt, but inevitably makes a few incorrect guesses and forwards corrupt symbols.

MIXIT's network code addresses two challenges in performing such symbol-level opportunistic routing over potentially erroneous data. The first problem is *scalable coordination*: the effort required for nodes to determine which symbols were received at each node to prevent duplicate transmissions of the same symbol is significant. MIXIT uses the randomness from the network code along with a novel dynamic programming algorithm to solve this problem and scalably "funnel" high-confidence symbols to the destination, compared to a node co-ordination based approach like ExOR [1].

The second problem is *error recovery*: because erroneous symbols do seep through, the destination needs to correct them. Rather than the traditional approach of requesting explicit retransmissions, MIXIT uses a rateless end-to-end error correcting component that works in concert with the network code for this task. The routers themselves only forward random linear combinations of high-confidence symbols, performing no error handling.

MIXIT incorporates two additional techniques to improve performance:

- **Increased concurrency:** MIXIT takes advantage of the flexibility that nodes are no longer required to ensure correct packet delivery to design a channel access protocol that allows many more concurrent transmissions than CSMA. As long as a transmitted symbol is correctly received by *some node* closer to the final destination, the protocol achieves good throughput.

- **Congestion-aware forwarding:** MIXIT's forwarding algorithm forwards coded symbols via paths that have both high delivery probabilities and small queues, unlike previous schemes that don't consider congestion information.

---

[1] A symbol is a small sequence of bits (typically a few bytes) that the code treats as a single value.

MIXIT synthesizes ideas developed in previous work, in particular opportunistic routing (ExOR [1] and MORE [2]) and partial packet recovery [8]. The key to this synthesis is its symbol-level network code. The result is a scheme that can perform much better than packet-level opportunistic routing, which cannot benefit from partial receptions. For example, these schemes often capitalize on sporadic receptions over long links, but these long links are inherently less reliable and likely to exhibit symbol errors. By insisting on forwarding only fully correct packets, current opportunistic protocols miss the bulk of their opportunities, as we show in our experiments. Similarly, prior proposals for exploiting partially correct receptions, like PPR [8], SOFT [29], and H-ARQ [16], limit themselves to a single wireless hop, trying to make that hop reliable. In contrast, we advocate jettisoning reliable link-layer error detection and recovery altogether in wireless mesh networks.

We evaluate MIXIT using our software radio implementation on a 25-node testbed running the Zigbee (802.15.4) protocol. The main experimental results are as follows:

- MIXIT achieves a $2.8\times$ gain over MORE, a state-of-the-art packet-based opportunistic routing protocol under moderate load. The gain over traditional routing is even higher, $3.9\times$ better aggregate end-to-end throughput. At lighter loads the corresponding gains are $2.1\times$ and $2.9\times$.

- MIXIT's gains stem from two composable capabilities: symbol-level opportunistic routing, and higher concurrency, which we find have a multiplicative effect. For example, separately, they improve throughput by $1.5\times$ and $1.4\times$ over MORE; in concert, they lead to the $2.1\times$ gain.

- Congestion-aware forwarding accounts for 30% of the throughput gain at high load.

MIXIT is a layered architecture that demonstrates cross-layer collaborations using clean interfaces: the network code can run atop any radio and PHY that provides SoftPHY hints, the system can run with any MAC protocol (though ones that aggressively seek concurrency perform better), and the routers are oblivious to the error-correcting code. This modular separation of concerns eases implementation.

2 Related Work

The idea of relaying partially correct information has been examined in information theory [14]. MIXIT builds on the intuition, but with two important differences that admit a practical design. First, intermediate nodes use SoftPHY hints to "clean" the symbols before processing and forwarding them, rather than just receiving, combining, and forwarding information at the signal level. Second, nodes use intra-flow symbol-level network coding, which allows them to coordinate and collaborate without requiring finely synchronized transmissions that many "distributed antenna" approaches entail.

MIXIT builds on prior work on opportunistic routing [1, 2], cooperative spatial diversity [18, 14], and wireless network coding [11]. In particular, it shares the idea of intra-flow network coding with MORE [2], but with three key differences: first, MORE operates on packets and cannot deal with packets with errors; second, MIXIT's symbol-level network code is an end-to-end rateless *error correcting code* while MORE's network code cannot correct errors; and third, MIXIT designs a MAC which exploits the looser constraints on packet delivery to significantly increase concurrent transmissions, MORE uses carrier sense and requires correct packet delivery which prevents
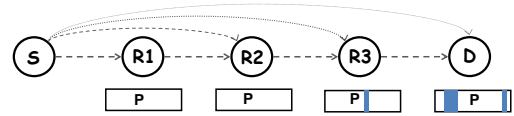
Figure 1: Example of opportunistic partial receptions: The source, S, wants to deliver a packet to the destination, D. The figure shows the receptions after S broadcasts its packet, where dark shades refer to erroneous symbols. The best path traverses all routers $R1$, $R2$ and $R3$. Traditional routing makes $R1$ transmit the packet ignoring any opportunistic receptions. Packet-level opportunistic routing exploits the reception at $R2$ but ignores that most of the symbols have made it to $R3$ and D. MIXIT exploits correctly received symbols at $R3$ and D, benefiting from the longest links.

it from achieving high concurrency. MIXIT's network code also builds on recent advances in extending network coding to scenarios with errors and adversaries [7, 12]. In contrast to all these schemes, MIXIT only codes over symbols above a certain confidence threshold, while using coding coefficients that reduce overhead.

MIXIT also builds on prior work on "soft information", whose benefits are well known [25, 4, 27]. Soft information refers to the confidence values computed in some physical layers when it decodes symbols. Recent work [8] has developed the SoftPHY interface to expose this information to higher layers in a PHY-independent manner by annotating bits with additional hints. Thus far, the use of these hints at higher layers has been limited to improving link reliability by developing better retransmission schemes [8] or to combine confidence values over a wired network to reconstruct correct packets from erroneous receptions [29]. In contrast, MIXIT uses SoftPHY hints in a new way, eschewing link-layer reliability in favor of spatial diversity to achieve high throughput and reliability.

MIXIT is philosophically similar to analog and physical layer network coding [10, 22], but it operates on symbols (i.e., bits) rather than signals; this difference is important because making a soft digital decision at an intermediate node improves efficiency by preventing the forwarding of grossly inaccurate information. And more importantly, it is a simpler design that fits in well with a layered architecture, so one can use the same network layer with a variety of physical layer schemes and radio technologies. MIXIT uses SoftPHY to propagate cross-layer information using a clean, PHY-independent interface.

3 Motivating Examples

This section discusses two examples to motivate the need for mechanisms that can operate on symbols that are likely to have been received correctly (i.e., on partial packets). These examples show two significant new opportunities to improve throughput: far-reaching links with high bit-error rates that allow quick jumps towards a destination *even when they might never receive entire packets correctly*, and increased concurrency using a more aggressive MAC protocol that induces higher bit-error rates than CSMA. The underlying theme in these examples is that one can improve throughput by allowing, and coping with, higher link-layer error rates.

First, consider Fig. 1, where a source, S, tries to deliver a packet to a destination, D, using the chain of routers $R1$, $R2$, and $R3$. It is possible that when the source broadcasts its packet, $R1$ and $R2$ hear the packet correctly, while $R3$ and D hear the packet with some bit errors. Traditional routing ignores the "lucky" reception at $R2$ and insists on delivering the packet on the predetermined path, i.e., it makes $R1$ forward the packet to $R2$ again. In contrast, recent opportunistic routing protocols (such as ExOR) capitalize on such lucky receptions (at $R2$) to make long jumps towards the destination, saving transmissions.
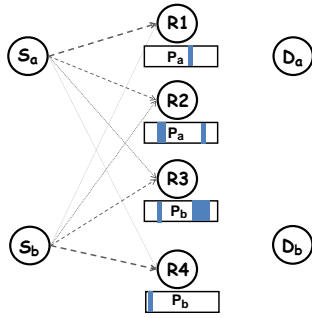
Figure 2: Concurrency example: The figure shows the receptions when the two sources transmit concurrently. Without MIXIT, the two sources S_a and S_b cannot transmit concurrently. MIXIT tolerates more bit errors at individual nodes, and hence is more resilient to interference, increasing the number of useful concurrent transmissions.

By insisting on forwarding fully correct packets, however, current opportunistic protocols miss a large number of opportunities to save transmissions and increase throughput; in particular, they don't take advantage of all the correct bits that already made it to $R3$ and even to the destination, D. Moreover, because of spatial diversity [18, 25], the corrupted bits at $R3$ and D are likely in different positions. Thus, $R3$ has to transmit only the bits that D did not receive correctly for the destination to get the complete packet. A scheme that can identify correct symbols and forward them has the potential to drastically reduce the number of transmissions required to deliver a packet.

Next, consider an example with potential concurrency as in Fig. 2, where two senders, S_a and S_b, want to deliver a packet to their respective destinations, D_a and D_b. If both senders transmit concurrently, the BER will be high, and no router will receive either packet correctly. Because current opportunistic routing protocols insist on correct and complete packets, the best any MAC can do is to make these senders transmit one after the other, consuming two time slots.

But interference is not a binary variable. In practice, different routers will experience different levels of interference; it is likely the routers close to S_a will receive packet, P_a, with only a few errors, while those close to S_b will receive packet P_b, with only some errors. A scheme that can identify which symbols are correct and forward only those groups of bits can exploit this phenomenon to allow the two senders to transmit concurrently and increase throughput. It can then "funnel" the correct symbols from the routers to the destination.

MIXIT aims to realize these potential benefits in practice. It faces the following challenges:

- How does a router classify which symbols in each received packet are likely correct and which aren't?
- Given the overlap in the correct symbols at various routers, how do we ensure that routers do not forward the same information, wasting bandwidth?
- How does a router forward traffic among its downstream nodes to maximize throughput and avoid creating hotspots?
- When is it safe for nodes to transmit concurrently?
- How do we ensure that the destination recovers a correct and complete version of the source's data?

The rest of this paper presents our solutions to these problems in the context of the MIXIT architecture, which we describe next.

4 MIXIT Architecture

MIXIT is a layered architecture for bulk transfer over static mesh networks. The layers are similar to the traditional PHY, link and network layers, but the interfaces between them, as well as the functions carried out by the network layer, are quite different. The physical and link layers deliver all received data to the network layer, whether or not bits are corrupted. Each packet has a MIXIT header that must be received correctly because it contains information about the destination and other meta-data; MIXIT protects the header with a separate forward error correction (FEC) code that has negligible overhead.

Rather than describe each layer separately, we describe the functions carried out at the source, the forwarders, and the destination for any stream of packets.

4.1 The Source

The transport layer streams data to the network layer, which preprocesses it using an error-correcting code as described in §9. The network layer then divides the resulting stream into batches of K packets and sends these batches to the destination sequentially. Whenever the MAC permits, the network layer creates a different random linear combination of the K packets in the current batch and broadcasts it.

MIXIT's network code operates at the granularity of symbols, which we define as a group of consecutive bits of a packet. The group could be the same collection of bits which are transmitted as a single physical layer symbol (*PHY symbol*) by the modulation scheme (e.g., groups of 4 bits in a 16-QAM scheme), or it could be larger in extent, covering a small number of distinct PHY symbols. The jth symbol in a coded packet, s'_j, is a linear combinations of the jth symbols in the K packets, i.e., $s'_j = \sum_i v_i s_{ji}$, where s_{ji} is the jth symbol in the ith packet in the batch and v_i is a per-packet random multiplier. We call $\vec{v} = (v_1, \ldots, v_K)$ the *code vector* of the coded packet. Note that every symbol in the packet created by the source has the same code vector.

The source adds a MIXIT header to the coded packet, protects it using FEC, and broadcasts it. The header describes which symbols were coded together. This description is easy to specify at the source because all symbols in a coded packet are generated using the packet's code vector, $\vec{v}$. The header also contains an ordered list of forwarding nodes picked from its neighbors, each of which is closer to the destination according to the metric described in §7.

4.2 The Forwarders

Each node listens continuously whenever it is not transmitting, attempting to decode whatever it hears. When the PHY is able to decode bits, it passes them up together with SoftPHY hints that reflect its confidence in the decoded bits. The network layer gets this information and uses it to classify symbols into *clean* and *dirty* ones. A clean symbol is one that is likely to be correct, unlike a dirty one. §5 describes how the MIXIT network layer classifies symbols.

When a node gets a packet without header errors, it checks whether it is mentioned in the list of forwarders contained in the header. If so, the node checks whether the packet contains new information, i.e., is "innovative" [13]. A packet is considered innovative if its code vector $\vec{v}$ is linearly independent of the vector of the packets the node has previously received from this batch. Checking for independence is straightforward using Gaussian elimination over these short vectors [13]. The node ignores non-innovative packets, and stores the innovative packets it receives from the current batch, preserving the "clean" and "dirty" annotations.

When forwarding data, the node creates random linear combinations of the clean symbols in the packets it has heard from the same batch, as explained in §6, and broadcasts the resulting coded packet. It also
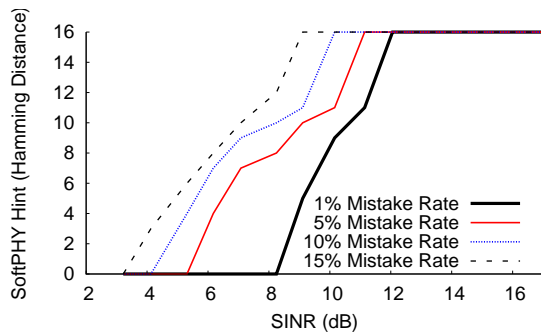
Figure 3: Decision boundary for different mistake rates as a function of SINR. At high SINR (> 12dB), all PHY symbols with Hamming distance less than 16 (the maximum possible in the Zigbee physical layer), will satisfy the mistake rate threshold. But at intermediate SINRs (5-12 dB), the PHY symbols have to be picked carefully depending on the mistake rate threshold.

decides how much each neighbor should forward to balance load and maximize throughput, as described in §7.

Any MAC protocol may be used in MIXIT, but the scheme described in §8.1 achieves higher concurrency than standard CSMA because it takes advantage of MIXIT's ability to cope with much higher error rates than previous routing protocols.

4.3 The Destination

MIXIT provides a rateless network code. Hence, the destination simply collects all the packets it can hear until it has enough information to decode the original data as described in §9. Furthermore, MIXIT provides flexible reliability semantics. Depending on application requirements, the destination can decide how much information is enough. For example, if the application requires full reliability the destination waits until it can decode 100% of the original symbols, whereas if the application requires 90% reliability, the destination can be done once it decodes 90% of the original symbols. Once the destination decodes the required original symbols, it sends a batch-ack to the source. The ack is sent using reliable single path routing, and causes the source to move to the next batch. For the rest of the paper, we will assume that the destination wants 100% reliability.

5 Classifying Received Symbols

MIXIT operates over symbols, which are groups of *PHY symbols*. A symbol is classified as *clean* if none of the constituent PHY symbols are erroneous with a probability higher than γ. It is classified *dirty* otherwise. We call the threshold γ, the *mistake rate*, and it is a configurable parameter of the system. To satisfy the mistake rate threshold, MIXIT's network layer picks a decision boundary on the soft values [8] of the PHY symbols. If all constituent PHY symbols in our symbol have soft values below this decision boundary, then the symbol is classified as clean, else it is dirty. The decision boundary depends on the mistake rate as well as the channel SINR [29, 25].

Fig. 3 supports this argument. The figure is generated using a GNU software radio implementation of the Zigbee protocol (see §10). The figure plots the decision boundary on soft values of PHY symbols for varying SINR at different mistake rates of 1%, 5%, 10% and 15%. Clearly the boundary depends both on the mistake rate as well as the SINR. The SINR measures the channel noise and interference, and hence reflects how much we should trust the channel to preserve the correlation between transmitted and received signals [29]. Factoring

in the specified mistake rate, we can use the above map to pick the right decision boundary to classify symbols.

MIXIT uses the SoftPHY interface proposed in [8], which annotates the decoded PHY symbols with confidence values and sends them to higher layers. We also augment the interface to expose the SINR. The SINR can be estimated using standard methods like that in [10]. The map in Fig. 3 can be computed offline, since the relationship between SINR, the confidence estimate, and the decision boundary is usually static [17]. The MIXIT network layer uses the PHY information to classify symbols as clean and dirty, and then performs symbol-level network coding over the clean symbols as described in the next section.

6 The MIXIT Network Code

When the MAC permits, the node may forward a coded packet. The symbols in a coded packet are linear combinations of the clean symbols received in packets from the same batch. To see how the coding works let us look at an example.

6.1 MIXIT in Action

Consider the scenario in Fig. 4, where the source S wants to deliver two packets, P_a and P_b, to the destination. Let the bit error rate (BER) be relatively high such that when the source S broadcasts P_a and P_b, the nodes in the network receive some symbols in errors. The network layer at each node classifies the symbols as either clean or dirty using the SoftPHY hints as described in §5. Fig. 4 illustrates the dirty symbols using shaded cells.

The objective of our symbol-level codes is to minimize the overhead required to funnel the clean symbols to their destination. Specifically, most symbols are received correctly by both $R1$ and $R2$. Hence, without additional measures, the routers will transmit the same symbols to the destination, wasting wireless capacity. To avoid such waste, MIXIT makes the routers forward random linear combinations of the clean symbols they received. Assuming a_i and b_i are the ith symbols in P_a and P_b respectively, router R picks two random numbers α and β, and creates a coded packet P_c, where the ith symbol, c_i is computed as follows:

$$c_i = \begin{cases} \alpha a_i + \beta b_i & \text{if } a_i \text{ and } b_i \text{ are clean symbols} \\ \alpha a_i & \text{if } a_i \text{ is clean and } b_i \text{ is dirty} \\ \beta b_i & \text{if } a_i \text{ is dirty and } b_i \text{ is clean.} \end{cases}$$

If both a_i and b_i are dirty, no symbol is sent. Similarly, $R2$ generates a coded packet P_d by picking two random values α' and β' and applying the same logic in the above equation. Since $R1$ and $R2$ use random coefficients to produce the coded symbols, it is unlikely that they generate duplicate symbols [5].

When R and R' broadcast their respective packets, P_c and P_d, the destination receives corrupted versions where some symbols are incorrect, as shown in Fig. 4. Thus the destination has four partially corrupted receptions: P_a and P_b, directly overheard from the source, contain many erroneous symbols; and P_c and P_d, which contain a few erroneous symbols. For each symbol position i, the destination needs to decode two original symbols a_i and b_i. As long as the destination receives two uncorrupted independent symbols in location i, it will be able to properly decode [5]. For example, consider the symbol position $i = 2$, the destination has received:

$$\begin{aligned} c_2 &= \alpha a_2 + \beta b_2 \\ d_2 &= \alpha' a_2. \end{aligned}$$
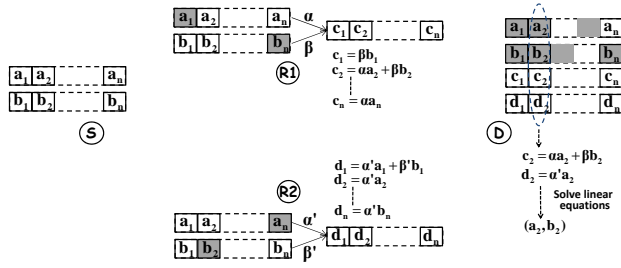
Figure 4: Example showing how MIXIT works: The source broadcasts P_a and P_b. The destination and the routers, $R1$ and $R2$, receive corrupted versions of the packets. A shaded cell represents a dirty symbol. If $R1$ and $R2$ forward the clean symbols without coding, they generate redundant data and waste capacity. With symbol-level network coding, the routers transmit linear combinations of clean symbols, ensuring that they forward useful information to the destination.

Given that the header of a coded packet contains the multipliers (e.g., α and β), the destination has two linear equations with two unknowns, a_2 and b_2, which are easily solvable (the details of the decoder are explained in §9). Once the destination has decoded all symbols correctly, it broadcasts an ACK, causing the routers to stop forwarding packets.

6.2 Efficient Symbol-Level Codes

The difficulty in creating a network code over symbols is not the coding operation, but in how we express the code efficiently. The length of a symbol is small, one or a few bytes. The MIXIT header in the forwarded packet has to specify how each symbol is derived from the native symbols so that the destination can decode. If all symbols in a packet are multiplied by the same number, then effectively we have a packet-level code, which can be easily expressed by putting the multiplier in the header. However, in MIXIT we want to code clean symbols and ignore dirty ones; i.e., only clean symbols are multiplied by a non-zero number.

Consider a simple example where the batch size is $K = 2$ with the two packets; P_a and P_b. Say that our forwarder has received two coded packets $P_c = \alpha P_a + \beta P_b$ and $P_d = \alpha' P_a + \beta' P_b$. Now our forwarder picks two random numbers v_1 and v_2 and creates a linear combination of the two packets it received.

$$P = v_1 P_c + v_2 P_d = (v_1\alpha + v_2\alpha')P_a + (v_1\beta + v_2\beta')P_b$$

Thus, the newly generated packet has a code vector $\vec{v} = (v_1\alpha + v_2\alpha', v_1\beta + v_2\beta')$. This vector would be sufficient to describe the whole packet if the forwarder received only clean symbols. Specifically, the clean symbol in the jth position in packet P, called s_j, is coded as follows:

$$
\begin{aligned}
s_j &= v_1 c_j + v_2 d_j, \quad \text{where } c_i \text{ and } d_j \text{ are clean} \\
&= (v_1\alpha + v_2\alpha')a_j + (v_1\beta + v_2\beta')b_j
\end{aligned}
$$

But because some received symbols are dirty, we need a more detailed description of how individual symbols in the packet P are derived from the native symbols. Depending on whether the forwarder has cleanly received the jth symbols in P_c and P_d, called c_j and d_j respectively, the generated symbol s_j might take one of four possible values, with respect to the native symbols.

$$
s_j = \begin{cases}
(v_1\alpha + v_2\alpha')a_j + (v_1\beta + v_2\beta')b_j & c_j \text{ and } d_j \text{ are clean} \\
v_1\alpha a_j + v_1\beta b_j & \text{only } c_j \text{ is clean} \\
v_2\alpha' a_j + v_2\beta' b_j & \text{only } d_j \text{ is clean} \\
0 \times a_j + 0 \times b_j & c_j \text{ and } d_j \text{ are dirty}
\end{cases} \tag{1}
$$
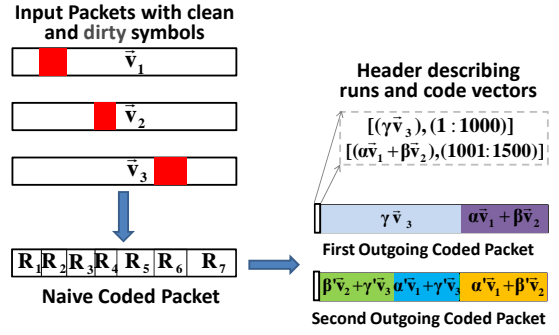
Figure 5: Creating coded packets with longer runs: The forwarder received 3 packets with code vectors v_1, v_2 and v_3. All packets contain dirty symbols represented as shaded areas. Naively coding over all clean received symbols results in a coded packet with 7 different runs. However, by ignoring some of the clean symbols, the node can generate coded packets with much fewer runs.

Each different value of the symbol is associated with a different code vector, the header has to specify for each symbol in a transmitted packet what is the symbol's code vector.

We address this issue using the following two mechanisms.

(1) Run-Length Encoding: Because wireless errors are bursty [18, 28], a sequence of consecutive symbols will have the same code vector. We can therefore use run-length encoding to describe the encoding of the transmitted symbols in an efficient manner. The header specifies a sequence of runs, each of which is described as `[(Code Vector of run),(Runstart:Runend)]`. For example, in Fig. 5, the header of the first outgoing coded packet will specify two runs, $[(\gamma\vec{v}_3),(1,1000)]$ and $[(\alpha\vec{v}_1 + \beta\vec{v}_2),(1001,1500)]$.

(2) Pick codes that give longer runs: We force the overhead to stay small by intentionally discarding clean symbols that fragment our run-length-encoding. Said differently, a forwarder can decide to ignore some clean symbols to ensure the header has longer runs of symbols with the same code vector, and thus can be encoded efficiently.

Consider the example in Fig. 5, where the forwarder has received 3 packets, each with some dirty symbols. Naively, applying the symbol-level network code along with the run-length encoding described above, we get a coded packet that has seven different runs. But, we can create fewer runs of longer lengths by ignoring some clean symbols in the coding procedure. For example, in the first five runs in the naive coded packet, we can ignore clean symbols from the first and second received packets. As a result, the five runs would coalesce to a single longer run with the code vector $\gamma\vec{v}_3$, where γ is a random multiplier and $\vec{v}_3$ is the code vector of the third received packet. Similarly for the last two runs, if we ignore clean symbols from the third received packet, we are left with a single longer run with the code vector $\alpha\vec{v}_1 + \beta\vec{v}_2$, where α and β are random multipliers and $\vec{v}_1$ and $\vec{v}_2$ are the code vectors of the first and second received packets. The resulting coded packet shown in Fig. 5 has only two runs with two code vectors, and requires less overhead to express.

But, what if the forwarder has to transmit a second coded packet? One option is to ignore the same set of clean symbols as above, but use different random multipliers, α', β', γ'. We would get a coded packet with two runs and their code vectors being $\gamma'\vec{v}_3$ and $\alpha'\vec{v}_1 + \beta'\vec{v}_2$. But this transmission will be wasteful, since the symbols in the first run are *not innovative* w.r.t the first coded packet the node already transmitted ($\gamma'\vec{v}_3$ is not linearly independent of $\gamma\vec{v}_3$). The solution is to split the first long run into two smaller runs by including clean symbols from the first and second packets, which we had previously ignored. The second coded packet, shown in Fig. 5 has 3 runs with 3 different code vectors $\beta'\vec{v}_2 + \gamma'\vec{v}_3$, $\alpha'\vec{v}_1 + \gamma'\vec{v}_3$ and $\alpha'\vec{v}_1 + \beta'\vec{v}_2$. The new packet

is innovative w.r.t the previously transmitted coded packet, and uses lower overhead in describing the codes.

6.3 Dynamic programming to minimize overhead

We now present a systematic way to minimize the number of runs in a coded packet, while ensuring that each packet is innovative with respect to previously transmitted coded packets. We formalize the problem using dynamic programming. Let there be n input packets, from which we create the *naive coded packet*, as shown in the previous example. Say the naive packet contains the runs $R_1 R_2 \ldots R_L$. The optimization attempts to combine consecutive runs from the naive coded packet into a single run, whose symbols all have the same code vector by ignoring some of the input clean symbols. Let C_{ij} be the combined run that includes the runs $R_i \ldots R_j$ from the naive coded packet. Note that the combined run C_{ii} is the same as run R_i.

Next, we show that each combined run can be assigned a cost, and that the optimization problem that minimizes the number of innovative combined runs exhibits the "optimal substructure" property, i.e., the cost of a combined run can be derived from the cost of two sub-runs.

The goal is to create an outgoing coded packet out of the smallest number of combined runs, while ensuring that the information we send out is innovative. Thus, we can formulate the cost of a combined run as follows:

$$Cost(C_{ij}) = \min \left\{ \mathbf{f}(C_{ij}), \min_{i<k<j} \{Cost(C_{ik}) + Cost(C_{kj})\} \right\} \quad (2)$$

where $\mathbf{f}(C_{ij})$ is given by:

$$\mathbf{f}(C_{ij}) = \begin{cases} \sum_i^j |R_i| & \text{if } C_{ij} \text{ is not innovative} \\ (2 \log S)/8 + K & \text{otherwise} \end{cases} \quad (3)$$

Intuitively, the function $\mathbf{f}(C_{ij})$ says that if the combined run C_{ij} is not innovative with respect to previous transmissions, the cost is the number of symbols in that combined run. But if the combined run is innovative with respect to previous transmissions, its cost is just the number of bytes required to describe it. This requires describing the start and end of the combined run, which can be done using $(2 \log S)/8$ bytes, where S is the packet size, and describing the combined run's code vector, which can be done using K bytes, where K is the batch size. The second component in Eq. 2 checks if splitting the combined run C_{ij} into two smaller runs incurs a smaller cost, and if it does, it finds the best way to split it.

The forwarder computes the dynamic program top-down using a table to memoize the costs. Because the algorithm coalesces runs in the naively coded packet, the table has at most as many entries as there are combined runs. The worst case complexity of the algorithm is $O(L^3)$, but in practice it runs faster due to the following heuristic. In Eq. 2, if C_{ij} is innovative, we do not need to check whether splitting it reduces the cost because $\mathbf{f}(C_{ij})$ will always be lower than the cost of the two sub runs, whose cost will at least be $2\mathbf{f}(C_{ij})$. Typically, the DP takes under a millisecond to run for a packet with $L \approx 15 - 20$.

7 Congestion-Aware Forwarding

In general, because wireless is a broadcast medium several downstream routers will hear any given symbol without error. For each symbol, the ideal situation is for the downstream forwarder with the best path quality to the destination to forward the symbol (after coding it). For example, in Fig. 6(a), routers $R1$ and $R2$ hear all the symbols from $S1$. However, $R2$ should be the one to forward the symbols because it can deliver them to the destination in fewer transmissions.
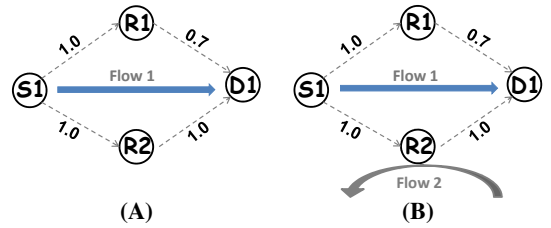
Figure 6: Example of congestion-aware forwarding: If there is a single flow in the network, $S1$ should always send all his traffic through $R2$ since he has the better link to the destination. But if $R2$ is involved in a second flow, then $S1$ should also send some of his traffic through $R1$ to avoid creating a bottleneck at $R2$.

Path quality is not the only consideration in making this decision because one ultimately cares about the *time* it takes for a symbol to reach the destination. If a path has high quality (as measured by a low error rate), but also has long queues at its forwarders, it would not be advisable to use it. Fig. 6(b) shows an example where a second flow being forwarded through $R2$ causes $R2$'s queues to grow, so having $R1$ forward some of its traffic would improve performance and avoid creating a bottleneck at $R2$.

These requirements suggest the following approach for a node to decide how its downstream forwarders should forward on its behalf. First, for each path (via a downstream node), determine the expected time it would take to successfully send a symbol along that path. This time, which we call C-ETS (for "congestion-aware ETS"), incorporates both path quality and node queue lengths (backlog). C-ETS via a downstream node i to a destination d is computed as C-ETS$(i,d) = $ PQ$(i,d) + kQ(i)$. Here, PQ is the path quality from i to d, which depends on the symbol delivery probabilities and captures the time taken to deliver a symbol to the destination in the absence of any queueing. In our implementation, we approximate this term using the ETS metric (defined as the expected number of transmissions required to deliver a symbol), instead of using a more exact formula for the expected number of transmissions using opportunistic routing [2].[2] $Q(i)$ is the total number of symbols (backlog) across all flows queued up at node i yet to be transmitted (k is a constant dimensional scaling factor that depends on the time it takes to transmit one symbol).

We now discuss how a node decides how many of its queued up symbols, $Q(i,f)$ for flow f, each downstream node should forward. (Because of the random network code, we don't have to worry about downstream nodes sending the exact same information.) The high-level idea is to favor nodes with smaller C-ETS values, but at the same time apportioning enough responsibility to every downstream node because no link or path is loss-free in general. Each node assigns responsibility to its downstream nodes by assigning credits. Credit determines the probability with which the downstream node should forward symbols belonging to the flow when they receive transmissions from the node. The downstream node with best the C-ETS has credit 1; the next-best node has credit $(1 - p_1)$, where p_1 is the symbol delivery probability to the best downstream node; the best one after that has credit $(1 - p_1)(1 - p_2)$, and so on. What we have done here is to emulate, in expectation, the best downstream node sending all the symbols it hears, the next-best one only forwarding a fraction that the best one may not have heard, and so on, until all the nodes down to the worst neighbor have some small responsibility.

How many transmissions should the node make? The node should make enough transmissions to make sure that every queued up symbol reaches some node with a lower C-ETS metric. If the symbol delivery

---

[2]Computing the ETS metric is simpler and does not change the paths used by much.

probability to downstream neighbor j is p_j, then the probability that *some* downstream neighbor gets any given symbol is $P = 1 - \prod_j(1 - p_j)$. Hence in expectation, the number of coded symbols from a batch that a node would have to send per queued up symbol is equal to $1/(1-P)$ before some downstream node gets it. Each node achieves this task by maintaining a decrementing per-flow *Transmit_Counter*, throttling transmission of the batch when its value reaches 0.

The above intuitions are formalized in Alg. 1.

---

1 Computing credit assignment at node i

> **while** $Q(i,f) > 0$ **do**
> > Update C-ETS of downstream nodes from overheard packets
> > Sort downstream nodes according to their C-ETS
> > $P_{left} = 1$
> > **for** node j in set of downstream nodes sorted according to C-ETS **do**
> > > $credit\_assgn(j) = P_{left}$
> > > $P_{left} = P_{left} * (1 - p(i,j))$
> > Increment *Transmit_Counter* of flow f by $1/(1-P_{left})$
> > Decrement $Q(i,f)$ by 1

Distributed Protocol: Each node, i, periodically measures the symbol delivery probabilities $p(i,j)$ for each of its neighbors via probes. These probabilities are distributed to its neighbors using a link state protocol. Node i includes the computed *credit_assgn* for each of its downstream nodes in the header of every packet it transmits. When downstream nodes receive a packet, they update their $Q(i,f)$ for that flow by the amount specified in the header. Further, whenever node i transmits a packet, it includes its C-ETS to the corresponding destination in the header. Upstream nodes which overhear this packet, use the C-ETS value in their credit assignment procedure.

The algorithm above improves on the routing algorithms used in prior packet based opportunistic routing protocols like MORE [2] in two ways. First, we use queue backlog information explicitly to avoid congested spots and balance network-wide load, prior work ignores congestion. Second, the algorithm works at the symbol-level, which is the right granularity for performing opportunistic routing on symbols. The algorithm is similar in spirit to theoretical *back-pressure* [19] ideas, but the exact technique is different and simpler. We also present an actual implementation and evaluation of this algorithm in §11.

8 Increasing Concurrency

Current wireless mesh networks allow a node to transmit only when they are sure that they can deliver the *packet* to the intended next hop with high probability. MIXIT however, has looser constraints:

1. It does not require the delivery of correct packets; it can work with partially correct packets.

2. Because of its opportunistic nature, MIXIT only needs to ensure that every symbol reaches *some* node closer to the destination than the transmitter; it does not need to ensure that a specific node gets the correct symbols.

MIXIT exploits the above flexibility to increase concurrency without affecting end-to-end reliability, improving throughput by enabling a more pipelined transmission pattern. MIXIT's concurrency design has two components: determining when concurrent transmissions are beneficial and building a distributed protocol to take advantage of concurrency opportunities. We describe both components below.

8.1 When should two nodes transmit concurrently?

MIXIT, similar to conflict maps [26], determines if two nodes should transmit concurrently by predicting the throughput under concurrent transmissions and comparing it to the throughput when the nodes transmit separately. The nodes independently pick the strategy with the higher expected throughput. Specifically, let $n1$ and $n2$ be two nodes transmitting packets of two flows l and k. $Ne(n1,l)$ and $Ne(n2,k)$ are the corresponding sets of downstream nodes for $n1$ and $n2$ for the respective flows. Symbol delivery probabilities on any link will depend on whether these nodes transmit concurrently or not. Let $p^c(i,j)$ be the symbol delivery probability on link (i,j) when the two nodes transmit concurrently and $p(i,j)$ when they don't. The symbol delivery likelihoods achieved by node $n1$ for flow l with and without concurrent transmissions are given by

$$
\begin{aligned}
D^c(n1,l) &= 1 - (\prod_{j \in Ne(n1,l)}(1 - p^c(n1,j))) \\
D(n1,l) &= 1 - (\prod_{j \in Ne(n1,l)}(1 - p(n1,j)))
\end{aligned} \tag{4}
$$

The symbol delivery likelihood is the probability that at least one node in $Ne(n1,l)$ receives the symbol correctly when node $n1$ transmits. The symbol delivery likelihood depends on other concurrent traffic, and could differ if $n2$'s transmission interferes with $n1$'s. Similarly, $n2$ can compute its symbol delivery likelihood under both conditions.

Each node then computes the following *concurrency condition*:

$$
D^c(n1,l) + D^c(n2,k) > (D(n1,l) + D(n2,k))/2 \tag{5}
$$

The above equation compares overall delivery likelihood under the two scheduling strategies. If the above condition is true, it implies that more information gets delivered per time slot when nodes transmit concurrently than when they transmit separately. Each node independently evaluates the above condition and decides its strategy.[3]
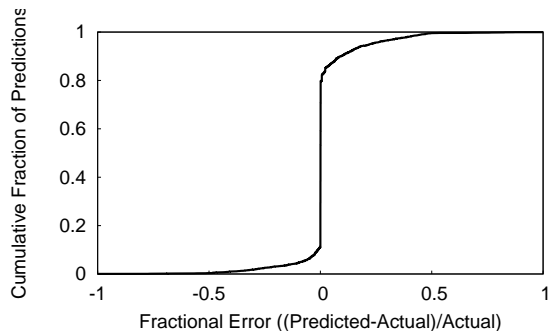
8.2 Estimating symbol delivery probabilities

The concurrency condition above depends on the symbol delivery probabilities. Empirically measuring these probabilities for all pairs of concurrent transmissions has $O(N^2)$ cost, where N is the number of nodes. Instead, MIXIT uses $O(N)$ empirical signal-to-noise ratio (SNR) measurements to predict these probabilities for any set of concurrent transmissions. The approach works as follows.

1. The SNR profile of the network is measured when there is little traffic. Each of the N nodes broadcasts probe packets in turn, while the rest of the nodes measure the received SNR and the fraction of correctly received symbols. The measurements are of the form $SNR(i,j)$ and $p(x)$, where $SNR(i,j)$ is the received SNR at j when i transmits and $p(x)$ is the fraction of correct symbols received when the SNR is x.

2. Nodes use the SNR profile to predict the signal-to-interference+noise ratio (SINR) at any node under concurrent transmissions. Specifically, if nodes $n1$ and $n2$ transmit concurrently, the SINR at node m is computed as $SINR(n1,n2,m) = SNR(n1,m) - SNR(n2,m)$ assuming $SNR(n1,m) > SNR(n2,m) \geq c$, where c is a threshold SNR below which no symbol can be decoded. The symbol delivery probability is then predicted to be $p(SINR(n1,n2,m))$, i.e., it is the same as if the signal was received at m with SNR of $SINR(n1,n2,m)$.

Fig. 7 plots the CDF of prediction errors using the above model. The results are from a 25-node testbed of GNURadio software nodes with

---

[3]The above conditions assumes a single radio transmission bit-rate; it can be adapted easily to handle variable bit-rates.

Figure 7: Prediction error CDF: The SNR based prediction model accurately predicts the symbol delivery probabilities under concurrent transmissions for 72% of the cases. The inaccuracies are primarily in cases where $(SNR(n1,m) - SNR(n2,m) < 4dB$, i.e., when concurrent transmissions will result in a signal being received with low SINR at the receivers.
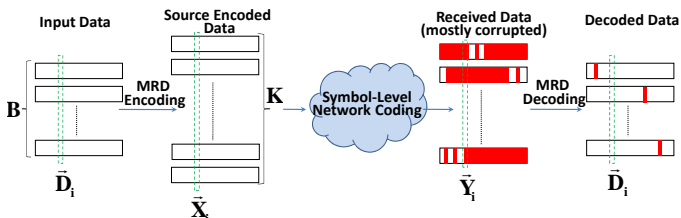
Figure 8: MIXIT's error correcting code design: The source preprocesses the original packets with a MRD code in groups of B symbols and transmits them. The network applies symbol-level network coding on clean symbols. Erroneous clean symbols may end up corrupting all the received symbols at the destination, but the destination can use the MRD code to decode most of the original data symbols.

USRP frontends, with two concurrent senders transmitting 802.15.4 packets. The figure demonstrates that the prediction model is quite accurate, with the inaccurate predictions occuring at low SINR (< 4 dB). But because the symbol delivery probability at low SINR is negligible, inaccuracies in the low SINR region do not affect performance. Furthermore, unlike prior proposals [24, 20] that try to predict packet delivery rates using a SINR model, MIXIT's model predicts symbol delivery likelihoods. The latter is simpler since packet delivery is a complex function of error rates, nature of interference etc. Finally, the concurrency condition is a binary decision, even if the predicted probabilities are slightly off, it is unlikely to affect the decision.

8.3 Distributed channel access protocol

A node uses a two-step procedure when it has packets enqueued for transmission. First, if it has not heard any on-going transmissions, it simply goes ahead and transmits. But if it has heard an on-going transmission, then it uses Eq. 5 to determine if it should transmit concurrently or defer until the on-going transmission has finished.

How does a node know which other nodes are transmitting at that time instant? Similar to prior work [8, 26], MIXIT encapsulates every packet with a header and trailer. The header includes the identity of the transmitting node, and the flow to which the packet belongs. Other nodes overhearing a packet use the header to identify the beginning of an active transmission and the trailer to signify the end.

9 Error Correction

Until now we have ignored the difference between clean and correct symbols and focused on delivering clean symbols to the destination.

But clean symbols can be incorrect. Moreover, an erroneous symbol that was incorrectly classified clean may end up corrupting other correct clean symbols due to network coding. Thus, the destination, could get all symbols corrupted due to a single clean but erroneous symbol. Fortunately, MIXIT comes with error correction capability that allows the destination to recover the original correct symbols. The error-correcting code is not affected even if all received symbols are corrupted; the only thing that matters is how many erroneous symbols were incorrectly classified clean. The code guarantees that if m erroneous symbols were incorrectly classified clean, then the destination needs only $B + 2m$ symbols to recover the original B symbols. This guarantee is theoretically optimal [30]. The code is simple, rateless and end-to-end; routers inside the network are oblivious to the existence of the error-correcting code.

MIXIT's error-correcting code is built on the observation that random network coding is *vector space preserving* [12]. Specifically, if we model the original data injected by the source as a basis for a vector space **V**, then the random network code acts only as a linear transformation **T** on the vector space. But vector spaces are preserved under linear transformations if no errors occur, and if errors do occur, the received vector space **U** is very close to the transmitted vector space **V** under an appropriately defined distance metric on vector spaces.

Recent work [12, 23, 7] has studied the problem of making network coding resilient to byzantine adversaries injecting corrupted packets. It has observed that low complexity Maximum Rank Distance (MRD) codes [3], with a small modification, can be applied to exploit the vector space observation and correct adversarial errors. The network coding in MIXIT is different, but the basic algorithm in MRD can be adapted to work with MIXIT's symbol-level network code. Fig. 8 shows the high level architecture of how MRD codes are integrated within MIXIT. The exact details of decoding MRD codes can be found in [23, 21, 3], we outline the main differences here:

- Symbol-level network coding along with the end-to-end MRD code functions as a rateless error-correcting code. The destination attempts to decode the original data vector $\vec{D}_i$ as soon as it receives $B < K$ coded symbols for that position. If no erroneous symbols had seeped through, then it will be able to recover the original correct data. If not, it simply waits to receive more coded symbols until it can decode. The code guarantees that if m erroneous symbols incorrectly classified as clean seeped through, then the destination can decode as soon as it receives $B + 2m$ coded symbols.

- MIXIT's rateless code provides flexible reliability semantics. Since the code works on groups of B symbols, there is no fate sharing across groups. Its likely that when the destination receives a few packets, it will be able to decode most of the groups of B symbols, but not some since they had more errors. Depending on the application, the destination could wait to receive more coded symbols until it can decode, or ignore the undecoded symbols and ask the source to proceed to the next batch by sending a *batch-ack* to the source.

10 Implementation

10.1 Packet Format

MIXIT inserts a variable length header in each packet, as shown in Fig. 9. The header is also repeated as a trailer at the end of the packet to improve delivery in the face of collisions [8]. The header contains the source and destination addresses, the flow identifier, and the the batch identifier. These fields are followed by a variable length *Code Vector Block*, which describes how the symbols in this packet have

Experiment	Section	Result
MIXIT in a lightly loaded network	11.2.1	MIXIT improves median throughput by 2.1× over MORE and 2.9× over SPR
Impact of concurrency	11.2.2	MIXIT exploits loose packet delivery constraints to increase concurrency.
Impact of symbol level diversity	11.2.2	MIXIT with plain carrier sense still outperforms MORE by 1.5×.
Impact of mistake rate threshold	11.2.3	MIXIT's error correcting code allows us to be flexible with the mistake rate. This reduces the fraction of correct symbols incorrectly labeled dirty and increases throughput.
Impact of batch size	11.2.4	MIXIT is insensitive to batch size, providing large gains for sizes as small as 8.
MIXIT in a congested network	11.3.1	MIXIT improves median throughput by 2.8× over MORE and 3.9× over SPR.
Impact of forwarding algorithm	11.3.2	MIXIT's congestion-aware forwarding prevents hotspots and keeps network capacity from dropping during congestion.

Table 1: A summary of the major experimental contributions of this paper.
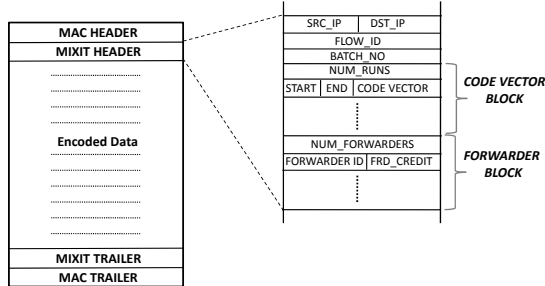
Figure 9: MIXIT's packet format.

been created. It has the format (Code Vector, Run Start, Run End); the values of these fields are obtained using the algorithm in §6.2. Following that is the variable length *Forwarder Block* that lists all the neighbors of this node ordered according to their C-ETS metrics. For each neighbor, the header also contains its credit assignment as described in §7. The *Code Vector Block* and the *Forwarder Block* are computed and updated by the forwarders. The other fields are initialized by the source and simply copied by each forwarder.

10.2 Node State

Each MIXIT node maintains per-flow state, which is initialized when the first packet from a flow that contains the node ID in the *Neighbor Block* arrives . The per-flow state includes:

- The *batch_buffer*, which stores the received clean symbols for each batch. This buffer is at most $K \times S$, where K is the batch size and S the packet size.

- The *credit_counter*, which stores the number of credits assigned to the node by the upstream neighbors for the batch. Upon the arrival of a packet from a node with a higher C-ETS, the node increments the credit by the corresponding credit assignment as indicated in the packet header.

- The *transmit_counter*, which is incremented by the credit assignment algorithm in §7. After a packet transmission, it decrements by one.

10.3 Control Flow

MIXIT's control flow responds to packet receptions. On the receiving side, whenever a packet arrives, the node checks whether it's ID is present in the *Forwarder Block*. If it is, then it updates the *credit_counter* for the corresponding batch of that flow by the credit assigned to it in the *Forwarder Block*. Next, the node picks out clean symbols from the received packet using the SoftPHY hints and adds them to the *batch_buffer*. If the credit is greater than one, it runs the credit assignment algorithm from §7. It then creates *transmit_counter*

coded packets using the technique in §6.2 and enqueues them. The MAC layer transmits these packets using the rule discussed in §8.1.

When the destination node receives a packet, it checks the symbol positions for which it has received atleast B coded symbols and decodes whichever of them it can. It sends a batch-ack to the source when it has decoded the required fraction (determined by the application's reliability requirements) of original symbols. The batch-ack is sent periodically until packets from the next batch start arriving.

11 Evaluation

We compare MIXIT with two routing protocols for wireless mesh networks: MORE, a state-of-the-art packet-level opportunistic routing protocol, and SPR, single path routing using the commonly used ETX metric. Our experimental results are summarized in Table 1.

11.1 Testbed

We use a 25-node indoor testbed deployed in a lab. Each node is a Zigbee software radio. The hardware portion of the node is a Universal Software Radio Peripheral [6] with a 2.4 GHz daughterboard, the remainder of the node's functions (demodulation, channel decoding, network coding etc) are implemented in software. The peak data rate on the link is 250 Kbits/s when there are no other transmissions in progress. Paths between nodes are between one and five hops long, and the SNR of the links varies from 5 dB to 30 dB. The average packet loss rate on links in our network is 23% for 1500 byte packets.

11.2 Single Flow

11.2.1 Throughput Comparison

Method: We run SPR, MORE, and MIXIT in sequence between 120 randomly picked source-destination pairs in our testbed. Each run transfers a 5 MByte file. The batch size of MIXIT is 12, but the error-correction preprocessing stage described in §9 converts it into 16 packets. To make a fair comparison, MORE uses a batch of 16 packets. We use the same batch sizes for MIXIT and MORE for all other experiments unless specifically noted otherwise. The packet size for all three protocols is 1500B. The mistake rate γ for MIXIT is fixed at 5% and the symbol size for MIXIT is 6 bytes unless otherwise noted. Before running an experiment, we collect measurements to compute pairwise packet delivery probabilities, which are then fed to SPR and MORE to be used in their route computations. The same measurement packets are used by MIXIT to compute the network's SNR profile as described in §8. We repeat the experiment for each source-destination pair five times and report the average throughput for each scheme.

Results: Fig. 10 plots the CDF of the throughput taken over 120 source-destination pairs in our testbed. MIXIT provides a median

throughput gain of $2.1\times$ over MORE and $2.9\times$ over SPR.

We note that MIXIT improves performance across the entire throughput range. Packet-based opportunistic routing protocols, like MORE, provide large throughput gains for dead spots, i.e., scenarios where all paths between the source and destination are of poor quality. The gains for high quality paths were relatively minor [1, 2]. Both MORE and ExOR exploit diversity at the packet level to build better quality links out of many bad links. But for source-destination pairs that are connected via good links, diversity does not help. Naturally, this makes one wonder whether MIXIT's gains over packet based opportunistic routing protocols arise from its ability to exploit concurrency, a question that we address in the next section.
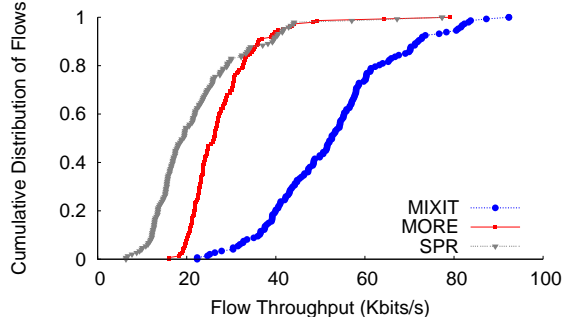

Figure 10: Throughput comparison: The figure shows that MIXIT has a median throughput gain of $2.1\times$ over MORE, the state-of-the-art packet level opportunistic routing protocol, and $2.9\times$ over SPR, a single path routing protocol based on the ETX metric.

11.2.2   Where do MIXIT's throughput gains come from?

MIXIT exploits both wireless diversity and concurrent transmissions. We would like to measure how much each of these components contributes to MIXIT's throughput gains.

Method: We first compare MIXIT with a modified version of MORE that takes advantage of concurrency at the packet level, which we call MORE-C. Like MORE, MORE-C performs packet based opportunistic routing. But MORE-C also allows nodes to transmit concurrently. To check whether two transmissions should be transmitted concurrently, MORE-C uses the same algorithm used by MIXIT and described in §8, but after it replaces symbol delivery probabilities with packet delivery probabilities.
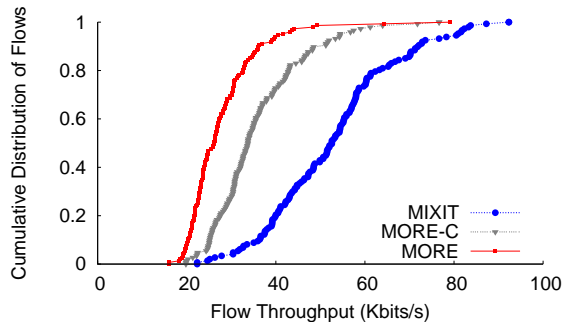

Figure 11: Impact of concurrency: The figure shows the throughput of MIXIT, MORE, and a concurrency-enabled version of MORE which we term MORE-C. Clearly concurrency helps but it is not sufficient to achieve the same throughput as MIXIT.

Results: Fig. 11 plots the CDF of the throughputs of MIXIT, MORE, and MORE-C taken over the same source-destination pairs as before. MIXIT provides a median throughput gain of $1.7\times$ over

MORE-C. The main result is that even when compared against a protocol that exploits both diversity and concurrency like MORE-C, MIXIT still does significantly better. The only extra property that MIXIT has beyond MORE-C is its ability to work at the symbol level. Is the median gain of $1.7\times$ over MORE-C due mainly to MIXIT's ability to exploit clean symbols, i.e., is symbol-level diversity the dominant contributor to MIXIT's overall throughput gain?

Method: To answer the above question, we prevent MIXIT from aggressively exploiting concurrent transmissions and use plain carrier sense. The intent is to limit its gains over MORE to be from being able to perform opportunistic routing over clean symbols. We call the resulting version MIXIT-CS.
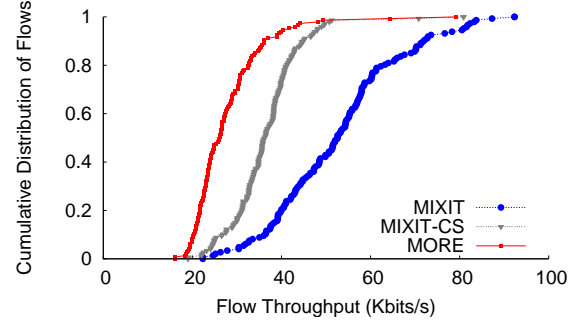
Figure 12: Throughputs for MIXIT with CS: The figure shows the throughput of MIXIT, MORE, and MIXIT-CS, a version of MIXIT which uses plain carrier sense and can only take advantage of symbol-level diversity. MIXIT-CS still performs better than MORE due to its ability to exploit long opportunistic receptions but with a few errors in them.

Results: Fig. 12 plots the CDF of the throughputs of MIXIT, MIXIT-CS and MORE. MIXIT-CS provides a median throughput gain of $1.5\times$ over MORE, i.e., significantly less gain than MIXIT. Thus, symbol-level diversity is not the dominant contributor to MIXIT's throughput gains. Indeed, comparing Fig. 12 with Fig. 11 shows that the overall gain of MIXIT over MORE is roughly *Gain of MIXIT-CS over MORE×Gain of MORE-C over MORE*, i.e $1.5 \times 1.4 = 2.1$. The multiplicative effect is due to the symbiotic interaction between concurrency and symbol-level opportunistic routing; concurrency tries to run the medium at high utilization and hence increases symbol error rate. But when the symbol error rate becomes high, almost every packet will have some symbols in error causing the whole packet to be dropped. Consequently, trying to exploit concurrency with a packet level protocol is limited by nature. Only a protocol that filters out incorrect symbols can push concurrency to its limits.

11.2.3   Impact of letting more errors through

Method: We evaluate how the threshold on classifying clean symbols affects throughput. As explained in §5, MIXIT has the flexibility to choose the threshold mistake rate γ. We vary this threshold and compare the average throughput. For the Zigbee protocol, the PHY symbol is 4 bits long, while the MIXIT symbol size is 6 bytes.

Results: Fig. 13 plots the average throughput across all source-destination pairs for different mistake rates. The average throughput surprisingly increases as we let more errors through! It peaks when the mistake rate is around 5% and drops at higher error rates.

This may sound counter intuitive, but recall that we are talking about a *probability* of error; if the router would know for sure which PHY symbols are incorrect, the best it can do is to drop all incorrect PHY symbols. But a PHY symbol that has a 5% chance of being in error has also a 95% chance of being correct. For our topology, at 5%
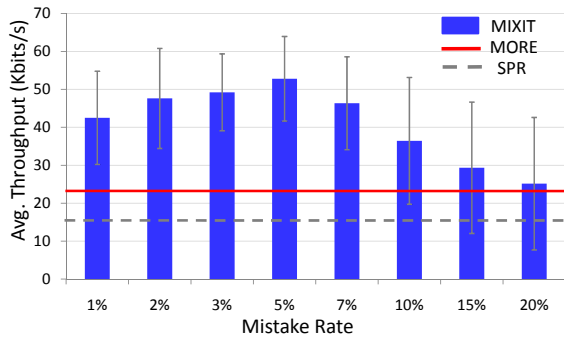
Figure 13: **Impact of changing the mistake rate:** The figure shows that many mistake rate thresholds provide significant throughput gains and hence MIXIT performs reasonably well even if the network is configured with a suboptimal threshold.
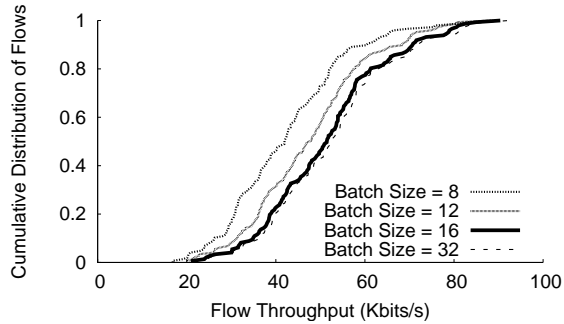
Figure 14: **Impact of batch size:** The figure shows the CDF of the throughput achieved by MIXIT for different batch sizes. It shows that MIXIT is largely insensitive to batch sizes.

mistake rate, the cost of correcting the error end-to-end balances the opportunity of exploiting correct symbols that made it to their next hops, maximizing the throughput.

The right mistake rate threshold depends on the network. We assume that the administrator calibrates this parameter for her networks. A large mistake rate like 30% does not make sense for any network.[4] The results however show that a wide range of choices provide good throughput and outperform packet-based opportunistic routing.

11.2.4 Impact of Batch Size

We evaluate whether MIXIT's throughput is sensitive to batch size. Fig. 14 plots the throughput for batch sizes of $8, 12, 16$ and 32. The throughput is largely insensitive to the batch size. The slight drop off at lower batch sizes is primarily because of higher overhead. A bigger batch size allows MIXIT to amortize the overhead over a larger number of packets, increasing throughput. Insensitivity to batch sizes allows MIXIT to vary the batch size to accommodate different transfer sizes. For any transfer larger than 8 packets, MIXIT shows significant advantages. Shorter transfers can be sent using traditional routing.

11.3 Multiple Flows

11.3.1 Throughput Comparison

Method: We run MIXIT, MORE and SPR in sequence, varying the number of random active flows in the network. The rest of the setup is similar to the single flow case. We run 50 experiments for each

---

[4]Even under optimal conditions, it takes at least two symbols to correct each incorrect symbol [30] and hence a mistake rate higher than 33% would never make sense.
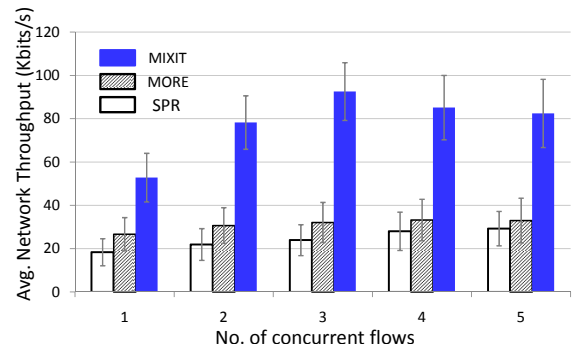
Figure 15: **Average throughput with multiple active flows:** The figure shows that MIXIT's throughput scales as offered load increases until the network is saturated. MORE and SPR become similar as load increases and perform worse than MIXIT because they cannot exploit concurrency opportunities.
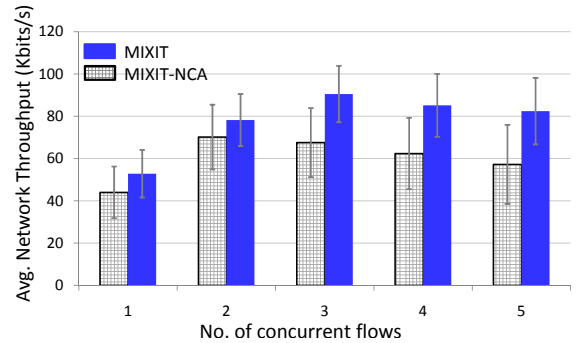
Figure 16: **The role of congestion-aware forwarding:** The figure shows that congestion-aware forwarding is particularly important when the number of active flows is large.

choice of number of flows, with each experiment repeated 5 times. We calculate the average throughput for each run.

Results: Fig. 15 plots the average throughput for MIXIT, MORE, and SPR with increasing number of flows. We see that MIXIT's throughput gain generally increases with load, and at its peak reaches $2.8\times$ over MORE and $3.9\times$ over SPR.

The higher gains as load increases are due to MIXIT's ability to aggressively exploit concurrency and perform congestion-aware forwarding. Both MORE and SPR, which rely on carrier sense, become conservative in accessing the medium as the number of flows increases. Thus, they cannot fully exploit the spatial diversity in the network. MIXIT however, can maintain high levels of concurrency because of its ability to deal with partially correct packets.

The throughput gains drop slightly as the network gets heavily congested. The primary reason is hidden terminals, whose effect is exacerbated by the fact that the USRP nodes, which perform all processing in user mode on the PC, do not have support for synchronous acks, and thus cannot quickly detect hidden terminals and backoff.

11.3.2 Impact of Congestion Aware Forwarding

Method: We evaluate the impact of MIXIT's congestion-aware forwarding component on performance. Node congestion is built into MIXIT's routing algorithm due to its use of the backlog parameter $Q(i)$, the number of symbols queued up at node i yet to be transmitted. Nodes that are backlogged will not be assigned credits by their upstream parents and thus traffic will be routed around hotspots. We compare this scheme with one where this component is disabled. Specifically, parent nodes assign credits to their downstream nodes based only on the path quality, i.e. based on the path ETS, and ignore

congestion information. We call this scheme MIXIT-NCA, for MIXIT with "No Congestion Aware" forwarding.

Results: Fig. 16 plots the average throughput for MIXIT and MIXIT-NCA for increasing number of flows. The figure shows that congestion-aware forwarding accounts for 30% of the throughput gain at high load. As load increases, the probability of the network experiencing local hotspots increases. MIXIT-NCA does not route around such hotspots, and insists on pushing the same amount of information through regardless of congestion. MIXIT adaptively routes around these hotspots and therefore increases throughput.

12 Conclusion

To the best of our knowledge, MIXIT is the first network architecture that profitably takes advantage of bit-level wireless spatial diversity to achieve high throughput. With MIXIT, as long as each symbol in every transmitted packet is correctly received by *some* neighboring downstream node, the packet is highly likely to be delivered to the destination correctly. Designing a network that has this attractive property is not an easy task because it needs to scalably coordinate overlapping symbol receptions and cope with erroneous symbol propagation. MIXIT solves these problems using a symbol-level network code that has an end-to-end rateless error correction component.

Instead of using link-layer error detection and recovery, MIXIT treats the entire wireless network as a single logical channel whose component links could run at high error rates. Because MIXIT can cope with individually high error rates, it encourages an aggressive MAC protocol that greatly increases concurrency compared to CSMA.

Although MIXIT exploits cross-layer information, its architecture is modular and layered: it can run atop any radio and PHY that provide suitable confidence hints, with the routers being oblivious to the end-to-end error correction mechanism. The gains may vary depending on the PHY and MAC used, but it can be used in any multi-hop wireless network with the following properties:

1. *Computational capabilities:* The coding/decoding algorithms in MIXIT are more demanding than traditional store and forward networks. In our proof-of-concept software implementation on software radios, the algorithms can achieve at most an effective throughput of 4.7Mb/s. In [9], we describe a hardware implementation using shift registers, which is similar to traditional Reed-Solomon (RS) hardware decoders. Because current RS decoders can achieve speeds of 80 Gigabits per second [15], we believe that computational considerations will not limit the applicability of our algorithms at high data rates.

2. *Memory:* MIXIT's nodes need to store packets from recent batches. The default batch size is 16, and typically there are two or three batches in flight, requiring storage space of roughly 70 KBytes, a modest amount for modern communication hardware.

The ideas in MIXIT may be applicable in sensor networks to ship data to sink nodes. Because most traffic in these networks is uni-directional, data from different sensors can be coded together to improve throughput. In addition, MIXIT could also be used to multicast data in a mesh network. Because all destinations require the same data, routers can keep transmitting coded data until all destinations can decode them.

Acknowledgments

We thank Kyle Jamieson, Szymon Chachulski, Robert Morris and the anonymous reviewers for their insightful comments. This work was

References

[1] S. Biswas and R. Morris. Opportunistic routing in multi-hop wireless networks. In *ACM SIGCOMM*, Philadelphia, USA, 2005.

[2] S. Chachulski, M. Jennings, S. Katti, and D. Katabi. Trading structure for randomness in wireless opportunistic routing. In *ACM SIGCOMM*, Kyoto, Japan, 2007.

[3] E. M. Gabidulin. Theory of codes with maximum rank distance. *Probl. Inform. Transm.*, pages 1–12, July 1985.

[4] J. Hagenauer and P. Hoecher. A Viterbi Algorithm with Soft-Decision Outputs and its Applications. In *IEEE GLOBECOM*, Dallas, USA, 1989.

[5] T. Ho, R. Koetter, M. Médard, D. Karger, and M. Effros. The Benefits of Coding over Routing in a Randomized Setting. In *ISIT*, Yokohoma, Japan, 2003.

[6] E. Inc. Universal software radio peripheral. http://ettus.com.

[7] S. Jaggi, M. Langberg, S. Katti, T. Ho, D. Katabi, and M. Médard. Resilient network coding in the presence of byzantine adversaries. In *IEEE INFOCOM*, Alaska, USA, 2007.

[8] K. Jamieson and H. Balakrishnan. Ppr: Partial packet recovery for wireless networks. In *ACM SIGCOMM*, Kyoto, Japan, 2007.

[9] S. Katti. *Network Coded Wireless Architecture*. PhD thesis, MIT, 2008.

[10] S. Katti, S. Gollakota, and D. Katabi. Analog network coding. In *ACM SIGCOMM*, Kyoto, Japan, 2007.

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

[12] R. Koetter and F. Kschischang. Coding for errors and erasures in random network coding. *IEEE Transactions on Information Theory*, 2007. To appear.

[13] R. Koetter and M. Médard. An algebraic approach to network coding. *IEEE/ACM Transactions on Networking, Volume 11, Issue 5, Oct. 2003, Page(s):782 - 795*.

[14] J. N. Laneman, D. N. C. Tse, and G. W. Wornell. Cooperative diversity in wireless networks: Efficient protocols and outage behavior. *IEEE Trans. on Inform. Theory, Volume 50, Issue 12, Dec. 2004 Page(s):3062 - 3080*.

[15] H. Lee. A high-speed low-complexity reed-solomon decoder for optical communications. *IEEE Transactions on Circuits and Systems*, 52(8):461–465, Aug. 2005.

[16] S. Lin and D. Costello. *Error Control Coding*. Prentice Hall, Uppser Saddle River, NJ, 2004.

[17] F. J. McWilliams and N. J. A. Sloane. *The Theory of Error-Correcting Codes*. North-Holland, 1977.

[18] A. Miu, H. Balakrishnan, and C. E. Koksal. Improving loss resilience with multi-radio diversity in wireless networks. In *ACM MobiCom*, Cologne, Germany, 2005.

[19] B. Radunovic, C. Gkantsidis, P. Key, S. Gheorgiu, W. Hu, and P. Rodriguez. Multi-path code casting for wireless mesh networks. In *CoNext*, New York, USA, 2007.

[20] C. Reis, R. Mahajan, M. Rodrig, D. Wetherall, and J. Zahorjan. Measurement-based models of delivery and interference in static wireless networks. In *ACM SIGCOMM*, Pisa, Italy, 2006.

[21] G. Richter and S. Plass. Error and erasure decoding of rank-codes with a modified berlekamp-massey algorithm. 2004.

[22] S. Zhang, S. Liew, and P. Lam. Physical layer network coding. In *ACM MOBICOM*, Los Angeles, USA, 2006.

[23] D. Silva, F. R. Kschischang, and R. Koetter. A rank-metric approach to error control in random network coding. *submitted*, 2007.

[24] D. Son, B. Krishnamachari, and J. Heidemann. Experimental analysis of concurrent packet transmissions in wireless sensor networks. In *ACM SenSys*, Boulder, USA, 2006.

[25] D. Tse and P. Vishwanath. *Fundamentals of Wireless Communications*. Cambridge University Press, 2005.

[26] M. Vutukuru, K. Jamieson, and H. Balakrishnan. Harnessing exposed terminals in wireless networks. In *USENIX NSDI*, San Francisco, USA, 2008.

[27] M. Wang, X. Weimin, and T. Brown. Soft Decision Metric Generation for QAM with Channel Estimation Error. *IEEE Transactions on Communications*, 50(7):1058 – 1061, 2002.

[28] A. Willig, M. Kubisch, C. Hoene, and A. Wolisz. Measurements of a wireless link in an industrial environment using an ieee 802.11-compliant physical layer. *IEEE Transaction on Industrial Electronics*, 49(6), 2002.

[29] G. Woo, P. Kheradpour, and D. Katabi. Beyond the bits: Cooperative packet recovery using phy information. In *Proc. of ACM MobiCom 2007, Montreal, Canada*.

[30] R. W. Yeung and N. Cai. Network error correction, part 1: Basic concepts and upper bounds. *Communications in Information and Systems*, 6(1):19–35, 2006.