Refector: Heuristic Header Error Recovery for Error-Tolerant Transmissions

Florian Schmidt, Muhammad Hamad Alizai, Ismet Aktaş, Klaus Wehrle
Chair of Communication and Distributed Systems,
RWTH Aachen University, Germany
{schmidt,alizai,aktas,wehrle}@comsys.rwth-aachen.de

ABSTRACT

High bit error rates reduce the performance of wireless networks. This is exacerbated by the enforcement of bit-by-bit correct transmissions and the resulting retransmission overhead. Recently, research has focused on more efficient link layer mechanisms and on tolerating payload errors. Header errors, however, still cause today's network and transport protocols to drop the erroneous packets.

Instead of retransmitting such packets, we investigate a novel concept (called *Refector*) of heuristically repairing header bit errors. Refector accepts erroneous packets on end hosts and exploits protocol knowledge and protocol state to assign packets to their correct destination applications. It operates on layers 3 and 4, is independent of the underlying MAC and PHY, and requires no changes to hardware, firmware, and communication behavior.

We evaluate the Refector concept via a prototype implementation deployed in an 802.11 network. Our results show that Refector reduces packet loss in the network by more than 25% when compared to payload-error-tolerant protocols such as UDP-Lite.

1. INTRODUCTION

A steady increase in the number of error-tolerant applications, for example, streaming audio and video, is playing a pivotal role in how the packet-switched networks have been evolving recently. For a long time both the (TCP/IP) network stack and the radio access catered to the demands for perfect data integrity of error-sensitive applications, for example, file transfers and SSH. However, error-tolerant applications, which are more sensitive to network delays and packet loss than to erroneous packets, have induced the following seriated evolution in packet-switched networking:

ACM CoNEXT 2011, December 6–9 2011, Tokyo, Japan.

Everything error-sensitive: The (TCP/IP) network stack and the medium access are error-sensitive. Packets need to be retransmitted if not acknowledged at the link layer. Protocols such as UDP are used to fasten the delivery of out-of-order yet error-free packets to streaming applications.

Error-aware medium access: While the medium access gains improved error handling, the network stack remains error-sensitive and can only handle error-free packets. It thus needs to recover erroneous packets transmitted over a medium, such as wireless, that is highly susceptible to frequent transmission errors. This has led to a wide array of low-level error handling mechanisms such as partial packet recovery [10, 14] and reconstruction from retransmissions [8]. The majority of these mechanisms requires significant modifications in medium access technologies, i.e., in the protocols and/or hardware at the link and physical layers.

Payload error-tolerant protocols: Finally, protocols themselves become partially error-tolerant: They only care about the integrity of packet headers, but the errors in the data are ignored, assuming they are acceptable for error-tolerant applications. UDP-Lite [19] is a prime example that leads this phase of evolution in packet-switched networking.

Considering this evolutionary pedigree, a logical next question should be whether we can make the network stack **also tolerant to header errors**—if even packets with those errors can be accepted. This paper explores the feasibility and limits of this novel approach in the form of *Refector* (*lat.*: repairer, mender): a system that accepts erroneous packets, even if the errors are within the protocol headers, and heuristically repairs these errors and identifies the correct communication end-point.

The utility of this approach depends on two key questions: (1) How many packets in a wireless network are lost due to header errors? Previous measurements targeted at 802.11 [26] have shown that bit errors, while bursty in nature, can appear anywhere in a packet, and that furthermore, a header bit is at least as likely to be corrupted as a payload bit. Therefore, a substantial amount of sent packets is lost at the receiver

due to header errors even when payload errors are ignored. This is particularly true for applications with small packets, such as VoIP, which have more header bits than payload bits. (2) Is delivering partially erroneous data to the receiver beneficial? Error-tolerant applications, such as VoIP, certainly benefit from partially corrupt packets [9]. We have also shown before in a simplified simulation [3] that, if header error tolerance and repair are feasible, there can be considerable gains to speech quality.

Refector runs on end-hosts and transparently integrates with the header processing of protocols, primarily at the network and transport layer. Thus, its main task is to assign erroneous packets to their correct applications even in the presence of header errors. This is based on two simple primitives: (1) Not all the header fields of a packet are required by the network stack to identify the target application at end-hosts. For example, an error in the TTL field at the end-host is immaterial as far as the delivery of a packet to the right application is concerned. Hence, at end-hosts, we can classify different header fields based on their importance in identifying the target application. This classification enables Refector to ignore errors in certain header fields. (2) Refector exploits the dynamic state of the network stack to repair header errors. For example, two packets with the same UDP source-port and different destination-port numbers, possibly due to errors, can be assigned to the same application if one of the destination ports is either invalid or not open.

While the concept of Refector is a very general one, in this paper we primarily focus on the network and transport layer and thereby stay independent of the underlying MAC and PHY layers. Our Refector prototype targets IEEE 802.11 based wireless LANs. In such setups, the wireless connection between the end host and the access point (i.e., the last hop over the Internet) is a major bottleneck for Internet communications. In Section 4.1 we show that Refector, when using NoAck schemes, reduces the average packet loss by more than 25%, rendering it highly valuable for error-tolerant applications in widespread 802.11 WLAN setups.

The rest of this paper is structured as follows. We briefly describe Refector and discuss our design goals in Section 2. The design of Refector is discussed in detail in Section 3. We evaluate Refector in an 802.11 WLAN setup in Section 4. Finally, we discuss limitations in Section 5 and related work in Section 6 before concluding the paper in Section 7.

2. SYSTEM OVERVIEW

Before exploring the details of how to repair header errors, we present the basic operational ingredients of Refector to facilitate a smooth sailing into the technical concepts discussed in Section 3. We further highlight our goals that justify our design decisions, explain the reasoning behind different design trade-offs, and comprehend the resulting limitations of Refector.

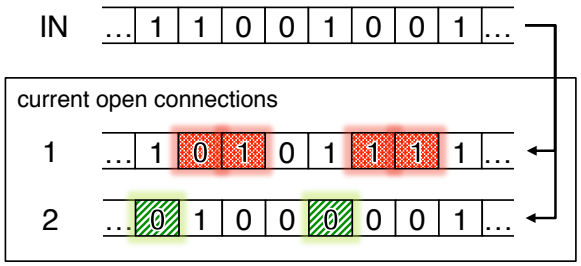


Figure 1: A simple example of port matching via Hamming distances. An incoming packet is matched against the expected values for two flows. Flow 2 is chosen in this example because of the lower Hamming distance (2 vs. 4).

2.1 Concept

The Refector approach follows a plain analogy with how a local postman delivers a postal package with a wrong address. Because of his experience in serving a locality, he can tolerate errors in the recipient's name or street address. Moreover, errors in certain fields of a postal address, e.g., the country code, do not concern a local delivery. Similarly, Refector makes use of what we call *domain knowledge*, which comes in two forms.

Time-independent domain knowledge is the knowledge of a person familiar with a certain protocol about how important the different header fields are, on the end host, to deduce the correct communication end-point. This knowledge can then be used to design a system that can tolerate header errors. We defer a detailed discussion on different fields to Section 3. One simple example is the TTL field in IP header. It is decremented at every intermediate hop, and the packet is dropped when the TTL reaches zero. However, on the end-host, this field is irrelevant, because the packet has already reached its destination. Despite the fact that a wrong TTL field has no effect on the header or payload processing, an error in any of the 8 TTL bits will result in the packet being dropped because the checksum did not match.

Time-dependent domain knowledge also takes into account the current state of the system. On the end-host, we know which connections are open at any given point in time. For each of these connections, a state is maintained that tells how a header for this connection should look like. The operating system needs this information to match a packet to a specific connection. As a result, we can match even an erroneous incoming packet to an ongoing connection by knowing which connections are currently open and which it most likely matches. A

simple example is given in Figure 1: A small header excerpt (for the sake of clarity), e.g. the *port* field, is matched against two ongoing communication flows at the end host. It does not perfectly match either of the two flows. However, since two bit flips are sufficient to match it to flow 2, as opposed to four bit flips to match it to flow 1, the decision is to assign the packet to connection 2.

As a similarity metric, we employ Hamming distances, i.e., the number of differing bits between two bit strings. It is a computationally inexpensive metric that computes bit similarity independent of the location of errors in the string.

2.2 Design Goals

The design of Refector leverages the following four primary goals.

Improving NoAck feasibility: NoAck schemes significantly improve application throughput by avoiding expensive acknowledgments and retransmission of highly time-critical data, as demonstrated in Figure 2. NoAck schemes eliminate the need for per frame *Ack* which, besides inducing additional communication overhead, has to honor the inter-frame timing constraints of 802.11. However, NoAck schemes suffer from high loss rates in lossy networks, and are barely usable when combined with full-coverage link-layer checksums. We want to enable NoAck schemes by introducing error-tolerance in the network stack. Unlike approaches such as UDP-Lite, we want to tolerate errors in packet headers, which means more packets are successfully delivered to the right communication end-point. Consequently, we can achieve the envisioned advantages of these schemes such as saving timeouts and retransmissions, reducing latency at the receiver's side, and improving the overall network performance by freeing air time for other transmissions.

MAC and PHY compatibility: We do not want to introduce specialized MAC and PHY layers. This is because changes in hardware or the overall infrastructure, such as WLAN, are expensive. Refector only imposes two requirements on the underlying layers: (i) the ability to pass through damaged packets, and (ii) the ability to disable MAC acknowledgments, that is, NoAck support. These features are commonly available in the IEEE 802.11e extension for quality of service supported by a wide range of commodity hardware. This extension defines access categories—flows with different QoS criteria—and also supports disabling acknowledgments on a per-packet basis via a flag in the MAC header. Thus, error-tolerant and error-sensitive flows can coexist by only disabling acknowledgments for packets belonging to error-tolerant flows.

Protocol compatibility: We want to confine the changes introduced by Refector to pure software changes. Moreover, the behavior of a Refector host to its communication partners or an outside observer should remain standard-compliant. This design goal enables Refector to integrate well with existing IP based networks because it neither changes the communication related behavior of protocols (i.e., UDP and IP) nor the transmitted data (both headers and payload).

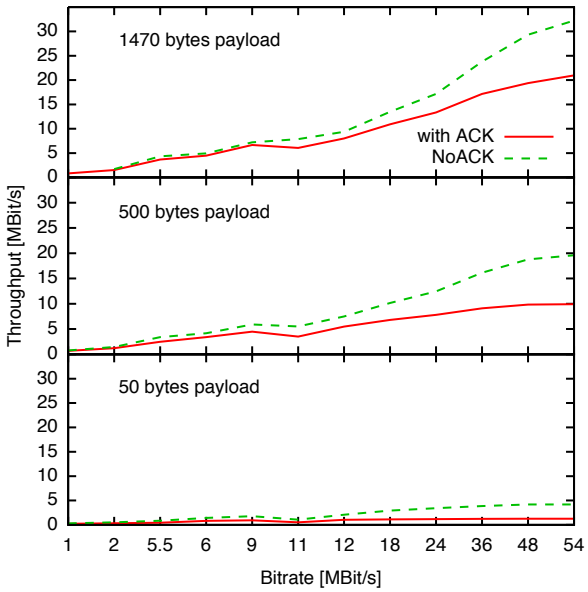


Figure 2: Application-level throughput on 802.11 with and without acknowledgments. Under good channel conditions, disabling acknowledgments can considerably improve throughput, especially at high bit rates.

Supporting heterogeneous application requirements: We want Refector to be an optional, rather than mandatory, feature for all applications running on an end host. Applications such as VoIP shall be able to opt-in to error-tolerant communication with Refector support, which leaves the standard case at completely error-free communication (e.g., FTP, but also control information such as ARP). Thus, the integration of Refector with the protocol-stack does not adversely affect the performance of concurrent applications.

Concluding, all our design goals favor accommodating common requirements—such as compliance to the existing standards, low deployment effort, and seamless integration into existing systems—over potentially more effective optimization efforts that require disruptive changes in existing networked systems.

3. REFECTOR

This section illustrates the design details of Refector. Approaches such as UDP-Lite filter out all packets with errors in UDP headers. Packets with errors in IP headers are also dropped before they reach UDP-Lite.

In contrast, the very purpose of Refector is to accept packets with errors in the UDP and the IP headers.

We particularly focus on the transport and network layers (i.e., UDP and IP) for two reasons. First, the network stack is a highly standardized design with the goal of interoperability between many heterogeneous systems. It is neither dependent on the underlying hardware, firmware, or driver, nor on the applications that use its functionality. This makes changes in this part, as long as they do not impair compatibility, effective and widely usable. Second, a plethora of solutions has already been proposed on the application layer, for example, streaming codecs to cope with partially erroneous data [7, 12], and on the MAC layer with [10] and without [20] hardware/firmware changes.

Over the 30 years since the original specification of UDP/IP protocols, their setup has remained unchanged. This is both a testament to the good design of the Internet protocol suite and to the issue that compatibility between hosts is vital for the functioning of the Internet. Changing specifications or introducing new protocols proves exceedingly difficult. The monotonousness of this Internet protocol specification, however, has lead to inefficiencies. Some techniques and their corresponding header fields have gradually fallen out of use, e.g., fragmentation and reassembly in the IPv4 protocol. Similarly, many fields are designed to contain a wide range of values, while in practice, at least on end hosts, only few of these values are used at any given point in time. Refector leverages these inefficiencies both by ignoring the contents of some fields, and by trying to introduce additional redundancies in others where possible.

3.1 Header Fields Categorization

To manipulate erroneous packets, Refector divides header fields of a packet into two categories, *don't-care* and *vital*. Don't-care fields are not required for identifying the right communication endpoint of a packet and are simply ignored by Refector. Vital fields however, if erroneous, have to be repaired.

In the following we show that each header field of IP and UDP protocols can either be categorized as don't-care or as vital (cf. Figure 3). Furthermore, we show how errors in vital header fields can be recognized and repaired.

3.1.1 Internet Protocol

The IPv4 header has a size of at least 20 bytes divided into the following fields.

Version and IHL: The first byte is split into a version identifier and the header length. We categorized both of these fields as *don't care*: The version field is redundant with information in the underlying MAC header that identifies the encapsulated network protocol. We also consider the header length static because header

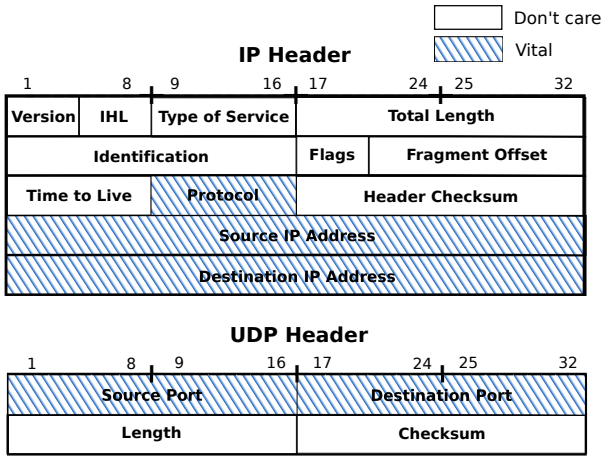


Figure 3: Classification of header fields in the IP and UDP header.

options (at least in IPv4) are practically non-existent. In a standard network, header options are rarely seen at all, and some are even deprecated and filtered by gateways (such as the originally defined routing headers).

Type of Service: The next byte contains DiffServ [24] and ECN [25] information. We categorize DiffServ as don't-care because it is not required once the packet has reached its destination. We also categorize ECN as don't-care because (i) it does not impact the behavior of IP layer, and (ii) TCP is not the focus of our work and will discard a corrupted packet in any case.

Total Length: Total length is a two-byte field that contains the length in bytes of the complete packet at this point in time. There is little practical use in this field since the actual packet length can easily be derived from other sources of information, such as the MAC layer or the length of the packet in data memory. After a packet is received by the network interface, it is copied into the main computer memory and the operating system is informed about the length of data in this packet. Therefore, we categorize it as don't care.

Identification, flag and fragment offset: The next four bytes contain information for IP-level fragmentation and reassembly. This feature has long fallen into obscurity due to MTU discovery and MAC-level fragmentation and is not even part of the basic IPv6 header any more. Therefore, the three fields are considered as don't-care in our approach.

TTL: The TTL field is not required for protocol processing at the receiving host and is therefore categorized as don't care.

Protocol: The protocol field identifies the next protocol handler to be run once the packet leaves the IP handler. This is one of the most important fields at the end host, however, it is not easily repairable: For example, UDP (encoded as 17; 0000**1**001) and ICMP (encoded as 1; 0000**0**001) are only separated by a single bit. Hence, a

single bit flip can result in the packet being forwarded to the wrong handler. We categorize this field as vital.
Checksums: The IP header checksum only covers the IP header itself. We categorized it as don't-care because (i) it depends on a large number of fields that we categorized as don't-care, and (ii) the checksum in itself does not provide us any information about the correct communication end-point.
Source and Destination Address: Finally, the last 8 bytes contain the sender and receiver IP address. These fields contain important information about the correct communication end-point and we mark them as vital. To repair these fields, we consider time-variant information about the currently ongoing communication flows: Refector maintains a list of IP addresses retrieved from packets that are received without any errors.

After receiving a packet with errors, Refector matches the source address of the packet against the list of IP addresses to find the best match in terms of the Hamming distance. We define a maximum Hamming distance threshold to avoid attributing packets to wrong flows. Practical tests have shown that a threshold of 3 effectively prevents misattributions (see Section 4.4) because other factors, such as UDP ports, are also considered before the packet is delivered to an application.

Refector performs a similar matching between the destination address in the header and valid addresses for the receiving host. This approach may seem less effective for numerically close destination addresses as might occur where several hosts in the same sub-network use Refector. Here, the small Hamming distance between addresses might increase the risk of misattributions and acceptance of packets that were destined for another host. However, before the packet reaches the IP layer, it has already passed another step of identification depending on the underlying medium access control techniques: A time or code division system or MAC address ensures that the packet was destined for the host. Compared to IP addresses, MAC addresses show a considerably higher randomness. Under the typical Ethernet MAC addressing scheme, even in a network of machines of the same type, three of the six bytes of the address do not show any noticeable correlation between the different hosts (the first three being a manufacturer prefix). Note that Refector, while it requires the MAC layer not to discard packets with CRC mismatches, does not apply repair techniques to the MAC layer.

3.1.2 User Datagram Protocol

The UDP header contains four fields of two bytes each. In effect, the only functionality it adds on top of IP are port numbers to allow demultiplexing of several flows on one host.
Source and Destination Ports: The first two fields contain the source and destination port of the flow the packet belongs to. These fields contain the most important piece of information to identify the right communication endpoint of a packet, and hence, we mark them as vital. Similar to IP addresses, Refector maintains a list of port numbers derived from packets that were received without any errors. For packets with erroneous port fields, Refector finds the best match from the list in terms of the Hamming distance. Again, to prevent misattribution, it is necessary to find a Hamming distance limit that facilitates effective repair without creating packet misattribution. However, unlike IP addresses, which are assigned to hosts in a fixed manner, port numbers can be actively chosen within a limited range, as discussed in the next section.
Length and Checksums: The remaining two fields belong to the don't-care category. This is because the length of the packet can be derived from the size of the data structure containing the packet data. The checksum covers the UDP header and payload, and is ignored by Refector.

3.2 Port Allocation

As discussed in the previous section, the *destination port* is the most important field in the UDP header with regard to identifying the communication endpoint at the receiving host. By carefully allocating ports to applications at the receiving host, we can make the port number fields more resilient to bit errors. Therefore, Refector modifies the kernel's port assignment algorithm to increase the Hamming distance between the ports numbers[1]. This reduces the risk of attributing a packet to a wrong application.

3.2.1 Selection Mechanism

The IANA defined "private ports" are in the range between 49152 and 65535, but many operating systems use a larger range typically starting from 32768. This is the range readily available for our custom port allocation scheme. Refector employs BCH codes [2] to select ports for error-tolerant applications. BCH codes construct "codewords"—port numbers in this case—that have a predefined minimum Hamming distance between each other.

A BCH code is a polynomial error-correcting code over a finite field $GF(q^m)$ defined by a generator polynomial. With UDP ports, we have a binary field of 16 bits. However, as all the port numbers with the most significant bit set to **0** are reserved, we need a BCH code that operates on codewords of 15 bit length. Therefore, Refector uses a BCH code over $GF(2^{15})$. The port numbers are then created from these codewords by prepending a most-significant bit **1**.

[1]Note that applications still can request specific ports, overriding the kernel's allocation. However, they lose the advantage of a guaranteed minimum Hamming distance.

generator polynomial	no. data bits	no. code bits	min. distance	corrects errors	no. ports
$x^4 + x + 1$	11	4	3	1	2048
$x^8 + x^7 + x^6 + x^4 + 1$	7	8	5	2	128
$x^{10} + x^9 + x^8 + x^6 + x^5 + x^2 + 1$	5	10	7	3	32

Table 1: Example generator polynomials for BCH codes. We used their property to create codewords of a given length and minimal Hamming distance to choose port numbers that are distant enough from each other to be resilient to a certain number of bit errors.

The choice of the generator polynomial influences the number of data bits and parity bits; a trade-off between its resilience to errors, and the number of ports we can choose with that resilience. A collection of example polynomials and their properties are shown in Table 1. We use the generator polynomial $x^8+x^7+x^6+x^4+1$ for our implementation, giving us 128 ports for the range starting at 32768 (or 64 ports for 49152) at a minimum Hamming distance of 5, and therefore, the ability to correct up to two bit errors.

We consider 128 ports sufficient because Refector is designed for end hosts, and the number of open connections at any time is low on such machines. We monitored the number of open sockets on a student lab machine during a normal workload of browsing, e-mail, instant messaging, etc., and never witnessed more than 65 simultaneously open ports. Furthermore, if our allocation scheme runs out of ports, it will fall back to the original scheme.

3.2.2 Analytic Approximation

After presenting our port assignment strategy, we now analytically compare this strategy with the standard case, that is, any port number in the private port range can be assigned if the application does not choose a specific one. We focus on the case of less than 128 concurrent connections and assume the worst case scenario under this condition: The minimum Hamming distance between two port numbers is 5 in the case of Refector and 1 in the standard case.

A Hamming distance of 1 means that two ports share all but one bit, and this bit is the deciding factor in the correct assignment (attribution) of a packet to the target application. The number of misattributions is therefore equal to the bit error rate (BER) assuming independent distribution of errors. With an increasing number of ports with 1-bit Hamming distance between them, the misattribution rate increases. This is because several bits in the 16-bit port field become deciding factors. For n ports with 1-bit Hamming distance, the misattribution rate therefore can be approximated as:

$$1 - \left((1 - BER)^n \right) \tag{1}$$

In the case of Refector, the minimum Hamming distance is always 5, that is, at least 3 of those 5 distance bits have to be flipped to cause misattribution. For two ports with 5-bit Hamming distance, the misattribution rate therefore becomes:

$$1 - \sum_{k=3}^{5} \binom{5}{k} BER^k \, (1 - BER)^{5-k} \tag{2}$$

For several ports with 5-bit Hamming distance between them, the exact misattribution rate depends on how many of the differing bits are overlapping between these ports. The absolute worst case, in which every port with a 5-bit distance is active, has a misattribution rate of[2]:

$$1 - \sum_{k=3}^{15} \binom{15}{k} BER^k \, (1 - BER)^{15-k} \tag{3}$$

Figure 4 summarizes the worst case misattribution of Refector when compared with the standard case. As pointed out, this analysis assumes that bit errors are independent of each other. However, as in practice errors in 802.11 tend to occur in bursts [26], we will compare this analysis with our real-world evaluation results in Section 4.4.

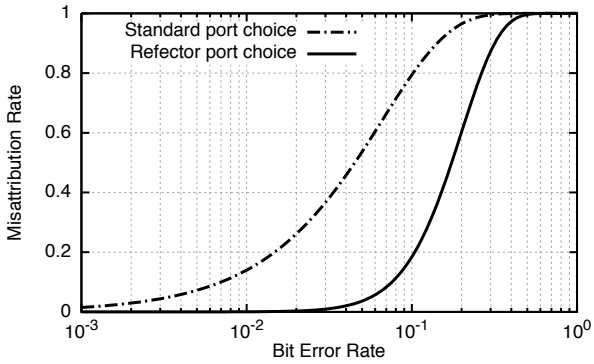
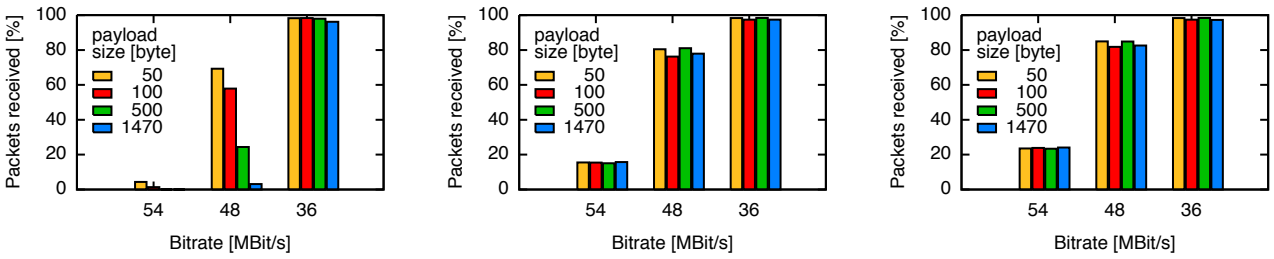


Figure 4: Packet misattribution rate: Refector's port selection algorithm creates a higher resilience to misattribution. Even at a BER of 10^{-2}, the worst-case misattribution rate is still below 0.05%, compared to 14% in worst-case standard assignment.

[2]This is actually strictly worse than what is defined by the chosen BCH code, because no port will have all 5-bit neighbors as valid choices.



(a) UDP (b) UDP-Lite (c) Refector

Figure 5: Influence of payload size on packet loss: As opposed to UDP with checksum coverage of header and payload, where payload size influences the reception rate, UDP-Lite with a header-only checksum coverage, and Refector, which tolerates errors in headers and payload, have a reception rate independent of payload size. (The graphs show results collected from one of our test machines.)

3.3 User-Kernel Interface

One of our design goals is to allow both error-tolerant applications (e.g., VoIP) and error-sensitive applications (e.g., FTP) to coexist on a Refector host. Refector extends the socket interface between the kernel and the applications with the additional option `SO_BROKENOK` which is set by the application when it creates the socket. This option allows Refector to determine whether or not to deliver erroneous payloads, that is, payloads contained in packets with checksum mismatches on any layer, to the application.

Refector also creates a signaling path in the opposite direction. The idea is to inform the application whether or not the delivered payload is error-free. This can be beneficial for an application even if it is ready to cope with erroneous data. When an application uses the `recvmsg` system call to receive incoming data, it also receives additional information in the form of a set of flags, such as whether the data fit the provided receive-buffer. Refector adds the flag `MSG_HASERRORS`, which is set when a CRC check failed on any layer, and thus indicates that the delivered data may contain errors.

4. EVALUATING REFECTOR

The evaluation of Refector is based on IEEE 802.11 based WLAN technology. Our evaluation focuses on three key parameters: packet delivery rate, packet misattribution, and the influence of encryption on the performance of Refector. We conclude this section by discussing the overhead introduced by Refector's heuristic matching.

In the following, we briefly describe our experimental setup before going into the details of our results.

4.1 Experimental Setup

Throughout our evaluation, we compare Refector with UDP-Lite, a standardized connectionless protocol that allows partial checksums which only cover part of a datagram, that is, header and an application-specified amount of the beginning of the payload. For our evaluation, we set UDP-Lite's checksum coverage to header-only for two reasons: (a) Including parts of the payload in the checksum negatively impacts the performance of UDP-Lite in terms of the number of packets delivered to the application at the end host, and, (b) securing the header clearly shows the effect of header error-tolerance—the key difference between the two techniques—introduced by Refector. Note that UDP-Lite has the same usage requirements as Refector: Both approaches require support to receive erroneous packets and disable MAC acknowledgments.

We implemented Refector for Linux kernel 2.6.32.27. Our experimental setup consisted of one AP and four hosts. All the machines used network cards with an ath5k [1] chipset. The AP machine provided AP functionality via hostapd [11]. To identify error-tolerant streams, the sender application sets the IP header's *Type-of-Service* field to a value that is mapped to the 802.11 access category "voice", which was configured at the AP to be sent out without acknowledgments.

We performed our experiments in an indoor setting at the computer science building of the RWTH Aachen University. The building has a very dense deployment of APs from the university network. We used 802.11 channel 5 which is the same channel used by a neighboring AP of the university network. Hence, the ongoing traffic on these neighboring APs served as background and competing traffic. Because we did not have direct influence on this traffic, we performed our experiments over the course of several days and nights. Our experiments for this analysis have two key characteristics: (1) Each experimental run comprised one UDP-Lite flow and one or several Refector flows per receiving host. (2) In each experimental run, the AP sent 10 000 packets per flow. Packets of all flows were sent concurrently; hence, slow changes in the channel quality did not influence the comparability of the results. We combined the results of

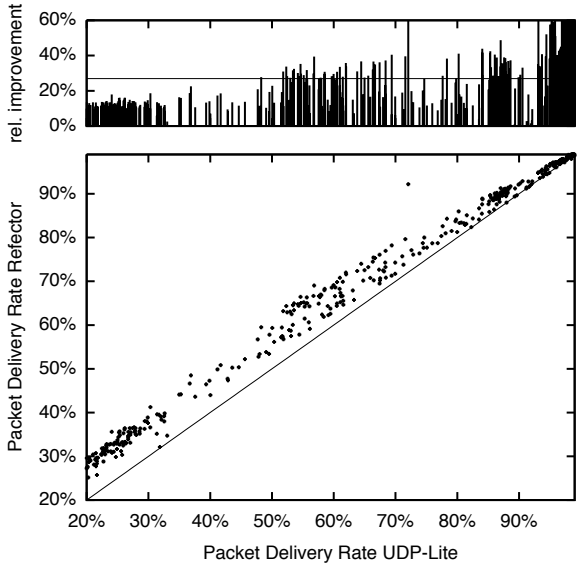


Figure 6: Lower Graph: Comparison of UDP-Lite and Refector by packet delivery rate over 400 runs from different stations. Upper graph: relative improvement (packet loss using UDP-Lite divided by packet loss using Refector) of Refector over UDP-Lite. On average, we witnessed an improvement of about 27%.

200 experimental runs on each machine to leverage the different reception qualities caused by their positioning and timing of the experiments. Thus, we received results for different error and packet loss rates.

4.2 Influence of payload size on packet loss

Before presenting detailed results on our key performance parameters, we need to answer one question: Does the payload size impact the performance of Refector and UDP-Lite? It is important to answer this question here because a negative answer will alleviate the need to use payload size as a defining parameter in the evaluation of Refector.

Figure 5 clearly shows that for UDP the size of the payload impacts the overall packet loss in the network. However, packet loss rates of UDP-Lite and Refector remain unaffected by varying payload sizes. This is because UDP creates a checksum that secures its header and the payload. For a fixed bit error rate, the number of non-matching checksums therefore increases with packet size. For our UDP-Lite setup, however, the coverage is fixed at 8 bytes. The number of non-matching checksums for a fixed bit error rate is now independent of payload size.

The same holds true for Refector because checksum mismatches are ignored, so the checksum coverage is effectively zero. This means that no erroneous packets are dropped because of checksum mismatches: The packet

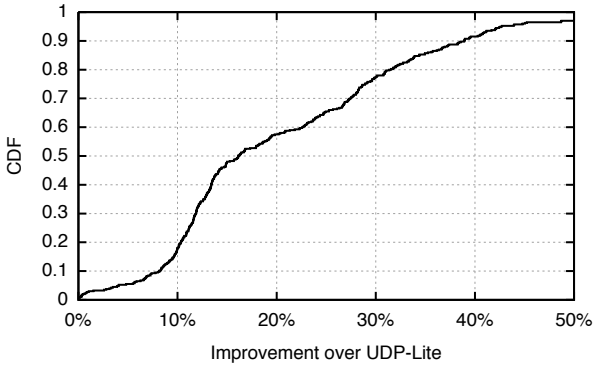


Figure 7: CDF of the relative improvement achieved by Refector over UDP-Lite (as defined in Figure 6).

is only dropped if no communication end-point can be derived after applying Refector's heuristic matching. Based on these results, for the rest of this evaluation section, we will not use payload size as a factor in evaluating Refector.

4.3 Packet Delivery Rate

Packet delivery rate, that is, the percentage of sent packets that are delivered by the receiver to the application layer, is one of our key performance metrics. Figure 6 shows our results. The lower graph compares the packet delivery rate of Refector with UDP-Lite. We exclude results with less than 20% and more than 99% delivery rate, because due to the very small and random amount of packets received or lost, respectively, these experiments showed severe variation in results, even after several days and nights of measurements. Each data point in the graph compares a UDP-Lite flow with a simultaneous Refector flow. Although varying channel conditions lead to diverse packet delivery rates, Refector outperforms UDP-Lite, which already tolerates errors in the payload, in more than 99% of the experiments.

The upper graph in Figure 6 summarizes the results by presenting the relative improvement achieved by Refector, computed by dividing the number of packets lost using UDP-Lite by the number of packets lost using Refector. We use this metric for measuring relative improvement because the difference between a packet loss of 1% and 2% can be significant for some error-tolerant applications: They will show their most noticeable quality degradation early on, that is, during the first few packet losses.

On average, Refector reduces the packet loss by 27% in the network when compared with UDP-Lite. Please note that this average improvement is sensitive to how many results at which delivery rate contribute to the average: Including an overwhelmingly large number of experiments at a very high or very low quality will skew

the average. Therefore, our experimental results presented here are evenly spread over the whole range of packet delivery rates. Note that the relative improvement typically increases with the channel condition, that is, at more practical packet loss rates of less than 10%, Refector tends to give a higher relative benefit. Finally, Figure 7 shows the CDF of relative improvement achieved by Refector in all the experimental runs. These results show the feasibility of Refector as a system that can improve the quality of error-tolerant applications, such as voice and video, by reducing packet loss in the network.

4.4 Packet Misattribution

Another important factor is packet misattribution: Because of the heuristic nature of Refector's header error tolerance, it is possible that packets are assigned to the wrong application. Although we analytically proved in Section 3.2.2 that packet misattribution is negligible when using Refector's port selection mechanism, these results were based on the assumption that bit error rates are independent from each other. This assumption contradicts the bursty errors rates observed in 802.11 based networks [26]. Hence, we investigate whether the analytical approximation holds true in a real world deployment.

There are two misattribution scenarios to distinguish: (1) misattribution between applications running on the same end host, and (2) misattribution between applications running on different hosts in the network. In all our experiments, we did not see a single instance of the second case. This is due to a packet's MAC address comparison with the destination's MAC address at the link layer.

To evaluate packet misattribution, we used two concurrent Refector applications running on each host in the network. We filled the application payloads with bit patterns specific to each flow and examined the received data to recognize misattributed packets. Figure 8 compares our empirical result with the analytical approximation (cf. Equation 2) from Section 3.2.2. Each data point is the result of 100 000 transmissions. It clearly shows that the packet misattribution rate is negligible: Except for rare outliers, the misattribution rate remains below 0.1% until the payload bit error rate increases above 3%, at which point packets are severely corrupted, and loss rate even with Refector exceeds 80%. Although these results are derived from two concurrent applications, we strongly believe that even in the case of multiple applications the misattribution rate will remain close to its respective analytical approximation as long as the Refector port selection mechanism is used to assign port numbers to applications.

It is also important to note that misattribution can only occur for error-tolerant applications that use Refector. Error-sensitive applications require correct packets (as indicated by correct checksums). Only erroneous packets are processed by Refector's heuristics, and those heuristics only take into account sockets of error-tolerant applications. Therefore, Refector will never assign an erroneous packet to an error-sensitive application.

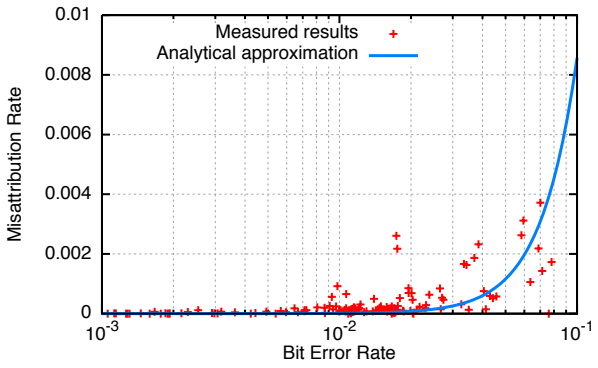


Figure 8: Packet misattribution rates for two concurrent Refector applications.

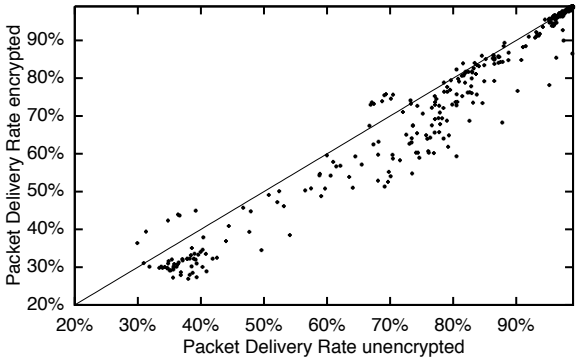
4.5 Influence of Encryption

Our evaluation results so far were based on unencrypted communications. However, to enhance the privacy of a wireless network's users, encryption is often applied on the MAC layer. Hence, we also measure the influence of encryption on the performance of Refector.

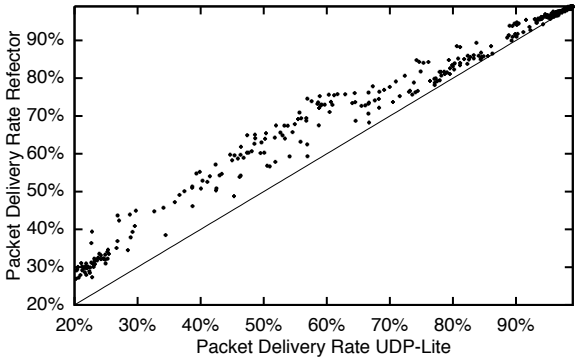
The 802.11 security subsystem provides two features: data integrity via a message identity code (MIC) and the actual encryption of data. Enforced data integrity protection is not possible with any error-tolerant transmission scheme because the two concept are mutually exclusive: either a receiver accepts errors in the received data, or it imposes strong integrity check via MIC.

The influence of encryption on error-tolerant transmissions primarily depends on the cipher's error propagation, that is, the number of additional bits that will be corrupted during decryption for each bit that was received incorrectly. 802.11 provides two encryption alternatives: CCMP uses AES and TKIP uses RC4. AES can corrupt up to 64 additional bits per bit error. However, the stream cipher RC4 does not corrupt any additional bits during decryption. For this evaluation, we therefore focus on TKIP/RC4 as the natural choice for error-tolerant transmissions.

For this part of our evaluation, we introduced two changes into the MAC layer implementation of the receiver. (1) We ignored decryption errors, that is, mismatches of the MIC, similar to accepting CRC mismatches. As with any checksum mismatch, error-sensitive applications will not receive payloads that could not be checked via the MIC, upholding data integrity



(a) Comparison of an unencrypted and an encrypted stream using Refector for error tolerance and repair.



(b) Comparison of an encrypted Refector stream with an encrypted UDP-Lite stream.

Figure 9: Influence of RC4 encryption on Refector's error tolerance: While more packets are unsalvageable, Refector still performs well under encryption, and still noticeably outperforms UDP-Lite.

protection for them. (2) We disabled countermeasures (against potential attacks on the encryption) that are taken by the MAC implementation if it witnesses too many MIC mismatches. These changes only required small modifications in the hardware-independent mac80211 [21] implementation inside the Linux kernel. It is important to note that these changes reduce the security because it opens possibilities for additional cryptanalytic attacks. Therefore, these changes are merely introduced (i) to measure the influence of encryption on the performance of Refector, and (ii) to support easy deployment of Refector in a network that encrypts all its frames on the MAC layer.

Figure 9(a) compares two concurrent UDP streams using Refector, one encrypted and one unencrypted. It clearly shows that Refector achieves higher packet delivery rate with unencrypted streams. This is because TKIP uses per-frame initialization vectors (IVs) to create a new key for every frame. If the IV, which is transmitted within the MAC header, is corrupted, the frame cannot be decrypted at all. However, at delivery rates above 80%, the performance differences are small.

To show the advantage of Refector over UDP-Lite in an encrypted network, Figure 9(b) compares a Refector stream with a UDP-Lite stream. This setup is directly comparable to Figure 6 for the unencrypted case. The results are very similar: Refector's advantage holds in the encrypted scenario. Note that UDP-Lite also benefits from our changes to the MAC encryption implementation. Without these changes, every bit error leads to dropping the frame, rendering UDP-Lite's payload error tolerance useless.

4.6 Overhead

Refector's heuristic matching introduces computational overhead in the network stack. However, this overhead is very small because:

1. Most header fields are categorized as don't-care, and therefore do not require any repair effort.
2. For the remaining fields (i.e., IP addresses and ports), Refector only computes their Hamming distance—a computationally inexpensive operation—to members of a list of previously encountered addresses and ports.
3. The number of these comparisons only linearly increases with the size of these lists. End hosts typically do not have many IP addresses, and port comparisons are only performed among ports of Refector-enabled applications. Hence, these lists remain small, introducing very low memory requirements and also keeping the number of Hamming distance calculations very small.

Consequently, this low computational overhead of Refector was not even measurable on our test machines.

5. DISCUSSION

In this section we highlight three major limitations of Refector and shed light on how these limitations can be dealt with.

Acknowledgment policies: Currently, the utility of Refector is limited to NoAck schemes. This is primarily because our initial design goal is the seamless integration of Refector with existing MAC standards.

It is due to the strict timing constraints of existing MAC acknowledgment schemes, such as in 802.11, that Refector is unable to coexist with such schemes. For example, the receiver has to decide within a short timeframe (the SIFS, typically between $10\,\mu$s and $16\,\mu$s) whether to acknowledge a frame or not. This time is insufficient even to hand over the frame from the network card to the operating system. Relaxing the timing constraints of acknowledgments at the MAC layer is a simple solution to overcome this problem and to extend

the benefits of Refector beyond NoAck schemes. This way, the receiver should be able to deliver the packet to the network stack and decide if the packet could be assigned to an application before sending out a positive or negative acknowledgment for that packet.

Rate Adaptation: While Refector's error-tolerant approach can significantly improve packet delivery rates at the application level, many rate adaptation schemes, such as the still widely used ARF [17] or the current Linux default Minstrel [22], do not work out of the box with NoAck schemes. These rate adaptation schemes rely on acknowledgments as feedback for future rate adaptation decisions.

If Refector's error-tolerant NoAck traffic constitutes a substantial portion of the traffic between a station and an access-point, the rate adaptation performs suboptimally. We are currently investigating a rate adaptation scheme that will work without explicit per-frame notifications via acknowledgments. To this end, we are working on a receiver-initiated rate adaptation that can integrate with existing rate adaptation approaches, upholding the principle of Refector being easily deployable in existing network infrastructures.

Multi-hop scenarios: A substantial amount of Refector's improvement stems from the local knowledge that can be leveraged at the receiving end-host. This does not necessarily hold true for relaying gateways. For example, unlike UDP ports, an IP address cannot simply be matched against the locally used address(es), because the packet may needed to be relayed to a different host that we know nothing about. It is therefore an open question how well Refector would perform in wireless multi-hop scenarios, and how it could be adapted to such scenarios. However, this question is of minor importance and beyond the scope of discussion in this paper because we specifically focused on WLAN setup, that is, a host directly connected to an access point.

6. RELATED WORK

Error tolerance within a communication system has fostered a great deal of research effort in recent years. However, the majority of these efforts has focused on PHY and MAC layer solutions. Figure 10 depicts the prominent techniques that deal with recovering erroneous packets.

Hybrid-ARQ [5] combines forward error correction and retransmissions: additional FEC information is sent instead of a simple retransmission. PPR [14] uses soft information to recognize errors and only retransmit erroneous parts. Similarly, SOFT [27] uses soft information received by several nodes, which then exchange this information via Ethernet to reconstruct a correct copy. ZigZag [8] uses the symbols received at the physical layer from retransmissions of the same packet to recover erroneous packets. These solutions require special hardware that gives access to soft information or the PHY symbols.

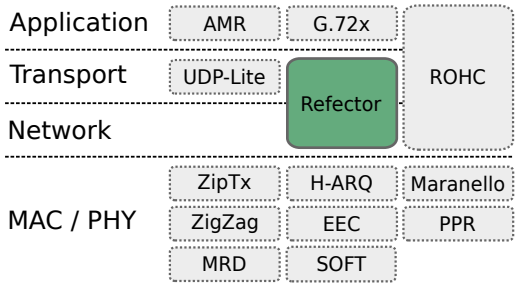


Figure 10: Current approaches to achieve more error tolerance within a communication system.

In ZipTx [20], the receiver is required to send negative acknowledgments to the sender to request parity information for recovering the packet. While it runs on commodity 802.11 hardware, it changes the MAC layer behavior. Maranello [10] creates multiple checksums over blocks of the transmitted packet. The receiver only requests the retransmission of erroneous blocks. It changes the 802.11 MAC protocol behavior in a way that does not interfere with standard nodes in the same network. MRD [23] combines information from multiple receptions of a packet, either by multiple nodes or through retransmissions. All these solutions aim at creating a correct copy of the packet to hand over to the network stack. The goal of EEC [4] is to estimate the BER and provide this information to sender and receiver. The estimation introduces additional sampling data into the packets and changes the MAC protocol.

Similar to Refector, Jiang [15] proposes to use redundancy in MAC addresses to heuristically correct errors at the MAC layer of 802.11. However, it is not clear how the system would perform in a real network, especially since it does not consider the inherent timing problems of such a solution in 802.11. UDP-Lite [19] is among the rare solutions that tackle erroneous packets at higher layers of the protocol stack. It has prominently featured in this paper as a benchmark and for comparative evaluation of Refector. UDP-Liter [18] introduces compatibility between UDP and UDP-Lite and presents an application-kernel interface similar to Refector.

Header compression schemes [13, 6, 16], while originally aimed at reducing overhead, support error-tolerant communications: By reducing the header size, they decrease the chance of bit errors in vital portions of the packet. One of the most advanced schemes, robust header compression (ROHC) [16], can reduce headers to about 10% of their original size, and is planned for use in LTE cellular networks. However, header compression needs to be deployed on all participating nodes on both sides of the wireless connection as it changes communication behavior via compressed headers. Also, to remove all redundancies, each sender needs to maintain

the current state of connections of all its communication partners. In contrast, Refector only needs local knowledge about connection, resides on the receiver side, and does not change the communicated data.

7. CONCLUSION

We presented Refector, a novel scheme that assigns erroneous packets to their correct applications even in the presence of header errors. Employing Refector in an 802.11 network without acknowledgments improves packet delivery rate by more than 25% using negligible computational resources. We also showed that the used heuristics produce very little error, and that it is possible to use Refector with encryption, depending on the used cipher.

We identify the following aspects as future work. (1) Analyze the wide applicability of Refector by evaluating it using other MAC protocols and physical layers. In fact, 802.11 is a rather unforgiving system for error tolerance because of its strict acknowledgment timing policies. We are currently investigating the effects of Refector in other scenarios. (2) Extend the use of Refector towards multi-hop wireless network scenarios such as wireless mesh networks and sensor nets. (3) Integrate Refector with error-aware medium access technologies, such as Maranello [10], to see if these approaches compliment each other to improve packet delivery and reduce retransmissions in a lossy wireless network. (4) Combine Refector with soft-information based schemes such as PPR's SoftPHY [14] for 802.11 networks.

Overall, this paper shows the applicability of Refector as a solution to enhance packet delivery in lossy wireless networks. Our evaluation under different networking conditions shows that Refector can realize its advantages in real world deployments.

Acknowledgments

We would like to thank Stefan Götz, Raimondas Sasnauskas, and Elias Weingärtner for many fruitful discussions and valuable feedback. This research was funded in part by the DFG Cluster of Excellence on Ultra High-Speed Mobile Information and Communication (UMIC).

8. REFERENCES

[1] Linux Wireless ath5k driver. [Online] Available `http://wireless.kernel.org/en/users/Drivers/ath5k` October 12, 2011.

[2] R. C. Bose and D. K. Ray-Chaudhuri. On a class of error correcting binary group codes. *Information and Control*, 3(1):68–79, 1960.

[3] T. Breddermann, H. Lüders, P. Vary, I. Aktas, and F. Schmidt. Iterative Source-Channel Decoding with Cross-Layer Support for Wireless VoIP. In *Proc. ITG SCC*, 2010.

[4] B. Chen, Z. Zhou, Y. Zhao, and H. Yu. Efficient error estimating coding: feasibility and applications. In *Proc. SIGCOMM*, pages 3–14, 2010.

[5] R. Comroe and D. Costello. ARQ Schemes for Data Transmission in Mobile Radio Systems. *IEEE Journal on Selected Areas in Communications*, 2(4):472–481, July 1984.

[6] M. Degermark, B. Nordgren, and S. Pink. IP Header Compression. RFC 2507, IETF, Feb. 1999.

[7] ETSI EN 301 704 V7.2.1 (2000-04). *Adaptive Multi-Rate (AMR) Speech Transcoding (GSM 06.90 Version 7.2.1 Release 1998)*, Apr. 2000.

[8] S. Gollakota and D. Katabi. Zigzag decoding: combating hidden terminals in wireless networks. In *Proc. SIGCOMM*, pages 159–170, New York, USA, 2008. ACM.

[9] F. Hammer, P. Reichl, T. Nordström, and G. Kubin. Corrupted speech data considered useful: Improving perceived speech quality of voip over error-prone channels. *Acta acustica*, 90:1052–1060, Dec. 2004.

[10] B. Han, A. Schulman, F. Gringoli, N. Spring, B. Bhattacharjee, L. Nava, L. Ji, S. Lee, and R. Miller. Maranello: practical partial packet recovery for 802.11. In *Proc. NSDI*. USENIX Association, 2010.

[11] hostapd. [Online] Available `http://w1.fi/hostapd/` October 12, 2011.

[12] ITU. *ITU-T Recommendation G.729*, Jan. 2007.

[13] V. Jacobson. Compressing TCP/IP Headers for Low-Speed Serial Links. RFC 1144, IETF, Feb. 1990.

[14] K. Jamieson and H. Balakrishnan. PPR: Partial Packet Recovery for Wireless Networks. In *Proc. SIGCOMM*, pages 315–326, August 2007.

[15] W. Jiang. Bit Error Correction without Redundant Data: a MAC Layer Technique for 802.11 Networks. In *Proc. WiNMee*, Apr. 2006.

[16] L.-E. Jonsson, K. Sandlund, G. Pelletier, and P. Kremer. RObust Header Compression (ROHC): Corrections and Clarifications to RFC 3095. RFC 4815, IETF, Feb. 2007.

[17] A. Kamerman and L. Monteban. WaveLAN-II: a high-performance wireless LAN for the unlicensed band. *Bell Labs Technical Journal*, 2(3):118–133, 1997.

[18] P.-K. Lam and S. Liew. UDP-Liter: an improved UDP protocol for real-time multimedia applications over wireless links. In *Proc. Wireless Communication Systems, 2004*, pages 314–318, Sept. 2004.

[19] L.-Å. Larzon, M. Degermark, S. Pink, E. Jonsson, and E. Fairhurst. The lightweight user datagram protocol (UDP-Lite). RFC 3828, IETF, July 2004.

[20] K. C.-J. Lin, N. Kushman, and D. Katabi. Ziptx: Harnessing partial packets in 802.11 networks. In *Proc. MOBICOM*, pages 351–362. ACM, 2008.

[21] Linux Wireless mac80211 framework. [Online] Available `http://linuxwireless.org/en/developers/Documentation/mac80211`, October 12, 2011.

[22] Minstrel rate control algorithm. [Online] Available `http://linuxwireless.org/en/developers/Documentation/mac80211/RateControl/minstrel`, October 12, 2011.

[23] A. Miu, H. Balakrishnan, and C. E. Koksal. Improving loss resilience with multi-radio diversity in wireless networks. In *Proc. ACM MOBICOM*, pages 16–30, 2005.

[24] K. Nichols, S. Blake, F. Baker, and D. Black. Definition of the Differentiated Services Field (DS Field) in the IPv4 and IPv6 Headers. RFC 2474, IETF, Dec. 1998.

[25] K. Ramakrishnan, S. Floyd, and D. Black. The Addition of Explicit Congestion Notification (ECN) to IP. RFC 3168, IETF, Sept. 2001.

[26] 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. *Industrial Electronics, IEEE Transactions on*, 49(6):1265–1282, Dec. 2002.

[27] G. Woo, P. Kheradpour, D. Shen, and D. Katabi. Beyond the bits: cooperative packet recovery using physical layer information. In *Proc. MOBICOM*, pages 147–158. ACM, 2007.