Heuristic Header Error Recovery for Corrupted Network Packets

Von der Fakultät für Mathematik, Informatik und Naturwissenschaften der RWTH Aachen University zur Erlangung des akademischen Grades eines Doktors der Naturwissenschaften genehmigte Dissertation

vorgelegt von

Diplom-Informatiker

Florian Schmidt

aus Köln

Berichter:

Prof. Dr.-Ing. Klaus Wehrle
Prof. Dr.-Ing. Wolfgang Kellerer

Tag der mündlichen Prüfung: 23. 10. 2015

Abstract

Wireless communication provides many advantages over wired communication, such as easier deployment due the lack of cabling infrastructure and higher mobility for users. However, one of its most important downsides is the significantly higher error rate. This is exacerbated by the fact that traditional Internet communication enforces perfect bit-by-bit correctness of each packet. While this ensures high reliability in the received data, it is very inefficient: even a single bit error leads to a packet drop, leading to high packet loss even at comparatively low bit error rates. Such a behavior is especially wasteful when considering error-tolerant applications. For example, many media codecs have been designed to tolerate and mask bit errors in their data.

To better support these types of transmissions, suggestions in the past have aimed at tolerating errors in the payload portions of packets, the most well-known example being UDP-Lite. However, these solutions suffer from several drawbacks. First, they suffer from low acceptance unless they ensure they stay fully interoperable with standard protocols. Second, they focus on single protocols, without taking the layered nature of protocol combinations into account. This is problematic because error tolerance in one protocol can be rendered useless by combining it with a lower-layer protocol that drops all packets that contain errors. Third, focusing only on payload error tolerance means any errors in the header portions of packets still lead to drops. Especially in small packets, headers form a large part of the packet, limiting the effectiveness of payload error tolerance.

In this dissertation, we design and present solutions that address these shortcomings. First, by introducing error tolerance into existing standard protocols, we ensure interoperability. Second, by taking the whole stack into account, we ensure that packets are not dropped before error tolerance can recover them. Third, by allowing errors to also occur in the header portions of packets, we increase the effectiveness of error tolerance. This last contribution means that control information in packet headers is not reliable any more. Thus, packets for one application could be misattributed to another. Hence, we will present solutions to identify the correct application a packet belongs to, even under header errors, as well as ways to repair corrupted header information, to prevent this misattribution.

Our first contribution is a solution to introduce header error tolerance and repair into existing protocols at the examples of IPv4, UDP, and RTP. As a second contribution, we design a protocol-independent approach that can identify which connection a packet belongs to, as well as repair certain errors in protocol headers, without requiring any knowledge about the protocols it works on. Our final contribution focuses on the popular 802.11 wireless technology. To fully unlock the potential of header error recovery in 802.11, we design a novel rate adaptation algorithm that can adapt to changes in channel quality without relying on acknowledgments, which is not possible with state-of-the-art solutions.

Kurzfassung

Drahtlose Kommunikation bietet gegenüber drahtgebundener viele Vorteile, so zum Beispiel eine einfachere Einrichtung aufgrund des Verzichts auf Kabel-Infrastruktur sowie höhere Mobilität für Benutzer. Andererseits ist einer der größten Nachteile die im Vergleich deutlich höhere Übertragungsfehlerrate. Dieses Problem wird durch die in klassischer Internetkommunikation erzwungene exakte Bit-für-Bit-Korrektheit der übertragenen Pakete noch verschärft. Dies garantiert zwar eine hohes Maß an Zuverlässigkeit, ist aber sehr ineffizient: selbst ein einzelner Bitfehler führt zum Verwerfen des gesamten Paketes und damit zu hohem Paketverlust bereits bei niedrigen Bitfehlerraten. Ein solches Verhalten ist besonders verschwenderisch bei fehlertoleranten Anwendungen. So gibt es beispielsweise Media Codecs, die speziell im Hinblick auf Toleranz und Maskierung von Bitfehlern entwickelt wurden.

Zur besseren Unterstützung solcher Anwendungsarten wurden in der Vergangenheit Ansätze vorgeschlagen, die Fehler in den Nutzdaten von Paketen tolerieren; das bekannteste Beispiel ist hier UDP-Lite. Allerdings leiden diese Verfahren unter mehreren Nachteilen. Erstens leiden Sie unter geringer Akzeptanz, wenn sie keine vollständige Interoperabilität mit bestehenden Standardprotokollen sicherstellen. Zweitens konzentrieren sie sich auf einzelne Protokolle, ohne dabei das Schichtenmodell und die daraus resultierenden Kombinationen von Protokollen in Betracht zu ziehen. Dies führt zu Problemen, wenn Protokolle niederer Schichten fehlerbehafte Pakete verwerfen. Drittens werden Pakete mit Headerfehlern weiterhin verworfen. Vor allem bei kleinen Paketen bilden Header einen großen Teil des Pakets, so dass eine Fehlertoleranz lediglich für Nutzdaten von beschränkter Wirkung ist.

In dieser Dissertation werden Lösungen für diese Nachteile entworfen und präsentiert. Erstens wird durch das Einbinden von Fehlertoleranz in bestehende Protokolle eine Interoperabilität mit anderen Systemen sichergestellt. Zweitens wird durch das Betrachten des gesamten Netzwerkstapels ein vorzeitiges Verwerfen von Paketen verhindert, bevor Fehlertolernzmechanismen das Paket behandeln können. Drittens wird durch das Zulassen von Fehlern im Headerbereich die Effektivität von Fehlertoleranz erhöht. Eine Herausforderung ist in diesem Zusammenhang die Tatsache, dass fehlerhafte Headerinformationen zu einer Fehlzuordnung des Pakets führen können. In diesem Fall werden Pakete einer Anwendung fälschlicherweise einer anderen zugeordnet. Die Vermeidung solcher Fehlzuordnungen, selbst bei Datenfehlern in den Identifikationsdaten der Paketheader, stellt einen der Hauptbeiträge dieser Dissertation dar.

Im ersten Teil dieser Dissertation wird ein Konzept für Fehlertoleranz anhand einer Implementierung in IPv4, UDP und RTP beispielhaft vorgestellt und untersucht. Im zweiten Teil wird ein protokollunabhängiger Ansatz vorgestellt, der die Zugehörigkeit von (potentiell fehlerbehafteten) Paketen zu einer Verbindung erkennt, ohne über Wissen über die verwendeten Protokolle zu verfügen. Schließlich wird ein neuartiger Ansatz für Ratenadaption in 802.11-Netzwerken vorgestellt, der aufgrund seiner im Gegensatz zu Standardverfahren von ACK-Paketen unabhängigen Adaptation die effektive Verwendung von Fehlertoleranz in WLAN-Netzen ermöglicht.

Acknowledgments

This dissertation concludes a long chapter of my life that I spent in Aachen, from my days as a young and confused undergraduate student to a "finished" PhD. This transformation, and the resulting work you are reading, would not have possible without the support and input by many people. Even though I am sure I will unjustly forget some of them in the following, I want to thank at least some of those that had the greatest influence on both me and my work:

My first thanks rightfully belong to my advisor, Klaus Wehrle. Many years ago, he put me on track as a young PhD student on a project that eventually would eventually branch out in many directions and form this dissertation. Through all this time, he gave me the support I needed, and fostered a creative research group with immense freedom to pursue your goals. I also want to thank Wolfgang Kellerer not only for agreeing to act as a second opponent, but also for valuable feedback during the final parts of this work.

While Klaus made sure that I found a great environment to flourish in, it is Elias Weingärtner who I am most grateful to for paving the way to even enter this environment. Elias picked me up as an insecure Diploma student and saw more scientific capability in me than I did myself. Without his encouragement, I would have never pursued a PhD.

During my work on this dissertation, I had the chance to collaborate with and advise many brilliant students, which helped me shape my work the way it stands now. Thank you, Caj-Julian, David, Dominik, Erwin, Mario, Matthias, Niklas, René, and Tobias, for our many discussions and your dedication to your work. I especially want to thank Anwar and Martin for their scientific rigor and high motivation to push their thesis topics to the limit. Both of them decided to pursue a PhD themselves, and I am sure that soon, I will see them defend brilliant dissertations.

Over the years, I met many great people at COMSYS who I wish to thank. Ismet and Raimondas, for being the best office mates to wish for, both for scientific discussions and non-scientific small talk. Jan and Torsten, for the same reasons, and for enduring my advice. Matteo and Oliver for, both in their own and different ways, opening up new insights for me into how academia works. Tobi, for making sure I would always go the extra mile to make every idea and result that little critical bit better and more well-rounded. Stefan, for teaching me more about scientific writing than anybody else, and for finding the most incredible movies for after-work movie nights. Petra and Ulrike, for always working around organizational issues and solving them for us. Kai, Rainer, and Dirk, for keeping my own and the whole chair's work–life balance in order. Additionally, all the others I did not mention here specifically: you made (and make) COMSYS a great place.

Finally, I want to thank my brother and my parents for their unconditional love and support in all the years. Without encouraging my curiosity from an early age, I might not have ended up a scientist; without the computer we finally bought after my nagging for a long time (and whose operating system I killed several times before I learned to fix it—sorry for all the lost files!), I might not have ended up a computer scientist.

Contents

1

Niggle was a painter. [...] There was one picture in particular which bothered him. It had begun with a leaf caught in the wind, and it became a tree; and the tree grew, sending out innumerable branches, and thrusting out the most fantastic roots. [...] Soon the canvas became so large that he had to get a ladder, and he ran up and down it, putting in a touch here, and rubbing out a patch there.

—J. R. R. Tolkien, *Leaf by Niggle*

Introduction

Wireless communication has become more and more important in recent years. With the proliferation of devices such as notebooks, smartphones, and tablets, which often do not even provide a wired network connection any more, the last hop of Internet connections has become more and more dominated by wireless communications. There are many reasons for this development. For example, wireless communication provides more convenient network access, since it allows a degree of mobility for devices and users that a wired connection cannot.

However, this convenience comes at a price. Wireless connections typically exhibit a much higher error rate than wired connections. Packet error rates (that is, the fraction of packets with at least one bit error in them) of 20–25% are not unusual in Wireless LAN (WLAN) communications [SHP+12]. This is due to *channel effects* such as attenuation, fading, and interference, which act on the wireless channel, while they are largely absent in wired connections. To overcome this effect, wireless networks use concepts to reduce and detect errors. Sophisticated modulation and coding with Forward Error Correction (FEC) techniques are used to prevent errors. However, since the magnitude of the channel effects varies over time, a complete prevention of errors is impossible. To recognize such errors, checksums are added to the transmitted data. In case of transmission errors which corrupt the data, the checksum will not match the data any more, and the packet will be discarded and potentially retransmitted. This so-called Automatic Repeat reQuest (ARQ) scheme has proven successful to ensure bit-by-bit correctness and allows reliable data transmissions, even in challenging conditions, and hence enables applications such as file transfers to work properly in wireless networks.

While ARQ is successful in guaranteeing reliable transmissions, it is also a potentially very inefficient scheme. Checksums can only signal whether a packet is completely correct or contains at least a single bit error, not which bits are corrupted. Even if only a single bit is corrupted, ARQ hence retransmits the complete packet, even

though most data was already correctly received. Thus, the retransmission occupies the wireless channel for longer than necessary and reduces the overall goodput. In addition, it introduces delay, since the packet cannot be delivered or forwarded before a successful retransmission has occurred.

This behavior is especially wasteful when considering that there are some application types that do not necessarily require bit-by-bit correct transmissions to function well. A prime example are media streaming applications, which have become very popular in recent years and by now account for up to 25% of all Internet traffic [GDFM+12, LAN+12]. For such applications, media codecs exist that are able to tolerate and mask bit errors in their received payload data (e.g., [ETSI00, SSJ+08]) and hence do not require perfect bit-by-bit correctness. On the other hand, those applications benefit from low latency between sender and receiver. This is especially true if they stream live media or are used for interactive communication, such as in the case of Voice over IP (VoIP). For example, the European Telecommunications Standards Institute (ETSI) and the International Telecommunication Union (ITU) both recommend an end-to-end delay of no higher than 150 ms [ETSI06, ITU03] for VoIP. For this class of applications, completely correct data (for example, as enforced by checksum checks and retransmissions) that arrives too late is practically useless, while corrupted data that arrives in time can be put to use [HRNK04].

This motivates approaches that hand corrupted data to these error-tolerant application instead of dropping and retransmitting the packet. Thus, these approaches not only reduce latency, because the packet can be processed immediately instead of waiting for a retransmission, but also increase the overall network throughput, because the time otherwise spent for the retransmission can be used for other data transmissions. The probably most well-known scheme to support such an approach is UDP-Lite [LDP+04], a transport-layer protocol that is derived from UDP. While UDP secures its header and the complete payload with a checksum, UDP-Lite allows to reduce the checksum coverage by only covering a subset of the payload, or to not cover the payload at all. If an error occurs in the unsecured payload, the UDP-Lite checksum will not produce a mismatch, and hence, UDP-Lite will not discard the packet.

Thus, payload-error-tolerant protocols such as UDP-Lite take an important step towards error tolerance. However, there are some shortcomings, which we will discuss at the example of UDP-Lite. First, on its own, without any support by the lower layers, UDP-Lite cannot adequately provide error tolerance. As soon as one of the lower layers employs a checksum that covers the whole packet, any payload errors will lead to drops on that layer, before the packet ever reaches UDP-Lite. This is not merely a theoretical problem, since checksums that cover the complete frame are the norm on the MAC layer, and are employed by, for example, WLAN. Second, since UDP-Lite is a different transport-layer protocol from UDP, it requires support from both sender and receiver to set up such an error-tolerant transmission. Finally, payload tolerance, while definitely helpful, is most beneficial for large payloads. When using small payloads, as is generally the case in audio streaming and VoIP, headers form a sizable portion of a packet, sometimes more than 50%, as shown in Figure 1.1. In such a situation, errors in a large portion of the packet still lead to packet loss, reducing UDP-Lite's potential.

Figure 1.1 UDP-Lite supports error-tolerance for a variable portion of its payload, which includes the application-layer payload (white) and potentially application-layer protocol headers (light gray). As long as errors are in this portion, UDP-Lite will not drop the packet. However, errors in its own header or lower-layer headers (dark gray) still lead to packet drops. In small packets, those portions can form more than 50% of the packet. The example shows the packet layout of an AMR Full-Rate [ETSI00] audio transmission that produces 32-byte packets, with typical headers in a WLAN scenario. Header and payload portions are to scale.

In this dissertation, we start from the basic idea of UDP-Lite as well as its limitations and design a system that allows for practical error tolerance by including the lower layers. Furthermore, we introduce error tolerance into already existing protocols instead of designing new ones, which facilitates easier and incremental deployment, since no support from the sender is required. Finally, and most significantly, we also take a conceptual step forward and tolerate errors not only in the payload, but also in the *header* portions of packets. However, doing so leads to novel challenges.

1.1 Challenges in Heuristic Header Error Recovery

By tolerating errors in the header portions of packets, we accept the fact that information in these packet headers may be corrupted due to bit errors. However, while there are error-tolerant applications, network stacks are not designed with tolerance to errors in mind, since those errors are expected to be prevented by checksums. Consequently, we have to consider the consequences of this decision to tolerate header errors.

These consequences are governed by the functions that headers fulfill. There are two such main functions. On the one hand, headers contain protocol-specific information such as version fields, or sequence numbers in a stateful protocol. A wrong sequence number in a streaming application can, for example, reduce quality, because data is played back at the wrong time. On the other hand, headers contain demultiplexing information, that is, information about which connection a packet belongs to. In a packet-switched network, this information is vital to identify the correct communication end-point of a packet that is transmitted via a channel that is shared between several connections. Errors in parts of headers that contain this demultiplexing information can therefore lead to packets being identified as belonging to another connection than they actually do. This *misassignment* or *misattribution* (we will use the two terms interchangeably in this dissertation) is highly problematic, because it can be detrimental or even fatal to error-sensitive connections such as file transfers: data is sent to a communication end-point (an application) that it doesn't belong to, corrupting the transmitted file.

From this simple example, it is clear that simply ignoring all errors in received packets is not viable. We need to make sure that header errors do not produce

negative side-effects. It is especially important to prevent misassignments due to their potential for fatal events. Therefore, we need to be able to correct header errors that could trigger such events.

As checksum mismatches only signal that at least one bit error occurred somewhere in the packet, but do not yield any information about the locations of such errors, we have to work with very limited information. In effect, all we have available is the received header content, which we cannot rely on to be correct. Therefore, we employ heuristics[1] to deduce the original header contents. Doing so, we have two ways of proceeding:

1. We can either require our repair to be perfect. To do this, we employ heuristics to repair the headers, then check the repaired headers (and potentially payload) against the received checksum, and only accept the packet if the checksum matches. This is a safe solution, but also provides limited improvement: Errors in the checksum itself would lead to such drops even if the repair was completely successful, as would payload errors if the protocol's checksum secures both header and payload. Furthermore, as we will see later, there are large header areas in standard protocol headers that are both irrelevant in certain situations and unrepairable, such as the TTL field in the IPv4 header. Errors in such fields would still lead to packet drops.

2. Motivated by the above observation, we can accept that even after error recovery, a checksum might not match. This approach is more risky because incorrectly repaired headers are not caught by the checksum, but has a higher potential performance gain.

This dissertation will focus on the more comprehensive second alternative because it promises higher rewards. Since thus we have to accept (1) that headers may contain errors even after all potentially employed forms of FEC, and (2) that packets might be imperfectly repaired, there is no way to reconstruct headers with absolute certainty. Employing heuristics to recover from header errors incurs the risk that those heuristics choose wrongly, leading to the aforementioned misassignments. Much of the work of this dissertation focuses on recovering as many corrupted packets as possible, while preventing and/or minimizing the chance of misassignments. We do so by designing robust heuristics and analyzing their results in scenarios that range from minimal to extremely high Bit Error Rates (BERs). To support robust heuristics, we make an important observation that stems from their use in an environment that we consider a typical use case.

1.2 Target Environment and Observations

From the opening motivating observation of this chapter – the strong increase in wireless connections for the last hop to an end host – follows the target environment

[1] Which heuristics we use will be shortly explained in Section 1.4 and presented in detail in the respective main chapters.

that we focus on in this dissertation. Such a scenario comprises a wireless access point that doubles as a gateway to the Internet, and one or several users with computers that act as wireless stations. The error-prone wireless link hence forms the last hop of the connection. A typical example of such a scenario is a WLAN setup as used in many households, companies, and public spaces. The strong increase in devices such as smartphones, tablets, and notebooks which can only communicate wirelessly and do not provide wired network connectors will make wireless connections the dominant system for last-hop Internet connectivity, if it has not already done so.

By focusing on the last hop, we can leverage a small, but significant information advantage when compared to any arbitrary network node in the Internet. In such a case, our heuristics reside on the receiving end host. We can therefore make use of the observation that an end host knows, at any given point in time, all its currently open connections. It furthermore knows the expected contents of those header fields that identify each communication end-point (e.g., ports in transport-layer protocols such as TCP and UDP), because this is the information that the network stack, matches against the header values of each received packet during processing. To a lesser degree, this is also true for state information such as sequence numbers. This is different from a router somewhere within the Internet, which does not have information about all currently ongoing connections between hosts that it forwards data for.

This information is important because it can be relied on, while header contents of corrupted packets are unreliable. We hence have information against which to check header fields in corrupt packets, and thus facilitate repair. Furthermore, header information that is purely concerned with the routing of a packet to an end-host is not relevant any more once the packet has reached that host. Consequently, errors in those fields are irrelevant and can be ignored.

This observation can be compared with the real-world example of how mail couriers deliver postal packages with erroneous addresses. Because they know the locality they serve, they can tolerate errors in the recipient's name or street address. Moreover, errors in certain fields of a postal address, for example, the country code, do not prevent local delivery, because the couriers will not consider them in their delivery attempt.

We will show in this dissertation how this knowledge about the recipients of packets can be leveraged to design highly performant and effective heuristics for header error recovery.

1.3 Research Questions

We now condense the previous challenges and observations into five research questions that we will answer over the course of this dissertation:

Q1: *Is it possible to heuristically recover from header errors?*
As a first step, we will have to design a system that can heuristically recover from errors. Furthermore, this system must be usable in practice.

Q2: *Is it possible to effectively prevent misattributions?*
If our system can recover from errors, but often repairs header incorrectly, its usefulness is dubious. We will have to show that we can prevent misattributions and their potentially catastrophic effects.

Q3: *Can we increase the robustness of protocol header information, while staying 100% compatible to protocol specifications?*
By introducing additional robustness, we can improve the identification process to support the prevention of misattribution, while at the same time increasing recovery rates. However, because we do not want to require any special support from the sender, we have to restrict solutions to those that do not change the protocol behavior.

Q4: *How much protocol-specific knowledge does a heuristic header error recovery scheme need to work properly?*
Designing a system to recover from header errors in certain protocols is certainly helpful. However, if it is possible to conduct such recovery even without protocol-specific knowledge, we could significantly broaden our solution by supporting legacy protocols for which no detailed protocol-specific solutions exist.

Q5: *How can we unlock the full potential of error-tolerant transmissions in WLAN?*
The presented heuristic header error tolerance concepts are designed to be as independent of the underlying MAC and physical systems as possible to allow a use in a wide range of different network technologies. However, to ensure their practical usefulness, effective support in WLAN, as today's most widespread wireless technology, is highly desirable.

We will affirmatively answer these questions by demonstration, by designing concepts to heuristically identify which connection a packet belongs to even under header errors and show their practical applicability in real-world systems at both low and high error rates.

1.4 Contributions

To address the questions stated above in the presented target environment, we consider the concept of heuristics across the whole network stack. First, we show how heuristics can be tailored to specific protocols [SAAW11, SOW13a]. Next, we present how classification can be used to create protocol-independent error recovery [SHW14]. Finally, we will solve a WLAN-specific challenge to error tolerance that otherwise prevents us from unlocking the full potential of heuristic header error recovery in WLAN networks [SHP+12]. In the following, we will present the salient

points of each concept. While doing so, we will also point out how these concepts relate to the above research questions, and visualize this relationship in Figure 1.2.

1.4.1 Protocol-Specific Heuristic Header Error Recovery

For Refector (Latin for mender, repairer), our first solution for heuristic header error recovery, we analyze several protocols and their headers. For this analysis, we choose IP and UDP [SAAW11] as well as RTP [SOW13a], because these protocols are designed to support non-reliable or streaming transmissions, and as such serve as good examples of protocols employed in error-tolerant transmissions. To an end host, the fields in those headers are of unequal importance, to the point where we can categorize them into groups of vital and don't-care fields. Don't-care fields are simply ignored: we do not recover from errors in them, but neither is this necessary. Vital fields, such as ports and addresses, are used to find the best matching connection, and are afterwards reconstructed to match the expected values of that match.

Such a search for a best match requires a distance metric. We show that, despite its simplicity, Hamming distance is an effective metric that allows us to create a heuristic header error recovery system that significantly improves the recovery rate over payload-only error tolerance, and thus answer Question 1.

To mitigate the problem of misassignments, we first focus on the most damaging case, in which a corrupted packet is misassigned to a non-error-tolerant application. Such an error can be fatal to, for example, a file transfer. We completely prevent such misassignments by requiring error-tolerant applications to signal their error-tolerance when they open a connection. If a corrupted packet is heuristically assigned to a connection that is not flagged as error-tolerant, it will be discarded instead. To reduce the risk of (non-fatal) misassignments to error-tolerant connections, we focus on making the heuristic as robust as possible by introducing a novel port selection method for transport-layer protocols. This answers Question 3, since such a method increases robustness to errors while not introducing any change in protocol behaviors. The effect of this is reflected in evaluations, which show that misassignment is an extremely rare occurrence, answering Question 2.

1.4.2 Protocol-Independent Heuristic Error Recovery

The previous approach works very well and can be fine-tuned to specific protocols. However, it incurs a non-trivial overhead in the design phase for each such protocol that is to be supported. For each new protocol, protocol headers have to be analyzed with regard to their significance for connection identification, and the methods have to be implemented. Protocols without such specifically implemented support cannot benefit from heuristic header error recovery.

We therefore investigate a machine-learning-based classification approach [SHW14] that replaces this design stage and can heuristically identify a packet's connection

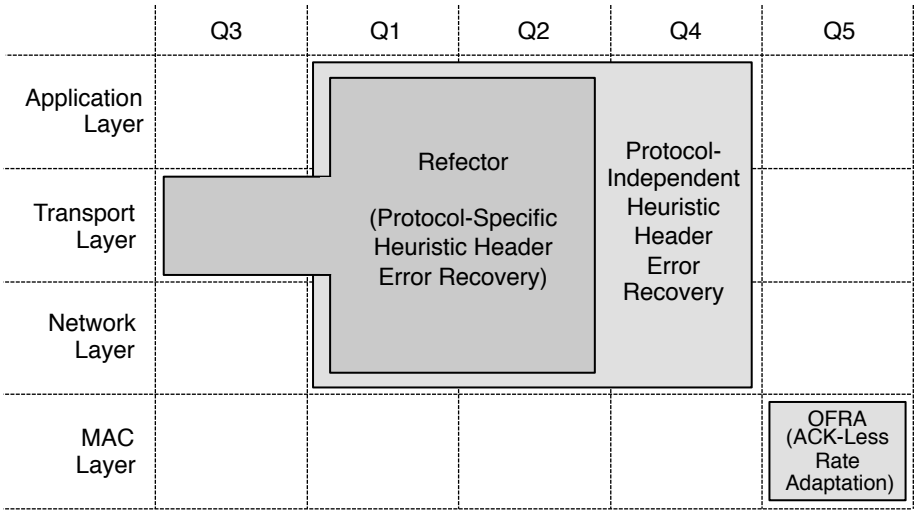


Figure 1.2 Visualization of our main contributions, how they relate to our research questions (note the ordering), and on which protocol layers they tackle those questions. Note that visualization only considers implemented solutions. For example, there is no fundamental conceptual problem that prevents the usage of Refector on the MAC layer, but this has not been investigated in our work.

without knowledge about the used protocols. For each connection, we analyze incoming correct packets to learn characteristic bit patterns that occur in those packets. Corrupted packets are then classified by matching them against those patterns, assigned to a connection, and headers repaired according to the learned patterns. Thus, there is no need for the classification algorithm to have any knowledge about the protocols it works on, which gives us an answer to Question 4. This significantly reduces implementation overhead and broadens applicability.

Our results show that the algorithm we designed can identify important header fields within a few packets from connection setup, and provides reliable identification (Question 1) without noticeable misassignments (Question 2).

1.4.3 Rate Adaptation for ACK-Less Communications

While we designed the previous two approaches to be generally applicable and independent of any specific MAC implementation, we also considered their use in practical systems. Due to its wide-spread use as wireless technology on the last hop, full support for WLAN (IEEE 802.11) is highly desirable. To effectively utilize error tolerance schemes, including heuristic header error recovery, in WLAN, we need to solve an additional challenge specific to that system.

As explained in the introduction, error-tolerant systems do not benefit from ARQ and the resulting retransmissions. However, while it is possible to disable retransmissions in WLAN by disabling acknowledgments of successful transmissions (ACKs), such ACK-less traffic is not well-supported by today's state-of-the-art rate adaptation systems. Because they use ACKs to recognize transmission successes, ACK-less

packets are treated as never successfully received, reducing transfer rates to the minimum and greatly reducing performance.

We therefore develop an On-Demand Feedback Rate Adaptation (OFRA), which instead monitors channel conditions and feeds back information to the sender, so that it can effectively choose a rate without relying on ACKs. OFRA allows effective and accurate rate adaptation for ACK-less traffic, thereby unlocking the full potential of error tolerant transmissions in WLAN, answering Question 5. Furthermore, network throughput with OFRA is higher than with state-of-the-art algorithms even for standard (ACKed) traffic. Thus, OFRA provides benefits even if no error-tolerant transmissions are used at all.

1.5 Heuristic Header Error Recovery

In this dissertation, we will rigorously investigate the concept of heuristic header error recovery. To make clear the meaning of this term, we will explain each constituent.

Our approach is heuristic, because it uses methods to reconstruct what the original contents of headers were, which might have been corrupted by transmission errors. It is heuristic because we lack information to guarantee coming to a certain, unambiguous, and correct decision. First, while checksum mismatches will tell us that some errors occurred during transmission, they do not contain any information about how many errors occurred, or which parts of the packet are corrupted. In the best case, the headers might not be corrupted at all, with errors being confined to the (application-layer) payload of the packet, but we cannot recognize this, either. The best we can do is hence to take the received data, match it against ongoing connections, and find the connection we believe the packet most likely belongs to. This means that we have to guess a good match without being able to guarantee that it always will be the correct match.

Since our approach is heuristic, there is a risk that we choose the wrong connection. A packet that should have belonged to connection A is then incorrectly assigned to connection B. We term this event a *misattribution* or *misassignment*. Since this is a highly problematic event that could potentially have catastrophic outcome for some connections, we need to make sure to keep this problem in check. As we will see in this dissertation, we will do this by setting up our heuristic header error recovery so that connections that are so sensitive to errors that even a single misattribution could cause a catastrophic failure are guaranteed to never see such an event, and by designing our matching heuristics so that error-tolerant applications will witness these events extremely rarely. Thus, our approach also is a heuristic in second way: it does not guarantee to always provide the correct and perfect solution, but we will show that it comes close enough for practical considerations.

Our approach deals with header errors because solutions to allow payload error tolerance already exist, and because applications that are error-tolerant likewise do. However, tolerating header errors is a field not yet investigated in detail.

Finally our approach considers error recovery. Recovery implies two factors: tolerance and repair. To be able to recover from errors, it is first necessary to be able to receive the corrupted packets. To do so, checksum checking has to be disabled or changed so that packets are not discarded prematurely. This is the first step of error tolerance. Next, if header fields are corrupted, especially those that identify the correct communication end-point (the socket), it is necessary to heuristically identify which socket the packet is destined for. By doing so, we will also know which information was contained in those identifying fields before corruption. From this follows repair, in that we now can repair those fields to the values that they should contain. We will see that we generally are not able to repair all fields, because some fields will contain information that cannot be deduced from knowing which connection a packet belongs to. Errors in those fields will still remain even after repair, but their potentially wrong content will be tolerated.

Thus, we arrive a the term "heuristic header error recovery".

1.6 A Note on Previously Published and Joint Work

This dissertation heavily bases on previously published work by the author, which in turn was based on Diploma, Bachelor, and Master theses by students who co-authored those papers. Throughout this work, those papers and theses are typically not referenced directly; such references would otherwise litter the text. If no reference is given, work in the following chapters or sections is based on the listed papers and theses, respectively:

- Section 3.2 is based on "Refector: Heuristic Header Error Recovery for Error-Tolerant Transmissions" [SAAW11], for which parts of the implementation were done by Mario Göttgens in his Bachelor Thesis "Heuristic Packet Repair for UDP/IP in the Linux Network Stack" [Göt11].

- Section 3.3 is based on "Iterative Source-Channel Decoding with Cross-Layer Support for Wireless VoIP" [BLV+10], a joint work with Tobias Breddermann, then a Ph.D. candidate at the Institute of Communication Systems and Data Processing (IND) at RWTH Aachen.

- Section 3.4 is based on "A Heuristic Header Error Recovery Scheme for RTP" [SOW13a] and the technical report "Support for Error Tolerance in the Real-Time Transport Protocol" expanding on it [SOW13b], which in turn are partly based on work done by David Orlea in his Bachelor Thesis "Error Tolerance for the Real-Time Transport Protocol" [Orl12].

- Chapter 4 is based on "Piccett: Protocol-Independent Classification of Corrupted Error-Tolerant Traffic" [SHW14], for which some fundamentals were laid in Martin Henze's Diploma Thesis "A Machine-Learning Packet-Classification Tool for Processing Corrupted Packets on End Hosts" [Hen11], and a refined algorithm was developed in Matthias Lederhofer's Diploma Thesis

"Classifying Corrupted Network Packets for Error-Tolerant Streaming Applications" [Led12].

- Chapter 5 is based on "A Receiver-Based 802.11 Rate Adaptation Scheme with On-Demand Feedback" [SHP+12], which in turn is based on Anwar Hithnawi's Master Thesis "An On-Demand Rate-Adaptation Mechanism for IEEE 802.11 Networks" [Hit11], co-authored and co-advised,respectively, with Óscar Puñal, then a Ph.D. candidate at the Mobile Network Performance Group at RWTH Aachen, who went on to amend this work for additional use cases. Several of the extensions proposed in Section 5.6 were devised in collaboration with Mario Göttgens, who implemented and evaluated them in his Master Thesis "On-Demand Feedback Rate Adaptation in the Linux Network Stack" [Göt13].

1.7 Outline

The remainder of this dissertation is structured as follows. In Chapter 2, we will present information about concepts that are fundamental to the topics of this dissertation, as well as work related to the field of error tolerance. We will present the concept of protocol-specific heuristic header error tolerance in Chapter 3, as well as give some insight into the practical implementation and evaluation results that show the feasibility and efficacy of our approach. In Chapter 4, we will do the same for protocol-independent heuristic error recovery. After we have presented these two error recovery strategies, we will focus on how rate adaptation for ACK-less communication can be implemented in 802.11 in Chapter 5. To this end, we will first explain why ACK-less communication is important for error tolerance. Then, we will explain our on-demand feedback rate adaptation mechanism, how it works and how it is implemented. Afterwards, we will provide in-depth evaluation to show the effectiveness of this novel rate adaptation scheme. Finally, we will conclude this dissertation in Chapter 6.

2

Science is made up of so many things that appear obvious after they are explained.

—Frank Herbert, *Dune*

Background and Related Work

Before we start with the presentation of our contributions in the following chapters, we will give some insight into topics that are relevant for this dissertation. In the first part of this chapter (Sections 2.1–2.5), we will give basic information about topics that form the basis of concepts that we assume as being known in later chapters. While many of the topics presented here will be discussed in more detail in later chapters, the basic knowledge contained in this chapter is vital for the understanding of many of the concepts of our contributions.

In Section 2.1, we will give a short introduction into the concept of communication protocols and protocol headers. We will then discuss why errors occur in transmissions, with a special focus on wireless transmissions, in Section 2.2 and the closely related topic of how checksums are employed to recognize these errors in Section 2.3. The concept of acknowledgments will be introduced in Section 2.4. Section 2.5 concludes the background section with a short overview over the basic concepts of Wireless LANs as defined in the IEEE 802.11 standard.

In the second part of this chapter (Section 2.6), we will discuss related work relevant to the topics of this dissertation, specifically to heuristic header error tolerance. We will present related work from the fields of error-tolerant protocols, heuristic error recovery, reduction of retransmissions via partial packet recovery or packet reconstruction, and header compression.

2.1 Internet Protocols

Protocols fulfill an important job in data communications. They are especially important in packet-switched communications, that is, in communications in which there is no dedicated, exclusive physical connection between the sender and the receiver. Instead, the connections run over a shared medium, called line or channel,

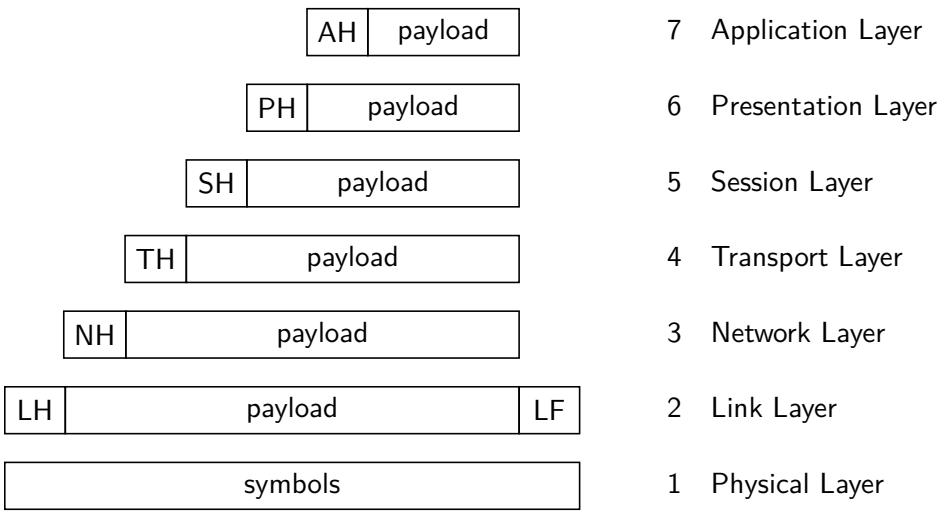


Figure 2.1 The 7-layer OSI model. When sending, each layer adds a header or footer to the payload, and passes it on to the lower layer, which considers the received data as its payload. Conversely, headers are removed bottom-up when receiving.

or a concatenation of such lines, shared by network participants. This is the case in the Internet, in which a large number of shared lines forms a mesh-like network, with special *gateway* or *router* nodes to forward the data along the correct lines to reach the receiver. To facilitate this sharing, participants group their data into packets, and those packets are then sent one after the other. Hence, in such a system, data from multiple users is multiplexed onto a common line.

This requires that each packet contains identification information that allows the routers to identify which connection a packet belongs to, and forward it accordingly, so that it eventually reaches the receiver. This information is typically carried in the form of an address that identifies the receiver and is carried in a piece of data called the header, which is prepended to the actual data (the payload). The specification of what additional information is carried in the header, how to write and read it, and specific communication behavior resulting from this information, forms a communications protocol.

Protocol Layer Models

In practical Internet communications, packets carry more than one header, and more than one protocol is used for each connection. This is due to the modular design of proctors: each protocol serves a specific purpose, and by using several protocols in combinations, the desired combined behavior can be reached. Protocols are organized in the form of a stack, in which several layers of protocols exist.

A widely-used theoretical model for this design is the OSI model, which defines 7 layers of protocol stacked on top of each other. A representation of the model is given in Figure 2.1. Each layer in the model fulfills a certain job: for example, the application layer only deals with the data that is to be sent to the receiver; the network layer deals with addressing the packet so that intermediate gateways know how to reach the receiver; and the physical layer transforms between the digital

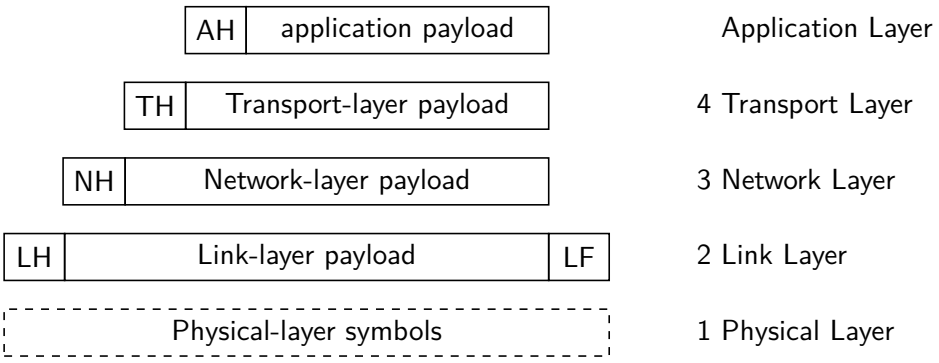


Figure 2.2 The Internet model reduces the number of layers compared to the OSI model. Note that the physical layer is not part of the specification, but is implicitly assumed to exist.

data and physical waveform representations of it that can be sent over a physical line (e.g., copper, fibre, wireless). On each layer, there may be several different protocols available. For each connection, one of those protocols is chosen that facilitates communication (e.g., it would not make sense to choose a physical-layer protocol designed for copper wires when using a fibre connection). Each layer is transparent to the others: at the sender side, a protocol p_n on layer n receives a chunk of data from protocol p_{n+1}, which contains that protocol's header, plus all the data it had received in turn from protocol p_{n+2}. In turn, it will add information that it requires the receiver to have in its own header, and forward payload and header to protocol p_{n-1}. In some cases, a footer may be added instead or in addition to a header. In Figure 2.1, this is done by the Link-layer protocol, a typical behavior of such protocols, which tend to put checksums there. On the receiver side, the opposite process is done: the stack is traversed from bottom to top, with each protocol removing its header and passing the payload up to the next protocol.

While the OSI model is widely used for theoretical modeling, a simpler model model is used in practice. One of the reasons is that layers 5 to 7 in the OSI model can easily be done by the applications themselves. The Internet model, the standard protocol stack for communication over the Internet, is shown in Figure 2.2. Note that the RFC-standardized version of the model [Bra89] only mentions four layers and does not concern with the physical layer. It implicitly assumes its existence, but considers it part of the network adapter hardware and hence out of scope of its specification. The number of different choices of protocols on each layer which are in practical use is very diverse: while each application can create its own application-layer protocol and hence, their number is very high, the number of transport- and network-layer protocols is very small. In today's practical use on the Internet, only three protocols are used: TCP and UDP on the transport layer, and IP on the network layer (though there exist two versions of IP in common use, IPv4 and IPv6). The number of link-layer protocols is again higher: each link technology typically defines its own Medium Access Control (MAC) protocol, a term that in practice is used interchangeably with the term link-layer protocol. Popular examples are Ethernet for wired connections and 802.11 for wireless connections, while mobile phone networks use their own link-layer technologies and protocols.

This discrepancy leads to the term of the "narrow waist of the Internet" [Ros08], in which two protocol layers are governed by a very small number of protocols. The practical reason for this is that these layers form the basis of compatible internetworking: the transport layer creates an end-to-end connection between two end hosts; the network layer ensures that packets can reach the end host of this end-to-end connection by implementing routing of packets among intermediate gateways. To fulfill these roles, all participants in the Internet need to speak the same protocols on these layers, which means that any new protocol on these layers suffers from the fact that it needs to be deployed throughout the Internet before it can be used. The fact that, despite the pressing problem of running out of IPv4 addresses, IPv6 took many years to be deployed in most of the Internet underlines this problem.

For this dissertation, this narrow waist is beneficial because recovery techniques that benefit protocols on the network and transport layer immediately have the opportunity to produce gains for virtually all users of network communications. Consequently, a large fraction of this dissertation will focus network- and transport-layer protocols as examples to implement heuristic header error recovery.

The Role of Headers

We already mentioned the role of headers in transmitting protocol-specific information. The arguably most important one (at least within the context of this work) that a header contains is the demultiplexing information: On a link which is shared by many connections from different users, it is important to recognize which packet belongs to which receiver. This information is important on layers 2, 3, and 4. The link-layer protocol is specific to the currently used link, and as such contains the layer-2 address (the MAC address) of the next hop that should receive the packet (called a *frame* in the context of layer-2 communications). To determine this next hop, intermediate routers inspect the layer-3 address (the IP address). Via a routing table that assigns each IP address range the corresponding MAC address of the next hop, routing is realized in the Internet. By using the IP address, the packet can be globally routed from the sender to the receiver. At the receiver, the demultiplexing information in the layer-4 header becomes important, which is given in the form of a port number. This port number identifies which of the numerous connections an end host might have open at any point in time a packet belongs to on that end host.

In addition to this demultiplexing information, headers can contain much additional information that is specific to the protocol. One field that is commonly contained in the header is the next-protocol field. In a layer-n protocol, it contains information about which protocol was used on layer $n+1$. Thus, the receiver always knows which protocol handler needs to process the packet on the next higher layer. Another common field is the checksum, which can secure either the header alone or both header and payload, and allows the receiver to recognize potential data corruptions during transport of the packet. Note that a protocol on layer n whose checksum secures the complete packet only will secure what it sees of the packet, that is, its own layer-n header as well the layer-n payload. Other fields can contain a hop count which is decreased at every intermediate hop and prevents infinite cyclic routing of

packets in case of misconfiguration of routes; sequence numbers that allow a protocol to recognize packet loss and reordering; or various flags (single-bit fields) that notify the receiver of certain conditions.

Sockets and Connections

One effect of the fact that protocols are designed to be as transparent as possible to other, higher-layer protocols that use them, is that sending and receiving data over the Internet is relatively easy for applications. The standard way of using connections under such commonly-used Operating Systems (OSs) as Linux, Unix derivatives, and Windows, is to create a so-called socket. To set up a socket, the application has to provide information such as the destination address and port, and which transport-layer protocol to use. Afterwards, the socket can be used like an ordinary file,[2] using the standard read and write procedures available on the OS.

Thus, any ongoing connection in a host maps to a socket that it was opened with. Conversely, every socket represents an ongoing connection for which data can be sent or received.

Note that by using this terminology, we somewhat extend the notion of a connection. In most literature, "connection" is only used for information exchange between two network participants for which a common context has been established. Thus, data communication via TCP, which requires setup of this context, is named a TCP connection. Conversely, data communication via UDP does not set up a context (at least not on the transport layer, as opposed to TCP), and as such, the term "UDP connection" is generally not used [TW11, KR13].

However, in this dissertation, we will use the term "UDP connection" to denote the fact that an application has opened a socket to receive UDP packets from another network participant. The main reason is one of convenience: there does not appear to be a concise equivalent of the term "connection" for UDP communications, and replacements such as "opened sockets awaiting reception of UDP datagrams" or "ongoing exchange of data (via UDP)" are cumbersome and inelegant.

While the above mentioned references tie the term "connection" to a setup of a common context, for us, the most important part is not that a connection setup has occurred, but that some application is waiting to receive data. Our notion of "connection" is hence more abstract and, in a literal sense, high-level, because it is concerned with the setup of an intent to communicate on the application layer by creating a socket.

[2] Or, more correctly, an ordinary file descriptor. A socket is treated as a stream, not a regular file, the difference being that ordinary files typically allow the user to seek (jump within the data contained in the file), while a stream requires sequential reading and writing, and does not allow seeking.

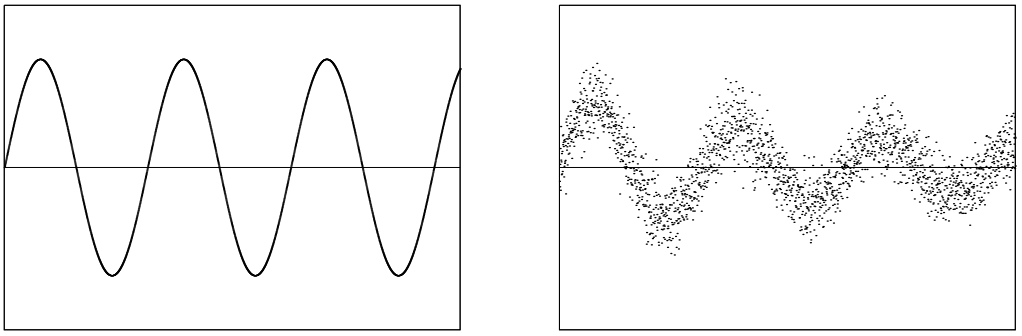
2.2 On Errors

One of the fundamental properties of data communications is that errors can occur during the process. In the following, we will discuss shortly why this is the case, why this especially is a problem for wireless communications, and what can be done to overcome this problem.

Data communication relies on analog signals. Even digital communication will have to convert the digital information into analog signals at some point. In fact, this is not specific to communications: data storage is effectively analog, as well. This conversion between digital and analog signals is the job of the modulator, which encodes the digital information in analog form onto an analog baseband channel at a certain carrier frequency. This translation does not have to use a bit-by-bit representation of the binary information: several bits can be, and for performance reasons generally are, combined into one so-called physical layer *symbol*. To transmit N bits in one symbol, an alphabet of 2^N distinct symbols is needed. These symbols can then be modulated onto a carrier wave in different ways, for example, by modifying the amplitude, the frequency, or the phase of the signal, each symbol changing the chosen criteria by different values. Demodulation occurs on the receiver's side by sampling the received analog values over that connection and retranslating them first into symbols and afterwards bits. As the number of distinct symbols increases (and all factors that we will abstract from here, such as bandwidth, stay the same), the possible differences between symbols decrease, as there is only a limited range of values available.

During transmission of a signal, as the waves cover the distance between sender and receiver, several effects act on the them. The first effect is the path loss that attenuates the signal. This attenuation depends on many factors, such as the frequency of the carrier wave (higher frequencies typically suffering more), the medium through which the signal propagates, and of course the distance between sender and receiver. In so-called free space models, which assume both sender and receiver to be in a completely empty area, floating in vacuum, the attenuation of electromagnetic waves is assumed to be in a quadratic relationship to the distance: if distance is doubled, the attenuation quadruples, and signal power at the receiver consequently is reduced to one quarter. In more realistic scenarios with obstacles, attenuation is even stronger. This effect further reduces the differences between symbols, making it harder to distinguish between them.

The second effect acting on the channel is noise. Typical sources of noise are due to unavoidable imperfections in the hardware, so-called thermal or Johnson–Nyquist noise, or from inadequately shielded electronic devices. The third effect acting on the channel is interference. This is the result of other transmissions that are received by the receiver at the same time as the intended signal, leading to superposition of the two signals. These superpositions can result in a stronger (constructive interference) or weaker (destructive interference) signal. In its weakest form, this can be nearly indistinguishable from noise, for example, if the source of the interference is another sender at a considerable distance. At closer distances, however, interference can lead to total loss of information, because the superposition changes the signal to



(a) Original wave (b) Wave after channel effects

Figure 2.3 Example of channel effects acting on a sine wave. Assume a simple sine wave was sent by a sender. At the receiver, the signal has suffered from path loss (overall reduced amplitude), fading (reduction in amplitude over time) and noise.

the point that the symbols are unrecoverable. A sender can also interfere with itself at a receiver: this typically occurs due to multipath propagation, in which a signal, when meeting an obstacle such as a wall, is scattered, that is, reflected in several different directions. After such reflections, the copies of the signal will have to travel different distances to the receiver and thus be received in multiple copies which arrive slightly offset from each other in time, leading to an interference of the signal with itself, a phenomenon termed inter-symbol interference, for its effect that one transmitted symbol interferes with an earlier or later symbol of the same transmission. Depending on the strengths of each signal, the effects can range from mild to severe. Such interference generally is not static over time; as the environment changes and signal reflectors move, whatever minutely, the strength of interference changes, and with it the measured amplitude of the signal. These time-dependent effects are termed *fading*.

The distinction between noise and interference can be somewhat vague, in that both act on the signal in the same way, by superpositioning unwanted changes onto it. Typically, the distinction is made by source. Interference is other signals that interfere, while noise is any unwanted modification of the signal from sources that are not recognizable as communication signal, and is often considered a random distortion, such as in the case of Gaussian noise, which is a popular noise model in simulations and analytical models.

Figure 2.3 gives a simplified example of channel effects acting on a sine wave. As can be seen, noise has distorted the sine wave pattern, and path loss and fading have reduced the wave's magnitude, which also changes over time.

To get a better understanding of the point at which errors occur, constellation diagrams are a helpful tool. These diagrams are used to visualize Quadrature Phase-Shift Keying (QPSK) and Quadrature Amplitude Modulation (QAM) modulations, two widely-used types of modulation in digital communications (for example, in WLAN), in which the phase of the signal changes, and those phase changes are measured by the receiver by regular sampling of the received analog signal. By

combining two orthogonal signals, that is, two signals that are independent of each other, such as the sine and the cosine function, these modulations can at least encode two bits in every symbol, by phase-shifting (or refraining to shift) either of the two constituent signals. Even more bits per symbol are reached by higher-order PSK or QAM modulations, which either use more fine-grained phase-shift steps, or add amplitude modulation to phase shifting, or a combination of both. For our purposes, it is simply important to know that different physical-layer symbols can be visualized by constellation points in a two-dimensional plane, each dimension visualizing the phases in one of the orthogonal signals.

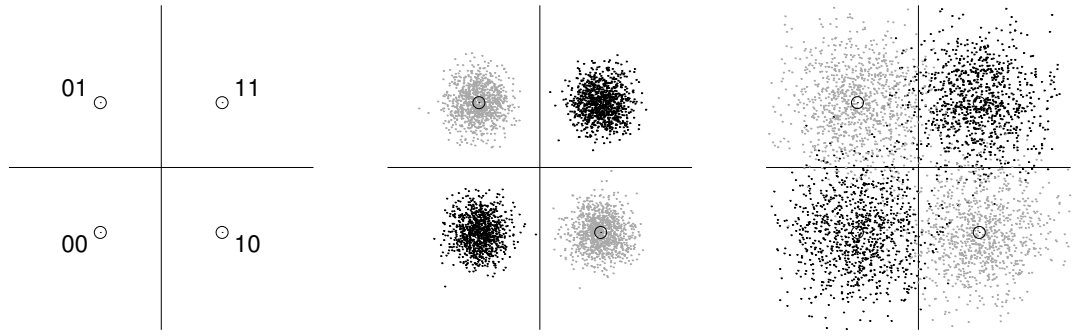
Figure 2.4a shows the constellation points for QPSK, in which the four different possibilities for phase modulation are denoted by points in a two-dimensional plane. These are the expected values for received signals: when regularly sampling the channel, each of the receiver's readings should, under perfect conditions, fall exactly onto one of those points. When noise is introduced, the readings become less accurate, and the data points form a cloud around each constellation point (cf. Figure 2.4b). The demodulator takes each imperfect reading and decides which symbol to report by calculating the Euclidean distance to each constellation point and taking the closest point's bits as result. Note that the constellation points are ordered in a form of Gray coding, in which the points which decode to information with larger Hamming distance have a larger Euclidean distance to each other, minimizing the number of bit errors. Such errors occur once the noise becomes so strong that samples read by the demodulator are closer to the wrong constellation point than the correct one. Under high-noise conditions, such as in Figure 2.4c, this can occur regularly and will lead to errors in the received data. This also gives a visual understanding of the tradeoff between robustness and speed of different modulations: as the number of bits per symbol is increased, speed increases, but also more points have to be added to the constellation diagram. As more points are added, the distance between points decreases, and errors occur at lower noise levels.[3]

It should be noted here that an error during demodulation is not tantamount to packet errors. To offset occasional demodulation errors, error-correcting codes can, and usually are, used. These allow the correction of a certain number of bits within a data block by introducing redundancy, with said redundancy being the deciding factor of error-correction power: the more redundancy is introduced, the more errors will be able to be corrected. This forms a tradeoff between robustness (being able to correct more potential errors) and overhead (not being able to send as many data bits in the same time).

We will go into more details about modulation, coding, and their effect on robustness and throughput when we investigate rate adaptation in Chapter 5. For now, it should suffice to understand the following:

No modulation and no coding, however robust, can guarantee error-free communication. For every modulation, we can introduce enough noise that demodulation errors will occur, and with noise being a random effect, a situation that corrupts at

[3]Provided other factors are kept equal; for example, increasing the amplitude increases the distance between points in the constellation diagram, but effectively simply means increasing the signal-to-noise ratio.



(a) constellation diagram (b) Low-noise samplings (c) High-noise samplings

Figure 2.4 Constellation diagram for QPSK. The four possible symbols are visualized in a two-dimensional plane, each dimension denoting values of one of the two constitutent orthogonal signals. The points denote the sample values expected at the sampling instants under no errors (left). As noise increases, the actual readings scatter around the expected values. As long as noise is low (center), the correct results can be recovered by taking the closest constellation point to the reading by Euclidean distance. As noise increases (right), readings divert from the correct constellation points so strongly that other points are closer: errors occur.

least one symbol will occur eventually. Or, figuratively, in the picture of Figure 2.4: however far we spread out the constellation points from each other by removing them farther from the origin (which, in effect, means an increase in amplitude, that is, transmission power), there exists a situation in which noise is strong enough to move a sampling point closer to another, wrong constellation point. Coding does not solve these problems absolutely: no matter how much redundancy is introduced, random noise will eventually, however unlikely, corrupt every single symbol in *some* packet, overcoming even the most robust error-correcting codes. There is no possibility to guarantee error-free communication.

Of course, as the modulation and coding become more robust, it becomes increasingly unlikely that enough errors occur to lead to residual errors after demodulation and decoding. However, these extremely robust modulation and coding techniques also become increasingly impractical, being extremely inefficient in situations with less noise. In practical scenarios, the goal is not to prevent errors at any cost, but to keep them at a rate that is low enough not to detract from the overall performance. In Chapter 5, we will discuss the concept of rate adaptation in detail. One of the tasks of rate adaptation is therefore to find a good tradeoff between the robustness and overhead as channel, and switch between different modulations and coding strengths as channel conditions change to stay at that tradeoff point.

2.3 On Checksums

Since it is not possible to create total reliance, and increasing reliance via error-correcting codes leads to increasing overhead, a typical approach is to send a small code with the packet that does not aid in correcting, but at least allows detection

of errors. The simplest of these codes is the parity, in which only one bit is added to a message. This bit is set to 0 or 1 to guarantee that the number of 1-bits is either always odd or always even, depending on implementation. This, however, is not a very robust method to detect errors: if an even number of bit errors occurs in a message, then the parity bit will not recognize the error.

For more reliable error detection, we need more sophisticated codes. The state-of-the-art in network communications, especially on the MAC layer, are Cyclic Redundancy Checks (CRCs) [PB61], which are based on special generator polynomials that have been chosen to detect errors. These polynomials are represented as elements of $GF(2)$, a Galois Field of order 2, that is, in binary representation. A simple example is the polynomial $x^4 + x + 1$, which is represented as `10011`. The bit stream that is to be secured by the CRC has a number of zeroes appended to it equal to the degree of the polynomial, and the bitstream is then divided by the generator polynomial in binary division, which is equal to eXclusive OR (XOR) operations. While the quotient of the division is ignored, the remainder is added to the outgoing data, replacing the zeroes added in the first step. These added bits form the checksum. An example of the CRC operation for a short example packet with the simple $x^4 + x + 1$ CRC is given in Figure 2.5. At the receiver's side, the data, with the checksum, is divided by the same generator polynomial. Since the sender added the remainder of the division to the sent data, the division is expected to not produce any remainder on the receiver's side. Hence, if the result of the division is 0, the packet is accepted. This is what we mean when we speak of "matching checksums" in the following: the checksum added to the packet lead, on the receivers side, to an acceptance decision. Conversely, if a checksum check on the receiver's side does not produce a result of 0, we speak of a "checksum mismatch" or a "broken" or "corrupted" checksum.

This position of the checksum as the remainder, added to the least significant bits of a packet, is one of the reasons that CRC checksums are often added to the packet as a footer, instead of forming part of the header.[4]

Note that CRCs, just as parity bits, cannot guarantee error detection with absolute certainty, though their failure risk is much lower. Since CRCs are shorter than the packets they secure, there are always several packets with different contents that share the same CRC. The goal of a good CRC polynomial is to detect as many possible error cases as possible, especially those deemed likely to occur in certain situations. For example, the CRC-32 polynomial $x^{32} + x^{26} + x^{23} + x^{22} + x^{16} + x^{12} + x^{11} + x^{10} + x^8 + x^7 + x^5 + x^4 + x^2 + x^1 + 1$, which yields a CRC of length 32 (4 bytes) and is widely used, for example, in Ethernet [IEEE12a] and WLAN [IEEE12b], can detect all single bit errors, all error bursts up to a length of 32, as well as any larger bursts of odd length [TW11, p. 235]. On the other hand, in a worst-case scenario, only 4 flipped bits in certain positions inside a packet of sizes typically used in network communications can already lead to non-detection: the packet contains errors, but the checksum matches. This is suboptimal, and it has been shown that

[4] Another reason is that this setup allows a just-in-time calculation of the CRC at the sender's side: the CRC calculation can be done with XOR gates added to shift registers, which the packet can pass while it is already being prepared for sending, or even sent. The result of the CRC calculation is available the instant the packet has cleared the shift register, and the checksum can then be added to the end of the bit stream.

```
Division:          1 1 0 1 0 0 1                 : 1 0 0 1 1

Calculation:       1 1 0 1 0 0 1 0 0 0 0 : 1 0 0 1 1
                   1 0 0 1 1
                   ---------
                     1 0 0 1 0
                     1 0 0 1 1
                     ---------
                         1 1 0 0 0
                         1 0 0 1 1
                         ---------
                           1 0 1 1 0
                           1 0 0 1 1
                           ---------
                               1 0 1

Resulting packet:  1 1 0 1 0 0 1 0 1 0 1
```

Figure 2.5 Example of a CRC checksum calculation for a simple 8-bit packet and the generator polynomial x^4+x+1. A number of zeroes equal to the degree of the polynomial are appended to the packet, which is then divided by the binary representation of the polynomial. The remainder is added in place of the added zeroes.

other CRCs of the same length do not suffer from this shortcoming, and can increase the minimum Hamming distance for an undetectable error from 4 to 6 for similar packet sizes [Koo02].

Another checksum approach is used by IPv4, UDP, and TCP to secure its header. In this case, a 16-bit checksum is used, which, in the words of the RFC [Pos81], is calculated as "the 16 bit one's complement of the one's complement sum of all 16 bit words in the header" (and payload, in the case of TCP and UDP). This effectively means to split the header (with the checksum field itself assumed to be 0) into 16-bit blocks, calculate the sum of all those blocks, take the carry bits that do not fit into the 16 bits, add them to the sum, and then invert it. So, for example, a checksum over the three values `0x9CBB`, `0xD2A3`, and `0xEAD0` would produce a sum of `0x25A2E`. Rolling over the carry bits produces `0x5A30`, whose one's complement is `0xA5CF`. This checksum has the disadvantage that as few as 2 bits can lead to undetectable errors. On the other hand, it is uniquely suitable for a protocol such as IP. Since the IP header contains the Time to Live (TTL) value, which is decremented at every hop, the header contents change at every intermediate routing system, and require a recalculation of the checksum. The layout of this checksum allows updating without having to do a full recalculation [Rij94].

To summarize, checksums are not a perfect measure to detect errors in packets: there always exist bit patterns that cannot be recognized by the checksum unless it contains at least as much information as the packet itself, defeating its purpose, especially since the checksum itself also has to be transmitted and hence is susceptible to bit errors.[5] However, checksums can be designed in ways that reduce the risk

[5]Furthermore, because a checksum cannot correct errors, a large checksum is undesirable because it increases the packet size and hence the risk of having at least one bit error in the packet, without offsetting this by error-correcting techniques, effectively increasing the risk that the packet is considered corrupted.

of missing an error to exceedingly rare corner cases. Consequently, checksums are generally considered a reliable measure to detect errors in packets: if the checksum matches, the packet is assumed to be correct; if it fails, then the packet is assumed to contain errors.[6]

Because checksums cannot exactly localize the error in the packet (because otherwise, they would be able to correct the error, making them error-correcting codes), they can only decide whether or not a packet contains at least one bit error. If an error occurs, the standard behavior of packet-switched networks is to discard the packet. This is a conservative approach that ensures that no transmitted payload will contain errors after transmission, and that no control information in the headers is corrupted that will interfere with packet processing. However, in a system that guarantees perfect bit-by-bit transmission of payload data from sender to receiver, this alone is not enough: the data needs to be retransmitted, to give it another chance to arrive at the receiver, this time in pristine, error-less condition. To ensure that all data is indeed transmitted, a feedback mechanism between receiver and sender is required.

2.4 On Acknowledgments

This feedback mechanism is typically a form of acknowledgments. In its original form, an acknowledgment is an information that a packet was received correctly, explicitly signaled from the receiver to the sender of the message. After reception, a special acknowledgment message is sent as feedback. If the original sender receives the acknowledgment, it considers the packet as correctly transmitted, and ends the transmission event for this packet; otherwise, it will retransmit the same packet, potentially several times, until it has been informed about correct reception (potentially up to a certain limit of retransmissions before giving up).

The way this signaling is done depends on the MAC protocol that is used, which is designed with the characteristics of the physical channel in mind. Such channels can differ strongly in the expected number of error events that can occur.

In Ethernet, for example, bit errors due to channel effects are exceedingly rare. Ethernet frames are secured by a checksum, but no acknowledgments are used by the Ethernet protocol. If an Ethernet frame was transmitted correctly, it is assumed to have been correctly received, as well. The main source of errors in Ethernet are collisions: the Ethernet frame cannot be correctly received any more because more than one network participant sends at the same time, leading to interference of signals.[7] If such an interference is recognized (typically, by the sender of a frame

[6] Note that this counter-assumption is not 100% reliable, either: if the checksum has been corrupted, but the packet has not, the packet will be still considered erroneous, even though the actual packet data is error-free.

[7] At least in classic Ethernet, this is the case. However, for many years, the standard in Ethernet deployments has been full-duplex, switched Ethernet: each participant has a dedicated connection to a dedicated switching infrastructure, and vice versa. In such a deployment, no collisions are possible.

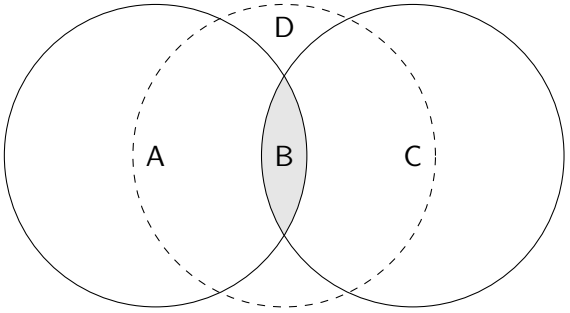


Figure 2.6 Hidden Station Problem: Both stations A and C want to send to B. Because they are outside of each other's reception range, they cannot hear each other, and may send at the same time, leading to a collision in the gray area, and consequently at B.

listening to the shared channel while sending, and noticing different signals than the ones sent by itself), a special jamming signal is sent that informs all network participants of the collision. This leads to a special case of acknowledgments, negative acknowledgments, in which the failure of a transmission is signaled instead of the success.

This can only be done successfully in settings where complete and unrecognized loss of a packet is impossible. Thus, such approaches are generally unsuitable for wireless communications, in which the medium is both shared and witnesses strongly varying channel effects not only over time, but also in space, such that the transmitter cannot infer the receiver's channel state from its own channel state information. An example that visualizes this problem is the well-known hidden station problem, given in Figure 2.6. The circles denote the range of each station, that is, the area in which their signal can be received. Both A and C send to B at the same time, leading to a collision of packets at B. Neither A nor C can recognize that such a collision occurs, because they are outside of each other's range.[8] Hence, an Ethernet-like negative acknowledgment is problematic: neither A nor C detect the collision, and therefore cannot react; and while B could, this would also lead to problems, since its jamming signal would also be received by station D. If that station were to be sending to an unrelated station, the B's jamming signal would cause it to retransmit its packet, even if no collision occurred at its recipient's location. Furthermore, in wireless scenarios, collision is only one channel effect that can lead to errors. The other channel effects explained in Section 2.2 (attenuation and noise) are also important. Again, the sender cannot infer from its local channel effects those at the receiver.

This leads to the use of per-frame acknowledgments in Wireless LAN (IEEE 802.11), the most commonly used standard for wireless communications outside of cellphone systems. After every frame, the receiver sends a special acknowledgment frame back to the sender if reception was successful.

Such a scheme is not without specific problems and drawbacks. First, acknowledgments can be lost just as data frames can. If channel conditions vary, the reception of the data frame can still have been successful, while the conditions worsened to the

[8] In Ethernet, by comparison, such a scenario is impossible because the signal attenuation over the copper wire, even at the maximum lengths mandated by the Ethernet standard, is negligible, and every network participant is guaranteed to hear each other.

point that the subsequent Acknowledgment (ACK) frame was not received correctly by the data frame's sender any more. In such a situation, an unnecessary retransmission is triggered. It is often argued that ACK frames are small, and consequently the risk of them being lost due to corruption is likewise rather small, since the risk of corruption increases with frame length. This is true for some types of transmissions, for example, file transfers, in which frames are as large as possible to reduce header overhead, but not for all of them. A counter-example is VoIP or live streaming of audio. Hence, this possibility should not be disregarded altogether. In addition, the sending of the acknowledgment itself introduces signaling overhead: while the acknowledgment is sent, the channel cannot be used to send data.

For our scenario of error-tolerant transmissions, acknowledgments pose an additional problem. The concept of acknowledgments, signaling the correct reception of a frame, is fundamentally at odds with error tolerance. This is exacerbated by the fact that, in 802.11, acknowledgments are only sent when a frame was received completely correctly, without any bit errors, and the absence of an acknowledgment causes the sender to retransmit the frame. Such a behavior undoes most of the advantages of error tolerance: if a frame is retransmitted until it is finally received without any errors, the advantage of using the first, error-ridden, reception, is dubious. Furthermore, the channel is unnecessarily occupied by retransmissions, which means that one of the advantages of error tolerance, freeing up channel capacity due to the reduction of retransmissions, is removed. We will discuss the consequences of this conflict, and possible solutions, in more detail in Chapter 5, when we present a novel rate adaptation algorithm for WLAN that does not rely on acknowledgments.

For the main parts of our work in Chapters 3 and 4, we will, however, simply assume that no acknowledgments are used for our error-tolerant transmissions. We will explain how 802.11 can be used without acknowledgments in the next section. Two solutions to the question of how to manage concurrent error-tolerant transmissions without and error-sensitive transmissions with acknowledgments will be discussed in Sections 3.2.3 and 3.5.

2.5 Wireless LAN

Considerable parts of the work presented in this dissertation focus on the use of IEEE 802.11 WLAN, often also called WiFi. This standard is by far the most popular one for Internet-based wireless communication networks.

In the following, we will give a short introduction into WLAN with specific focus on the salient points with respect to this dissertation. Note that many of the aspects presented here assume a typical setup of infrastructure-mode, 2.4 GHz or 5 GHz range, with 20 MHz bandwidth channels using Orthogonal Frequency-Division Multiplexing (OFDM). Rarely used options, such as ad-hoc mode, different bandwidths, or different frequency ranges (or even substituting radio waves for infrared communication), often show differences in protocol behavior as well as in timing.

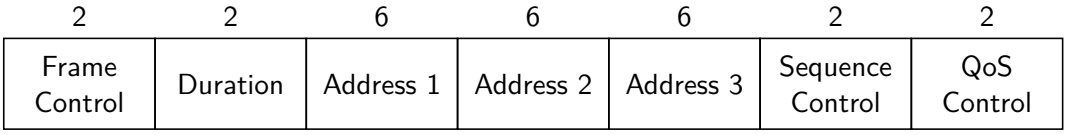


Figure 2.7 Layout of a typical 802.11 MAC frame header. The numbers above each field denote the size in octets. Depending on the type of frame (signaled in the Frame Control field), field can be added or removed from the header.

Communication Behavior and MAC frame format

As the name suggests, many of WLAN's design concepts stem from the idea to create a wireless system that works as a natural extension of wired networks. An infrastructure 802.11 setup comprises an Access Point (AP) and one or more Stations (STAs), with the AP playing the role of the centralized network infrastructure, like a hub or switch in the case of star-topology Ethernet, while the STAs are network participants such as laptops or mobile phones. 802.11's status as an "extended, wireless Ethernet" can be seen from the frame format (cf. Figure 2.7): as opposed to Ethernet frames that contain a source and a destination address, 802.11 frames contain typically three addresses (though in special cases only one or up to four). This mirrors the path frames take: even if two STAs within one network communicate with each other, they do so via the AP; if a STA sends data to the Internet or receives data from it via the (typically) wired connection that connects the AP with the Internet service provider, it obviously has to do so, in any case.

802.11 frames hence contain the sender MAC address, the receiver MAC address, and the MAC address of the AP, which doubles as the so-called Basic Service Set Identification (BSSID), which identifies the physical wireless network.[9] In the case of communication to and from outside the wireless network, one of these addresses is the MAC address of the "outbound" port on the AP. While the details on which address is put into which address field change depending on the type of message, the one constant is that the Address-1 field always contains the MAC address of the intended next immediate receiver of the message, which facilitates such options as power-saving by only listening and decoding received frames up to this point, and stopping reception if the address doesn't match the device's address.

The other header fields provide additional ancillary information. The Frame Control field contains various information about the frame, most prominently the type of frame (e.g., data frame or acknowledgment), which is important because the rest of the frame format depends on the type of frame. For example, an ACK only contains a single address: the receiver of the ACK, that is, the sender of the original data. The duration field informs all receivers of the frame of the length (in microseconds) that the current frame exchange will take, so that, for example, in a data frame, the duration also includes the time required to send the subsequent ACK. Since the field is early in the header, before the Address-1 field, every network participant will be informed about how long the channel will be occupied, and can go into a power-save mode for that time, if desired. The Sequence Control field, finally, contains

[9] As opposed to logical 802.11 networks, which can comprise several APs; but this is outside of the scope of this introduction

a sequence and fragment number. Thus, if fragmentation of frames received from upper layers is necessary, the fragments can be identified and reassembled at the receiver.

In WLAN, just as in the original Ethernet, all network participants share a common medium. Furthermore, even different networks may have to share the same medium, since the available spectrum is limited. To somewhat reduce this problem, several frequencies within the available spectrum, so-called channels, can be used. In the still popular 2.4 GHz range, only 3 non-overlapping channels exist, while in the 5 GHz range, this number is much higher (exact numbers vary based on regional regulatory restrictions). Due to the shared nature of the medium, a scheme to deal with collisions is necessary. However, as we have already seen in the previous section, Ethernet's collision detection is not feasible for WLAN. Instead, it uses a scheme called Carrier Sense Multiple Access with Collision Avoidance (CSMA/CA), which, while similar to Ethernet's Carrier Sense Multiple Access with Collision Detection (CSMA/CD), skips the detection step. In CSMA, the sender listens to the channel to check whether another network participant is currently sending. If this is not the case, it will send its data frame and expect an ACK. However, as in Ethernet, collisions are still possible due to timing issues and signal propagation delays. If both stations now immediately tried to retransmit their data, another collision would occur, leading to an endless sequence of failed transmissions.

To solve this problem (just as in original Ethernet), a random backoff is introduced. Both stations roll a random number from a certain range, termed the *contention window*, typically between 1 and 15 in the first round, multiply this number with the *slot time* of 9 µs, and wait that long before retrying the transmission, starting with the carrier sensing step again. Thus, whichever station rolls the lower number will be able to start, while the other station will notice that the channel is busy and wait until the channel is available again. If both stations happen to roll the same random number, or a third station interferes, another collision occurs, and another round of random backoff is introduced, with an increased contention window, up to a maximum of 1023 after several rounds, which reduces the risk of collisions while increasing overhead due to long backoff times.

Note that this, by itself, does not solve the hidden station problem, as depicted in Figure 2.6: A would still not be able to hear C and send its data, causing a collision at B. However, due to the missing ACK, A (and C) would notice a failure in transmission of its data. Nevertheless, even if both stations rolled different random numbers, a collision would still be highly likely to occur, because the frame transmission is likely to take longer than the difference in backoffs. While this problem tends to solve itself as the contention window increases, the price of several collisions is very high. Thus, CSMA/CA introduces a so-called RTS/CTS mechanism, in which a station that wants to send first sends a small Request To Send (RTS) frame that includes the time it will need to reserve the common channel to transmit its data. The receiver (if it is ready for reception and not currently receiving other data) will then answer with a Clear To Send (CTS) frame, in which it repeats this time information. The idea of the CTS frame is that it will be received by all network participants that are in range of the receiver, denoted by the dashed circle in Figure 2.6. Assuming channel reciprocity, that is, if B can receive data from A,

	TIDs	CW_{min}	CW_{max}	AIFSN
AC_BK (background)	1, 2	15	1023	7
AC_BE (best effort)	0, 3	15	1023	3
AC_VI (video)	4, 5	7	15	2
AC_VO (voice)	6, 7	3	7	2
legacy 802.11 (no AC)	–	15	1023	2

Table 2.1 Parameters for different access categories (ACs) in 802.11e. Lower CW_{min}, CW_{max}, and AIFSN increase chance of timely sending and hence QoS. The standard category for traffic in 802.11e is AC_BE.

then A can receive data from B, the CTS will be received by network participants that could cause a collision at B. Hence, once A sent an RTS and B answered with a CTS, C knows that it cannot communicate with B for the time announced in the CTS. The main downside of RTS/CTS is its increased overhead: instead of sending two frames (data and acknowledgment), a data exchange requires the sending of four frames. This is exacerbated by the fact that RTS and CTS, while small, are sent with a more robust modulation and coding.[10] Hence, RTS/CTS is typically not used for all data frames, but, if at all, only based on certain properties, most often a size threshold: if the frame is larger than the threshold it is secured with RTS/CTS, otherwise not. Thus, the otherwise extreme overhead of RTS/CTS for small frames can be avoided.

802.11e QoS Extensions

In original 802.11, all data was treated equally; there were no provisions for Quality of Service (QoS) on the MAC layer. However, the 802.11e extension, part of the 802.11 standard since 2007, provides for different priority classes of frames.

When frames are received from the upper layers and prepared for sending, they are inserted into a transmission queue, which either exists in software or directly in hardware on the network adapter. Instead of a single queue, 802.11e introduces four queues, for four different priorities, termed Access Categories (ACs). The main difference between these queues is the settings of CW_{min} and CW_{max}, the minimum and maximum size of the contention window, and the AIFSN, which governs the length of the Arbitration Interframe Space (AIFS), the time between sending subsequent frames. Due to the complexity of the topic of different interframe spaces in 802.11, and the fact that it is not important for the topics discussed in this dissertation, we will refrain from an in-depth discussion of this aspect.

The default values, as mandated by the standard, for these parameters are given in Table 2.1. Note that, while the default AC in QoS mode, AC_BE, has a higher

[10] The reason for this choice is that, especially for the CTS to work properly, it is important that all network participants in B's range are able to properly receive the CTS. By choosing the most robust available modulation and coding, the CTS has the maximum probability of being received correctly, and to reach even those network participants farthest away from B.

AIFSN than legacy traffic, and thus it seems as if it might be disadvantaged, subtle changes in the behavior of how to increment and decrement the contention window size in 802.11e lead to a similar performance [BTS05]. The Traffic Identifiers (TIDs) allow, in theory, a more fine-grained classification, modeling the 8 priority levels in Ethernet [IEEE14]. In practice, however, there is no difference between the two TIDs in each AC. Note that TID 0, the default, is assigned to AC_BE for backwards compatibility.

One effect of these different queues is that there are two layers of contention when trying to send a frame: in addition to the channel contention that all network participants go through if they want to send data simultaneously, there is also a local contention between different queues. Simply sending data from the highest-priority queue that has data available could lead to starvation for the lower-priority queues. Therefore, whenever a network participant wants to send data, it rolls random numbers for each queue it has data in, between 1 and that queue's current contention window size. The queue with the lowest roll wins and proceeds to contend for channel access. The lower lower values for CW_{min} and CW_{max} hence help both in the local as well as in the contention period to provide, on average, higher priority access to the channel.

The 802.11e QoS extensions introduce a second concept that is less known, but of vital importance for this dissertation. Whereas in standard 802.11, every frame is acknowledged, 802.11e allows switching off ACKs on a per-frame basis. This is done by setting a flag in the QoS control block of the MAC header, that signals to the receiver that it should not send an ACK. Since the sender has no feedback from which to infer whether transmission was successful, it only sends the frame once. This is especially helpful if retransmissions are assumed to be of little benefit, for example, due to the delays introduced for time-critical data. As we hinted at in the previous section, ACKs are problematic in combination with error tolerance. The ability to switch them off for certain transmissions is therefore helpful. In addition, not using ACKs has a beneficial side-effect. Under normal (ACKed) conditions, the sender reserves the channel both for the duration it takes its data frame to be transmitted, as well as the time it takes the sender to calculate the checksum and send the ACK. In No-ACK transmissions, this extra time is saved, and the channel yielded earlier, increasing the potential throughput.

Figure 2.8 shows results from our measurements [SAAW11] that underline the potential of No-ACK transmissions. For this experiment, we placed a STA close to an AP to produce good channel conditions and minimize frame loss, so that the limiting factor is the speed at which frames can be sent over the channel. The potential improvements are significant. Note that both small and large frames benefit. The largest absolute increase is realized with large frames, but small frames show the largest relative increase. This is because continuously sending small frames means more frames are sent, which means more time is spent sending ACKs in the standard case, which No-ACK can use to send more frames.

The fact that switching off ACKs is done on a per-frame basis is both beneficial and problematic. Beneficial, because a setting in which all traffic would be either always sent with or always without ACKs is infeasible: all error-sensitive traffic

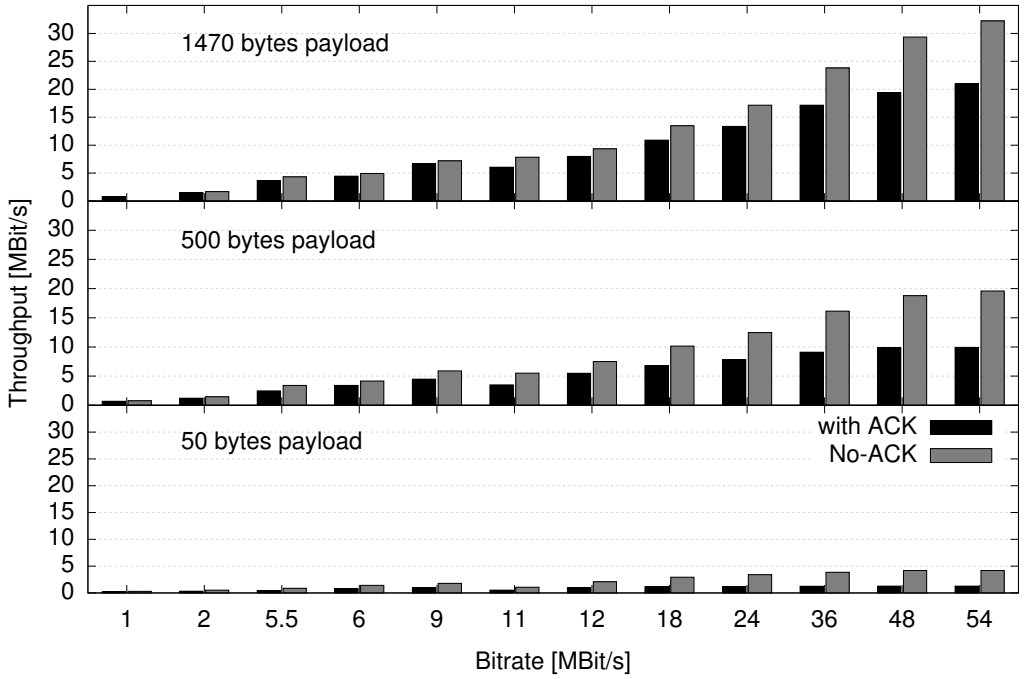


Figure 2.8 Application-level throughput of an 802.11e transmission under good channel conditions, with and without ACKs. Under these conditions, No-ACK increases throughput significantly. Larger frames produce higher absolute improvements, while smaller frame sizes benefit the most relative to with ACKs.

would suffer. On the other hand, the setting is per-frame, and hence has no concept of flows. By default, there is no provisioning in the standard of how to signal to the MAC layer which frames should be sent without ACKs. This is similar to the field of rate adaptation, which we will discuss later in this section: while it is to be assumed that there is some logic in place to switch between rates, the 802.11 standard does not mandate or suggest any ways to do so. Over the course of this dissertation, we will present two approaches of signaling to the MAC layer to make error-tolerant traffic ACK-less, while keeping ACKs for error-sensitive traffic: the first solution was implemented and will be described in Section 3.2.3, while a potential improvement will be discussed in Section 3.5.

Rate Adaptation

While wired communications can guarantee a steady communication quality at any point in time, the aforementioned channel effects lead to strongly varying signal quality in wireless systems. One approach to cope with these variations is to use different strengths of modulation and coding, and adapt their use to the current channel quality. This approach is called *rate adaptation* and allows the use of high-speed communication under good channel conditions, while reducing the rate to increase robustness under bad conditions. In 802.11, several rates, that is, preset combinations of modulation and coding that produce a nominal throughput rate, are defined. Depending on the version of the standard, the number of choices can be as low as two (in the original 802.11 standard) and as high as 32 (in 802.11n).

While these rates are defined in the standard, there are no defined ways to switch between rates. Much like congestion control in TCP, this has lead to many algorithms being proposed and used over the years.

One of the fundamental problems of rate adaptation in 802.11 is the scarcity of information, both spatial and temporal. Regarding spatial effects, the choice which rate to use in the next frame has to be done by the sender; however, the information about reception quality is only available at the receiver. Hence, some feedback mechanism is required to transfer this information. Many approaches use ACKs as implicit feedback mechanisms: if an ACK is received, the frame was received correctly, which means that at that point in time, the chosen rate was robust enough. However, this does not provide any information whether a higher-speed rate would also have been successful. To feed back more detailed information from the receiver to the sender, more sophisticated mechanisms are required that either send explicit feedback, or rely on the concept of *channel reciprocity*: if A receives frames from B at a certain channel quality, it is assumed that frames sent from A to B will be subject to the same channel conditions.

With respect to temporal effects, the problem is that a 100% reliable prediction of channel behavior in the future is impossible. Real-world channels show a chaotic behavior that can only be predicted to a certain degree from previous readings. The general approach is to assume that channel quality will not change significantly within a time frame termed the *coherence time*, so that channel quality for a frame that is to be sent can be inferred from the channel quality that recently received frames witnessed. The more chaotic the channel, the smaller the coherence time, and the more challenging predictions become. We will discuss the challenges and the different types of approaches to rate adaption in detail in Chapter 5, when we present a novel rate adaptation mechanism that forms part of the contributions of this dissertation.

The PLCP

One result of rate adaptation is that, depending on the data rate, frames are encoded with different strengths of convolutional coding, and then modulated in different ways. Especially the latter point leads to the problem that the receiver of a frame needs to demodulate the received waveforms into symbols, without knowing which demodulation technique to apply. However, using the wrong demodulation scheme produces a completely different output, corrupting the frame completely.

To solve this problem, each 802.11 frame contains additional information in front of the MAC header, the Physical Layer Convergence Protocol (PLCP) preamble and the PLCP header. The preamble contains a well-defined pattern that is the same for every frame, regardless of modulation, and allows the OFDM decoding unit to both recognize that a frame is about to be received, and to adapt its reception unit to the reception, by, for example, setting the correct gain and account for potential frequency offsets.

The PLCP header contains information about which modulation and coding the rest of the frame (starting with the MAC) header will use, and about the length of the

frame (in bytes). This information is always sent at the base rate: in OFDM at 6 Mbit/s, with Binary Phase-Shift Keying (BPSK) at rate 1/2, so that it is always clear how to demodulate and decode the PLCP header.

Within this dissertation, the PLCP header is not examined in detail and generally abstracted from. This is for two reasons, a fundamental and a practical one.

Fundamentally, an error in the PLCP header generally leads to a catastrophic failure during reception. If the length field is broken, the frame will either be truncated, or the receiver will listen to the channel after the frame transmission has finished, potentially much longer, leading to unexpected lockups or communication problems (since the receiver will be locked into reception mode). If the field denoting the modulation and coding is broken, the remainder of the frame will be completely corrupted.

Practically, the possibility to receive frames with errors in the PLCP header is much less prevalent in consumer hardware than to receive those with correct PLCP checksums, but broken MAC checksums. Consequently, when we investigated error tolerance in an 802.11 setup in Section 3.2.4, we let the hardware drop all frames with errors in the PLCP header[11] and considered them lost.

2.6 Related Work

Before starting with presenting our contributions in the next chapter, we will discuss work that is related to the topics of this dissertation. Note that this section deals with related work with respect to error-tolerance approaches or other approaches that try to assuage problems with erroneous links. Related work for rate adaptation is discussed separately in Section 5.3.

2.6.1 Error Tolerance

Error tolerance is one of the core concepts and motivators for the works presented in this dissertation. To give an overview over the current state of research in this field, we will first discuss protocols that introduce payload error tolerance, before discussing related work in the field of heuristic header error recovery.

Error-Tolerant Protocols

UDP-Lite, proposed in 1999 [LDP99] and later standardized as an RFC [LDP+04], is the initial point from which much of the following research was motivated. It changes the behavior of the UDP checksum. In standard UDP, the checksum is calculated over the pseudo header (cf. Figure 2.9), an extension of the UDP header

[11] Practically speaking, we did not have a choice on some cards; On others, we preliminarily investigated the feasibility of recovering from PLCP errors, but the aforementioned prevalence for catastrophic failures convinced us to not further consider this approach.

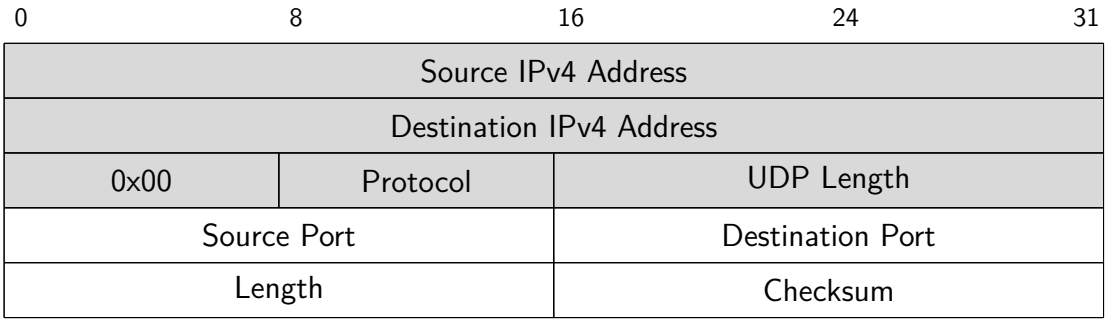


Figure 2.9 The UDP pseudo header (when used with IPv4) is a combination of a subset of the header fields of the IP header (light gray) and the UDP header (white) and is used for calculation of the checksum. The checksum is calculated over the fields of the UDP pseudo header (with the checksum field set to 0 during calculation) and the UDP payload.

with a subset of fields from the IP header, and the payload. UDP-Lite changes this behavior by redefining UDP's length field. Instead of containing the length of the UDP payload (which is redundant, because it can be calculated from the Total Length field of the IP header by subtracting length of the IP header itself), it contains the checksum coverage, which denotes over how many bytes the checksum was calculated. In addition to the header, the checksum can cover a variable portion of the payload, from none to all. This allows the application to, for example, put sensitive information in the beginning of the datagram and have it be secured by a checksum, while the remaining part of the payload is not secured. Hence, errors in this part will not lead to drops.

Even though protocols that did not secure the checksum were no new concept – IPv4, for example, does not secure the payload at all and only secures its own headers with its checksum – UDP-Lite was, to our knowledge, the first time a protocol was specifically changed to introduce error tolerance, and be standardized in this way. However, one major problem is the lack of backwards compatibility with standard UDP. Since the meaning of a header field is incompatibly changed, UDP-Lite is effectively a new protocol, despite its clear pedigree. To use UDP-Lite, both sender and receiver therefore need to support UDP-Lite by having a protocol implementation for this new protocol available in their network stack.

This downside is solved by UDP-Liter [LL04], which takes the idea of error tolerance from UDP-Lite, but implements it in a way that stays backwards compatible. Instead of using a checksum coverage field, UDP-Liter uses standard UDP checksums, but it does not discard packets when the checksum fails. Instead, it still forwards the packet to the application, but notifies it if the checksum failed. UDP-Liter took two very important steps towards heuristic header recovery: First, it introduced the concept of ignoring checksum mismatches and living with the consequences. It is not yet an error recovery concept, because there are no provisions to recover from or repair errors; there is simply the idea to ignore checksum hints that an error occurred. Hence, even though the paper acknowledges the problem, there are no provisions to prevent misattributions. The authors' rationale is that the problem is considered "small enough [...] to ignore". Second, UDP-Liter introduced the notion of notifying the application layer of checksum mismatches, a concept that we also introduced

in Refector. While the concepts are the same, UDP-Liter's solution is somewhat more cumbersome to implement and use, because it introduces additional system call functions to replace the standard socket creation and reception functions, while we solve this problem by an additional socket option and message flag, and keep using standard functions.

Heuristic Error Recovery

The field of heuristically recovering from errors is a relatively novel one, with a rather slim body of related work. Furthermore, there has been little coordinated effort in the area, with related work scattered among the fields of protocol design, wireless communications, and coding theory. While the prevalence of the word "Lite" in many of these these works' titles suggest that they have been inspired by the concept of UDP-Lite, none of them seem to have been aware of other, similar efforts. This leads to the effect that the basic concept of classifying header fields into categories such as "vital" and "don't-care" has apparently been discovered independently multiple times.

To the best of our knowledge, this dissertation is the first time that the concept of heuristic header error recovery has been tackled from both a comprehensive and practical point of view, and also the first time that this body of related work has been compiled into one discussion.

Alfredsson and Brunstrom propose TCP-L [AB03] and give some insights into the basic idea of their heuristic recovery. Similarly to Refector (protocol-specific header error recovery) in this dissertation, they recognize the advantage of categorizing fields by importance, and that heuristic recovery is possible if header fields of incoming messages are matched against expected values for ongoing connections. However, their approach is hampered by the focus on TCP. While they can show that their approach increases throughput in a simulated wireless scenario, they do not discuss the open questions that stem from the interaction between their approach and TCP's complex control loops, for example, the setting of correct receive and congestion windows. The question of if, how and when to send TCP ACKs in response to corrupted packets is also not discussed. Finally, the fundamental problem of misattribution due to erroneous heuristic repair choices is mentioned, but its probability of happening within their scheme is not discussed or evaluated by the authors.

Jiang presents a scheme [Jia06] that aims at introducing heuristic header error recovery into the 802.11 MAC protocol. In many ways, the contributions of this work mirror those of TCP-L. While there is no explicit classification of header fields into categories, Jiang also implicitly distinguishes between important and unimportant fields. However, the approach again suffers from idiosyncrasies of the chosen protocol. Since the 802.11 MAC protocol requires extremely timely acknowledgment of received packets, the question of whether or not to acknowledge a corrupted packet becomes much more pressing, due to the fact that the decision has to be done before any complicated recovery technique can feasibly identify whether a packet can be salvaged or not. In Refector, we avoid this problem by using the special No-ACK

capabilities of the 802.11e extension. Jiang, however, does not discuss this problem except for a suggestion to fundamentally (and incompatibly) change the ACK procedure in 802.11 networks. Finally, just like Alfredsson and Brunstrom, he recognizes the possibility of misattribution, but only gives a rough estimation of such probabilities depending on the bit error rate, and does not provide any evaluation results.

Khayam and Radha [KR07] propose a scheme for video transmissions that shares many similarities with Refector. For example, they propose header error recovery that spans more than one protocol. They also recognize that there are, in their words, "critical header fields", which are vital to the identification of a connection a packet belongs to. However, for identification of header errors, they rely on having an accurate bit error distribution model for the received data, which is hard to acquire, and, worse, cannot be generalized, since such distributions strongly depend on the used hardware [HJL+09, HJL+12]. Furthermore, this complicates their estimation calculations; conversely, we can show that a simple but computationally efficient Hamming distance calculation is sufficient. More so than other related work, Khayam and Radha seem aware of the problem of misattribution and try to alleviate it. However, their solution to the problem is specifically tailored to their use case of video streaming: to reduce their relatively high misattribution rate of, depending on scenario, more than 1%, they propose to provide additional FEC to the application-layer H.264 video header, which contains a sequence number, and to drop the packet on the application layer if an unexpected sequence number is received. Additionally, it is not clear how they prevent misattribution towards other concurrent connections that are not error-tolerant. This is exacerbated by the fact that they decide to always send ACKs, even when packets are corrupted: concurrent error-sensitive traffic will hence witness large packet loss rates.

In Mac-Lite [MLKD10], the authors approach the problem of header error recovery from coding theory. They also recognize that header fields are of different importance and predictability. Furthermore, they notice that header fields in protocols of different layers can rely on each other and hence carry additional redundancy. They give the example of the 802.11 PHYsical layer (PHY) and MAC length fields. While their approach is well-founded in theory, the practical applicability is questionable. One problem is that they use a decoder with exponential complexity with respect to the size of the received frame. This is exacerbated by the fact that they use soft-decision decoding, in which each bit is not hard-decoded to a 0 or 1, but instead bit probabilities (likelihood that a bit has a certain value) are used, which further increases the real-world computational overhead. Consequently, they only show simulation results and do not discuss the practical feasibility of their approach. Furthermore, the use of soft information precludes the use of consumer hardware. Finally, they do not discuss or even acknowledge the problem of misattributions at all. Nevertheless, if these problems could be overcome, their solution might be a natural extension to the concepts for heuristic recovery presented in this dissertation: since, while our concepts are applicable generally, our current implementations only consider heuristic recovery for protocols from the network layer upwards, and their approach focuses on the MAC and PHY layer, a combination might produce further improvements.

2.6.2 Reducing Retransmissions

Much work has focused on improving the efficiency of retransmission schemes. The standard behavior of retransmitting a complete packet is extremely inefficient if only small parts of the data have been corrupted. This motivates work that combines information from a retransmission with the original transmission to make it very likely that no additional retransmissions are needed. Other work, instead of focusing on reducing the number of retransmissions, tries to reduce their size by identifying the erroneous parts and only retransmit those.

The approaches can be roughly categorized depending on the amount of information that is used and hence required to be available. In the following, we will use two categories: approaches that do not require additional data other than what is available on the MAC layer; and approaches that require additional information from the PHY layer. The motivation for this categorization is that, while solutions in the second category have the potential for stronger improvements, their need for PHY information precludes their use on commodity hardware, and restricts them to experimental software radios, such as the popular GNURadio USRP [USRP].

With the exception of ZigZag, all of these approaches require support from both sides of the communication (transmitter and receiver).

No Additional Information

Maranello [HSG+10] partitions the 802.11 MAC frame into blocks, with each block secured by its own checksum. If the overall checksum fails, the receiver checks the sub-checksums and replies with a specially-crafted Negative Acknowledgment (NACK) frame, which contains information about which blocks were corrupted, instead of the normal ACK. In response, the sender only retransmits the corrupted blocks. One of Maranello's downsides is that its negative ACKs are much larger than standard ACKs, because they have to carry information about which blocks were corrupted. Since the sender reserves the time for their data frame's ACK, collisions can occur if the negative ACK uses the channel for longer than reserved. Bologna [HGC10], an extension to Maranello, solves this by both reducing the size of the NACK considerably, and increasing the channel reservation to always cover the NACK, which is still slightly larger than a standard ACK.

Instead of sending partial checksums and partial retransmissions, ZipTx [LKK08] uses a Reed–Solomon code to add additional redundancy to an 802.11 MAC frame, but does not send the redundancy with the initial transmission. If a packet is not received correctly, ZipTx sends the redundancy incrementally in two additional frames that replace the classic retransmissions. This scheme benefits from the fact that, by sending redundancy data instead of retransmitting original data, even partially correct receptions of the RS code blocks can be enough to restore the data. Like our recovery schemes, ZipTx needs a specially crafted rate adaptation algorithm to show its full potential.

TVA [MM13] is an approach that, while general in its theoretical foundation, is crafted for use in 802.15.4 [IEEE11] (ZigBee, sensor net) networks. It recovers the

original content of corrupted messages without any additional redundancy information, only using the information contained in the CRC to infer original frame contents. Since the CRC was not designed for recovery, there are typically multiple repair options. Hence, TVA has to cope with a problem very similar to misattribution, in which they misrepair the packet to a content that matches the CRC, but is not the correct. TVA solves this by calculating another checksum with a different generator polynomial over the packet, sending this CRC to the sender for cross-check, and only accepts frames if the sender confirms the match of this checksum. The main challenge of TVA is that repair via CRC either is computationally expensive or requires large lookup tables. Thus, it requires domain knowledge about typical error patterns specific to the hardware, which explains the focus on sensor network: it has been shown (both by us [SCW13,SCHW14] and by others [MPC+10,HWM+14]) that 802.15.4 hardware is prone to very specific error patterns which TVA can leverage.

MRD [MBK05] also tries to reconstruct corrupted packets by checking against the CRC. However, it uses additional information by exploiting spatial diversity: its use case is a dense deployment where every packet sent by a STA is received by at least two APs, which furthermore are interconnected via Ethernet. A central controller then takes both receptions, splits them into blocks, compares the blocks, and exhaustively tries all combinations of erroneous blocks from both frames to find a combination that matches the received CRC. The division into a small number of n blocks allows reducing the number of combinations that are needed to check and thus the otherwise prohibitive overhead of $\mathcal{O}(2^n)$.

Badam et al. [BKH+11] also focus on a system with both wired and wireless connections. However, their use case is a fast, but unreliable wireless network, combined with a reliable, but slow (or otherwise restricted, e.g., for pricing reasons) wired network, where all network participants are connected via both methods. They offload all control traffic onto the wired connection and add a partial checksumming scheme very similar to Maranello. Thus, they significantly increase the throughput in their wireless network.

Physical Layer Information Available

This group can be further split into two subgroups, depending on the amount of information that is required from the PHY layer. Solutions from the first group require only soft information, while the second group works directly on the physical layer symbols. The first group is arguably more likely to see support in consumer hardware, since it only requires an enriched interface from the PHY to the MAC: instead or in addition to the hard-decoded bits, the per-bit soft information is handed over. The second group requires direct implementation in the PHY, because those solutions directly modify the symbol decoding.

SOFT [WKSK07] and PPR [JB07] belong to the first group. PPR uses soft information to estimate which bits are likely corrupted, and to selectively request those for retransmission. To keep the size of this feedback small, it dynamically partitions the received packet into variable-size blocks to minimize the messaging overhead, since signaling feedback to the receiver increases with the number of single bits that need

to be retransmitted, while requesting large blocks reduces the feedback overhead, but leads to retransmission overhead from the sender. SOFT is conceptually very similar to MRD as described above. However, SOFT makes use of the richer information available to repair corrupted bits, and therefore outperforms MRD significantly.

The rest of the works presented here belong to the second group.

Aman et al. [ASC14] propose an improvement of PPR. Instead of using per-bit soft information, they use per-symbol Error Vector Magnitudes (EVMs), a quality metric that effective describes the Euclidean distance between the received symbol's position in the constellation diagram and that of the closest reference point. By using this richer information, they can better recognize which parts of a packet are corrupted.

ZigZag [GK08] recovers packets that have been the victim of collision on both their initial transmission and their first retransmission. Instead of a second retransmission, ZigZag analyzes the two receptions, is able to recognize the start of the collisions within the receptions, and, unless they happen to be at exactly the same point in both packets, can then iteratively decode each next PHY symbol by using information from one reception to decode the other, and vice versa (hence the name "ZigZag").

MISC [OZL14] also reconstructs data from two transmissions, even under strong noise. To do so, it uses a different constellation map, that is, assignment of bit values to physical layer symbols, on its retransmission. Visually, taking Figure 2.4a on page 21 as a reference, this would mean to, for example, exchange the values `01` and `00` so that, on a retransmission, `01` and `11` have a larger distance to each other, and it is easier to distinguish between the two. After reception of the retransmission, decoding is done by calculating the sum of both Euclidean distances for each symbol value's two reference points and decoding to the bits of the best fit.

A very different way to solve the problem of retransmissions is presented in SoftCast [JK09]. In its goal it is similar to the work of this dissertation, in so far as it proposes a way to efficiently send media streams via error-prone wireless links. However, its solution is completely different: instead of keeping the restrictions of packet-based transport and working with them, SoftRate modulates the video data directly onto the OFDM signal of 802.11. This leads to a higher video quality over a large range of Signal-to-Noise Ratios (SNRs) compared to using the standard rates, and since no addressing is used any more, broadcast is inherently supported. By using the H.264 [SMW07] scalable video coding extension, such broadcast is even possible to stations with different reception qualities, by graceful degradation of the reception quality. However, such a massive change, circumventing all protocol layers, requires support to a degree that the practical applicability of such a scheme in an 802.11 network is questionable. Consequently, the authors do not discuss questions such as the coexistence of SoftCast traffic with other 802.11 traffic.

2.6.3 Header Compression

Header compression's relation to the works presented in this dissertation is less direct. It can be considered an alternate approach, with the two coming from different ends. The goal of Header compression is to reduce packet size, which increases efficiency; the fact that reduced packet size also reduces packet loss rates because smaller packets contain fewer bits that can be corrupted is a welcome side-effect. Heuristic header error recovery, on the other hand, tries to solve the problem of packet loss by recovering from errors, which leads to more timely reception and less wasteful communication; an increased efficiency is a welcome side-effect. Both have at their core the realization that packet headers contain redundancies that can be leveraged: in one case, they are removed to reduce header size, in the other, they are used to recover from bit errors in the headers.

One concept that is shared by all header compression schemes that reduce size by eliminating redundancy is that the eliminated information has to be transmitted at least once at the beginning, so that the receiver knows the contents of a so-called reference header, from which it can take the static information during decompression.

The idea of reducing header size by only transmitting information that has changed since the last packet can be traced back to Thinwire [FDC84], which proposed several different compression schemes for TCP/IP and UDP/IP. The simplest form, Thinwire I, can reduce the per-packet header overhead of TCP/IP from 40 to 17 bytes. It is notable that this method is almost independent of the used protocols, because all it needs is the size of the header that is to be compressed, and then sends updates of only those bytes (marked with length/offset descriptors) that changed. Thinwire II exploits more specific information about which header fields stay static and which change over the lifetime of a connection and can thus decrease the overhead further to a typical size of about 13 bytes (depending on which TCP fields and flags were changed).

Van Jacobson Header Compression [Jac90] is specific to TCP/IP connections, but leverages additional effects that lead to further compression: compressed headers are between 3 and 16 bytes, with a typical size of about 5 bytes. At the core of this, and many later approaches, is the idea that, just because a header field changes its value, it does not necessarily have to be transmitted completely. It is sufficient to transmit the difference between the new and the old value, which generally is a much smaller number and takes less space to transmit. Several years later, Casner and Jacobson proposed a compression scheme [CJ99] for streaming data using UDP/IP with RTP. Under favorable conditions, it can reduce the 40 bytes of header to 4 bytes. They extended the previous idea by denoting fields that increment in a regular fashion by a simple bit switch. A field that always increases by 1 or another static amount can hence be compressed even further. However, the stronger the compression, the larger the risk that due to undetected packet losses, fields are incorrectly decompressed at the receiver side. IP Header Compression [DNP99] proposes another scheme that supports IPv4, IPv6, TCP, and UDP. While it generally does not reach the same compression performance as the previous algorithms, this is a conscious tradeoff to provide better performance in scenarios with unreliable links, such as wireless net-

works, where packet loss is common. The overhead due to less efficient compression is justified by larger robustness to packet loss events.

Robust Header Compression (ROHC) [BBD+01] is a framework of concepts that can then be applied to protocols by protocol-specific profiles. Standardized profiles exist for UDP/IP with and without RTP as well as ESP/IP (all [BBD+01]), general IP without transport-layer protocol encryption [JP04], IP/UDP-Lite with and without RTP [Pel05], and TCP/IP [PSJW13]. ROHC tries to combine the advantages of high compression and robustness to potentially desynchronizing errors by defining three states. In the Initialization-and-Refresh (IR) state, uncompressed packets are transmitted, which is necessary during initial connection setup and if synchronization between sender and receiver was lost due to packet losses. In the First Order State, compression is increased, but not yet optimal. In this state, unexpected irregular header contents can be dealt with by transmitting them without compression. The Second Order State uses all compression available. In addition to the compression approaches described above for previous solutions, ROHC uses sophisticated compression schemes in this mode that, for example, dynamically adapt the number of bits necessary for each differential value that has to be transmitted. However, this sophistication in combining robustness with high compression comes at the price of a very complicated setup and implementation compared to other approaches.

6LoWPAN defines a header compression scheme [HT11] that grew out the task to make IPv6 with its large headers suitable for low-power wireless networked devices, following the Internet Of Things concept. In addition to some of the standard approaches mentioned above, 6LoWPAN's header compression makes specific use of the typical environments and network topologies in which such devices are used. For example, it compresses all IPv6 addresses, even on the first use, by using an administrator-defined address prefix and only a small local part, due to the fact that those devices are generally expected to communicate only locally with each other and a gateway.

All these header compression schemes presented so far share the problem that they are protocol-specific. One header compression solution that solves this problem is the Generic Header Compression (GHC) [Bor14] scheme defined for 6LoWPAN. While it is less efficient than a specifically tailored header compression, it offsets this downside with its generality: all headers, in theory, and even payload can be compressed. The idea is to simply use a generic lossless data compression algorithm, the LZ77 algorithm [ZL77] to compress all headers above the 6LoWPAN IPv6 header. Since LZ77 is dictionary based and, under normal circumstances, would need to send its compression dictionary with the compressed data, it would struggle to effectively compress the often very small header. To improve the compression rate, GHC uses a precomputed dictionary that is already available and does not need to be sent with the header, and that is specifically set up to aid compression of header values considered to be likely.

3

By this art you may contemplate the variation of the twenty-three letters, which may be so infinitely varied, that the words complicated and deduced thence will not be contained within the compass of the firmament.

—Robert Burton, *The Anatomy of Melancholy*

Refector: Protocol-Specific Heuristic Header Error Recovery

In this chapter, we will present one of our two approaches to heuristic header error recovery. As a first step, we will discuss the feasibility of such recovery.

We will then present in Section 3.2 the fundamental concept of heuristic header error recovery, with a focus on IPv4 and UDP, which we termed *Refector* (Latin for *mender, repairer*). In this part, we will discuss how to find a matching connection for a packet with header errors, and present the classification of header fields depending on their necessity for recovery. Finally, we will evaluate Refector's efficacy in a real-world 802.11 (WLAN) testbed.

In Section 3.3, we will investigate a combination of Refector with Iterative Source–Channel Decoding, a highly efficient approach for data encoding, and thus answer two questions: (1) How much speech quality improvement can Refector give us? (2) How does so-called soft information improve the recovery performance of Refector?

Finally, in Section 3.4, we will present an extension to Refector. This extension was motivated by the fact that RTP, the Real-Time Transport Protocol, is a standard protocol often used to support media streaming. Our original version of Refector only supported static header fields that are the same in every packet that belongs to a stream, such as port numbers. On the other hand, RTP contains dynamic fields such as sequence numbers, which change from packet to packet. Using RTP as an example, we will present the necessary extensions to recover from errors in such fields. Again, we will first discuss the conceptual solution before presenting evaluation results, this time from a simulated environment.

3.1 Introduction

The original motivation for Refector was one of efficiency. The classic layered protocol approach of the theoretical ISO/OSI and the more practical TCP/IP network stack requires, implicitly or explicitly, total correctness of content. Enforced by checksums, even single bit errors lead to checksum mismatches and, consequently, to packet drops. This is highly inefficient, especially in networks which regularly produce bit errors – which includes virtually all wireless networks.

One idea to improve the performance is to devise approaches to reduce the overall bit error rate, without unduly reducing the overall throughput performance; another is to make recovering from errors more efficient than by retransmitting complete packets. We, however, decided to investigate towards a different goal: accepting that unrecoverable errors will always occur, and finding ways to cope with those errors.

This solution is obviously not suitable for every kind of data transmission: the enforcement of bit-by-bit correctness is vital for any type of data transmission that relies on correctness. A typical example is the transfer of files, especially executable binaries, in which bit errors are unacceptable, because they will change the behavior of the contained code, in unpredictable and possibly catastrophic ways.

However, there exists a group of applications which, by their very nature, are error-tolerant. A prime example from this group is media streaming, in which input data is already subjected to a non-recoverable (lossy) reduction in size by applying models that separate unimportant information from important information with respect to human sensual perception. Especially for transmissions over media that are expected to produce errors, these media codecs are already designed with a certain amount of error concealment in mind. While some of them have been specifically adapted to the case of packet loss [WSBL03, VVT12], there also exist codecs, especially those originally designed for mobile telephone networks, that have strong concealment techniques for bit errors inside a received data stream [ETSI00, SSJ+08].

For these applications, the current state is doubly problematic. Not only do retransmissions of such packets produce overhead that could be prevented by handing the applications the (potentially corrupted) data. Retransmissions also introduce delay into the transmission of data. However, media streaming data is highly time-conscious: data needs to be available at the receiver at the time it is played out. Partially corrupted data that still arrives in time is helpful [HRNK04] and can be used for playback after potentially applying error concealment, while correct data that arrives only after one or possibly several retransmissions might be too late to be of any use at all any more. This sensitivity to delay can be somewhat reduced by introducing buffering on the receiver's side: the delay arising from those buffers give the application more time to receive potential retransmissions. However, those so-called jitter buffers[12] are very limited in their applicability in two-way

[12] These buffers are named for the effect they aim to mitigate, the so-called jitter, which denotes variance in the reception delay perceived by the receiver. Retransmissions are one cause for this variance in delay.

communication scenarios, such as video conferencing or VoIP. For example, for VoIP, more than 150 ms of end-to-end delay are considered detrimental to perceived quality [ETSI06, ITU03].

UDP-Lite as a "Spiritual" Predecessor

This suggests that support for error tolerance in transmissions is beneficial and can improve both performance of some types of applications and the overall performance of the network by reducing retransmissions. However, such a scheme needs support by adapting network protocols to allow such error tolerance. One of the most well-known examples of such an adaptation is UDP-Lite [LDP99, LDP+04]. By redesigning the UDP protocol to allow dynamic coverage of only a part or none of the payload by its checksum, error tolerance is facilitated for payloads on the transport layer, only keeping its header secured. This does take an important step towards supporting error tolerance; however, it suffers from two main problems:

1. By redefining the header contents of UDP, UDP-Lite is incompatible to UDP. It hence requires support by both the sender and the receiver, and by both the application (that chooses to use this new transport-layer protocol) and the operating system (that supplies an implementation of UDP-Lite) on either side.

2. UDP-Lite is a stand-alone solution that depends on support by lower-layer protocols. While IP does not pose a problem to payload error tolerance because its checksum only covers its own header, MAC-layer protocols typically secure the complete packet, and hence drop any packets with errors anywhere in it.

Furthermore, while UDP-Lite is a well-known example that recognizes the benefit of error tolerance for certain applications, we argue that it only "goes half the way": errors in headers still lead to packet drops. Especially in audio streaming applications, individual packets tend to carry only small payloads: media compression algorithms reduce the amount of payload to be carried significantly, and to keep the aforementioned timings, packets have to be sent out regularly, for example, every 20 ms. In such situations, headers form a sizable part of each packet, more than 50% in extreme cases. Introducing header error tolerance therefore promises to further improve on solutions such as UDP-Lite.

Nevertheless, there are good reasons why this approach has been shied away from and not taken yet or investigated by anybody in a comprehensive and detailed manner. This approach comes with considerable risks. The main problem is that media codecs that are expected to cope with erroneous information were designed to do so. Protocol headers, on the other hand, were not designed with such error tolerance in mind, but are rather expected to contain completely correct information when received. To give an example, if an error-tolerant and an error-sensitive application, such as a file transfer and a VoIP call, are receiving data concurrently, each of them is identified by, among other information, the destination port that serves as the main demultiplexing information of different concurrently active connections on a single

system. It is conceivable that an error in the header portion of a packet corrupts the port number in a way that a packet that was destined for the VoIP connection now appears to belong to the file transfer, which may result in a corrupted file reception, an almost catastrophically detrimental outcome that we term a *misassignment* or *misattribution* Any solution that targets header error tolerance therefore needs to prevent such misattribution to the largest extent feasible. Any benefit of increased error tolerance needs to be weighed against the risk of misattributions.

Reducing misattributions is generally a goal of the algorithmic design of our error tolerance scheme, and will be discussed when presenting the respective solutions. However, we add one general safeguard to our error-tolerance schemes that vastly reduces the problem of misattributions. We require all applications that are error-tolerant to signal this capability to the protocol stack. This way, we can completely prevent any misattributions toward error-sensitive applications: a file transfer will never be corrupted by a misattributed VoIP packet. The only possible misattribution is the one toward error-tolerant applications: these can "catch" packets from other error-tolerant applications, as well as packets from error-sensitive applications that were misattributed to them. Note that for an error-tolerant application, such a misattribution should never be immediately fatal: to such an application, a misattributed packet will simply appear to contain extremely corrupted data. The concept of error tolerance requires an application to even cope with such a situation. Obviously, such a misattribution is still highly undesirable: it will almost certainly reduce decoding quality because unrelated data has been introduced, and (at least in the case of a misattribution between two error-tolerant applications) the data will be missing from another connection.

While requiring this opt-in means that error-tolerant applications need to be changed to gain the benefit of Refector, we consider this a worthwhile tradeoff. First, the required changes are limited, and application developers who want to make use of error tolerance can be expected to have self-interest in adapting their applications for such support. Second, the opposite approach (providing error tolerance by defaulting and opting out) is infeasible because this would require a change of a large amount of applications to keep the previous behavior, and to otherwise suffer from potentially catastrophic communication failures without any benefit to them. Third, a solution such as UDP-Lite likewise requires changes to the application by switching from one transport-layer protocol to another. Furthermore, this change has to be implemented on both the sender and the receiver side. As we will see, Refector does not require any changes to the sender (although it requires a certain amount of support from the local network gateway, typically a wireless AP, instead).

Header Error Recovery

After this short excursus, let us conclude this introduction with a discussion on what header errors mean to a communication system, and a first motivation why heuristically recovering from those might be feasible.

Under header errors, we need a reliable system to recognize and recover from these errors. Our approach is motivated and modeled by a real-world analogy. A human

mail courier, especially one who has served his locality for a long time, will know the people living in the streets they serve. Simple typographical errors or smudged address parts will often not lead to problems in delivery, because the courier can use their experience (the *domain knowledge*) to still deliver the letter to the correct recipient. Such a mail courier can also recognize, even with partially unreadable addresses, whether a letter was erroneously put into their satchel, and can refuse to deliver that letter.[13]

An end host on a network works quite similarly. At any given point in time, an end host knows which connections are open (which people live in his locality), because applications open new connections via the OS. It furthermore knows how a packet for each of these connections is supposed to look like (what the address of every person is), because the OS needs this information to recognize incoming packets and assign them to the correct application. A port number, in that respect, is analogous to a street address. Partially corrupted header information should therefore, in theory, still result in a correct delivery.

Of course, this reasoning assumes that there is sufficient redundancy in the header information. Human language is notoriously redundant;[14] machine-readable information is not necessarily so. However, even a cursory look at the setup of typical network protocols hints at the amount of redundancy in network protocol headers. For example, transport-level network protocols such as TCP and UDP use port numbers to distinguish between different connections on the same machine, and use 16-bit port numbers for that. There are hence 65 536 possible port numbers, and a connection is moreover identified by the 4-tuple ⟨local IP address, local port, remote IP address, remote port⟩,[15] further increasing the possible combinations; however, on a typical end host, there are rarely more than 100 connections open at any given point in time.[16] The fact that header compression schemes such as Thinwire [FDC84] and RObust Header Compression [BBD+01] exist further reinforces the point of redundancy in network protocol headers. As a rough estimate of the amount of redundancy, the field-based Thinwire could already reduce headers by about 50% in size, with ROHC being able to reach more than 90% compression in perfect conditions.

In the next section, we will look at the setup of IPv4 and UDP headers and assess the redundancy from a slightly different point of view. To us, redundancy in an information-theoretic understanding, while certainly important, is not the only factor in assessing the feasibility of error recovery. The IPv4 protocol header is a good example of a header that contains a large amount of information that may or may

[13] And hopefully bring it back to the post office, where it will be handed to the right courier – but this point is not part of our analogy any more.

[14] This can easily be seen by compressing text documents with a file compression algorithms such as Lempel-Ziv [ZL77] or Lempel-Ziv-Welch [Wel84].

[15] This 4-tuple is often extended to a 5-tuple by adding the IANA-assigned number [IANA15] that identifies the transport-layer protocol.

[16] While we do not have a large body of data to support this claim, the reader is encouraged to look up the number of current Internet connections on their machine via tools such as `netstat`. In our experiments, some of which are presented in one of our papers [SAAW11], as well as (independently) by Martin Henze in his diploma thesis [Hen11], we rarely witnessed more than 60 and never more than 100 connections under normal usage conditions on end host machines.

not be compressible well, but that, more importantly, is of no use to end hosts. A prime example here is the TTL field, which contains the number of routing hops a packet is allowed to take before being discarded (to prevent infinite routing loops). Once a packet has reached its final destination, the information is of no further practical use to the receiving application; an error inside this field, however, will still lead to checksum mismatches and packet drops. We therefore can identify a class of *don't-care* header fields, which, if they contain errors, should not lead to a drop of the packet. Coming back to our mail courier analogy, a letter with a misspelled or unreadable country information will not lead to delivery problems on the local level, either.

Now that we have established that header errors should be, in theory, recoverable, we will present our work that shows the practical application of these ideas. However, before we start this presentation, we will conclude this introduction with the design goals that guided our conceptual decisions for Refector:

1. Refector should be effective at recovering from header errors and at repairing headers where necessary. Conversely, it should prevent misattribution of packets to wrong applications.

2. Refector should be easily and incrementally deployable. This means Refector is a receiver-side solution. It should not require any support from the sender, nor from the intermediate gateways the packet has to pass from sender to receiver.[17] Furthermore, Refector should only incur software changes, since these can be deployed relatively easily; changes to firm- or hardware must not be part of the changes.

3. Refector should be as independent as possible from the underlying MAC and PHY layers. Since protocols on these layers strongly depend on the used network access technology, this facilitates the use of Refector in a large number of networks. As long as the MAC protocol provides a way to pass on corrupted packets to the upper layers without dropping them, Refector should be able to work on top of it. Furthermore, if the MAC protocol retransmits packets based on some rules, a way to influence those rules is necessary for good performance.

4. Refector's protocol changes should keep compatibility. This is a direct consequence of rule 2. While the protocol implementations are changed to facility error recovery, the communication behavior with other parties (especially the sender) must stay the same as before, to stay compatible and allow incremental deployment.

5. Error-tolerant traffic should be an optional addition, and error-tolerant and error-sensitive traffic must be able to coexist. This means that error-tolerant

[17] We will see in this chapter and in Chapter 5 that, at least for 802.11 connection, some support by the AP is necessary to reach good performance. However, we consider this acceptable, since our target environment, as specified in Section 1.2, is wireless networks for end users and small-to-medium non-private deployments, in which the AP is under the control of the users, as opposed to senders in other networks.

applications need to opt in, that is, signal their error tolerance to Refector. Only then will they receive corrupted packets. This is done to make sure that all other traffic will not receive corrupted packets, so that, for example, a file transfer will not generate corrupt files, and a DNS lookup will not produce the wrong lookup data because a corrupted packet was received.

3.2 Refector for Stateless Protocols

As a first step, we will present the practical implementation of a header recovery scheme for stateless protocols. For this work, we focused on IPv4 and UDP. The reasoning was that both these protocols are relatively simple due to their statelessness; that they contain a large amount of fields that can be leveraged for header error recovery; and, most importantly, that their position in the network stack at the "narrow waist of the Internet" [Ros08] means that almost every connection will use one or both of these protocols and therefore will be able to benefit from Refector.[18] This is opposed to the MAC and application layer, where no such concentration on a few protocols exists, and any solution would only be applicable to a small subset of scenarios.

This section is structured as follows. First, we will discuss the categorization of IPv4 and UDP header fields. In Section 3.2.2, we will discuss how to recover fields that are important to identify the connection a packet belongs to, if errors occurred in those fields, and possibilities to increase the robustness of those fields against bit errors. We will then give a short overview of the implementation of a prototype into the Linux kernel, before presenting evaluation results in Section 3.2.4 and summarizing in Section 3.2.5.

3.2.1 Header Fields Categorization

As a first step, we will now analyze the protocol headers of IPv4 and UDP. We already noted previously that there exist header fields that are of no importance to the end host, and that we named those *don't-care* fields. Conversely, we classify fields that we care about as *vital*.

The IP header is shown in Figure 3.1. Disregarding the rarely-used header options, it has a size of 20 bytes and is divided into the following fields.

Version and IHL: Both these fields contain static values that never change and are the same for all open connections. The version field contains a 4 to identify the header as IP, version 4. The Internet Header Length (IHL) contains the length of the header, expressed in the number of 32-bit words. At a standard header length of 20 bytes, this is statically 5. Both of these fields do not provide any helpful insight and are therefore classified as don't care. The version field is redundant

[18] TCP is arguably the more widely-used of the two dominant transport protocols. However, its statefulness means that it does not fit the requirements of this section. Furthermore, the focus of TCP on reliable data stream transmission is almost the opposite of our error tolerance approach.

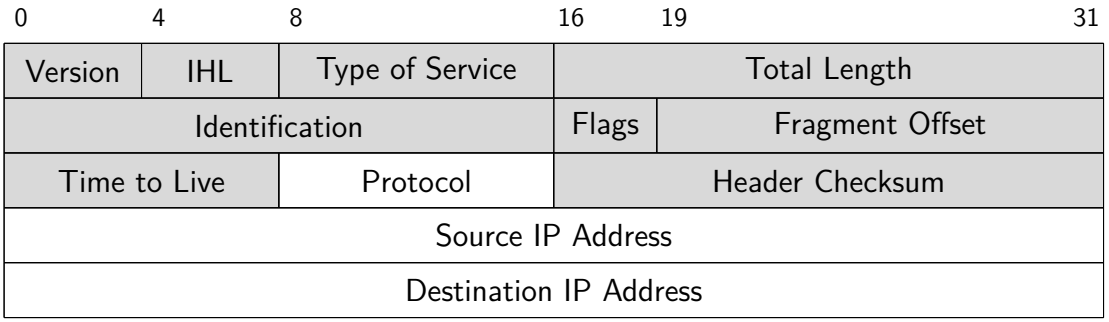


Figure 3.1 Classification of header fields of the IPv4 header. Vital fields in white, don't-care fields in gray.

because, by the time the header is parsed, the end host has already decided to parse it with an IPv4 parser (typically because the MAC header contains a *next-protocol* field that denotes IPv4 as network protocol). The IHL field is redundant under the assumption that header options are not used. This is reasonable because, in real networks, IPv4 header options are practically non-existent. A large number of them has been officially deprecated [PG12]; others are actively filtered by network gateways because of their harmful potential (such as the originally defined routing headers).

Type of Service: This field has seen different uses over the years with the goal of implementing QoS schemes for IP, such as IntServ [BCS94], DiffServ [BBC+98], or ECN [RFB01]. However, by the time a packet reaches an end host, the QoS information is not needed any more. We hence classify this field as don't-care.

Total Length: This two-byte field contains the length of the packet, counting from the first byte of the IP header. There is little practical use in that information, because the length of the packet is known from the data received, and we classify it as don't-care. In theory, this field could identify whether padding had been added to the payload on the IP layer. However, there is little reason to add such padding, especially not on the IP layer since IP does not require a minimum frame length. If another protocol (such as Ethernet) requires minimum packet/frame sizes, it is that protocol's job to add padding, and such padding will be transparent to IP and not be reflected in this header field, because it will already have been removed by the time the receiver processes the IP header.

Identification, Flags, and Fragment Offset: These three fields share four bytes and implement fragmentation for IP. The identification field contains the same number for fragments of the same IP packet, and different numbers for different IP packets. The fragment offset denotes the relative position of a fragment in the original packet, in 32-bit words. The flags contain information about whether a packet must not be fragmented, or a received packet is a fragment of a larger IP packet. Overall, just like header options, IP-level fragmentation is a rarely-used feature. Fragmentation due to MAC/PHY constraints is either done at the MAC layer, or at the transport or even application layer due to Maximum Transmission Unit (MTU) settings or automatic MTU discovery [MD90,MH07]. Again, these fields are classified as don't-care.

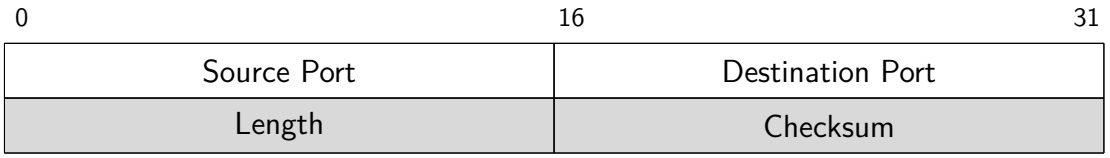


Figure 3.2 Classification of header fields of the UDP header. Vital fields in white, don't-care fields in gray.

TTL: TTL has already been mentioned above as an example of a don't-care field. It is used to prevent infinite routing loops and contains the number of routing hops a packet is allowed to take before being discarded, and is decremented on each intermediate hop. By the time the packet is received by the end host, the TTL does not serve any practical purpose any more.

Protocol: This field denotes which transport-layer protocol is used, that is, which protocol handler should be called after IP processing has finished. This information is vital because there is no other easy way to identify the next protocol, and it is necessary to identify the correct protocol handler for packet processing.

Header Checksum: This field contains the IPv4 header checksum as introduced in Section 2.3. As this field denotes errors in the header (of which we already categorized large parts as don't-care), while our goal is to ignore the fact that header errors occurred and to recover from them, we classify this field as don't-care.

Source and Destination IP Address: The last 8 bytes of a standard IPv4 header contain the IP addresses of the sender and receiver, respectively. We consider these as vital. Specifically the Source IP Address helps us identify a packet as belonging to a certain connection: at any given time, a typical end host will have open connections to a number of different hosts with different IPs. The large size of the field helps identify the correct connection. On the other hand, the destination address is typically of less interest: most end hosts will only use one IP address for communication with the outside world, and hence all connections will have the same field content. However, this means that there is no harm in adding this field: all connections will either match perfectly, or match in the same non-perfect way. In the rare case that an end host uses multiple IP addresses, however, this can further increase recovery performance.[19]

In the same way as the IP header, we will now analyze the UDP header. The header contents are shown in Figure 3.2. As the only job of the UDP header is to demultiplex connections for different applications, without any additional functionality such as ordering or reliability, the header is quite simple and contains only four fields of two bytes each.

Source and Destination Port: Together with the source and destination IP address, this forms the 4-tuple that identifies a connection (often extended to a 5-tuple that

[19]Incidentally, this opens up new possibilities for IPv6 header recovery: because even end-users are often assigned a whole subnetwork for private use by the communications provider, using different addresses for different connections can be used to increase recovery performance; see Section 3.5.

includes the used transport-layer protocol). As such, these fields are vital for recovery.

Length: Similarly to the IP length field, this field does not give us any information that we cannot deduce from other, more reliable sources. We therefore classify it as don't-care.

Checksum: Although this field contains a checksum over both the UDP pseudo header (cf. Figure 2.9, page 34) and the payload of a packet and hence is not the exact counterpart of the IP header checksum, it similarly and for the same reasons is of no interest to us and classified as don't-care.

3.2.2 Recovery of Vital Fields

In the last section, we have seen that a large amount of header fields in the IPv4 and UDP header are of no interest to the end host. More than 50% of the header bits belong to fields that can be safely ignored on end hosts in today's communication systems. This means that bit errors in these fields will lead to a packet drop under normal circumstances (because the checksum does not match any more), but does not if using Refector.

However, a sizable amount of bits is still necessary for identification and demultiplexing of packets to the correct communication end-point. Errors in these header fields cannot simply be ignored. Because this information, the 5-tuple ⟨source IP, destination IP, source port, destination port, protocol⟩ is used by the end-host to identify the correct application, an error inside these fields will mean that identification fails. In most cases, this will mean that the host cannot find an ongoing connection with the (erroneous) 5-tuple, and consequently drop the packet. Even worse, if this information breaks in such a way that it happens to match the 5-tuple of a different connection, a packet might be assigned to an wrong application. This *misattribution* is extremely undesirable, because it can lead to catastrophic failures of applications. For example, assigning a VoIP packet to an FTP connection means that the transmitted file will be corrupted.

We will first focus on the first problem: recovering from errors to prevent packet drops if the 5-tuple contains erroneous information, and will discuss the problem of misattribution and how it can be alleviated afterwards.

3.2.2.1 Heuristic Recovery of Header Fields

At the beginning of the chapter, we presented our mail courier analogy (page 46) and how a real-world courier can cope with address errors. We also pointed out how this analogy fits the situation of an end host, and that protocol headers contain redundancy that can be leveraged for heuristic error recovery. It is now time to consider how exactly to do this leveraging.

The approach is quite straightforward. Bit errors that occur as the effect of transmission problems manifest themselves as bit flips in a data stream. With every bit

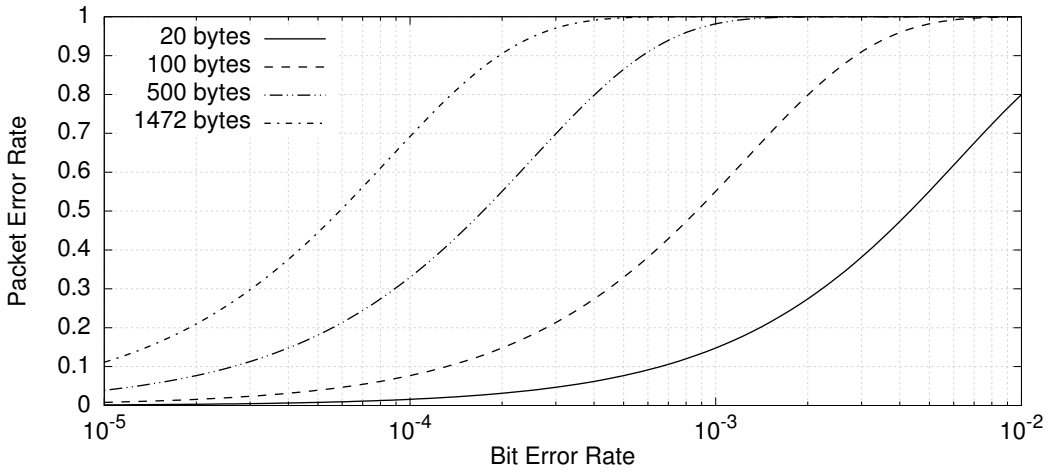


Figure 3.3 Even low BERs already lead to high PERs. Even a 100-byte packet has a 7.6% packet error rate at a BER of 10^{-4}, and more than 50% at 10^{-3}.

flip, the received data loses some similarity to the originally transmitted data, and the distance between the two is the Hamming distance.

Let us assume that $|\bar{b}| \ll |b|$, that is, the number of flipped bits is much lower than the number of unflipped bits, which means that the Bit Error Rate (BER) is low. This is a reasonable assumption, considering that, under normal communication schemes with packet drops on checksum mismatches, even very low BERs already produce high Packet Error Rates (PERs) and render communication impossible (cf. Figure 3.3), because even a single bit error leads to an erroneous packet:

$$PER = 1 - \Big((1 - BER)^n \Big)$$

That is, the chance that a packet of length n (in bits) is erroneous increases not only with BER, but also strongly with packet length. Furthermore, even relatively robust media coding schemes start degrading noticeably above BERs of 10^{-4} [NOCW07].

At these relatively low BERs, the received erroneous packet will, with a high probability, contain values for the vital fields that still closely resemble the expected fields. We can therefore exchange the standard "perfect match" algorithm, in which a packet is only assigned to a connection and the application that opened that connection if the 5-tuple exactly matches, with a heuristic matching that, if no perfect match is found, finds the next closest match, with Hamming distance as a metric. An example is given in Figure 3.4.

One different approach that we considered is not to use Hamming distance as measurement, but instead a metric that is adapted to the idiosyncrasies of the modulation and coding scheme of the used MAC and PHY technology. Hamming distances can be expected to work best if bit errors are independently distributed over the length of a packet (and hence each header field only suffers from few errors), a property that real-world MAC/PHY systems cannot always satisfy. However, we decided to keep Hamming distances as metric for several reasons. First, using specialized distance metrics that take effects of underlying technologies into account

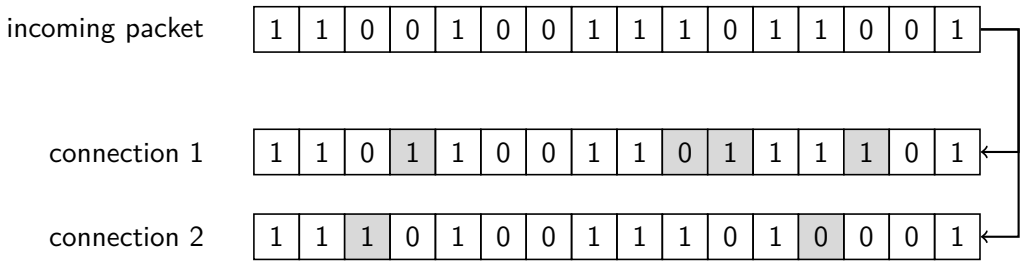


Figure 3.4 An example of port matching via Hamming distance. An incoming packet's port field of 16 bits is matched against the expected values for two flows. Flow 2 is chosen in this example because of the lower Hamming distance (2 vs. 4).

means that our approach loses some of its generality, and thus goes against design goal 3 (independence from MAC and PHY). As it stands, it considers the network and transport layer, and can be used with any set of upper and lower layer protocols. Specialized distance metrics would dilute that generality. Second, while many widely-used wireless communication technologies, including 802.11, do exhibit bursty tendencies of unequal bit error distribution, such distributions are not easily expressible and often occur on longer scales than the 8–32 bit field sizes we consider [WKHW02]. Furthermore, the bit error distribution strongly depends on the used hardware [HJL+09, HJL+12], introducing yet another variable to consider in a tailored approach, requiring complicated and cumbersome measurements and testing for any devices to be used, which significantly complicates deployment and hence contradicts design goal 2 (easy and incremental deployment). Third, as we will see over the course of this and the next chapter, the Hamming distance method already works so well that it is questionable whether potential performance improvements justify this additional effort.

3.2.2.2 Port Allocation

Recovering from errors works better the larger the Hamming distance between the header fields of different connection: if a connection has at least a distance d to all other connections, up to $\left\lceil \frac{d}{2} \right\rceil$ bit flips can be corrected, that is, up to that many bit flips can be tolerated by our scheme before misattributions occur. It is therefore desirable to increase the Hamming distance between open connections. However, the possibilities for this are limited. An end host typically has one fixed IP address, so this field is the same for all connections. The remote host's IP address cannot be influenced; however, the IP addresses of different remote hosts will typically show large differences (which is to our benefit) unless most communication is within local networks. The remote port, as well as the protocol, are similarly unchangeable. However, the end-host has some choice in assigning a local port to a new connection. For non-server connections, the Internet Assigned Numbers Authority (IANA) defined a "private ports" range between 49152 and 65535 [CET+11]. Many operating systems extend this range to start from 32768. Within this range, a host can choose a port to assign to a new connection. This port choice can be done in a way that maximizes the Hamming distance.

generator polynomial	data bits	code bits	minimum distance	corrects errors	usable 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 3.1 Some examples for generator polynomials for BCH codes. We used their property to create codewords of a given length and guaranteed minimal Hamming distance to choose port numbers that are distant enough from each other to be resilient to a certain number of bit errors.

One possibility to achieve this would be to calculate a port with maximum Hamming distance to all other ports on-the-fly, whenever a new port is opened. However, this not only means additional overhead; it can also mean that later, additional ports have to cope with comparatively low Hamming distances: As an example, consider one open connection with destination port number p_1 = `000000` (in a theoretical system with 6-bit ports). Once a second connection opens, the maximum Hamming distance is achieved by flipping all bits, that is p_2 = `111111`. A third connection can now only achieve a Hamming distance of 3, for example, `111000`: for each bit it distances itself from p_1, it approaches p_2. If we had catered for a third connection from the beginning, a higher minimum Hamming distance would have been possible by assigning `000000`, `001111`, and `111100`, guaranteeing a minimum Hamming distance of 4. A greedy approach hence does not deliver satisfactory results.

Instead, we decided to use a system that precalculates admissible ports. We employed BCH codes [BRC60] to create a set of ports that all have a guaranteed minimum Hamming distance from each other. A BCH code is a polynomial error-correcting code over a Galois field $GF(q^m)$ defined by a generator polynomial. Calculation of codewords has close similarities to the calculation of CRC codes (cf. Section 2.3 for details): the data to be encoded, in binary notation, is padded with zeroes according to the degree of the generator polynomial, and then divided by the binary representation of this generator polynomial. The remainder is added to the data and contains the redundancy. On the receiver's side, a number of bit flips, specific to the used generator polynomial and the redundancy added by it, can be recognized and corrected. The larger the generator polynomial, the more redundancy is added, but the more overhead is introduced.

In our case, the choice of generator polynomial for the BCH code balances the number of usable ports against the introduced robustness. As the port field has 16 bits, but the first bit has to be 1 to start at port number 32768, we have 15 bits available, and as such, Refector uses a BCH code over $GF(2^{15})$. The port numbers are then created by taking the BCH code words and prepending a binary 1. Several examples of generator polynomials are given in Table 3.1. For Refector, we decided to use the generator polynomial $x^8 + x^7 + x^6 + x^4 + 1$, as it allows us to use 128 ports concurrently. This should suffice for typical end hosts, as we already discussed

at the beginning of this chapter.[20] Using this generator polynomial, all 128 possible binary values, from `0000000` to `1111111`, are BCH-encoded with the above generator polynomial, creating the used ports. For example, for `1011011`, the steps to create the port number are:

1. Pad `1011011` with 8 zeroes, according to the degree of the generator polynomial, producing `101101100000000`.
2. Divide `101101100000000` by the binary representation of the generator polynomial $x^8 + x^7 + x^6 + x^4 + 1$, `111010001`, resulting in the remainder `1101101`.
3. Add the remainder in place of the padding, resulting in `101101101101101`. Finally, prepend the 1, resulting in `1101101101101101`, or port 56173.

All 128 ports created this way are guaranteed to have at least a Hamming distance of 5 to every other port such created, allowing the correction of at least to bit errors. Again, note that this set of ports is precalculated; there is hence no computational overhead introduced by this scheme at runtime when the OS has to assign a new port, since it only has to pick a port from the static, precalculated list.

3.2.2.3 Analytical Approximation of Port Selection Performance

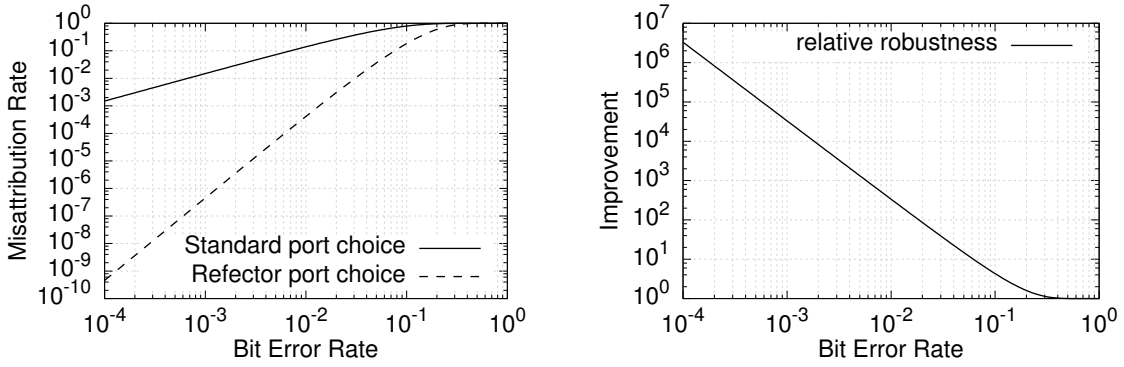
This guaranteed minimum Hamming distance of 5 greatly increases the robustness of the destination port field against random bit errors. To quantify this gain, we will give an analytical approximation. Note that in this section, we assume a binary symmetric channel, that is, the probability of each bit being flipped solely depends on a static BER, not on its value or any other previous or successive error events: the errors are independent from each other. We will only give the final terms for each worst-case approximation in this section. Appendix A gives a more detailed derivation.

Under an OS's standard port allocation scheme, which does not aim to maximize Hamming distance, the distance between two ports can be as little as one. The differing bit then becomes the deciding factor for the choice of attributing a packet to one or the other port (all other information being equal). This means that the misattribution rate is equal to the BER. With increasing numbers of connections, the worst case becomes a situation in which *all* 1-bit neighbors of a port are used, and therefore any bit error in the port field leads to a misattribution. The misattribution rate can then be approximated as:

$$\Big((1 - BER)^{15} \Big) \tag{3.1}$$

Refector's port selection algorithm, on the other hand, guarantees a minimum Hamming distance of 5 by employing the aforementioned BCH code. The worst case can

[20] In our implementation, the port allocation scheme falls back to the default scheme if the 128 ports are ever exhausted. If the OS honored the IANA suggestion to start the private port range at 49152 instead of 32768, our BCH code would degrade to 64 available ports, after which, again, the standard port scheme would be used.



(a) absolute difference (b) relative difference

Figure 3.5 Comparison of the standard port choice and Refector's port choice in worst-case scenarios via analytical approximation. Note the logarithmic scales on both axes. Refector's port choice is significantly more robust. Furthermore, its robustness translates very well at low to moderate BERs: The right graph shows the quotient of the standard port allocation's misattribution rate, divided by Refector's misattribution rate. At low to moderate BERs, Refector is several orders of a magnitude more robust than the standard port allocation.

therefore be approximated by a situation in which all 5-bit neighbors of a port are used, and *any* combination of 3 bit errors produces a misattribution.[21] This can be expressed as:

$$1 - \sum_{k=3}^{15} \binom{15}{k} BER^k \left(1 - BER\right)^{15-k} \tag{3.2}$$

Figure 3.5 shows the difference in robustness for the two port allocation schemes by plotting BER against misattribution rate. Two effects are apparent here. First, Figure 3.5a shows the misattribution rate of the two port allocation schemes in the worst-case scenario as a function of the BER. At unrealistically high BERs, the difference becomes negligible as both schemes fail. However, at less pathological BERs, it is apparent that Refector outperforms the standard port choice significantly. For example, at 10^{-4}, the standard port choice produces a misattribution rate of $\approx 1.5 \cdot 10^{-4}$, while Refector's misattribution rate is only $\approx 4.5 \cdot 10^{-10}$, while at a BER of 10^{-2}, the misattribution rates are $\approx 14\%$ and $\approx 0.04\%$, respectively. Moreover, the lower the BER, the higher the relative improvement in robustness, as can be seen in Figure 3.5b. At a BER of 10^{-2}, Refector is already more than 100 times more robust, while at a BER of 10^{-4}, the difference is more than 6 orders of magnitude.

[21] This is in fact strictly worse than the result of applying the BCH code; while 5 is the guaranteed minimum Hamming distance, no code word created by the BCH code (and therefore no port in Refector's port selection scheme) will have that minimum distance of 5 to *every* other port; there are always some ports with a higher Hamming distance.

3.2.3 Implementation

We implemented Refector in the network stack of the Linux kernel, version 2.6.32. This required patching the kernel in several locations, which can conceptually be divided into four fields: the error-tolerance implementations for IPv4 and UDP themselves, changes to the 802.11 MAC layer's handling of corrupted frames, extension of the socket interface, and support for ACK-less traffic.

Repair Techniques

As a first step, we need to keep track of IP addresses and ports that are open at any given point in time. While this information is already contained in the kernel, it is kept in highly optimized hash table data structures designed for quick lookup of, for example, a socket indexed by a port number, not for traversal of all port numbers. Hence, we created an additional linked list that contains information about currently open ports. To keep this list updated, we hooked into the functions that create and delete IP devices and sockets, and let them update this list.

Furthermore, we changed the main packet processing routines for IPv4 and UDP to run our heuristic header error repair at the beginning. Hence, we patched `ip_rcv` and `__udp4_lib_rcv` accordingly. The actual repair was done in two functions, `repair_ip_hdr` and `repair_udp_hdr`, which were called if a packet had a checksum mismatch. These functions checked every header field and repaired it as well as possible, according to the design laid out in the previous section. Static fields were repaired to static values; IP addresses port numbers were repaired by finding the nearest Hamming distance match.

MAC Support

While we focus on IPv4 and UDP in this part of our work, we still need a modicum of support from the underlying MAC layer. This is due to the fact that MAC protocols typically (and 802.11 is no exception) employ checksums covering the complete frame to recognize reception errors. Therefore, we need to patch the MAC protocol implementation to not drop those frames. In the case of MAC, due to its close relation to the hardware being used, there is an additional problem. Checksum checking is often already done in hardware on the network adapter, and frames with failed checksums are dropped before they ever reach the kernel. This behavior is governed by the network adapter's chipset. For our purposes, if it is present, it must be disengageable. Thankfully, most consumer cards, while they do drop frames in hardware as standard behavior, have flags with which the driver can instruct the card to deliver all frames.[22] Two (at the time of the experiment) popular chipsets that support this behavior are the Atheros ath5k [ath5k] and the Broadcom b43 [b43]. While the b43 driver worked without any changes, the ath5k driver's code dropped

[22]Note that this is different from such settings as monitor mode, in which the card is put into a promiscuous sniffing mode. In our approach, the drops due to checksum checks are switched off, but the card is kept in infrastructure mode.

such frames regardless and needed to be changed in that respect. In addition, the aptly named function `should_drop_frame` function in the Linux kernels mac80211 hardware abstraction layer needed to be patched to not drop frames with failed checksums.

Note that these changes do not introduce any error recovery; if fields in the MAC header were corrupted, frames were typically dropped during processing. These changes merely allow receiving corrupted frames at all.

Finally, we used the fact that the MAC-layer checksum covers the complete frame to keep track of corrupted frames. Hence, we added a `l2_check_failed` flag to the `sk_buff` structure that contains all information about a network packet in the Linux kernel. This way of keeping track whether the packet had any errors was used for additional signaling in the socket interface between the kernel and the application.

Kernel–User Interface

We extended the socket interface between user and kernel space with additional signaling capabilities. Signaling as such is already present in the socket interface. Special socket settings can be changed by using the `setsockopt` system call, for example, to change the sizes of buffers associated with the socket. We added a socket option `SO_BROKENOK` which signals to the kernel that this connection is error-tolerant. With this opt-in approach, we ensure that we never will assign broken packets to applications that cannot deal with them. All packets, regardless of whether they belong to an error-tolerant or an error-sensitive connection, will pass through the heuristic repair stage if they are broken. This is indeed necessary since we do not know which connection a packet belongs to before it has passed the protocol handling routines, and our heuristic repair has to take place before those routines so they do not fail on erroneous header contents. However, if a packet, after repairing, is identified as belonging to an error-sensitive connection, it is dropped instead of assigned to that connection's socket.

We also implemented signaling from the kernel to an error-tolerant user-space application. If a packet is corrupted, it might be beneficial for the application to be informed of that situation; for example, a codec might put less emphasis on its contents than on those guaranteed to be correct. The `recvmsg` system call to receive data from a socket already has provisions to signal ancillary data belonging to that packet. We added a `MSG_HASERRORS` flag that can be read by the application to check whether the received data potentially contains errors. To decide whether a packet contains errors, we check the UDP checksum: if it does not match, then there are errors somewhere in either the (UDP pseudo) header or the payload of the packet. Note that this information errs on the side of caution: since the UDP checksum covers the payload, the UDP header, and parts of the IP header in the form of the UDP pseudo header, there might be no errors in the payload of the packet, and the data received from the socket may be error-free. This, however, is the most exact information we can provide, since there is no checksum provided that only secures the payload. Conversely, if `MSG_HASERRORS` is not set, the application can be sure

that there are no errors present (save for very rare situations in which an error was not recognized by the UDP checksum).

No-ACK for Error-Tolerant Streams

We suggested before (cf. Section 2.4) that ACKs for error-tolerant streams are problematic due to the question of what an ACK is supposed to signal in such a case. Since the 802.11e QoS extensions allow the sending of frames without ACK, we decided to use that scheme. However, while we introduced it in Section 2.5, we deferred a detailed explanation of how to identify which packets to send without ACKs, since the 802.11 MAC layer has no concept of different upper-layer streams, and the setting is done per-frame.

In our implementation, we use the IPv4 Type of Service (ToS) field, which already was designed to hold QoS information. In fact, the Linux kernel already implements a translation between ToS and 802.11e's Access Categories (ACs) in `ieee80211_set_qos_hdr` function. The 256 possible ToS values[23] are mapped to the 8 Traffic Identifiers (TIDs) and then further to the 4 ACs by only considering the 3 most-significant bits of the ToS. Hence, for example, all ToS values between 0 and 31 are assigned TID 0 and to AC_BE, while all values between 192 and 223 are assigned 6 and AC_VO, respectively.

We added a logic to the above function, which also governs whether 802.11 frames are sent with or without acknowledgments, that allowed to set certain ACs as No-Ack classes. By setting a flag that we made available via the `proc` filesystem, we were able to, for example, send all frames that had a TID of 5 or higher without ACKs. In our experiments, we then had the sender send its packets with a ToS that would translate into a TID that we set as No-ACK.

3.2.4 Evaluation over 802.11

The central questions our evaluation has to answer directly stem from the previous explanations about the benefits and drawbacks of Refector, namely:

1. How does the Packet Delivery Rate (PDR) improve compared to existing approaches?

2. How often does misattribution occur?

In addition, we will also provide insight into the question of whether error tolerance is compatible with encryption.

[23] Note that the number 256 refers to the internal representation in the Linux kernel. This generic, 8-bit ToS value is then mapped onto the available precision inside protocol headers. For example, if Differentiated Services [NBBB98] are used, the field is is mapped onto the 6 available bits in the IPv4 header, the remaining two bits of the original ToS field being reserved of other uses, such as Explicit Congestion Notification (ECN).

To answer Question 1, we chose to compare ourselves to UDP-Lite. The reasoning for this is that UDP-Lite is a standardized protocol that implements tolerance to (application-layer) payload errors.[24] It is readily available as it is implemented in the Linux kernel, and forms a reasonable baseline as state-of-the-art. Refector's advantage over UDP-Lite is tolerance to header errors, so comparing ourselves to UDP-Lite answers two questions simultaneously whose meanings are equivalent: (1) What is the performance gain of Refector over a state-of-the-art solution? and (2) What is the performance advantage of header-and-payload error tolerance over payload-only tolerance, that is, what is the advantage of header error tolerance?

3.2.4.1 Experimental Setup

To answer both questions put forward, we set up a small 802.11 network of five machines, one emulating an AP via `hostapd` [hostapd], the other four running as STAs with our Refector changes, which we implemented into Linux kernel 2.6.32.27. All machines used wireless network adapters with the ath5k [ath5k] chipset. The machines were deployed in the RWTH Aachen computer science building, which meant numerous other competing and interfering networks at the same location (first and foremost eduroam, but also other smaller, local deployments). We consciously chose a channel already in use (channel 5) to stimulate the competing and interfering nature of the traffic. This scenario models our target scenario well: a small home network with one AP and one or several STAs, and various interfering 802.11 networks from neighbors.

As the other networks within the building were not under our control, we did not have controllable and reproducible settings for our experiments. We offset this disadvantage by lengthy measurements over the course of many days and nights. Additionally, we provoked different error rates in the experiments by setting static transmission rates (from 1 Mbit/s to 54 Mbit/s) and switching between them between experimental runs.

Furthermore, we interleaved the sending of packets of the Refector streams and the UDP-Lite streams that we used as comparison point to produce results in which at least slow fading and other slow channel changes would not influence our results.

Each experimental run consisted of 10 000 packets per flow, with one Refector and one UDP-Lite flow being sent concurrently to a machine. We evaluated results from 800 runs, 200 for each receiving machine. The witnessed PDRs spanned the range from below 20% to 99%. Connections that resulted in PDRs outside this range were removed from consideration. The reasoning for this was as follows. First, PDRs below 20% over an extended time frame mean that channel quality is so low that there is a high probability that even the basic management frames such as beacons and association frames will not be reliably received any more. In this case, association of a STA to the 802.11 network, and therefore any data transmissions, would fail, or the association would be lost during the experiment, terminating the connection. Because broken connections do not give much insight into the performance of

[24] In the following, whenever we discuss UDP-Lite, we assume the checksum coverage to be set to 8 bytes to only cover UDP-Lite's header, thereby implementing payload error tolerance.

Refector and UDP-Lite, we removed these runs from consideration. Second, almost every run showed a small amount of packets that were completely lost, that is, either dropped by the MAC (for which there is no Refector support currently) or not even sensed as packets by the PHY. This is most likely due to collisions on the wireless channel and an effect of sharing the channel with other networks. At PDRs above 99%, this effect accounted for virtually all packet losses. Because neither Refector nor UDP-Lite have any effects on these losses and because, due to their randomness, results did not produce much meaning, we removed these runs from consideration. After removing such broken connections from our runs, as well as connections which did not produce any errors whatsoever, 505 runs remained for consideration.

3.2.4.2 Influence of Packet Size on Packet Loss

Note that above, we did not discuss packet size as parameter in our experimental setup. This is because for this evaluation, results are independent of packet size. Hence, there is no need to look at different packet sizes in our evaluation. This point warrants a short discussion however, as to the reasons for this unimportance. Under normal conditions, different packet sizes produce strongly different PDRs under the same environmental conditions (cf. Figure 3.3). However, for both UDP-Lite and Refector, packet size is of no consequence.

This is substantiated by a short experiment we ran between our AP machine and one of the STA machines. For several packet sizes, we set up UDP, UDP-Lite, and Refector connections and measured the PDR under comparable environmental conditions (see above for the explanation about interleaving of connections) and with statically set 802.11 rates to provoke different levels of BERs. The results are presented in Figure 3.6.

As expected, Figure 3.6a shows a decrease in PDR as packet size increases. On the other hand, both UDP-Lite (Figure 3.6b) and Refector (Figure 3.6c) show no significant difference in PDR between different packet sizes. This is, of course, because, while with standard UDP, an error in any part of the packet leads to a packet drop (therefore increasing the chance of a packet drop with increasing packet size), with both UDP-Lite and Refector, the size of the packet areas in which errors can lead to a drop are of static size. In UDP-Lite, the payload is error tolerant: checksum mismatches and subsequent packet drops only occur if the error is in the UDP-Lite header, not the payload. In Refector, both the payload and (most) headers are error-tolerant, so only errors in the MAC header or during reception on the PHY lead to packet drops. Note that the UDP-Lite results benefit from our changes to the MAC layer that allow corrupted frames to pass, as described in Section 3.2.3, which were activated during these and all following experiments. A UDP-Lite without this support would in effect behave like UDP: since frames are dropped on the MAC layer if they contain any errors, UDP-Lite's payload error tolerance is irrelevant, and consequently, its packet drop rates will increase with packet size. Also note that these graphs already foreshadow results from our evaluation proper: Refector's PDRs are slightly, but consistently, above those of UDP-Lite.

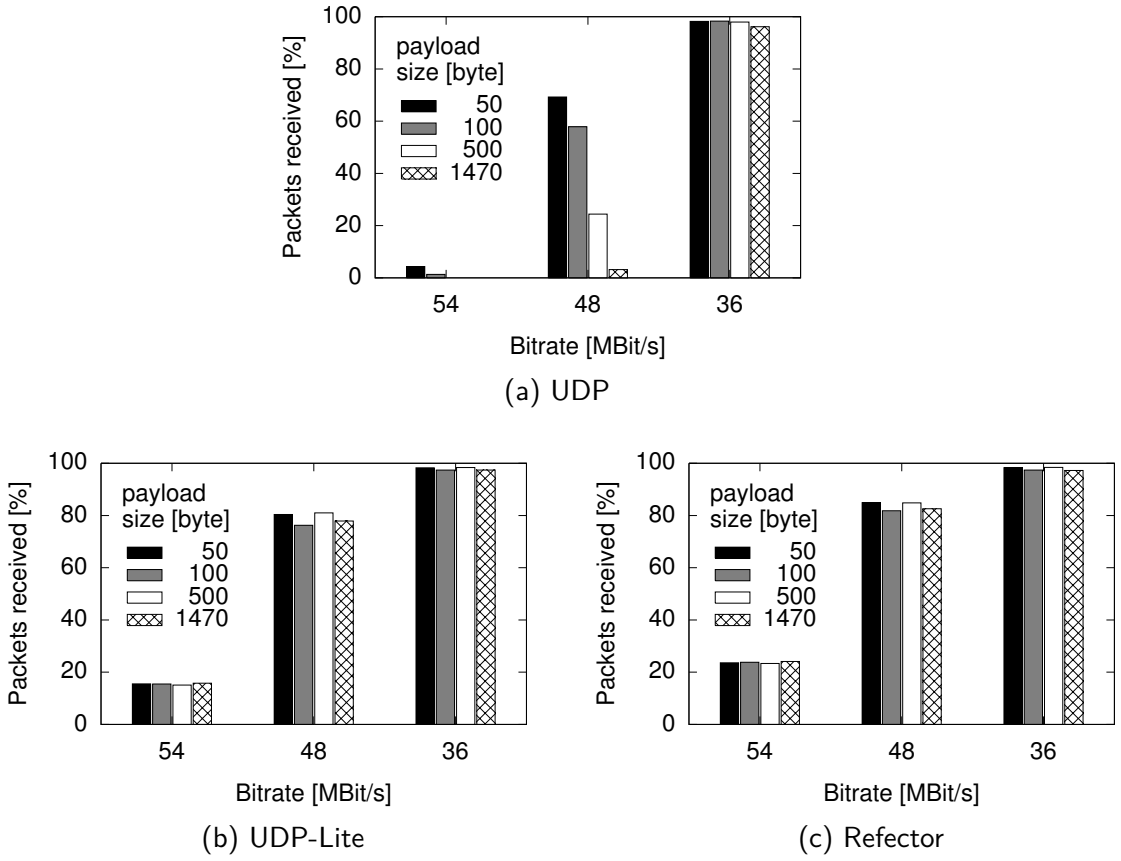


Figure 3.6 The influence of payload size on packet loss depends on the used protocol, and whether it provides tolerance to errors. UDP does not provide such tolerance, with its checksum covering both header and payload, so the payload size has a strong influence on the packet loss rate. Contrarily, UDP-Lite with a header-only checksum coverage, and Refector, which tolerates errors in headers and payload, have a reception rate independent of payload size.

Due to these results, we can abstract from packet size as a parameter for the remainder of this evaluation section. Nevertheless, we used different payload sizes (50, 250, 500, 1000, and 1470 bytes) for different runs in our evaluation as a safety measure to rule out potentially unaccounted effects. As we did not witness any such effects, we will combine them in one result and not split them by packet sizes for the rest of the evaluation.

3.2.4.3 Packet Delivery Rate

We can now focus on answering the two questions put forth in Section 3.2.4. We will first investigate the increase of PDR due to header error tolerance as provided by Refector. Because, as mentioned above, environmental effects on the wireless channel were beyond our control, we always interleaved a Refector and a UDP-Lite connection to the same receiver, alternating back and forth by sending one packet for one connection, followed by one for the other, and so on. We then compared these two connections, which would have witnessed similar environmental conditions, to each other, and looked at the PDR increase of Refector compared to UDP-Lite.

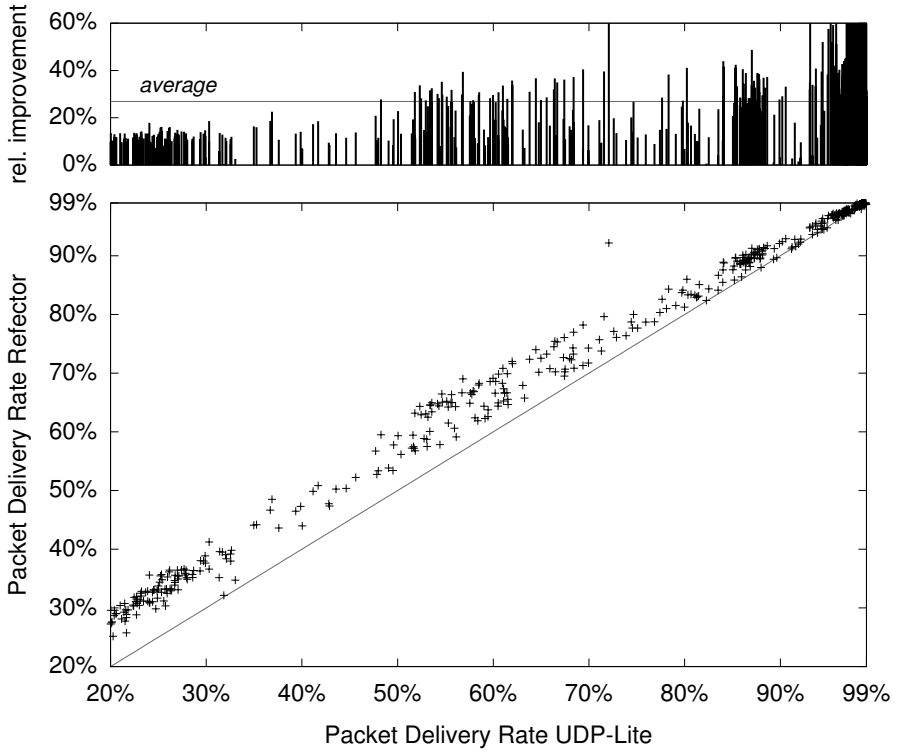


Figure 3.7 Comparison of UDP-Lite and Refector by packet delivery rate over 400 runs from different stations. The lower graph shows a direct comparison between each UDP-Lite/Refector pair of runs. Data points above the diagonal line denote improved performance of Refector over UDP-Lite. The upper graph shows the 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%.

The results are shown in the lower portion of Figure 3.7. Each data point denotes a connection pair, with the PDRs for UDP-Lite and Refector as the x and y values of the coordinate system, respectively. This means that whenever a data point is above the diagonal line, Refector performs better than UDP-Lite, which is virtually always the case.

Results with PDRs below 20% and above 99% missing is due to the filtering explained in Section 3.2.4.1. The fact that there are more results towards the extreme ends than in the middle is due to difficulty of setting up wireless channels to produce "medium quality" results, that is, PDRs that are neither very high nor very low. This is chiefly due to the fact that most Modulation and Coding Schemes (MCSs) show a very fast transition from near-perfect to near-zero transmission success, and consistently provoking an error rate somewhere in between requires careful setup of the network, taking even such effects as exact antenna positioning into account. Even then, minute and uncontrollable outside effects such as people moving along the hall of the building and attenuating the signal can move results out of that narrow area again.

While the lower portion of Figure 3.7 already shows that Refector outperforms UDP-Lite, the gains might not seem overly impressive. One factor that reduces Refector's performance is that we only apply recovery techniques to IP and UDP. The 802.11

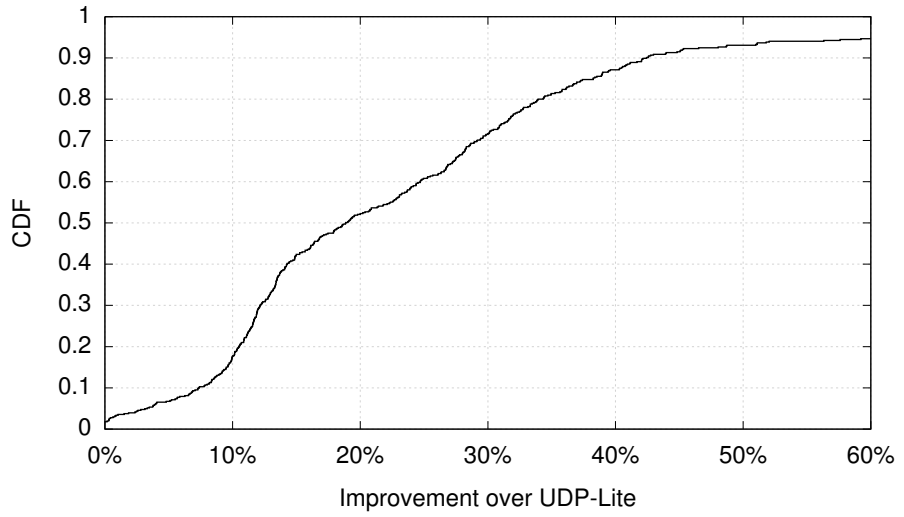


Figure 3.8 Relative improvement of Refector over UDP-Lite, using the same dataset as in Figure 3.7, visualized as CDF.

MAC header, at a size of 26 bytes almost as large as the IP and UDP headers combined, does not have any error tolerance implemented. While we disabled frame drops due to MAC layer checksum mismatches, errors in the MAC lead to frame drops with a high probability (for example, if the receiver MAC address is corrupted). However, even considering this, the visualization in the lower portion of Figure 3.7 is somewhat unfavorable because it disregards the relative improvements provided by Refector. For example, towards the high end, Refector's improvements are very low: if only 1% of all packets would be lost with UDP-Lite, Refector's potential improvement is only those 1%. On the other hand, those 1% can make a noticeable difference in quality: many error-tolerant applications, such as audio streaming codecs, will show their most noticeable quality degradation early on, so the difference between 1% and 2% packet loss, or 1% and 0.5%, can be significant. In contrast, the 10% difference between a UDP-Lite stream with 50% PDR and a Refector stream with 60% PDR will probably be almost inconsequential, because at such high loss rates, meaningful communication will be impossible. We therefore also visualize Refector's improvement over UDP-Lite as *relative improvement*, by dividing UDP-Lite's packet loss rate by Refector's:

$$\frac{PLR_{UDP-Lite}}{PLR_{Refector}}$$

These relative improvements are shown in the upper portion of Figure 3.7, with each data point visualized as bar to show the improvement, at the x-axis position at which it occurred in the lower graph. That means that the x-axis of the upper graph can be read as PDR: to the left are the worst and to the right the best channel conditions. This shows that relative improvements are especially high when the overall PDR is already high, that is, Refector can recover a large portion of those packets that are lost by UDP-Lite in those conditions. The average relative improvement of Refector over UDP-Lite was 27% over the course of our experiments. In the area of 95% and more PDR, this relative improvement was generally over 40% (often much higher),

meaning that packet loss could be almost halved (or even more than halved, in many situations).

Figure 3.8 shows the overall relative improvements as a CDF. While the average relative improvement was 27%, the CDF shows that the median was 19%. However, this is mostly due to lower relative improvements at high loss rates. It also shows that in this experiment, both average and median are of little expressiveness on their own. They can be almost arbitrarily skewed by the number of experiments in certain quality ranges. The results and their behavior over the range of PDRs is of more importance: Refector virtually always outperforms UDP-Lite, and the relative improvements are largest in the area where they matter most: when Packet Loss Rate (PLR) is still low enough to allow reasonable communication quality.

3.2.4.4 Misattribution

As discussed before, the main disadvantage of Refector, and header error tolerance in general, is the possibility of misattribution. We hence need to investigate whether such misattributions occur, and how often they do. While we gave an analytical approximation of misattribution rates (cf. Section 3.2.2.3), we assumed independently distributed bit errors. This is a simplification, and real-world measurements have shown bursty tendencies to varying degrees [WKHW02, HJL+09, HJL+12]. Consequently, we measured misattribution rates in our experiments and compared them to the analytical approximation.

Note that there are two types of misattribution that are conceivable: On the one hand, a packet can be misattributed to the wrong application, but on the correct host. This can happen if the destination port is corrupted. On the other hand, a packet can be misattributed to (a wrong) application on a different host. This could happen if the destination IP address is corrupted, and a host accepts a packet as its own that was destined for another one. In this case, misattribution can actually involve duplicate processing of the packet on two or more hosts, because one host accepting a packet does not preclude another host from also accepting it. However, in all our experiments, we never saw even a single occurrence of this latter case.[25] We assume that this is due to the fact that, to accept a packet erroneously, in addition to a corruption in the IP address that coincidentally matches another host's IP address, the MAC address would also have to be corrupted in a way that matches the other host's MAC address. Since we did not employ heuristic repair techniques at the MAC layer, such a corrupted address would have to exactly match the other host's MAC address, which is extremely unlikely considering the MAC address's size of 48 bits.

One problem in measuring misattributions is that these typically occur when error rates are high. This means header contents are not reliable any more and cannot be

[25] This is also the reason why the analytical approximation, which only investigated misattributions due to errors in the port field, and the real-life measurements can be compared. Since no misattributions between different hosts occurred, all misattributions were between applications on one host, and the decision which application to hand the packet to relies on the destination port field.

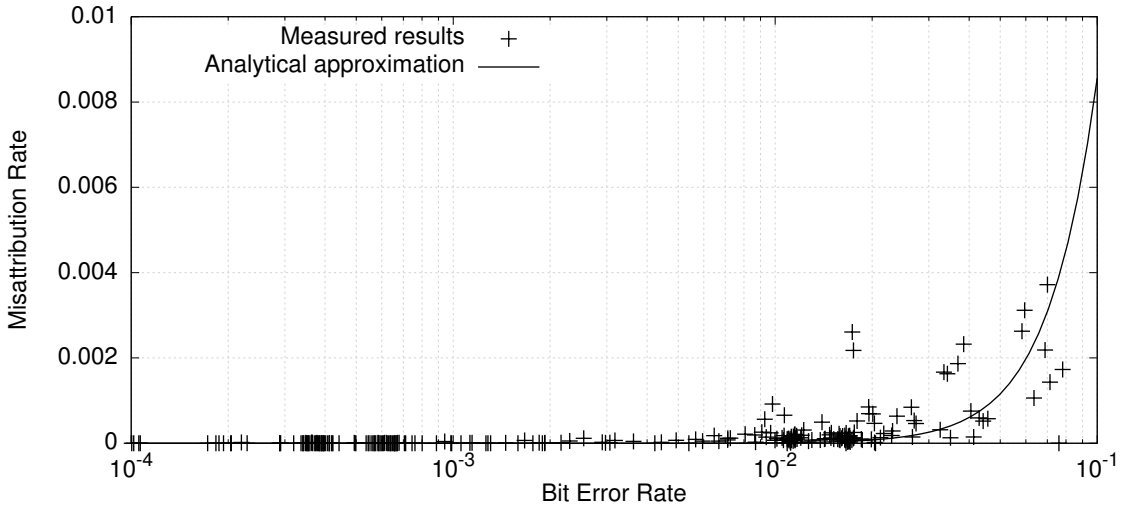


Figure 3.9 Packet misattribution rates for two concurrent Refector applications, measured in our evaluation setup, and compared to the analytical approximation for destination port misattribution given in Section 3.2.2.3.

used to evaluate whether a misattribution occurred. We therefore needed additional information to identify packets that would still be available if headers had witnessed a large degree of corruption. Hence, we filled the payloads of packets with bit patterns specific to each application flow. Even if a packet header was completely corrupted, this still allowed identifying the original sending and receiving application.

To evaluate misattribution, we ran two instances of a Refector-enabled test application that recorded packets on each of our four test hosts, and consequently 8 sending applications on the AP. Due to the very low volume of misattributions, we increased the number of packets per flow per experimental run to 100 000 packets (from 10 000 in the PDR tests).

The results from these experiments are shown in Figure 3.9. However, two remarks are important. First, a misattribution is not immediately fatal to an error-tolerant application. This is due to the design of such an application: to it, a misattributed packet will merely appear to have an extremely high error rate (because the contents strongly differ from the expected contents); nevertheless, such a misattribution is undesirable because it will decrease decoding quality of the application. Second, the BER noted on the x-axis of Figure 3.9 was calculated from the bit errors seen in the payload of the packet. Since each application flow had its payload filled with a characteristic bit pattern, the BER could be calculated from counting the erroneous bits in the *payload*. For misattribution, however, the BER in the *header* portion is important. This means that the BER given in the figure is only an approximation to the true BER. However, it has been shown [WKHW02, HJL+09, HJL+12] that, while the frequency of bit errors does not stay the same over the length of a packet, the differences stay within a reasonable range. Depending on modulation, coding, and the hardware used, bit rates between the start and the end of a packet typically deviate by less than a factor of 2 which, while not insignificant, has little influence

on the range of values shown in our evaluation. Therefore, in this case, header BER can be reasonably approximated by payload BER.

Coming back to the evaluation results, up to BERs of 10^{-2}, misattribution rates stay very low, with often not a single misattribution occurring. As we already pointed out, such a BER is already extremely high and barely usable (cf. [NOCW07] and Figure 3.3). At more realistic usage scenarios, with BERs at or below 10^{-3}, misattribution are effectively non-existent, with not a single occurrence in our experiments. Finally, it can be seen that the analytical approximation from Section 3.2.2.3, given the uncertainties about the actual BER within the header's destination port field, provides a reasonably close match to the measured results.

3.2.4.5 Encryption

So far, the evaluation abstracted from the concept of encryption and used unencrypted communications. However, the vast majority of wireless links uses some sort of encryption because the overhearing of wireless transmissions otherwise makes all communications susceptible to eavesdropping from other users in the vicinity.

Encryption algorithms can be sorted into two categories. One category are so-called stream ciphers, in which the data is encoded as a stream of bits, bit-by-bit. Other algorithms are block ciphers, that is, they operate on a complete block of a certain size of bits as an indivisible operation. The most important difference between the two from the point of view of error tolerance is the property of error propagation, that is, whether a single bit error in the encoded stream produces only one bit error in the decoded stream, or more than one. Typically, stream ciphers do not have any error propagation: a single bit error in the encoded stream maps to a single bit error at the same position in the decoded stream. For block ciphers, the error propagation strongly depends on the block chaining technique used with the algorithm. The effects can vary from single bit errors in the received data producing only single bit errors in the decoded data, to corruption of the block of data in which the bit error occurred, the block and its successor, the block and all its successors, to the complete corruption of the whole message [BV09, Riv97].

802.11 provides both types of ciphers. As stream cipher, it uses Temporary Key Integrity Protocol (TKIP) with RC4, while for a block cipher, it uses CCM mode Protocol (CCMP) with AES. For this evaluation, we will use TKIP/RC4, because it lends itself more to the use case of error-tolerant transmissions. Note that in the most recent version of the standard [IEEE12b], TKIP/RC4 has been deprecated due to security concerns. The evaluation in this section should therefore not serve as a suggestion to set up an 802.11 with this coding scheme, but rather as a general investigation of the influence of encryption on error-tolerance. It is important to consider that stream ciphers themselves are not insecure; rather, 802.11's implementation of RC4 with TKIP is not specified and realized in a secure fashion.

There are two more problems regarding this evaluation that are specific to 802.11. One problem is that, if encryption is used, 802.11 introduces a second checksum into the frame, the Message Integrity Code (MIC), which provides another check for

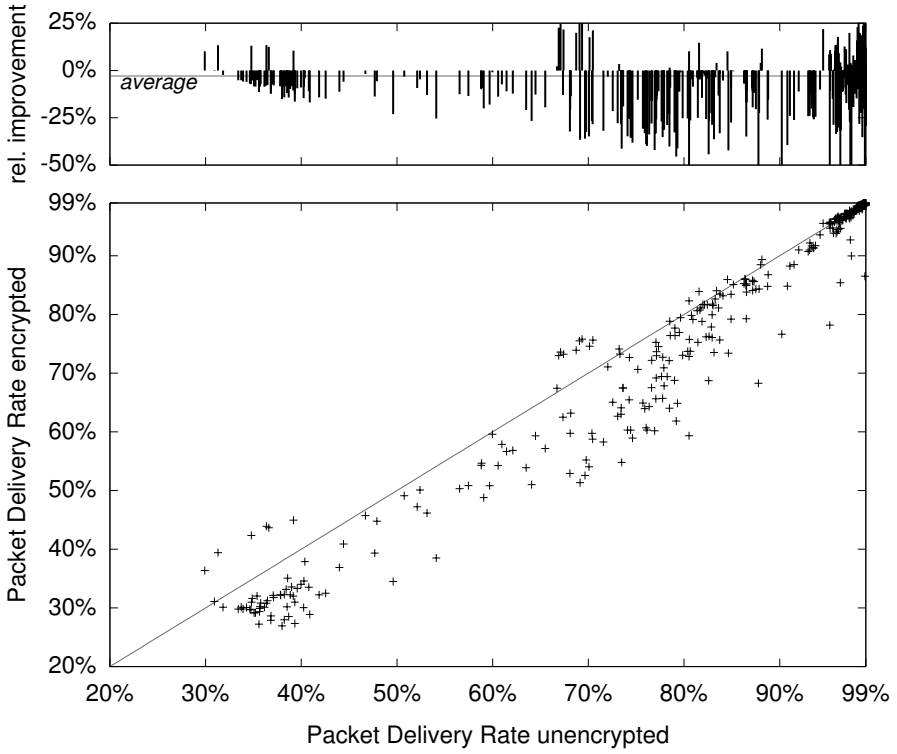


Figure 3.10 Effect of (TKIP/RC4) encryption on Refector. Due to the fact that bit errors in the Initialization Vector (IV) produce unrecoverable errors during decryption, encryption reduces the effectiveness of Refector. However, at good channel conditions, the effect is not as pronounced, and on average, the loss in effectivity is low.

message integrity. As with the CRC, the MIC checks have to be disabled to accept frames with mismatches. This has potential security implications, especially in combination with our second change that we need to introduce: Standard 802.11 MAC implementations take countermeasures against potential attacks if too many MIC mismatches occur, which we also had to disable. This again shows that the results presented in this section are not supposed to support the idea that the presented setup provides a feasible combination of robust error tolerance and high security, and should be deployed. Rather, it gives a more general insight into the influences of encryption on error tolerance. The specific problems of this implementation are not fundamental; they are rather effects of 802.11's flawed implementation of security features. Robust and (for their respective times) secure stream cipher algorithms have been deployed in mobile communications networks for many years (for example, GSM and UMTS [3GPP03,3GPP09]) and continue to be specified and used (for example, LTE [3GPP06, 3GPP11]).[26]

Coming back to the evaluation results, we will first have a look at the amount of degradation that an encrypted stream introduces into Refector's error tolerance scheme as opposed to an unencrypted stream. Figure 3.10 shows that at low and medium delivery rates, there is a noticeable degradation of up to 40%. This strong

[26]While GSM's A5/1 and especially A5/2 ciphers are often given as examples of insufficiently secure encryptions even for their time, this is less a problem of the algorithms themselves, but chiefly due to the limited key size used for political reasons [Sch96, p. 389].

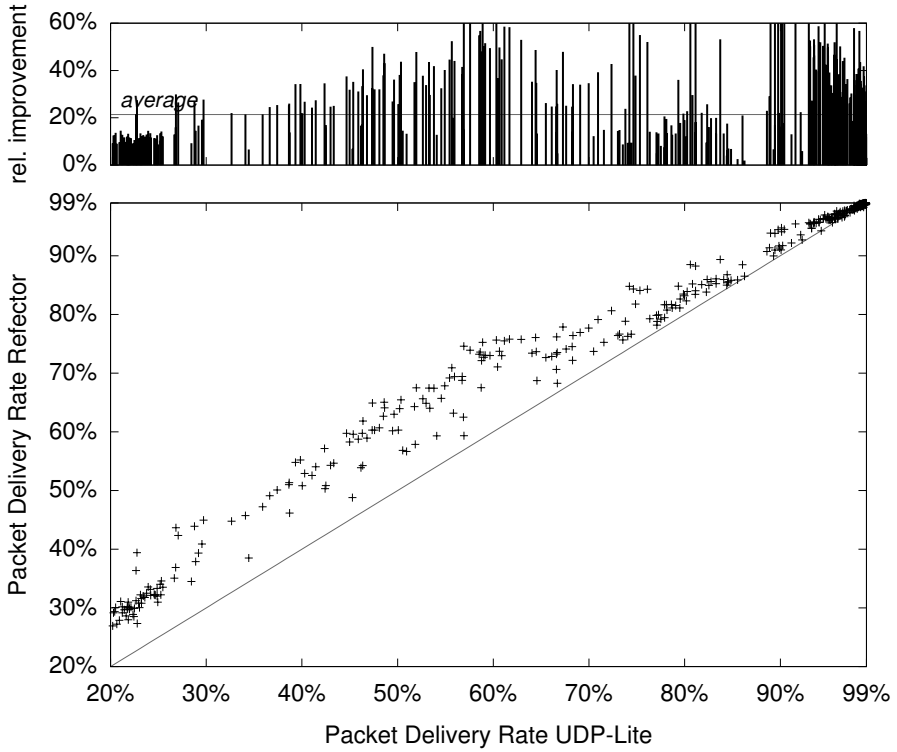


Figure 3.11 Comparison of UDP-Lite and Refector when (TKIP/RC4) encryption is used. The results are directly comparable to those in Figure 3.7. While there is more diversity and scattering in the results, the general trend remains the same: Refector virtually always outperforms UDP-Lite, with an average relative improvement of about 22% in our experiment series.

degradation is chiefly due to the fact that TKIP uses per-frame keys, created from the main encryption key via so-called Initialization Vectors (IVs). The 32-bit IVs are sent with their respective frames, and any bit error in them renders decryption impossible. Therefore, at high error rates, many frames are unrecoverable due to bit errors in the IVs. With high quality connections, IVs are much more rarely corrupted, and so the effect is not as pronounced. In fact, at high PDRs of above 95%, degradation and improvement occur in almost equal shares, a result that is reflected in the low average degradation of only 3% of the encrypted channel compared to the unencrypted one over all our experiments. At such high PDRs, the BER is low enough that the low number of occurrences of errors in the 32-bit IV are negligible compared to errors that occur in the vital IP and UDP header fields (comprising 104 bits) and might or might not be recoverable. This is encouraging, because it means that, on average, encryption (at least of such a type, which sends IVs to create per-frame keys) is less of a problem at high channel qualities, the case that we already established is especially interesting for error tolerance.

Finally, we will investigate the respective effects of encryption on UDP-Lite and Refector. Figure 3.11 shows the results of this evaluation. Evaluation setup and result aggregation are directly comparable to the results from Figure 3.7, with only the addition of encryption as difference. The results show more scattering and less homogeneity in the results. This is due to the effect of unrecoverable errors in the IV

as described above. While this leads to more diverse relative improvement values, the average improvement is similar and only somewhat reduced due to the IV effect, and a slightly different distribution in results across the range of PDRs. Again, it should be noted, just as in Section 3.2.4.3, that the average relative improvement can be easily skewed by the distribution of results across the PDR range, and can only give a rough estimate of general performance.

3.2.4.6 Performance

Refector's heuristic packet matching algorithm indubitably introduces additional computational overhead into the packet handling routines of the network stack. This overhead can be roughly categorized by the following thought experiment:

If a packet arrives without any errors, Refector does not introduce any additional overhead, because the heuristic error tolerance routines are not executed. A correct packet is handled exactly the same after the introduction of Refector. If a corrupted packet arrives, Refector's routines produce a linear instead of a static overhead: under normal circumstances, a lookup for the one or several of the parameters destination port, source port, destination IP address, source IP address are done. These lookups are typically (for example, in the Linux kernel) done on a hash map, which produces a constant lookup time. In contrast, if no perfect match is found, Refector needs to traverse the list of currently open connections one by one and match each connection's parameters against the received values, which produces a linear overhead. The actual matching can be done very efficiently by one of the various algorithms available to count Hamming distances between bit fields [War07], especially considering that most fields were categorized as don't-care and only a small amount of bits need to be compared.

The overhead can therefore be calculated as $\mathcal{O}(PER \cdot n \cdot H)$, with PER being the packet error rate, n the number of open connections and H the overhead of computing the Hamming distance: whenever a corrupted packet is received, the Hamming distance calculation has to be done for all open connections. With H being a constant factor and PER being a value between 0 and 1, we arrive at a linear complexity of $\mathcal{O}(n)$. To estimate n, we monitored the number of open connections during normal computer usage by several computer science students specializing in communication systems.[27] Results were typically in the range of approximately 25–35 concurrent connections, rarely reaching 50–60, never increasing above the 100 connection mark.

Finally, packet processing on today's end hosts, even at high speeds, produces only negligible overhead to start with. Consequently, when we deployed Refector on our test machines, we did not notice the computation overhead nor were we able to measure it. While our experiments only explored 802.11a/b/g data rates up to 54 Mbit/s, this suggests that even fully utilizing the higher data rates provided by 802.11n/ac should produce no performance problems.

[27] It stands to reason that their use of network communications is at least as intensive as an average user's.

3.2.5 Summary

From these results, we can see that Refector is an effective tool that reduces packet loss under challenging conditions. Even in direct comparison to UDP-Lite, which already introduces payload error tolerance, we saw an average improvement of 27% in our experiments. We did not compare ourselves directly to UDP because of the problems in comparing the two approaches, specifically because UDP, while it does not give any reliability guarantees that data arrives at all, when it does, it is guaranteed to be error-free (disregarding error that are not recognized by the checksum). However, if we ignore this fact, Refector can produce immense reductions in packet loss, especially with larger packets, which we showed in Figure 3.6.

The largest problem of heuristic header error recovery – misattribution – can be shown to be of minimal consequence, with misattributions only occurring exceedingly rarely until BER is in excess of 10^{-2}. Furthermore, even encryption does not completely rule out the use of error tolerance. Depending on the used cipher type and mode, heuristic error recovery can still be effective. In our experiments using a standard 802.11 encryption, we could see that, while packet loss increases somewhat, the relative improvement of Refector over UDP-Lite is roughly equal to the unencrypted case. While these results are somewhat tentative, they show that encryption in itself does not pose an unsurmountable problem to error tolerance. Finally, we gave insight into the computational overhead of Refector when added to a network stack, and argued that it is negligible, an argument that was confirmed by our inability to measure any significant difference between a normal and a Refector-enabled network stack's runtime performance.

3.3 Use Case: Refector-ISCD

Most of the evaluation presented in this dissertation focuses on connection- and heuristics-related metrics, such as packet loss and misattribution rates. These have the advantage that they are easily comparable and independent of the used applications.[28] Since the improvements are independent from these factors, focusing on those metrics keeps our results clear and focused.

However, since one motivation for this work is support for media codecs, it stands to reason that at least some investigation should be done into the effect that heuristic header error tolerance has on the perceived quality of those types of transmissions, and whether Refector can indeed improve quality, and if so, by how much.

For this investigation, we focused on a special group of media codecs using so-called Iterative Source–Channel Decoding (ISCD). The codec for this investigation was developed at the Institute of Communication Systems and Data Processing (IND) at RWTH Aachen University, and the combination of Refector with ISCD was part of a joint work with Tobias Breddermann from that institute.

[28] And in many cases, though not in the previously presented evaluation, are also independent of the underlying MAC and PHY.

We will first give a very short introduction into ISCD and the concept of soft information. Since this is an entire field of research in itself, we will only provide as much information as is necessary to understand our approach and evaluation, and refer to the literature quoted in the following for further information. After this introduction, we will present results from the combination of Refector with ISCD and show what quality improvements can be reached by employing Refector. Conversely, we will also show how direct access to PHY decoder output, in the form of so-called soft information, can improve Refector's recovery mechanisms.

3.3.1 Introduction to ISCD

ISCD [Gör00, ACS08] is a special use case of the turbo coding [BG96] principle. In turbo codes, instead of one coder/decoder (codec) pair, with the encoder residing at the sender's and the decoder at the receiver's side, two codec pairs are employed. At the receiver, the output of the first decoder is used as (part of) the input of the second decoder, and vice versa.[29] Via several iterations of this process, traversing each decoder multiple times, the decoding quality can be significantly increased by approaching the Shannon limit much closer than other codes of the era.

ISCD changes this standard use case of two codec pairs on the physical layer as a combined channel codec, to a scheme where one codec is used as physical layer codec, and the other as so-called source codec, encoding and decoding the source material, that is, the media payload. The source codec plays a vital role in this scenario, because not only does it encode the payload, it also is in charge of adapting to potential channel effects, a job that is typically done by the channel coder. In ISCD, the channel coder is kept at a fixed rate of 1, that is, no redundancy at all, which is typically a very subpar choice for a channel codec. However, in this special case, the combination of rate-1 channel code, an adaptive source code, and a specially crafted interleaver that evenly distributes information over all bits in the packet, a further improvement can be realized [AVC05], approaching the Shannon limit even closer (given enough iterations between source and channel coder).

The two decoders iteratively exchange so-called *soft information*: Upon frame reception, the demodulator transforms the physical symbols not into bits, but instead into bit probabilities (that is, how likely this bit is to be a 0 or a 1) in the form of log-likelihood ratios:

$$L(x|\vec{z}) = ln\frac{P(x=1|\vec{z})}{P(x=-1|\vec{z})} \tag{3.3}$$

where x denotes a bit in bipolar notation (that is, 1 denotes a 0 and -1 denotes a 1) and $\vec{z} = (z(1)\dots z(n))$ the received frame of length n bits. The sign $\hat{x} = \text{sgn}(L(x|\vec{z}))$ denotes the *hard decision* output (whether the bit should be considered 0 or 1 at this point), and the magnitude $|L(x|\vec{z})|$ the reliability of this decision. The goal of iterative decoding is to steadily increase the magnitude of each soft value, until correct decoding is achieved to a high degree of probability.

[29] This behavior is the origin of the term "turbo code". Like in a turbocharged motor, where the exhaust pressure is used to produce additional air intake into the motor, a turbo decoder uses the output of one decoder to feed into the other decoder [BG96, HOP96].

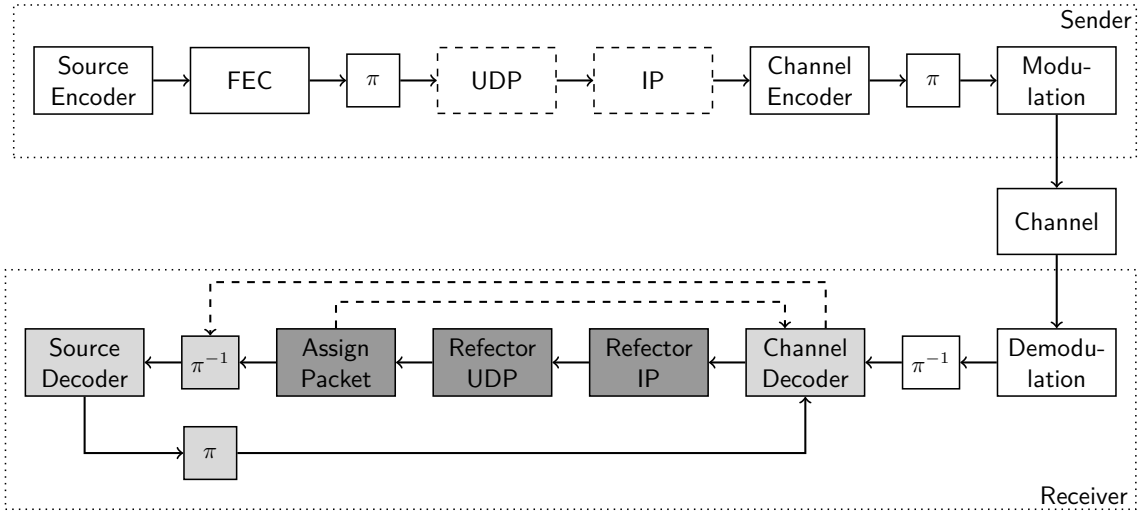


Figure 3.12 ISCD transceiver system. In a circuit-switched setup, Channel (PHY) and Source (APP) decoder are directly connected (light gray boxes only). In a packet-switched network, transport and network layer need to be traversed, and the packet needs to be assigned to the correct application (dark gray boxes).

The main problem with ISCD in our case is that it was not designed with packet-switched networks in mind, but rather for systems in which the receiver always knows what application a packet is destined for, like it is the case in circuit-switched networks with dedicated connections, or in systems that multiplex and demultiplex connections on the physical layer due to time slots (TDMA), dedicated frequencies (FDMA), or combinations thereof. In fact, ISCD performs very poorly in packet-switched networks. This is because the iteration step that improves decoding quality can only begin once the packet has traversed the network stack and we know which application a packet belongs to, and hence which source decoder should work in tandem with the channel decoder. However, the initial output of the channel decoder is of quite poor quality because of its use of a rate-1 codec which is very susceptible to errors during transmission. This means that the packet has to traverse the network stack while potentially containing many bit errors (in payload and headers, but only the latter is of consequence at this point), leading to a high packet drop rate before ISCD ever has the chance to produce high-quality decoding.

The overall ISCD setup is shown in Figure 3.12, with the sender side at the top and the receiver side at the bottom. The network stack is added between source and channel coder. In a circuit-switched ISCD setup, the receiver's side is set up so that channel and source decoder can directly interface with each other, separated only by the interleaver π or the deinterleaver π^{-1} which translate between the expected bit layouts. In a packet-switched network, the packet has to be assigned before the source decoder is found and can start iterations with the channel decoder. Note that the network stack only has to be traversed once to find the correct source decoder; the dashed arrows denote that the channel decoder is informed of the source decoder it may interface with in future iterations, and can afterwards circumvent the network stack.

Refector therefore is a natural choice to allow usage of ISCD in packet-switched networks: by being able to find the correct application a packet belongs to, even under header errors, we can employ the efficient ISCD decoding scheme in such networks. Conversely, ISCD's channel decoder provides Refector with per-bit soft-information. We will also see how this soft information increases the assignment rate of Refector, by allowing more packets to be assigned correctly.

3.3.2 Experimental Setup

To evaluate the effects of the combination of Refector with ISCD, we used a custom simulation setup. The main reason for this approach is that the ISCD implementation available to us was developed as a simulation model for a custom simulator used at IND. This simulator comprises both the source and channel decoders, as well as a channel model applying Additive White Gaussian Noise (AWGN) onto the simulated PHY symbols, in one monolithic block. There were no provisions for a packet-switched system and its network stack.

We decided to split this monolithic simulation into three functional blocks: source (de)coder, channel (de)coder, and channel model. These blocks were then encapsulated as stand-alone simulation models for the ns-3 [LH06, HRFR06, ns3] simulator. The decision to use ns-3 is motivated by its packet modeling. As opposed to many other network simulators such as OMNeT++, where packets are modeled in an abstract fashion as data structures of the used programming language, ns-3 uses a byte-by-byte exact representation of packets and headers as they would be used in a real system.[30] This type of modeling allowed an easy interfacing with the channel (de)coder by inserting packets into it and receiving (corrupted) packets out of it. Furthermore, due to this packet representation, the network stack model of ns-3 closely matches the Linux kernel's implementation compared to other simulators, which made it easy to transfer Refector concepts between the two implementations.

The metric for our experiments was speech quality. For different channel qualities, we sent speech samples from the ITU-T standardized voice sample collection [ITU01], compressed with the GSM AMR [ETSI00] codec at 12.2 kbit/s, which, with payload FEC by the application decoder, produced 97 bytes of payload at a sending rate of 50 frames/s (one packet every 20 ms). Speech quality was measured by Mean Opinion Scores (MOSs), using the standardized Perceptual Evaluation of Speech Quality (PESQ) tool [ITU01] for synthetic and objective speech quality values in the range from 4 (good) to 1 (poor). As a baseline setup to compare against, we used the circuit-switched setup, in which the speech data is sent without any headers over a dedicated link; this was done by simply running the original simulation stand-alone. We investigated the performance by changing the channel quality in increments of 0.1 dB and repeating experiments 20 times for each such data point. Error barrs

[30] The trade-off here is generally that an abstract modeling of packets is more generalizable to approaches that use different types of messages that do not fit the packet paradigm well. Conversely, ns-3 was designed to support network emulation, that is, interfacing a simulation with real systems, in which case the simulation needs to create and receive "real" packets, and therefore chose this exact representation of packets.

denote 95% confidence intervals. The original stand-alone simulation, however, did not create any noticeable variance, so we will refrain from showing error bars for it. The number of ISCD iterations between source and channel decoder before evaluating the resulting speech quality was set to 6 in the experiments. Cursory tests at other iteration numbers showed the same relative behavior between the different approaches presented in the following, so we expect the iteration number to not have any significant influence on the validity of our results.

Finally, the ISCD simulator was not designed for being run in parallel; in fact, the code does not even allow to run several completely separate instances on the same system. Hence, not only could we only use one ISCD connection in each experiment, the channel model also only allowed data from this ISCD connection to pass through. Since only having one connection does not adequately model a packet-switched network, we used a different approach to model concurrent connections. We set thresholds in our Refector implementation for IP address and port fields. If the received data for those fields differed from the correct data by more than a preset number of bits, the packet would be dropped. This distance modeled other concurrent connections that, due to the high error rate, would have "caught" the packet due to misattribution. Since this only models one direction of misattribution (own packets being lost to other connections, not foreign packets captured from other connections), we set these to be more stricter. For example, we presented a solution in Section 3.2.2.3 that guarantees a minimum Hamming distance between ports of 5 (unless a large number of ports is used concurrently); consequently, we set the threshold in our experiments so that any Hamming distance of 3 or greater would result in a packet drop, which is stricter, because no port has ever all distances of 3 resulting in a misattribution (cf. Section 3.2.2.3).

3.3.3 Packet-Switched ISCD

Figure 3.13 shows the results from our investigation. Several results are immediately apparent. First, none of our approaches, which we will describe and explain in detail in the following, performs better than the baseline circuit-switched ISCD. This, however, is expected. Note that channel quality is measured in signal-to-noise ratio per information bit (E_b/N_0), therefore abstracting from factors such as per-symbol transmission energy compared to per-symbol SNR (E_s/N_0). "Information bits" in this respect mean the raw voice data before any encoding. In the circuit-switched approach, this abstracts from code rate effects: for example, a rate 1/2 code introduces a bit of redundancy for every information bit; however, at the same E_b/N_0, each of these coded bits can then only use half the transmission energy to be sent. In our packet-switched approaches, however, header bits also need to be sent, and are not classified as information bits. Hence, we immediately have less energy budget per bit we need to transmit (header plus payload) than the circuit-switched version, putting us at a large disadvantage. Overcoming this disadvantage is near-impossible, though the figure already shows that, given the right cooperation between Refector and ISCD, we were able to approach the baseline case very closely. On the other hand, while we never reach the baseline scenario, we provide advantages

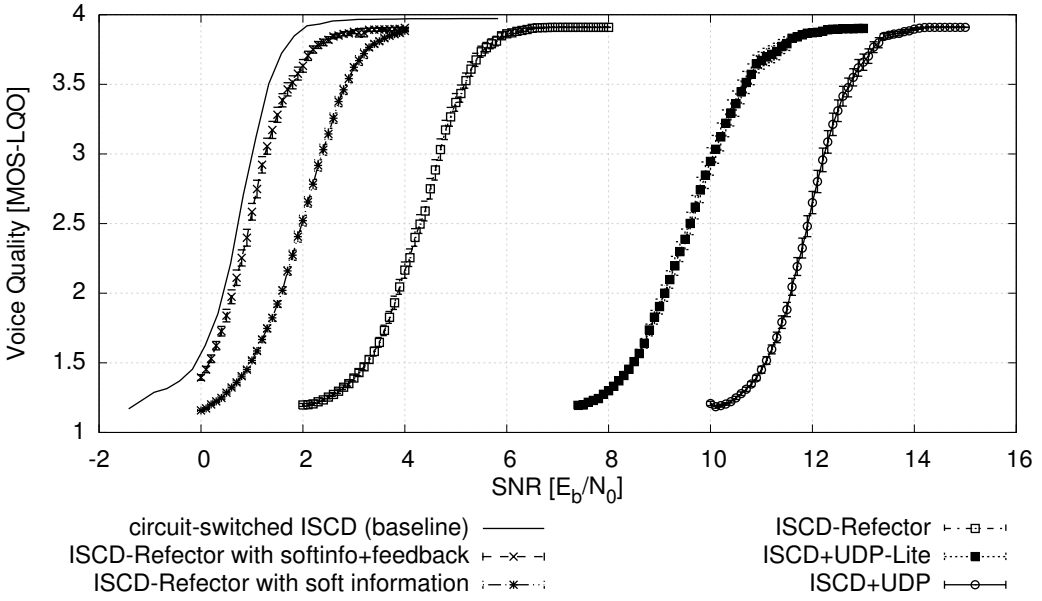


Figure 3.13 Comparison of ISCD in a circuit-switched, headerless scenario to different setups in packet-switched scenarios. Due to the additional overhead by introducing headers, the circuit-switched scenario (which assumed all connection control is done out-of-band) outperforms all packet-switched scenarios, but by different margins. UDP and UDP-Lite both require a massively improved SNR to provide the same quality (11 dB and 9 dB, respectively). Refector already performs much better at a degradation of roughly 4 dB. By extending Refector to make use of the soft information provided by the channel decoder, performance can be improved by another 2 dB. Finally, by feeding back repaired headers from Refector to the channel decoder, the degradation can be reduced to roughly 0.5 dB.

that packet-switched networks have over circuit-switched networks: primarily, since we use a full IPv4 and UDP implementation, we can multiplex different connections onto the same channel. We also need no out-of-band signaling that is generally necessary to set up and manage a circuit-switched setup. While this is outside of the scope of this investigation, by using packet-switched ISCD, this out-of-band overhead could be eliminated, further increasing its performance compared to circuit-switched ISCD.

Coming back to Figure 3.13, we will first observe the performance of a packet-switched ISCD without Refector, that is, simply encapsulating ISCD's voice payload into standard IP/UDP packets and sending them. The results show a massive degradation of 11 dB. This is because on the receivers side, all bits have to be correctly decoded on the physical layer by the weak rate-1 channel decoder, which requires very good channel conditions. Otherwise, the packet will be dropped due to checksum mismatches. This effectively renders ISCD useless: either the packet gets dropped, or it is already perfectly decoded after the initial channel decoding step, and no iteration can lead to further improvement.

Next, we replaced UDP with UDP-Lite as transport-layer protocol, setting the checksum coverage to the minimum of only covering its own header. This already leads to a significant improvement of 2 dB, but the degradation is still extremely severe at

9 dB compared to the baseline case. The improvements are due to the decrease in packet losses. Still, any packet containing bit errors in the IP and UDP-Lite headers after the initial channel decoding stage are dropped, leading to a strong quality degradation.

From these results, we can conclude that standard approaches to packetizing ISCD streams are impractical.

3.3.4 Refector-ISCD

We next combined ISCD with Refector. At this point, checksum mismatches do not lead to packet drops any more. Only drops due to not being able to correctly repair packets occur at this stage. This immediately leads to a large performance improvement of 5 dB over UDP-Lite and 7 dB over UDP. Still, to reach the same voice quality as the circuit-switched setup, 4 dB more energy has to be expended: this shows the burden of the additional overhead introduced by the packet headers.

However, we can improve Refector in a way that was not possible in our WLAN scenario. Since ISCD is a soft-information coding system, the output of the channel (and, less relevantly, source) decoders is per-bit soft information as described in Section 3.3.1. This gives us much more detailed information about the state of potentially broken bits. The per-bit log-likelihood ratio (cf. Equation 3.3) denotes the confidence that the bit is a 0 (or a 1). We can use this additional information to Refector's advantage. Consider the following example: For the destination port, we tolerate at most two bit errors, that is, require a Hamming distance of 2 or less. The decoding leads to a distance of 3. However, the soft information indicates that all the matching bits are correct with a high probability, while the non-matching ones are of low confidence.

To leverage this information, we calculate a reliability metric over the bit field $(x_1, \ldots, x_n)$, $x_i \in \{-1, 1\}$ in bipolar notation:

$$\sum_{i=1}^{n} x_i \cdot L(x_i)$$

This produces a number that increases with the reliability of the match. We then accept packets for an application if the reliability of the match is above a minimum threshold. Thus, we further increase the amount of packets that can be identified and handed over to the application, increasing the voice quality at the receiver's end. Figure 3.13 shows that this process improves the performance of ISCD by a further 2 dB. The large reduction compared to the standard ("hard information") Refector might seem surprising, but the reasons for this are twofold: First, packet loss has a very large negative influence on the quality of the AMR speech decoder. Second, the rate-1 channel decoder produces results that are so error-prone that packet drops occurred in almost every experiment, leading to said lost packets which did not reach the application. By using the available soft information, packet drops could be reduced to almost 0.

However, there is further room for improvement. Remember that the interleaver spreads information of each bit over several (in the theoretical optimum, all) physical layer symbols. Conversely, if some bit information can be recovered, it will positively influence the decoding quality of the other bits it was encoded with. The interleaver will, during its work, interleave header and payload bits to have the two groups share symbols with each other. Furthermore, after we traverse the network stack, we have an understanding of the correct contents of several header fields: those that are vital (and had to be repaired) and those that are static over all connections. We can therefore set the log-likelihood values of these bits to the maximum values $L(x) = +\infty$ or $L(x) = -\infty$. We then feed back these values to the physical layer decoder. In the following iterative decoding process, these values derived from header fields will boost the decoding of the payload bits, improving the voice quality. This boosts the following iterative decoding process between source and channel decoder and leads to an improvement in voice quality. Note that this feedback needs to be done only once: once the relevant header bits are set to maximum log-likelihood values, there is no benefit in feeding back information between Refector and channel or source decoder. This is denoted in Figure 3.12 by the two dashed arrows: when the packet assignment stage is reached, information is fed back to the channel decoder once. From then on, instead of progressing to the Refector IP implementation, the channel decoder interfaces with the deinterleaver (π^{-1}), setting up the standard iterative source–channel decoding loop. We showed the theoretical upper bounds of this in [BLV+10]. Figure 3.13 show that in a more practical setup, we can also benefit greatly from this feedback. A combination of Refector and ISCD, with Refector exploiting soft information and feeding back maximum log-likelihood values for (potentially repaired) header fields, almost reaches the performance of a circuit-switched ISCD setup, despite transmitting additional information in the form of headers over the channel at the same energy budget.

3.3.5 Summary

This use case, and the results from it, give us two important insights. First, while connection-oriented metrics, such as number of recovered packets and misattributed packets, strongly suggest performance improvements for error-tolerant applications, we showed in this section by example that this is indeed the case: Refector produces a significant quality increase for error-tolerant transmissions compared to other solutions (such as a standard UDP-based setup, or UDP-Lite), all other conditions being equal.

Second, Refector can, in turn, benefit from rich information provided by channel decoders. We showed that an extended version of Refector that replaces Hamming distance matches with a weighed metric that takes bit probabilities into account can further improve Refector's accuracy and consequently quality for media applications.

With these two questions answered, we will, for the rest of this dissertation, again focus on connection-related metrics due to their higher degree of abstraction and independence from specific application scenarios. While promising, we will also forego the benefits of soft information for assignment decisions in the following.

This decision is based on the fact that consumer hardware does not provide such rich information to the software, and therefore, for all its benefits, is not a realistic source of information for our envisioned use case and contradicts our design goals of easy deployment. Nevertheless, we consider the interaction between soft information and heuristic header error recovery a promising field that might warrant further investigation at a future point, especially if soft information becomes a more mainstream solution also found in consumer hardware.

3.4 Refector for Stateful Protocols

In the previous sections, we have shown a feasible and effective design and implementation for header error tolerance in some protocols. The protocols investigated there (IP and UDP) share the property that they do not carry a state: each received packet is, in effect, independent from packets received before and after it (with the exception of fragments due to IP-level fragmentation, which is not used in practice). However, many communication protocols have a state that is communicated via the packets that are sent. A typical example for a stateful protocol that is used with media streaming and hence could benefit from error tolerance is RTP. In this section, we will explore the concept of error tolerance for stateful protocols using the RTP [SCFJ03] as an example.

3.4.1 The Real-Time Transport Protocol

The Real-Time Transport Protocol (RTP) was designed to support the transmission of streaming content. It is an application-layer protocol, that is, it is used on top of IP and UDP. While it is possible to also combine it with TCP, this is typically not done due to the conceptual design of RTP. The protocol's idea is to provide certain desirable traits (ordering and timing information) that are absent from UDP without introducing the overhead that a TCP connection incurs for properties that are not considered important for streaming (reliability through acknowledgments and retransmissions, flow and congestion control). The reasoning is that reliability is not considered important enough to warrant the considerable overhead: keeping track of sequence numbers and receive windows, and acknowledging all received packets complicates protocol behavior significantly; in addition, the time-critical nature of streaming data makes retransmissions problematic if the information arrives too late to be useful. This argumentation follows the same reasoning as the one we base our motivation for error-tolerance on. On the other hand, ordering and timing information is considered important to give the decoder of the stream the information when to play which part of the received data to produce a coherent output. Instead of acknowledgments, RTP is typically used in combination with a control stream using the RTP Control Protocol (RTCP), which, in regular intervals, sends so-called reports that feed back information such as packet loss and jitter, to which the participants then can try to react, for example, by choosing a more robust media coding scheme or different bit rates (if packet loss was due to exceeding the

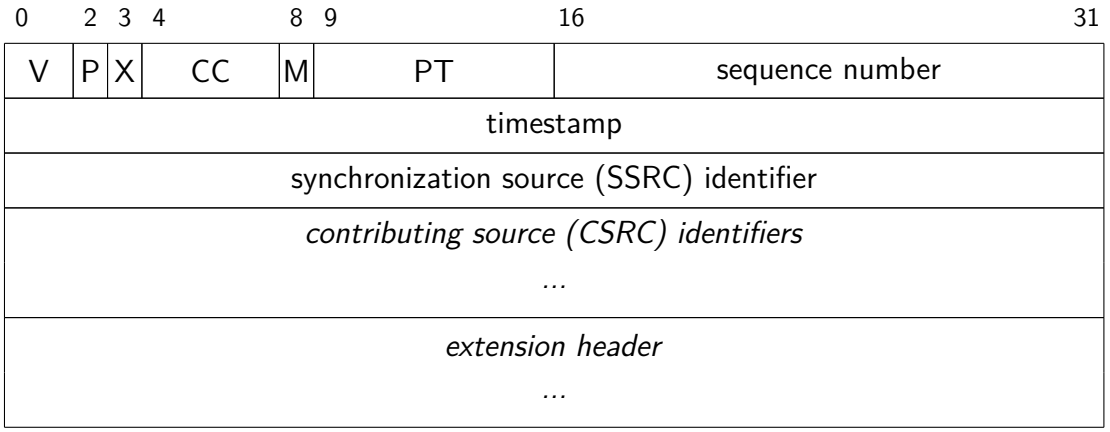


Figure 3.14 Contents of a Real-Time Transport Protocol (RTP) header. Names in italic script denote optional fields. V: version; P: padding; X: extension; CC: CSRC count; M: marker; PT: payload type.

available bandwidth). Due to the low volume of RTCP compared to RTP, and since we consider this control information more important and less error-tolerant than the media data itself (with errors in the packet loss and jitter information, for example, potentially leading to erroneous adaptations), we will not present an error-tolerance scheme for RTCP and instead focus on RTP.

The RTP header layout is shown in Figure 3.14. We will quickly explain some of the more basic header fields and flags and then discuss those that contain information about some fundamental concepts of RTP in more detail. The *version* (V) field, in the same way as the field of the same name in the IP header, contains a value that denotes the protocol version. The *padding* (P) flag denotes whether the packet was padded at the end. If padding is used, the amount of padding is denoted by writing its length into the padding itself. The *extension* (X) flag denotes whether an extension header is present, which allows to add additional information to the header by extending it. Again, if such a header is present, its type and length are contained in the extension header itself. The *marker* (M) flag can be used to mark certain packets. The meaning of this marking is specific to the payload type.

The *payload type* (PT) contains an identifier that identifies which codec (and optionally codec parameters) should be used to decode the stream. This list is prespecified, in a manner similar to IP protocol numbers [IANA15]. Like this list, the list of payload types is administered by Internet Assigned Numbers Authority (IANA) [IANA14] as per RFC 3551 [SC03]. Note that while it is possible to thus multiplex different types of media over the same RTP connection (e.g., multiplexing the video and audio stream of a video conference setup by sending the two logical streams with different payload types over the same RTP connection), the standard actively discourages this setup and strongly suggests using separate RTP connections (i.e., to different receiver ports) for multiplexing.

Sequence number and timestamp give information about the order of packets. The sequence number allows to recognize packet loss and reordering, while the timestamp gives information about when data contained in this packet should be played back relative to other received data.

The Synchronization Source Identifier (SSRC) and Contributing Source Identifier (CSRC) are used to identify the source of the stream. Each RTP stream has exactly one SSRC that identifies the source, and is randomly chosen to reduce the chance of collisions between SSRCs of independent streams. If an intermediator modifies or aggregates streams, it updates the SSRC and adds each input stream's SSRC as CSRC. For example, during a conference call, a central aggregator could mix all streams and then add each caller's SSRC as CSRC to each aggregated output packet. The number of CSRCs in the header is encoded in the CRSC Count (CC) field. In general, the CSRCs of a stream stay largely static, with no CSRCs at all in the basic scenario of a one-to-one VoIP connection, and a static number of CSRCs in the presence of a mixer, which only changes when callers join or leave.

3.4.2 Header Fields Categorization

With this information in mind, we can now categorize RTP's header fields in a similar fashion to the way we did with IP's and UDP's in Section 3.2.1. There are, however, two vital differences between RTP on one side and IP/UDP on the other. First, the concept of identifying the right stream is different. Since by the time the packet reaches the RTP handler, it can be assumed to belong to that connection (disregarding very rare cases of misattribution resulting from using Refector on the lower layers), so the stream identification problem is much diminished. Only if there are two or more logical streams (identified by different SSRCs) that are multiplexed within one RTP connection,[31] misattribution can occur at all, and then also only between this (typically) low number of multiplexed streams. Second, RTP contains header fields that are not static over the course of the connection, but rather change from packet to packet and convey information about the content in each packet. Following this reasoning, we refrain from categorizing header fields into "vital" and "don't-care" fields; the only field considered vital for stream identification would be the SSRC field. Instead, we consider fields to either be *static*, *predictably dynamic* or *unpredictably dynamic*, and investigate ways of reconstructing dynamic header fields in case of corruption.

Static fields are fields that can be expected to not change during the lifetime of an RTP stream. They are either the same for every RTP stream or specific, but unchanging, for each specific stream. These do not pose any special problem to repair, because repair techniques can simply follow the same ones laid out in the case of stateless protocols in Section 3.2.

Predictably dynamic fields are fields that are expected to change from packet to packet, but in a pattern that allows inferring future field contents from investigation of previous contents of those fields. Thus, in a corrupted packet, such fields are repaired by inferring the value from previously learned contents in uncorrupted packets.

[31] For the remainder of this section, the terms stream and connection will be used to distinguish between one or several logical *streams*, denoted by their SSRCs, multiplexed into one RTP *connection* that matches a socket connection via lower-layer protocols.

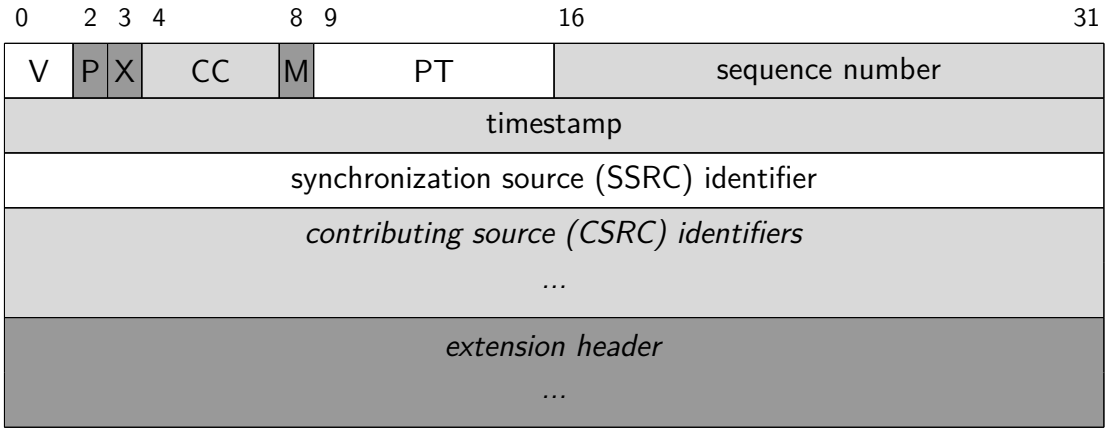


Figure 3.15 The same RTP header as in Figure 3.14, but with fields colored according to the recoverability via our heuristic header recovery scheme: white denotes static fields, light gray predictably dynamic fields, and dark gray unpredictably dynamic fields. Most fields are recoverable; even those that are categorized as unrecoverable are salvageable in some circumstances.

Unpredictably dynamics fields are fields that might or might not change from packet to packet, even within the same stream, and without any possibility to infer future field contents from previous contents.

We will now discuss the categorization of each header field in detail. Figure 3.15 visualizes this categorization.

The *version* field works the same as in the IP header. Its job is to identify the version of the RTP protocol. As there is only one standardized version of the RTP protocol, identified by the number 2, this field has to contain the static bit pattern 10 in every RTP packet.

The *padding (P)* field denotes whether the packet has been padded by the RTP instance on the sender's side. This field cannot be easily predicted, at least if RTP tries to enforce fixed packet sizes while the payload data created by the codec is not of static size. In that case, the padding value might change from packet to packet in an erratic fashion, making the field unpredictably dynamic. However, this use case (dynamic rate codecs producing variable size output combined with enforced padding to create uniform packet sizes) is rather exotic; in most cases, there is no reason for RTP to pad the packet, because lower-layer protocols will know better whether there are compelling reasons to pad the packet (e.g., padding Ethernet frames to the minimum frame size required for correct functioning of Ethernet). Thus, assuming that this field is 0 can be considered a good educated guess; while not currently implemented in our solution, watching the padding field in all correct packets and statically setting it to 0 if no other value was ever seen is a reasonably safe solution, especially considering that even if padding is present and by mistake not removed from the packet prior to handing the payload over to the codec, the results will be codec-dependent and might not have any negative results whatsoever.

The *extension* field denotes the existence of an extension header. If the field is set to 1, then an extension header follows the main RTP header. Vital information, such as the length of that extension header and whether a second extension header

will follow, is contained within that first extension header. In many ways, the field behaves like the padding field with respect to predictability and repairability. While extension headers might only be present in some packets and there might not be a way to predict from previous packets whether the next one will contain such an extension, these extensions are exceedingly rare in practice. In fact, the RFC [SCFJ03] actively discourages the use of extension headers due to their overhead, and most popular predefined audio and video payload types (e.g., G7xx [Cas07], AMR [SWLX07], H.264 [WEKJ11]) do not define any extension headers. With the same reasoning as in the padding case, we therefore argue that, when in doubt, repairing the field to 0 is a reasonable choice, though not by all means perfect.

The *CSRC count (CC)* field denotes the number of 32-bit CSRC records present in the header. As described above, the number of CSRCs is expected to change very rarely over the course of a connection, if at all. It is therefore possible to learn the content of the CC field by monitoring its contents in previous packets: for each SSRC (if there are even more than one present within one RTP connection), the CC can be heuristically repaired by setting to the last value seen before.

The *marker (M)* bit is, in the words of the RFC, "intended to allow significant events [...] to be marked in the packet stream" [SCFJ03]. How to exactly define the concept of a significant event is at the liberty of the payload type used. For example, codecs that do not send data continuously, but only if significant data is available (e.g., talkspurts in voice codecs) can use this field to denote the first frame after a silence period; video codecs that send frames larger than the maximum packet size can denote the start of a frame in a packet. Due to the diversity and unpredictability of this field, we consider it unpredictably dynamic.

The *payload type (PT)* field denotes the profile, that is, the type of content within the payload of the RTP packet and the codec to be used to decode it. As described above, this field is not expected to change over the course of a stream: for each SSRC, this field is static and therefore is learned after seeing the first uncorrupted packet for a stream.

The *sequence number* identifies the order of packets. It is incremented by 1 with every packet sent. As such, it is the prime example of a predictably dynamic field: once we have seen one packet, we can predict the value for all future packets, provided there is no reordering of the corrupted packets. However, we consider this a reasonable assumption: while packet reordering does happen in Internet connections, it seems to show bursty tendencies, with most connections only showing numbers of reordering evens [ZvM04]. Furthermore, reordering is most prevalent when bursts of packets are sent and tends to occur more often as packet sizes increase [PBB05]. However, at least for audio streams, we can expect to not have any burst behavior, but, in fact, pauses between individual packets. Since audio data is comparably small, there is no need to send large amounts of data spanning several packets at the same time; thus, audio packets are sent at regular intervals (e.g., every 20 ms) to satisfy end-to-end latency requirements [ETSI06, ITU03].

The *timestamp* field contains the time (in a unit predefined in the payload type) at which the contents of a packet should be played at the receiver. At least for

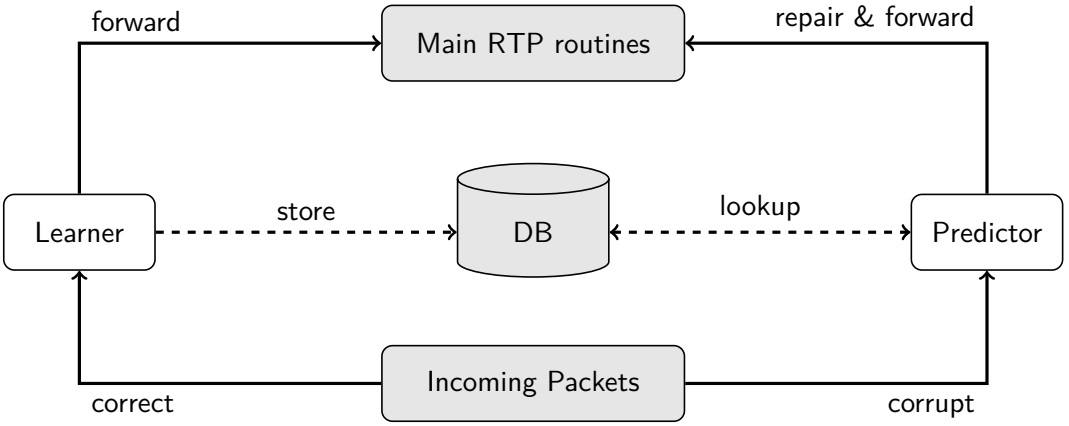


Figure 3.16 Packet flow through the extended RTP library. Correct packets will be inspected by the learner, their contents saved to a simple database, effectively a list of entries indexed by SSRCs, and forwarded to the main RTP processing routines. Conversely, corrupted packets will be passed to the predictor, matched against expected header field values for ongoing streams, repaired to match the best stream, and afterwards forwarded.

audio streams, this field will increase predictably due to the aforementioned regular intervals between packets.[32] The timestamp is then tightly coupled to the sequence number: for every increase in the sequence number, the timestamp increases by a static value. Thus, the timestamp field is also predictably dynamic, at least in audio applications.

As described above, the *SSRC* is not expected to change over the lifetime of a stream, and is thus static.

The *CSRCs* can change over time, but as pointed out above, they are expected to do so very rarely, if ever. As such, they could be considered static; however, a better choice is to consider them predictably dynamic and assume that, in a corrupted packet, the CSRCs are the same as in the last uncorrupted packet. This covers the wider range of rarely changing CSRCs reliably, without preventing occasional CSRC changes.

3.4.3 Stream Identification: The Learner–Predictor Scheme

The previous sections already hinted at the fact that, to properly support the repair of predictably dynamic fields, we need some way to learn packet contents and then apply this information to corrupted packets. This information can then be saved in a very simple database. Since an RTP connection can encompass several streams, and these streams are identified by their SSRC, this field will act as the primary key of our database. A conceptual diagram of the behavior is shown in Figure 3.16.

Whenever a correct packet arrives,[33] its header fields are read and information saved to the database by the *learner* unit. Static fields only need to be saved once, because

[32] In video streams, several packets with the same timestamp, but increasing sequence numbers, might arrive, due some frames containing too much data to fit into one packet.

[33] We will discuss how to identify error-free RTP packets in Section 3.4.4.

they never change afterwards. The way predictably dynamic fields are saved depends on their type: *CC* and *CSRCs* are always saved to the database, to keep the last seen contents of those fields. *sequence number* and *timestamp* are also saved. In addition, a stepping factor is saved, which calculated by

$$\frac{ts_{this} - ts_{last}}{seq_{this} - seq_{last}}$$

where ts_{this} denotes the timestamp of the current packet, ts_{last} the timestamp of the last previously correctly received packet, and seq_{this} and seq_{last} the sequence numbers, respectively.[34] Finally, unpredictably dynamic field contents are not saved, because they do not aid in reconstruction of corrupted packets.

Whenever a corrupt packet arrives, the *predictor* unit first tries to look up the learned information for the SSRC received in the RTP packet. This lookup can either produce a result immediately, if the SSRC was uncorrupted. If no perfect match is found, a best match will be found by using the exact same scheme as the one used by Refector to find matching IP addresses and ports: the SSRC with the lowest Hamming distance to the received value is used. Once the entry is found, the predictably dynamic fields will be repaired by the predictor. Mostly static fields, such as CC and CSRCs, are simply replaced in the packet. The sequence number and timestamp are repaired by taking the last seen sequence number, incrementing it by 1, and writing it into the received corrupted packet. Similarly, the timestamp is repaired by taking the last seen timestamp, incrementing it by the stepping factor, and writing it into the packet. Finally, the fact that a packet was repaired towards the values of this stream is saved by incrementing an internal stream-specific counter. Thus, if another corrupted packet arrives for the same stream, the sequence number for that packet is repaired by incrementing the saved sequence number value by 2, and the timestamp by incrementing the saved timestamp by incrementing it by twice the stepping factor. This continues until a correct packet is received for a stream, at which point that packet's sequence number and timestamp will be saved, and the counter reset to 0. Thus, static and predictably dynamic fields can be repaired.

The main problem with this approach is that is disregards the possibility of losing RTP packets completely: in this case, sequence number and timestamp are repaired incorrectly. If a correct packet with a sequence number of n is received, the packet with sequence number $n + 1$ is completely lost, and then a corrupted packet that was sent with sequence number $n + 2$ is received, the number will incorrectly be repaired to $n + 1$. While this sounds like a large problem, note that the resulting desynchronization is rather light: first, sequence number and time stamp are kept in lock-step, so no incoherence occurs here; second, as soon as a correct packet is received, the predictor resynchronizes itself automatically; and third, while a lost packet will lead to incorrect repairs of every corrupted packet that is received afterwards until a correct packet is received again, the *relative* synchronization within this spurt of incorrectly repaired packet is kept: only at the point of packet loss and subsequent reception of a correct packet, slight de- and resynchronizations occur.

[34] The stepping factor is, of course, only calculated if valid values for ts_{last} and seq_{last} exist, that is, not for the very first received packet of a stream.

3.4.4 Implementation for RTP in libortp

To test the feasibility and performance of our concept, we implemented it in an RTP implementation. We decided to use oRTP 0.16.5 [ortp], a dynamic library written in C. This library is, for example, used by the linphone [linphone] project. Our changes are small and self-contained, and so should be easily portable into other RTP libraries. The only interface of our learner–predictor implementation with existing library code is the requirement to have access to the RTP headers early in the processing, to feed the learner and potentially repair the packet via the predictor. All the necessary changes are contained to the `rtpsession_inet.c` file. The learner and predictor both hook into the `rtp_session_rtp_recv` function, the wrapper function for the library's main parsing routines. The database that contains the per-stream data (e.g., SSRC, payload type, last seen sequence number and timestamp, stepping factor, counter of incorrect packets since last packet to calculate the correct sequence number and timestamp) as a linked list of C `structs`.

The second part that requires integration with existing code concerns the interface with the operating system's underlying socket architecture to differentiate between correct and corrupt packets, a problem we already mentioned in the above section, but deferred discussing until now. When we focused on IP and UDP earlier, the difference was easy to recognize because both protocols had a checksum that allowed to immediately see whether a corruption had occurred. RTP, however, does not have any checksum: as application-layer protocol, it relies on the integrity checks implemented by lower layers and refrains from adding another superfluous integrity check itself. This means that RTP itself does not have an easy way to recognize whether a packet was corrupted in transit. However, the `MSG_HASERRORS` flag that we introduced in Section 3.2.3 and that signals whether a packet potentially contains errors when using the `recvmsg` system call solves this exact problem. We simply need to check its value to decide whether to run the learner or the predictor on a received packet.

3.4.5 Evaluation

As in the first part of this chapter when we evaluated the performance of Refector for stateless protocols, we now investigate the behavior of our repair techniques for stateful protocols. Before we started this evaluation step, we considered the lessons from the first evaluation setup.

Evaluating Refector as presented in Section 3.2.4, within a kernel implementation, using commodity WLAN adapters and a testbed of real-world machines, provided irrefutable evidence for the feasibility of the approach. It also showed that there were no hidden effects that rendered such a system impossible in practice, due to unforeseen and unaccounted-for side effects from other protocols, operating system behavior, or hardware and channel effects.

On the other hand, one large problem that we faced during the evaluation was that reproducibility of experiments was nearly impossible. Only by careful setups,

by interleaving Refector and UDP-Lite streams, and by very long measuring series were we able to produce results that provided credibility and soundness. Even then, environmental effects made it very hard to achieve reproducible results over the complete investigation range. For example, effects from underlying layers, such as MAC errors, influenced the results, while being uncontrollable themselves. These MAC effects also watered down the quantitative results and limited the insight into how much of an improvement Refector produced for those protocols that it supported.

For this next step in developing Refector, we therefore decided to use a simulative approach to have more control over environmental factors, and to focus our investigation onto those parts of the protocol stack that we "refectorize".

3.4.5.1 Experimental Setup

Our focus in this evaluation is RTP, represented by the oRTP library enhanced with our heuristic header error repair techniques. Hence, we decided to disregard all factors from lower layers. This can be done reasonably easily due to the fact that oRTP is a self-contained implementation of RTP. All we need is a small wrapper program for sending that triggers creation of RTP packets by the library, and a second such wrapper that receives these packets. The communication between the two can then be done via Unix domain sockets. These sockets allow inter-process communication by sending and receiving data in the same way as Internet socket. Changing from one socket type to another only requires trivial changes in the oRTP library that have no effects on the rest of the code base and its behavior. To do so, we only needed to change the socket types from Internet sockets to Unix Domain sockets by changing the relevant lines in the `create_and_bind` and `rtp_session_set_remote_addr_full` functions in the `rtpsession_inet.c` file. Communication between the two oRTP instances afterwards behaves exactly as if an Internet connection had been set up before the start of the communication.

This, however, only produces an error-free, direct channel between the two instances. Such communication is uninteresting from the view point of error tolerance, since there are no errors to tolerate and repair. Our first idea was to use netem [netem], a framework for simple network emulation tasks that is a standard tool in Linux distributions, to inject bit errors into the communication. However, netem is not designed for this specific use case. Even though it has the functionality to corrupt packets with a certain error rate, an error is only provoked by flipping a single bit in the packet, invalidating the checksum. However, since we are interested in packets that are potentially strongly corrupted, we need to be able to flip more than a single bit.

We therefore created a simple application to corrupt the packets ourselves, which was then inserted between the sender and receiver: instead of the sender directly connecting to the receiver via a Unix domain socket, it instead connected to the corruptor, which read data, applied bit errors to the complete packet with a certain

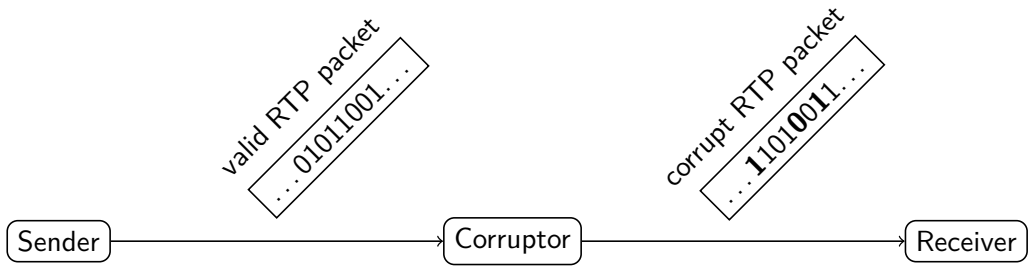


Figure 3.17 Diagram visualizing the experimental evaluation setup. Sender and receiver are two instances of the oRTP library, with the receiver extended with our heuristic header error recovery mechanisms. They communicate with each other via Unix domain sockets. In between the two resides the corruptor, which passes through data from the sender to the receiver and can be instructed to introduce various amounts of errors into the RTP packets.

per-bit probability, and then sent it to the receiver. The setup is depicted in Figure 3.17. If the corruptor introduced at least one error, it signaled so to the receiver, emulating a checksum failure on a lower layer and the resulting `MSG_HASERRORS` flag.

The corruption followed a simple Bernoulli process, modeling a binary symmetric channel: we set a BER for the experiment, and each bit would then be flipped with that probability. The effect is that the errors are independently distributed from each other within the packet. This is a somewhat dangerous assumption; in fact, real-world measurements [WKHW02, HJL+09, HJL+12] have shown that errors in WLAN are generally not distributed independently. We used two approaches to cope with this problem: first, we investigated the whole BER range from 0 to 0.5, that is, completely randomized output. As the error rate increases, the length of error bursts also increases. Hence, while a nominal BER might lead to longer average error bursts in real-life WLAN deployments than it does in our simulation, we do not neglect the investigation of the effect of error bursts on our results. Second, we also investigated a Markov model that simulated error bursts; we will show some results from the measurements and see that, in fact, the difference to a simple Bernoulli process, while significant (depending on the parameterization), is mostly irrelevant for realistic scenarios.

Note that in our setup, there is no modeling of packet loss or reordering. This means that the phenomenon of incorrect repairs of sequence number and timestamp, as described in Section 3.4.3, is a more rare occurrence. However, it is not completely removed from our evaluation due to "crosstalk" effects in the case of misattribution, which we will describe when presenting the respective results in Section 3.4.5.3.

Without packet loss, misattribution is the main metric that we investigate in this evaluation, more so than in Section 3.2.4. On the other hand, we will more closely investigate effects of misattribution, and how to further curb its detrimental effects, even under extreme error rates.

Our test included setups of 2, 3, and 4 concurrent RTP streams (over the same RTP connection). While a single-stream connection is arguably the most common use case of RTP, especially when using VoIP, it is also a very boring setup: since we do not model packet loss, and misattribution cannot occur, the result would always be

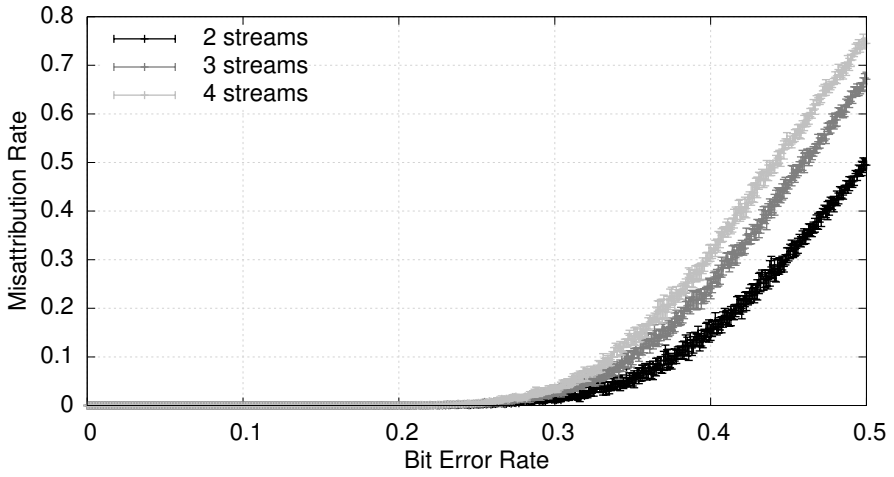


Figure 3.18 Misattribution rates for two, three, and four concurrent streams. For each investigated bit error rate (increments of 0.001 from 0 to 0.5), the mean and 95% confidence intervals are shown. Even at a bit error rate of 20%, we witnessed virtually no misattributions.

a stream of packets with perfectly repaired static and predictably dynamic fields, regardless of error rate. Each test run comprised 10 0000 packets for each stream in each experiment. For each data point, we executed 10 test runs unless otherwise noted. (Error bars denote 95% confidence intervals.) This is especially important since oRTP follows the advice of the RFC [SCFJ03] and randomizes its SSRCs, so Hamming distances between the SSRCs can differ strongly from run to run. To quickly reach a steady-state in which the learner–predictor database was seeded with content, the first two packets of each stream were not corrupted.[35] This was especially important at very high BERs: if correct packets are received only rarely, then results from run to run depend on random chance to a degree that conceal the actual results: if the first two correct packets are only received after dozens or even hundreds of corrupt packets, then all preceding packets will have been mis-repaired because of lack of information in the database.

3.4.5.2 Misattribution

As a first step in our evaluation, we investigated the misattribution that occurred with two, three, or four concurrent RTP streams. Figure 3.18 shows the results from our experiments, in which we investigated BERs between 0 and 0.5, in increments of 0.001.

The results show that misattribution is virtually nonexistent until BERs exceed 20%.[36] At such high BERs, the payload will suffer such corruption that typically,

[35] This can be considered to model a scenario in which the initial setup of the RTP connection was successful and link quality degradation happened subsequently. Note that any application could also enforce this behavior itself in a real-world scenario by only setting the `SO_BROKENOK` header option after receiver several packets, guaranteeing that these packets are correct.

[36] Also note that at 50% BER, the misattribution rate for n streams approaches $\frac{n-1}{n}$, showing that attribution is completely random, as can be expected when the error rate effectively randomizes the header contents.

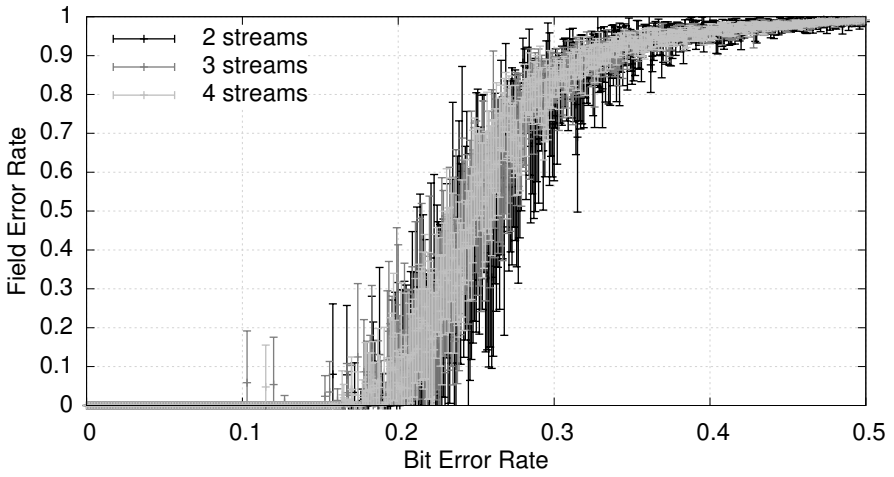


Figure 3.19 Fraction of header fields with errors after repair due to incorrect repairing. The high variance is due to error propagation: if a packet is misattributed, all following packets will suffer from incorrect repair of some header fields until a correct packet is received for each affected stream again. At high BERs, the rarity of correct packets leads to large and highly varying numbers of incorrect repairs.

no useful reception can be expected any more, even with voice codecs designed for error tolerance [NOCW07]. For practical scenarios, misattribution can therefore be considered a negligible problem. These results again show the practical feasibility of heuristic error recovery.

3.4.5.3 Field Errors

The previous result only considered misattribution of a packet to an incorrect RTP stream. However, even when packets are not misattributed, important fields can be repaired incorrectly. We therefore investigated how often field errors occurred, that is, how often a field was misrepaired and contained a different value after repair on the receivers side, compared to the original value with which it left the sender. The results are presented in Figure 3.19. Note that this only considered errors in static and predictably dynamic fields. Unpredictably dynamic fields are not considered in this situation due to the fact that they are not repaired and hence, errors in those fields do not provide any insight into the repair performance.

The much higher field error rate (compared to the misattribution rate) is mostly due to the effect of error propagation, coupled with long series of corrupted packets without any correct packets in between. For illustration, consider the following scenario: Two streams, A and B, have most recently sent the sequence numbers seq_A and seq_B, respectively, so that the next expected values are seq_A+1 and seq_B+1. If now a corrupted packet p_n that belongs to stream A is received, but misattributed to stream B, then its sequence number will be misrepaired to $seq_B + 1$ instead of $seq_A + 1$. If the next packet p_{n+1} is also corrupted, it will now invariably also suffer from misrepair, even if no misattribution occurred: if it belongs to A, its sequence number will be misrepaired to $seq_A + 1$ instead of the correct $seq_A + 2$; if it belongs

to B, its sequence number will be misrepaired to $seq_B + 2$ instead of the correct $seq_B + 1$. This error will propagate until both the stream p_n correctly belonged to and the stream it was misattributed to receive a correct packet to resynchronize the sequence number to the correct values. At high bit errors rates, correct packets without any errors are rare, so it can take many packets for this resynchronization to occur. This same behavior leads to an error propagation in the timestamp field.

While this behavior at first seems to be a large problem, remember the explanation in Section 3.4.3. Error propagation overestimates its effect on the stream, since disruptions in the "natural" progression of sequence number and time stamp only occur at the original point of desynchronization and at the point that resynchronization takes place. Furthermore, field errors are not a significant problem below 10–15% BER and rarely appear at these, already quite high, error rates.

The very large confidence intervals in Figure 3.19 are a result of this error propagation, combined with the fact that at high BERs, correct packets are rare. Hence, in each experiment, once an error occurred, the number of packets that suffered from error propagation until a correct packet was received varied strongly. This shows that our decision to always send the first two packets of each stream without any errors to instantly reach an initialized state in our database supported the lucidity of our result presentation: without such a measure, the results of our misattribution rates in Figure 3.18 would show confidence intervals that were similarly, if not quite as large.

3.4.5.4 Reduction of Misattribution

While Section 3.4.5.2 already showed very satisfactory results, we still wanted to consider measures to reduce misattribution even further. Our motivation was that, even in streams that show BERs low enough to be useful, and in which misattribution generally does not occur, short effects that lead to a sudden drop in reception quality (e.g., strong, short fading effects, interference) could momentarily produce very high BERs for a single or a few packets, or even only parts of a packet, for example, the RTP header, without degrading lower-level headers and streaming payload. In such a situation, an otherwise perfectly working heuristic recovery could produce undesirable misattribution.

We therefore investigated when such misattribution tended to happen, and whether there were signs that could be extracted from received packets to suggest a potential misattribution. In our search, we limited ourselves to such information that was available to the receiving oRTP library, so that the results could be directly implemented to produce a misattribution avoidance scheme.

One such information is the minimum Hamming distance, that is, the Hamming distance of the packet's header contents to the expected values of the closest match. A low Hamming distance means a near-perfect match, while a high Hamming distance signifies large difference between the expected and the witnessed results. It therefore stands to reason that, if the Hamming distance to the chosen match is high, then the risk of misattribution is larger, because it is relatively unlikely for random bit errors to occur in a way that changes a header to perfectly match a different one.

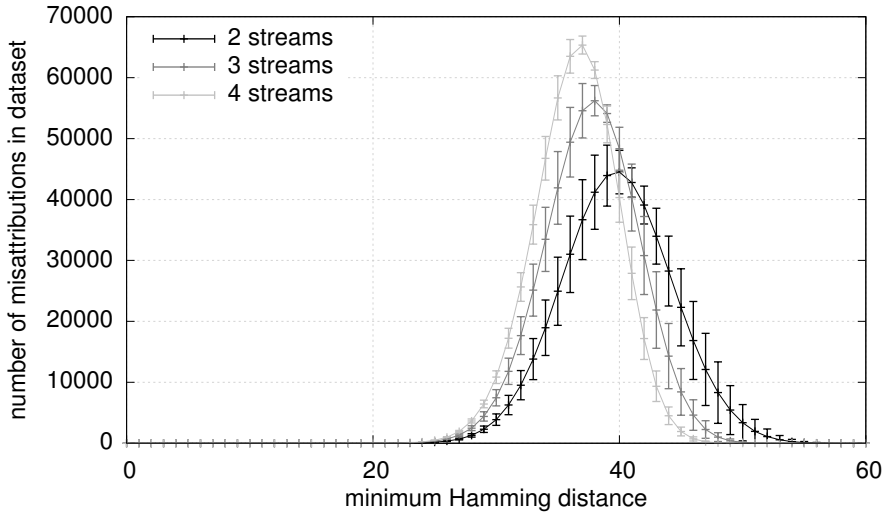


Figure 3.20 Misattribution occurs due to high bit error rates. A side effect of high BERs is that even the best match often shows a high Hamming distance to its expected header values. Hamming distance can therefore be used as an estimator for risk of misattribution. At low Hamming distances, no misattribution occurs. Extremely high Hamming distances are uncommon because it is more likely another match with lower distance exists instead.

To investigate the relationship between minimum Hamming distance and misattribution, we checked the minimum Hamming distance every time a misattribution had occurred in our data sets. Figure 3.20 shows the relationship between the two metrics. It can be seen that relating the two metrics in such a manner produces a Gaussian distribution. Misattributions virtually never occur at Hamming distances below 20. They peak at a value between 36 and 40, depending on the number of concurrent streams, then become rarer again.[37] The fact that there is a peak misattribution rate at certain minimum Hamming distances is due to the fact that two factors influence these results: On the one hand, higher Hamming distances are a result of higher BER, which also produces higher misattribution rates. On the other hand, the numbers given in the graph are absolute: very high Hamming distances are less likely to occur, even at high bit error rates, because there are always several matches to choose from (two, three, or four, depending on the number of streams): at some point, the maximum distance cannot be increased because the header otherwise will start to match a different stream's expected values more closely. This is also why, as the number of streams increases, the peak of the curve is reached at lower Hamming distances, and the curve is steeper.

As a result of these insights, we decided to extend our oRTP implementation with a *cutoff* value. If the minimum Hamming distance ended up being larger than this cutoff, the packet would be dropped instead of assigned to the stream. We then reran our experiments with the same setup as previously, but with different cutoffs. The results are presented in Figure 3.21. It shows the misattribution rate at different cutoffs, for the 4-stream scenario. We focused on this one because it showed the highest misattribution rate in our original scenario. The results show that we

[37]For reference, a standard RTP header has 96 bits.

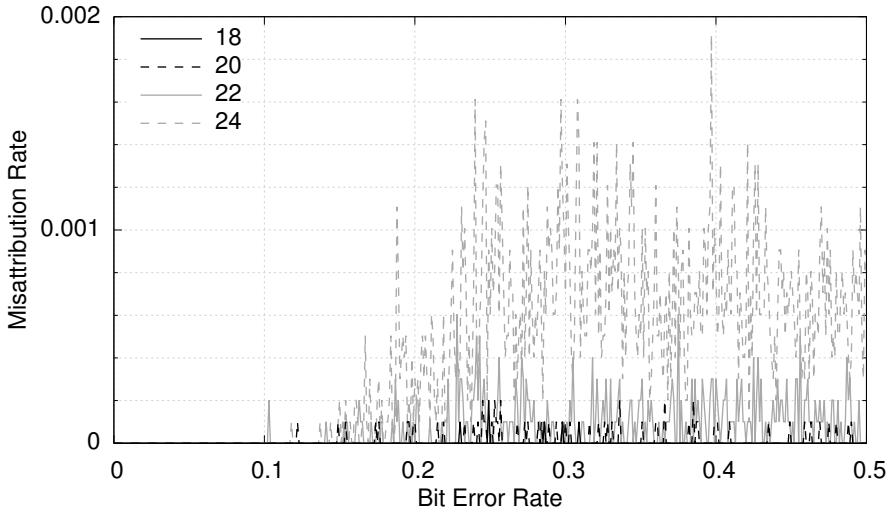


Figure 3.21 Misattribution rate in a 4-stream scenario for different Hamming distance cutoffs. By setting a cutoff and discarding all packets whose best match has a higher distance, misattribution can be reduced dramatically. Even at a cutoff value of 24 (one quarter of the 96 bits of the standard RTP header), misattribution stays below 0.2%, regardless of BER, and becomes exceedingly rare at stricter cutoffs.

can virtually prevent any misattributions by settings a relatively strict cutoff at a minimum Hamming distance of 18, over the full BER range from 0 to 0.5. Even at more lenient cutoffs, the misattribution rate becomes small and manageable (less than 0.0002 at 20, less than 0.0005 at 22, less than 0.002 at 24). Thus, a simple change that is easily to implement into the RTP library can effectively prevent misattributions even under extremely high BERs.

However, this prevention comes at a price, since we now discard packets. Some of these discarded packets might in fact have reached the correct application had they not been cut off, and therefore are lost even though they would have been useful. At high BERs, the number of discarded packets becomes significant. To fairly compare the original (non-cutoff) and cutoff performance, we need to investigate the packet drop rate.

Again, we focus on the 4-stream scenario, and present in Figure 3.22 the packet drop rates for the previously chosen cutoffs of 18, 20, 22, and 24. Indeed, the drop rates are sizable. In fact, drop rates surpass misattribution rates for most BER values (cf. Figure 3.18). However, for the most relevant BERs (below 10%), drop rates are low. For the strictest cutoff value of 18, drops start to occur at about 4% BER, and reach 10% drop rate at 10% BER. The less strict cutoff of 20 only has a a drop rate of about 1% at 10% BER, and the most lenient cutoff of 24 rarely drops any packets until the BER is in excess of 15%.

It is hard to exactly derive a one-size-fits-all suggestion. However, a cutoff of 20 seems to perform reasonably well, and will only perform better in scenarios with fewer concurrent streams. It produces low packet drop rates, while preventing misattributions effectively.

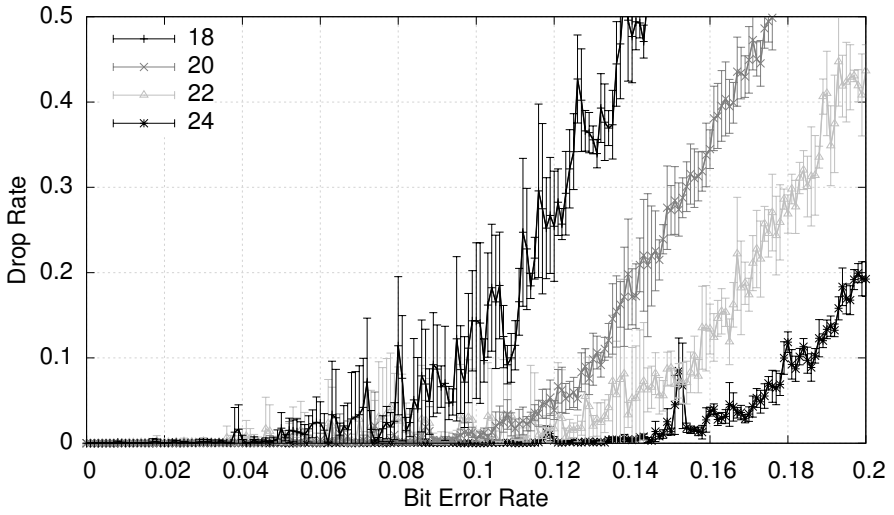


Figure 3.22 Drop rate in a 4-stream scenario for different Hamming distance cutoffs. When using cutoffs, misattribution rate is reduced, but packets are outright dropped instead. However, even at a strict cutoff rate that prevents virtually all misattribution (see Figure 8), drop rates stay minimal until BER is in excess of 5%.

3.4.5.5 Markov Chain Model Performance

We promised in Section 3.4.5.1 to discuss the differences between using a Bernoulli process and a Markov model on the misattribution rate. To understand the differences, let us first quickly restate the concept of Markov chains and some of their properties. We will not go into the details of formal definitions via random variables, but will instead keep in mind our application scenario.

A Markov model, often visualized as a Markov chain, is a probabilistic process that comprises several states, with each state having certain transition probabilities to other states. Such a model is assumed to be memoryless: the transition probabilities only depend on the current state the system is in, not on any previous states. In our scenario, we used a simple Markov chain in our corruptor (cf. Figure 3.17) to model burst errors. This chain only comprised two states, a "non-flip" (do not change bit) and a "flip" (change, i.e., corrupt, bit) state. For each bit of the input stream, a random number was created to model state transition. Compared to the simple Bernoulli scenario with a single bit error probability p that was completely stateless, our Markov model had two error probabilities, p_1 and p_2.

The model is visualized in Figure 3.23. Note that in our case, p_1 models a transition from q_1 to q_2, while p_2 models the chains staying in state q_2. This is to use those two as error probabilities: p_1 models the probability to produce an error after a no-error event, while p_2 models a subsequent bit error after an error had already occurred, and therefore an error burst. At $p_1 = p_2$, the model becomes the previous Bernoulli model.

Since now we had two scenario variables, our number of experiments would explode if we were to investigate them all in the same detail as before. We therefore decided to only focus on a few values for p_2, modeling different tendencies to produces burst

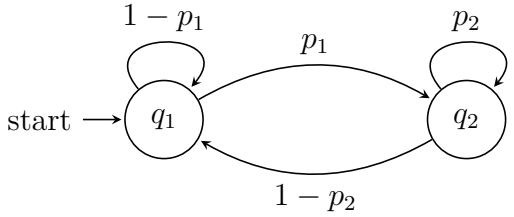


Figure 3.23 A simple two-state Markov Chain. From the starting state q_1 (in our application: the error-free states that does not flip a bit), the model can move to state q_2 (in our application: the error state that flips bits) with the probability p_1. While in state q_2, the model will stay in that state with the probability p_2.

errors, while letting p_1 run from 0 to 0.5. In the following, we will present results for $p_2 = 0.4, 0.6, 0.8$. These model different tendencies to produce burst errors. We can calculate the expected error burst lengths by taking the expected value of state transitions until changing from q_2 back to q_1, and adding 1 (because changing into state q_2 already corrupted one bit). The expected length of staying in a state is the reciprocal value of the probability of transitioning out, so the expected error burst length is $1 + \frac{1}{1-p_2}$. To calculate the effective BER, we need to calculate the steady-state probability of being in state q_2. We can do this by taking the transition matrix of our Markov chain,

$$\begin{pmatrix} 1-p_1 & p_1 \\ 1-p_2 & p2 \end{pmatrix}$$

and solving the following linear equation, where (x_1, x_2) are the probabilities of the Markov chain's stationary (steady-state) distribution:

$$\begin{aligned} \begin{pmatrix} x_1 & x_2 \end{pmatrix} \begin{pmatrix} 1-p_1 & p_1 \\ 1-p_2 & p_2 \end{pmatrix} &= \begin{pmatrix} x_1 & x_2 \end{pmatrix} \\ \Leftrightarrow \quad \begin{pmatrix} x_1(1-p_1) + x_2(1-p_2) & x_1 p_1 + x_2 p_2 \end{pmatrix} &= \begin{pmatrix} x_1 & x_2 \end{pmatrix} \end{aligned}$$

Displayed as a linear system, with the added constraint that the probabilities must add up to 1, so $x_1 + x_2 = 1$:

$$\begin{aligned} x_1 - p_1 x_1 + x_2 - p_2 x_2 &= x_1 \\ p_1 x_1 + p_2 x_2 &= x_2 \\ x_1 + x_2 &= 1 \\ \\ \Rightarrow \qquad x_1 &= \tfrac{(1-p_2)x_2}{p_1} \\ \Rightarrow \qquad x_2 &= 1 - \tfrac{(1-p_2)x_2}{p_1} \\ \Rightarrow \qquad p_1 x_2 &= p_1 - x_2 + p_2 x_2 \\ \Rightarrow \qquad (1 + p_1 - p2) x_2 &= p_1 \\ \Rightarrow \qquad x_2 &= \tfrac{p_1}{1+p_1-p_2} \end{aligned}$$

The effective BER is therefore $\frac{p_1}{1+p_1-p_2}$. Table 3.2 shows the expected burst error length and effective BER for the three chosen values for p_2.

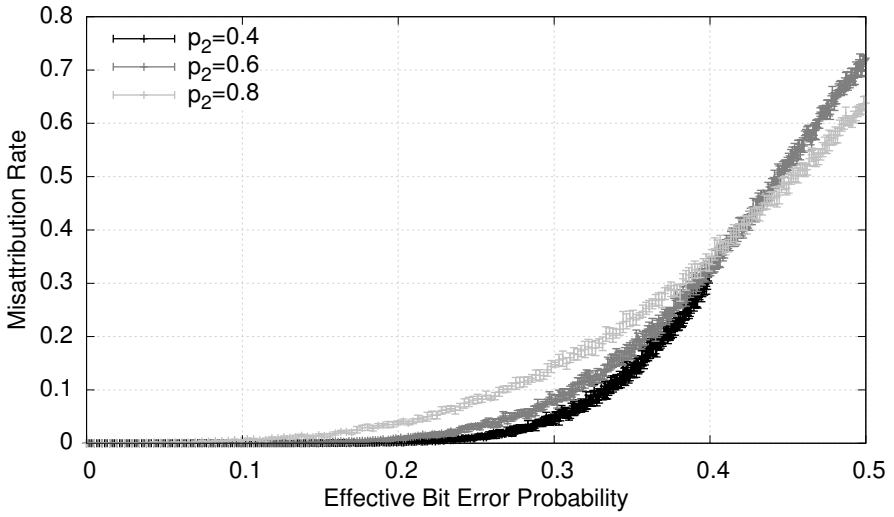


Figure 3.24 Misattribution rates for three different values of p_2. The effective BER is calculated from the values given in Table 3.2. While at large susceptibility for burst errors, the misattribution rate is noticeably higher than in the Bernoulli model (cf. Figure 3.18), the results are still comparable. Again, we only witnessed misattributions at BERs that are high enough to already cause serious problems in media decoding.

Figure 3.24 shows the misattribution rates for the three presented values of p_2. Again, we focus on the most challenging 4-stream scenario. The effective BER is calculated as given in Table 3.2, and the results are normalized to this BER for each data set. This explains the somewhat counter-intuitive result that, at high effective BERs, the misattribution rate for $p_2 = 0.6$ is higher than for $p_2 = 0.8$: at these high error rates, the normalization produces artifacts. For example, at an effective BER of 0.5, $p_1 = 0.2$ if $p_2 = 0.6$, but $p_1 = 0.1$ if $p_2 = 0.8$. As the chance to produce long error bursts increases, the tendency to produce long error-free bursts also increases to produce the desired effective BER.

When focusing on the more relevant low-BER areas of the graph, it can be seen that, while misattributions occur at a lower effective BER than in the Bernoulli case (cf. Figure 3.18 on page 90), up to 10% BER, they are still exceedingly rare. These results suggest that while a Bernoulli distribution slightly underestimates misattribution rates if long burst errors are the norm, the results are still comparable.

p_2	Expected burst error length	Effective BER
0.4	$2.6\overline{6}$	$\frac{p_1}{0.6+p_1}$
0.6	3.5	$\frac{p_1}{0.4+p_1}$
0.8	6	$\frac{p_1}{0.2+p_1}$

Table 3.2 Expected error burst length and effective BER (depending on p_1) for specific values of p_2.

3.4.6 Summary

In this section, we showed that it is possible to design header recovery techniques that are not only applicable to static header fields for which values never change over the course of a connection, but also for fields that are changing, but in a predictable fashion.

Our application of these concepts to RTP allowed us to create recovery techniques that produce correct assignment of packets to RTP streams, even under very high error rates. Depending on the configuration, bit error rates up to 20%–25% BER do not produce any misattributions. If misattributions should reliably be prevented even under extreme conditions, then a cutoff value can be introduced to minimize or completely prevent misattributions, regardless of error rates, at the cost of an increased packet drop rate. In our results, we could completely prevent misattributions and still assign virtually all packets correctly up to BERs of 5–7%.

Finally, we discussed the differences between a simple Bernoulli and a Markov error model, and showed that, while there are differences between the two, differences do not become relevant until BERs become very high, a result that motivates the use of a similar Bernoulli-model driven setup in the next chapter.

3.5 Summary and Discussion

In this chapter, we proposed several techniques to heuristically recover from errors in headers of communication protocols. We showed that, since an end host always knows which connections are open and how headers for those connections should look like, can match incoming corrupted headers against those expected values to find the most likely match for the packet, and that the Hamming distance is a simple and effective metric to do this matching.

We investigated headers of standard protocols and identified significant areas of those headers as irrelevant for recovery on end hosts, allowing our recovery techniques to ignore potential errors in those fields. For other fields, we proposed techniques to repair them, either by matching them against expected values, or by predicting future values from past ones. Additionally, we investigated the possibilities to choose header fields in a way that increases robustness without changing the protocols themselves, keeping compatibility.

In our evaluation, we showed that heuristic header error recovery is an effective solution. Even compared to previous approaches such as UDP-Lite, our solutions produce significant performance improvements, recovering more erroneous packets. We also showed that the specific problem of our approach, misattribution, can be kept in check, effectively preventing it except for very rare occurrences. Finally, in addition to general performance metrics, a specific use case employing audio decoding showed that heuristic header error recovery indeed can improve the perceived quality significantly.

To conclude this chapter, we will discuss some loose ends and concepts that we did not implement or investigate in detail, but that could form the basis of future extensions of our work.

Selective NoAck Signaling

In Section 3.2.3, we explained how to facilitate parallel error-tolerant traffic without ACKs and error-sensitive (legacy) traffic with ACKs. To this end, we used the preexisting mapping from IPv4 ToS fields onto ACs, and added logic to send traffic from some ACs without ACKs.

This approach has one significant drawback, in that the ToS field needs to be set accordingly, which is under the sender's control. Therefore, this setup, while working, goes counter to our design goal of easy and incremental deployment by only introducing local changes at the receiver's side. Even worse, some core routers may reset the ToS field to 0.

Therefore, we suggest a better suited approach. Protocols that manage aspects of the traffic between the AP and the STAs in a WLAN network have existed for a long time. Some well-known examples are the Universal Plug and Play (UPnP) protocol suite [ISO11] which allows network discovery and setting up functionality, and more specialized approaches such as the NAT Port Mapping Protocol (NAT-PMP) [CK13] or the Port Control Protocol (PCP) [WCB+13], which allow setting up Network Address Translation (NAT) traversal from a local WLAN to the Internet. With these protocols, a client (the STA) can request from the server (the AP) to set up a NAT traversal for traffic inbound to a certain port. In the same spirit, one of these protocols could be extended (or a similar one created) that allows the STA to signal to the AP that traffic inbound to a certain port or address/port combination should be sent over the wireless link without ACKs.

This solution relieves the sender from any need to participate in our recovery scheme, reinstating our requirement to be independent from the sender and have a receiver-only solution. Only the AP in the receiver's network has to be changed, but we consider this requirement to be in line with our envisioned use case of private users who have full control over their AP.

Extensions for RTP Recovery

We consider two extensions of Refector recovery for RTP. The first considers the Hamming cutoffs as introduced in Section 3.4.5.4. We already noted that the number of concurrent RTP streams had an influence on the Hamming distance distribution with respect to misattributions. Eventually, we suggested a cutoff at a Hamming distance of 20. From our results, this seems to produce a good tradeoff between minimal misattribution and high recovery rates. However, since the number of streams has an influence, it might be beneficial to use a dynamic cutoff scheme, in which the cutoff is chosen by the number of concurrent connections (stricter cutoffs at higher number of concurrent connections). To further increase the detail of information for the dynamic cutoff choices, it might also be beneficial to (additionally

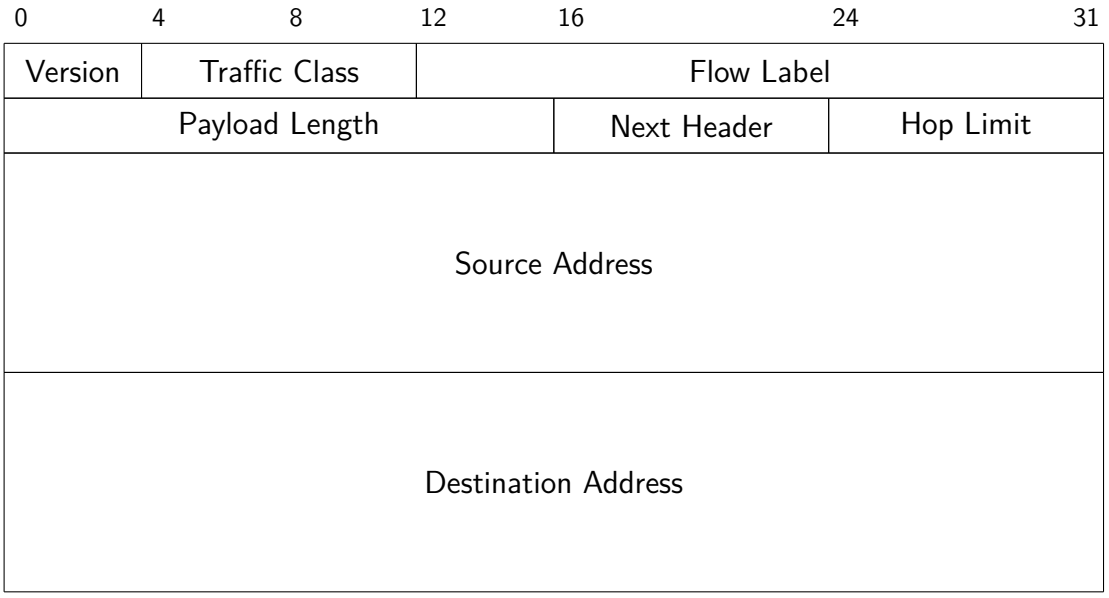


Figure 3.25 Layout of the IPv6 header (without header options).

or instead) investigate the Hamming distances between significant fields (first and foremost the SSRC) between concurrent connections, and choose the cutoff accordingly: the higher the minimum Hamming distance between fields, the more lenient the cutoff value.

This leads to the second possible extension. Just as we created a system for IP to produce high Hamming distances between ports, a system to produce high Hamming distances between SSRCs of concurrent streams within one RTP connection would be highly beneficial, especially since the number of concurrent streams within one RTP connection is typically very small, while the SSRC header field is even larger than the port field. We did not implement this because the SSRC is chosen by the sender, not the receiver. To influence SSRC choice would therefore conflict with our goal to have a solution that is independent of the sender and easily deployable. However, adding such a scheme whenever recovery techniques are implemented in additional RTP libraries might nevertheless be beneficial. The changes do not change the protocol's communication behavior and can therefore be rolled out gradually. While the user of such an extended library would not benefit from these changes themselves, their communication partner potentially might; as these high-distance SSRC extensions were to be gradually implemented into RTP libraries, the users would start to benefit whenever their communication partner also used such an extension.

Refector for IPv6

While IPv6 deployment is still a topic of discussion more than 15 years after its original standardization, we expect that eventually, IPv6 will supplant IPv4 in many areas and use cases. With respect to future developments, it therefore is beneficial to also consider Refector for IPv6. While we did not implement and test solutions for IPv6, we will give some suggestions on how to recover from IPv6 header errors.

The layout of an IPv6 header is given in Figure 3.25. Two changes compared to IPv4 are apparent. First, many fields that we classified as don't-care, such as the header checksum and the fragmentation fields, have been removed. The reasoning for this by the designers of IPv6 is similar to our reasoning for classifying those fields as don't-care: they either provide support for features that are unnecessary because they are already provided by other layers or not used any more. Second, the IP addresses are much larger, now forming the by far largest part of the header, and are the main contributor to the fact that the header is now larger than before, despite the fact that some fields were removed without any replacement.

The larger header suggests that header error recovery for IPv6 is even more beneficial than for IPv4, because the risk of errors increases with size. On the other hand, we categorized addresses as vital fields, and thus the larger addresses are problematic with respect to error tolerance.

However, we can leverage a side effect of how IPv6 is deployed in practice. Due to the massive increase in the number of available addresses, they are handed out much more freely than in IPv4. Even end users often receive not only a single IP from their Internet Service Provider, but a subnet of addresses. This allows us to use "coded" IP addresses in a similar way to destination ports (cf. Section 3.2.2.2), since an end host can be addressable via more than one IP address. Within the ISP-provided subnet of IP addresses, we can then choose IP addresses with large Hamming distances to identify different connections. Note that in this case, it is sufficient to use different IP addresses for error-tolerant streams only, and have all error-sensitive connections use the original IP address. Since there is no misattribution possible towards error-sensitive connections, there is no need to further distinguish between different such connections; only the fact that the packet belongs to *some* error-sensitive connection is important, because an corrupted packet identified as belonging to any error-sensitive connection is discarded in any case. Only having to use one additional IP address per error-tolerant connection significantly reduces the number of addresses needed, in turn increasing the robustness by using more bits of the subnet range as coding. Setting up new IP addresses for error-tolerant connections requires support in the network stack of the receiver, so that the OS will assign connections from error-tolerant applications to unused IP addresses from the available range. Support by the AP is also beneficial, because, if the network comprises several STAs that want to use error-tolerant communication, the AP can act as the central entity that ensures that all addresses used in the network are assigned according to rules that maximize Hamming distances. Such support can be implemented in similar ways to those described above for NoAck signaling.

In fact, such a system could even be deployed to great use in any NATed network (for both IPv4 and IPv6). By using, for example, the private 10.0.0.0/8 network, 24 of the 32 bits in an IPv4 address can be used to create coded addresses with large Hamming distances. We expect this approach to further reduce the risk of misattributions and increase the overall robustness of our recovery mechanisms.

4

[T]here are known knowns; there are things we know we know. We also know there are known unknowns; that is to say we know there are some things we do not know. But there are also unknown unknowns—there are things we do not know we don't know.

—Donald Rumsfeld

Protocol-Independent Heuristic Header Error Repair

In the previous chapter, we presented Refector, a system to heuristically recover from errors in protocol headers. We showed that the approach is feasible and can produce large gains over alternative solutions, and that its main downside, the risk to misassign a packet to the wrong application, can be prevented effectively.

However, there is one disadvantage of Refector as presented in the previous chapter: for heuristic recovery, it is necessary to investigate the protocols that are to be supported in a very detailed fashion.

In this chapter, we will present a solution that fundamentally expands the concepts of Refector. Instead of being specific to certain protocols, it can recover from errors independent of the protocols being used. To do so, it recognizes which connection a packet belongs to, even if headers are corrupted, and without any knowledge about the semantics of those headers.

We will first introduce our problem space and challenges, motivate our approach, and propose design goals in Section 4.1. In Section 4.2, we will then present the algorithmic design that stems from these goals. We will give some insight into the implementation of the algorithm in Section 4.3. In Section 4.4, we will show evaluation results that underline that our solution is both effective in classifying packets correctly – recognizing which connection a packet belongs to – and efficient – doing so quickly, both in per-packet processing time as well as adapting to new connections. We will present observations regarding classification via packet size and inter-arrival time in Section 4.5, an approach that, while we ultimately abandoned it, provides valuable insights into the inherent problems of using such information. Finally, we conclude this chapter in Section 4.6.

4.1 Introduction and Motivation

As mentioned above, the motivation for the work presented in this chapter is that Refector's recovery techniques are always specific to certain protocols. This means that extending Refector for new protocols involves a considerable amount of manual labor: each protocol needs to be investigated by someone familiar with it. Of course, we learned general approaches and basic concepts from our Refector implementation for IP, UDP, and RTP, for example, the classification of header fields into different classes of importance. However, this does not relieve us from repeating these steps for every protocol we want to support. Extending Refector to new protocols thus requires specific knowledge about these protocols. This knowledge then needs to be transformed into an implementation. Until this point, those protocols cannot benefit from from heuristic header error recovery.

This idea led us to a new challenge: the design of an algorithm that can automatically apply heuristic header error recovery to protocols, *without knowing anything about those protocols*. At first, this task seems daunting. How could an algorithm learn the intricacies of protocols without any previous knowledge? This is compounded by the fact that such an algorithm should not take too much time producing a good understanding of the used protocols: if it takes dozens, hundreds, or even thousands of packets to properly identify how to introduce heuristic header error tolerance into a protocol, then the connection might be closed by the time it has come to that conclusion; or at the very least, the connection will not be able to benefit from error tolerance for a long time.

Nevertheless, just as in the case of Refector, where we started with the simple mail courier analogy to break the problem down to a manageable core problem, we start with a simple observation. We are especially interested in protocols that fulfill the job of multiplexing and demultiplexing several logical streams onto one connection, because this demultiplexing step is the source of harmful misattributions. Indeed, all protocols presented so far feature such a demultiplexing unit: IP and UDP (and IP/TCP in the same fashion) together contain IP addresses and port numbers to demultiplex connection that share a common physical line. RTP demultiplexes streams identified by SSRCs that share a common connection. Finally, MAC protocols, while not a focus of this dissertation, similarly identify which of the receivers that share a common channel a frame belongs to by a MAC address.

We observed that the information identifying which logical stream a packet belongs to is invariably static: MAC and IP addresses, port numbers, SSRCs. As such, they should be easily identifiable in packets. Of course, other fields can likewise be static in a packet (version number, unused fields, etc.). However, these will either be the same in all packets over all connections, in which case it is simply a question of designing the algorithm in a way so that static and important fields do not "drown" in a mass of static, but unimportant fields. Or these fields will not be the same over all connections, in which case, we can actually use those to identify the connection a packet belongs to in addition to the fields designed for that purpose.

From this basic insight, we can now see that it should be possible to design a machine-learning algorithm that identifies important fields (in the sense of Section 3.2.1 and

Figure 3.1, that is, fields important to the demultiplexing decision) without any outside information that is not contained in the packet headers themselves. However, such an algorithm needs to fulfill several requirements:

1. The algorithm must be effective. This means that it should identify packets correctly (in the terms of pattern recognition: have a high hit rate) with very low misattribution.

2. The algorithm must work on streaming data. It is expected to learn its pattern online, while data is being received, and to adapt the pattern it matches to continuously. Such so-called data stream mining algorithms [BBD+02,GZK05] form a specific sub-field of data mining algorithms.

3. The algorithm should have a cutoff threshold. As opposed to standard classification algorithms used in machine learning, which always assign the input to a class, the algorithm should drop packets that are too different from all existing classes, that is, expected header values for connections. As we saw when investigating RTP, such a cutoff threshold is beneficial to prevent undesirable misattribution rates at high bit error rates.

4. The algorithm must be fast to learn new characteristic patterns for newly opened connections. As said above, an algorithm that takes dozens or even hundreds of packets to produce a robust (in the sense of producing high recall rate) pattern will not be able to reach the desired performance.

5. The algorithm must be fast to classify incoming packets. Even a 802.11a/g connection produces several packets per millisecond at high speeds, and this number only increases for 802.11n or 802.11ac. If the algorithm takes more than a few microseconds to classify a packet, it is at risk of being overwhelmed; furthermore, it would produce a very noticeable computational overhead while receiving data.

6. The algorithm should not use floating point numbers and operations. This, in contrast to the previous requirements, is less of a design and more of a practical requirement. While floating point operations are slower than integer operations (which could interfere with requirements 4 and 5), the main reason is that this algorithm should be easily portable into an operating system kernel's network stack. In many kernels, among them the Linux kernel, the floating-point unit is reserved for user-space calculations [Tor04] for performance reasons.

These specific requirements mean that popular standard classification algorithms such as support vector machines [CV95] or ensemble methods such as AdaBoost [FS97] or random forests [Bre01] are unsuitable for this task. On the other hand, our original idea only requires us to identify static areas in header fields, and such an algorithm can be implemented to be very simple and fast.

4.2 Design

In the following, we will first discuss further considerations and preliminary investigations we conducted to come to an informed design decision. Afterwards, we will present the classification algorithm to heuristically identify and recover corrupt packet headers.

4.2.1 Design Considerations

One of the cornerstones of our approach is the assumption that static header fields can be easily recognized. To reach this goal, bits in a packet header need to either be static over the course of the connection, or near-random so that their values will change at least once within the first few packets. Conversely, bits that do change, but only rarely, might reduce the effectiveness of the approach, because it takes longer for them to be recognized, and thus the algorithm will take longer to produce effective patterns to match incoming corrupted packets against.

To investigate the question of how header bits behave, and for continuous testing of the algorithm during development, we created a data set of different types of Internet connections with diverse use cases and connection behaviors, among those ICMP pings, web surfing sessions on Wikipedia, several RTP-based online radios, as well as YouTube video streams via HTTP and RTP, capturing the received data. To create an error-free baseline data set, all data for these connections was transmitted via Ethernet. When required, we injected errors into the error-free stream by means of a corruptor, similar to the one described in Section 3.4.5.1.

On this data set, we investigated the randomness of bits over the course of a connection. As a representative example, we show the results from a 7269-packet YouTube video stream in Figure 4.1. For each bit position, the average value over the course of the connection is calculated. For example, if four packets were received, and the nth bit was 1 in the first packet and 0 in the others, the average value is 0.25. Thus, values equal to 0 or 1 denote static bits that did not change over the course of the connection. Conversely, bits with an average value close to 0.5 were almost completely random. The figure shows that almost all bits fall either into the "completely static" or "completely random" category. The small amounts of nearly-but-not-quite static fields were mostly due to incrementing fields such as sequence numbers: the high-order bits stayed at one value for a long time before changing eventually.

In this specific example, we truncated each packet to 46 bytes, the size of the smallest packet received for that connection. This truncation was done for two reasons: first, since even the smallest packets need to have headers, truncating to the smallest size focuses our result on the header portions of the packet. Second, not doing any truncation does not change the overall message of the graph, but makes it more cumbersome to read: since the payload content changes from packet to packet, payload bits are almost completely random. Not truncating the packet would lead to a very large value for the two center bins, and make the lowest and highest bin harder to discern. The resulting conclusion would still be the same: bits are almost always either static or highly random.

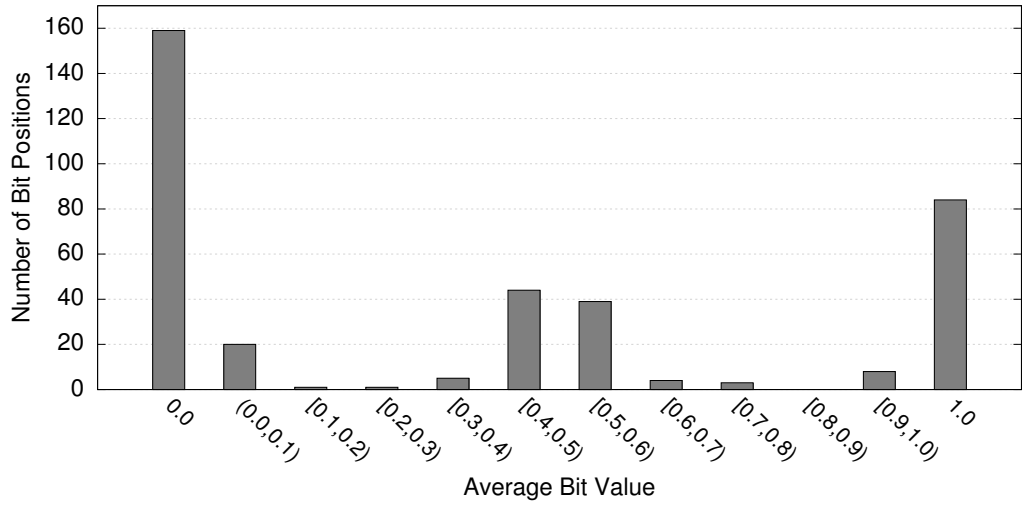


Figure 4.1 Average values for each bit position in the first 46 bytes of a 7269-packet YouTube stream. Bits are generally either completely static (0.0 or 1.0) or highly random (close to 0.5); only very few positions show different behaviors.

With our assumption substantiated by real-world numbers, we can now design an algorithm that recognizes static bits in connections and uses those to identify the correct connection for corrupted packets. Note that this does not yet give us any insight into whether choosing these bits will actually produce an *effective* classification: so far, we only assume so, based on expectations we have from results in the previous chapter. In the end, only a classification performance evaluation will answer this question.[38] For now, however, we will focus on designing an algorithm that is fast to run, fast to find static bits in new connections, and fast to adapt to changing conditions, that is, if a hitherto static bit changed its value; a problem that in data mining is known as *concept drift*.

4.2.2 Algorithmic Design

Similarly as in the solution designed for RTP (cf. Section 3.4.3), we designed the algorithm in two parts. This follows the observation that we can distinguish two classes of input for our algorithm: credible input from uncorrupted packets that is used to create the classes, and potentially non-credible input from corrupted packets that is classified against the classes created from uncorrupted packets. Just as before, we will term the two algorithm parts *learner* and *predictor*.

The task of the learner is to create classes, one for each open connection (that is, Internet socket); when a socket is closed, the respective class is removed. The defining characteristic of each class is the position and value of static bits, that is, bits that always had the same value in every packet belonging to that connection. This can easily be done by keeping two bit fields for each class, the *mask* and the *value*. The mask denotes which bits were static: a `1` denotes a static bit, while a `0`

[38] And indeed, we will present evaluation results in Section 4.4 that show exactly this effectiveness.

denotes a variable bit. The value contains the bit value of each bit. For static bits, this contains the value that was seen in every packet; for variable bits, the content is ignored.

For each incoming packet, the mask can now be updated with simple and very computationally efficient bitwise operations:

$$mask_{new} = (value \odot packet) \wedge mask_{old} \tag{4.1}$$

where $\odot$ denotes an XNOR operation (that is, a logical NOT applied after an XOR operation), and $\wedge$ denotes an AND operation. XNOR is effectively a comparison operation: if the compared bits are equal, it returns 1, and keeps the mask bit set if it is still set. Otherwise, it returns 0 and marks the respective bit as nonstatic in the mask.

The second field, the value, only needs to be initialized once, by copying the contents of the connection's very first packet into it, and afterwards not changing it at all: if a bit never changed, there is no need to update it; if it did, then the mask will mark it as unimportant, and it will be ignored.

This algorithm has several defining characteristics apart from its computational simplicity: it is also very space-efficient, because it does not keep a large backlog or window of data: all previous packets are aggregated into two bitfields of a static size n that is a parameter of the algorithm. If the packet is larger than this size, only the first n bytes will be taken into consideration; if the packet is smaller, it is padded for the sake of this calculation. The idea here is that we focus on header contents, so only considering a certain amount of bytes at the beginning of the packet makes sense. Choosing a very small size will make calculations faster, but potentially reduce accuracy, because only parts of the packet headers will be classified. Choosing a very large size will slow down calculations, and likely introduce large amounts of (near-random) data into the classifier. We settled on a size of 40 bytes, which, for example, covers a combination of IPv4 and TCP, or of IPv4, UDP, and RTP.

The algorithm furthermore relies on non-static bits changing regularly to achieve fast convergence to a characteristic pattern. This is another reason why the results presented in Figure 4.1 are so important: these suggest that this is the case. We will further investigate this point during the evaluation. Finally, concept drift is only a problem for the algorithm if bits that used to be random become static, a case that we discounted as unlikely in our design considerations. The opposite case, a static bit becoming random (a more likely case due to high-order bits in counting fields such as sequence numbers) does not introduce any problems: as soon as the bit changes for the first time, the algorithm will immediately note this.

For predicting, we created a scoring algorithm that likewise only uses simple bitwise operations. Whenever a corrupted packet arrives, a score towards each open connection is created,

$$score = \frac{\#((packet \oplus value) \wedge mask)}{\#mask} \tag{4.2}$$

training packets	1. packet	0	1	1	0	1	0	0	0	
	2. packet	0	1	1	0	1	0	1	1	
	3. packet	0	1	1	0	0	1	0	1	
learned pattern	*mask*	1	1	1	1	0	0	0	0	
	value	0	1	1	0	-	-	-	-	
scoring packets	packet A	0	1	1	0	*1*	*1*	*1*	*1*	score 0.0
	packet B	0	**0**	1	0	*0*	*1*	*0*	*1*	score 0.25
	packet C	**1**	**0**	**0**	0	*0*	*0*	*0*	*0*	score 0.75

Figure 4.2 Example of the scoring algorithm. Three packets are received for a certain connection (top block). After learning from those three packets, *mask* shows the first four bits to be static, with the static values in *value* (center block; unimportant values are denoted as "–" for clearness; in the actual implementation, they are simply left unchanged). Incoming packets are then be assigned a score (bottom block). The score of a packet is the ratio of non-matching (static) bits to all (static) bits. Non-matching bits for to-be-scored packets are denoted in bold, masked (ignored) bits in italic.

where # denotes a bit-counting operator, counting the number of 1-bits in a bitfield, and $\oplus$ a binary XOR. In effect, the score is therefore calculated as the fraction of bits that are static, but do not match the expected value, of all static bits. Lower scores hence denote better matches: a score of 0 means a perfect match, while a score of 1 means that all static bits were exactly the opposite of the expected value.[39]

Figure 4.2 gives an example of how the learner and predictor parts of the algorithm work. For this example, packets are considered to be only 1 byte (8 bits) long. After receiving three uncorrupted packets, the mask has already reduced the number of static bits to half the packet's length. After this pattern is created, three corrupted packets are received that potentially can contain header errors. For each of these packets, the score towards the learned pattern is calculated. Packet A matches perfectly, while packet B and C show different amounts of deviations from the expected pattern.

In a real system, a multitude of such patterns are expected to exist at any point in time, one for each open connection. An incoming packet is matched against all open connections and classified to belong to the connection is has the lowest score towards.

First test results with this algorithm already showed it to be promising, but one large problem persisted. Since the algorithm creates the mask from monitoring packets for a connection, it only converges over time and starts out overspecific: after the very first packet reception for a new connection, the mask does not mask out any bits, and if a corrupted packet for that connection is received, the scores even towards that correct connection will be very high, since, on average, it can be expected that about half of all random bits (subsequently recognized as such when

[39]Note that scores are presented as values between 0 and 1 conceptually, suggesting that such an algorithm might use floating-point numbers, violating requirement 6. However, this can easily be circumvented by using fixed-point arithmetic and scaling the number of differing bits by a scaling factor, then using integer division.

refining the mask) will not match. The resulting score tended to be so high that packets for new connections were routinely misassigned to the wrong connection. While this problem could be alleviated by setting a strict cutoff value (that is, if even the best score is above a certain threshold, drop the packet instead of assigning it to that connection), this could not completely prevent this behavior, and to at least somewhat reign it in, the cutoff value had be set to such strict values that we considered the resulting high drop rates to be unacceptable.

Instead, we designed an extension to our scoring algorithm that creates *combined scores*. Such a combined score is designed to specifically highlight the differences in important bits that two connections share, by calculating a *combined mask*:

$$cmask_{1,2} = mask_1 \wedge mask_2 \wedge (value_1 \oplus value_2) \tag{4.3}$$

A combined mask is created by pairwise AND-ing of two standard masks of two connections. This allows to "boost" a non-specific mask from connections in their early phases with more specific masks of other connections. The idea behind this comes from the observation that protocols typically have fixed header layouts, so that the important bits are always in the same position inside the header. While there are many different protocols, and, for example, the protocol-specific mask for a connection using IPv4 looks different from one using IPv6, only focusing on bits that are known to be important in at least *some* connections is nevertheless helpful.

An example of combined scoring is presented in Figure 4.3. The first step from Figure 4.2, showing correct packets that trained the learner, is omitted; we immediately start with two connections and their respective masks and values at some point in time. As can be seen, the mask of connection 2 is completely unspecific: because only one packet had been received so far, no bits have been masked out yet. From the two connections, their combined mask is calculated. We then consider a corrupted incoming packet. Only considering the score for a packet, we can see that the score for connection 1 is 0.2, while for connection 2, it is 0.375: the packet should be assigned to connection 1. However, this decision is dominated by the second half of the packet, which might already belong to the payload, or contain non-static bit fields. This is the exact situation that led to misattribution in our preliminary evaluation. Note that this effect is much more problematic in real-life scenarios where packets are larger and there tend to be many more non-static than static bits. In this example, it seems that simply setting a cutoff at 0.2 would solve the problem. With real-world packets, however, this threshold had to be much stricter to reliably rule out misassignments, thus increasing unnecessary packet drops. The combined score for the two connections paints a different picture. In this example, they only differ in one bit, and this one matches for connection 2 and does not match for connection 1, leading to scores of 0.0 and 1.0, respectively. This suggests that connection 1, which was suggested by the normal scoring, might not be a good match after all.

As we will see in the evaluation, combined scoring proved to be very effective at preventing misattributions while keeping packet drop rates low. However, it is not without drawbacks. Because we have to do a pairwise comparison between all ongoing connections, we have increased the computational complexity of our algorithm

connection 1	mask1	1	1	1	1	0	1	0	0	
	value1	1	0	1	1	-	1	-	-	
connection 2	mask2	1	1	1	1	1	1	1	1	
	value2	1	0	1	1	0	0	1	1	
combined mask		0	0	0	0	0	1	0	0	
incoming packet		1	0	1	1	1	0	0	0	
score for connection 1		1	0	1	1	*1*	**0**	*0*	*0*	0.2
score for connection 2		1	0	1	1	**1**	0	**0**	**0**	0.375
combined score for connection 1		*1*	*0*	*1*	*1*	*1*	**0**	*0*	*0*	1.0
combined score for connection 2		*1*	*0*	*1*	*1*	*1*	0	*0*	*0*	0.0

Figure 4.3 To improve the classification algorithm, we introduced combined masks. Two connections are open. Connection 2 has only recently been opened, and its mask does not yet mask out any bits (e.g., because only one correct packet was received so far). To specifically compare the two connections, a combined mask is computed from the two masks and values (top block). A corrupted packet is received and needs to be classified (center block). The scoring algorithm suggests that it more likely belongs to connection 1, due to the large number of mismatching bytes for connection 2. The combined score, however, suggests that the packet should not be assigned to connection 1, either.

from $\mathcal{O}(n)$ to $\mathcal{O}(n^2)$. This can quickly become problematic. In the previous chapter, we noted that we typically saw fewer than 60 open connections on an end host during typical use, and never more than 100. However, this means that doing combined scoring requires, instead of 60–100 score calculations, 3 600–10 000 score calculations (plus the respective calculations of the combined masks themselves), which can transform an algorithm whose overhead is hardly noticeable into an impractically slow one.

We therefore limited the use of combined scoring. It is only used if

(a) a new connection exists[40] whose mask had not had the chance to stabilize yet. In this case, combined scoring will only be done between this new connection and all other connections.

(b) there are several connections that show a "good" score toward a packet. In this case, combined scoring will only be done between these good connections, to serve as an additional safeguard against misassignments. The definition of "good" is somewhat arbitrary; in our experiments, 0.2 showed to be a good number that did not negatively influence classification quality, while keeping the computational complexity low and the algorithm quasi-linear.

The two scoring methods are combined into one scoring algorithm by executing the following steps for each incoming corrupted packet:

1. Calculate the score of the packet towards each connection. If the score is above a preset threshold θ_s, remove the connection from consideration. Order the remaining connections by increasing score.

[40] We defined "new" as having received fewer than 10 correct packets. The reasoning behind this seemingly arbitrary number will be discussed in the evaluation.

2. For the first item in the list, if combined scores should be calculated as per the rules above, calculate the combined scores for that item between this item and other eligible items in the list. If none of these combined scores is above a preset threshold θ_{cs}, continue. Otherwise, remove the item from consideration and repeat with the next item, until a suitable item is found or the list is empty.

3. If the list is empty, drop the packet under consideration. If the top item of the list is a connection marked as error-tolerant, assign the packet to this connection; otherwise, drop the packet.

The last check is important because it preserves the opt-in property that we already noticed as important and implemented in Refector: corrupt packets can be recognized as belonging to an error-sensitive connection, but they will never be assigned to it and instead be dropped.

4.3 Implementation

In the following, we will present how to integrate the classification algorithm into a network system. For our approach, we used the concepts of the Linux kernel's network stack. Other operating systems that allow modifying the network subsystems code to hook into will, however, allow similar modifications. We will answer three main questions:

1. Where in the network stack should the learner and predictor be hooked into?

2. How do we deal with corrupted information in broken packets?

3. What other parts of the network stack need to be adapted for error tolerance?

4.3.1 Integration into the Network Stack

If we want to learn from correct packets, we need two pieces of information: we need the packet headers' contents to learn patterns from, and we need to know which connection the packet belongs to, so we can create a distinct class for each connection. The latter information is not available at reception time; after all, one important task of network protocols is to demultiplex incoming packets onto different concurrent connections. This suggests that learning should occur as late as possible, when demultiplexing has taken place and it is clear which connection a packet belongs to. Consequently, the learner sits at the top of the network stack; in a typical system, this means between the transport protocol and the socket interface, hooked into the function that assigns payload data to a socket. Conceptually, the integration of the learner is shown in Figure 4.4b.

In the Linux kernel, hooks in this position already exist to use the Linux Socket Filter (LSF), a socket-level implementation of the Berkeley Packet Filter [MJ93]:

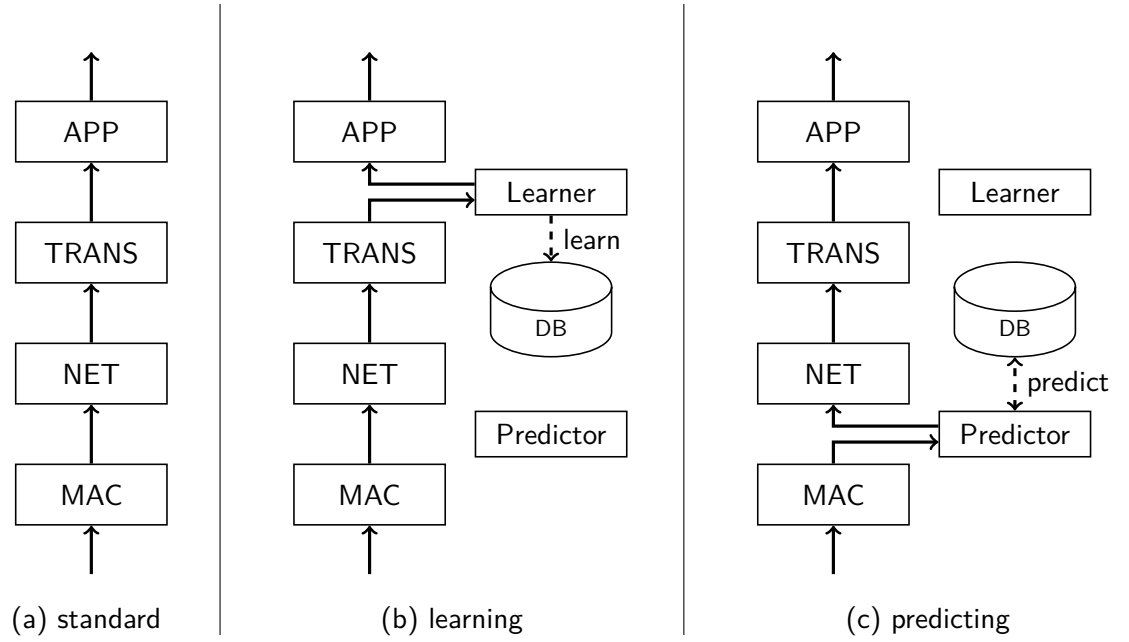


Figure 4.4 Integrating the classification algorithm into the network stack: In standard behavior (left), packets are simply passed up the stack and handled by protocols. When using the classification algorithm, correct packets are passed through the stack (center). Before handing them over, the pattern from the packet's headers and its destination (by then known) are saved to the DB. Corrupted packets (right) are intercepted early on, classified and, as far as possible, repaired by the predictor, then passed on.

every transport-layer protocol calls `sk_filter()`, which provides for filter functions to be called. While we do not add ourselves as a filter function for performance reasons – LSF is designed to allow user-space programs to hook themselves into the socket filter, and the resulting context switch overhead and memcopies between user and kernel space memory, if done for every single packet, is considerable [BBC+04] – we can add ourselves into the `sk_filter()` function itself, because every packet is expected to pass this function.[41]

At such a late point in the processing, one might think that the protocol headers are not available any more, because they have already been stripped from the packet after processing. However, this is not the case in the Linux kernel. Instead, an incoming packet is saved into an `skb_data` structure, which is effectively simply an allocated memory area. Two pointers are set to point to the first byte (`data`) and the last byte (`end`) of the packet, respectively.[42] As the packet traverses the stack, each protocol handler will simply move the `data` pointer accordingly to always point to the first byte of the following protocol header. This is more efficient than actually removing the headers from memory. Such a behavior can therefore be expected to

[41] Every packet that is destined for a user-space application, that is. If support for other types of packets is desired that never leave the kernel, such as ICMP messages, their handlers need to be patched separately.

[42] This allows using uniform-size memory regions for each packet, regardless of size: the remaining parts of the `skb_data` area are simply left unused. Thus, a so-called slab cache, a number of uniform-size memory blocks that are reused instead of allocated and deallocated every time, can be used, which greatly increases speed.

also be used in other operating system network stacks. Thus, if the original value of the `data` pointer is saved for later use, all headers are still available. Note that this assumes that protocol handlers do not rewrite information inside the packet headers themselves. However, we consider this a reasonable assumption: none of the standard protocol handlers show such behavior. Furthermore, it would be of questionable use, because the header area could only be used as temporary memory area, and such rewriting would open new possibilities for bugs, requiring careful implementation to not overwrite data that might be needed afterwards.

For the predictor, it is important to intercept the packet as early as possible. In a typical network stack, this means after the MAC and before the network layer. Conceptually, this is shown in Figure 4.4c.[43] The main reason for this is that corrupted packets can lead to severe problems if they are processed by protocol handlers not designed for this case. If important information, such as the destination port, is corrupted, the packet will be either dropped or misassigned.

It is hence beneficial to classify such corrupted packets as early as possible to predict the correct connection. In the Linux network stack, this can be achieved by hooking in the `netif_receive_skb()` function, which all MAC protocol handlers call after they have finished processing, and in which the next protocol handler to send the packet to is decided on. Hooking the predictor into this function again makes it easy to classify and repair the packet. However, at this point, we still need to ensure that the packet has a good chance to traverse the network stack. Simply bypassing all protocol handlers might seem appealing, because it both saves processing time, and we already know which connection the packet most probably belongs to, in any case. However, this is typically not possible, and cannot be done if we do not know anything about the used protocols, for two reasons. First, we still need to remove the protocol headers so that we can send the payload to the socket without headers. Protocol handlers will do this for us; if we wanted to circumvent them, we would need to know how much data to strip from the packet. This could be solved by saving the length of the headers with each connection's class. This length can be calculated by subtracting the original value of the `data` pointer from its final value. However, this only works if protocol headers have a static length: a typical scenario, but not one we can guarantee if we do not know the underlying protocols. Second, by bypassing the protocol handlers, we potentially change the communication behavior by discounting side effects. These can range from acknowledgments to flow or congestion control.

4.3.2 Repairing Header Contents

To send corrupted packets through standard protocol handlers, we need to make sure that vital information is repaired. In our predictor, we do this by applying the expected values of a connection class onto the packet. That is, if bits marked by the *mask* as static differ, they are written into the packet from the *value*. Again, this can be done with binary operations,

$$packet_{rep} = (value \wedge mask) \vee (packet_{corr} \wedge \sim mask) \tag{4.4}$$

[43] In a full SoftMAC system, where all MAC handling is done in software, it would be possible and desirable to intercept the packet even before MAC handling.

where $\wedge$ denotes a bitwise AND operation, $\vee$ a bitwise OR, $\sim$ a bitwise NOT, $packet_{corr}$ the incoming corrupted packet and $packet_{rep}$ the repaired packet.

Of course, this does not guarantee that all important bits are repaired, only static bits. However, our assumption in designing the algorithm was that identifying bits are typically static; hence, we can assume that identification information (addresses, ports, etc.) can be repaired in this fashion. If, however, one of the protocols implements sequence numbers and uses them for schemes such as in-order delivery, this solution will not help. This is a problem that we have to accept and for which our classification algorithm does not provide any solution. To support such a solution would complicate the algorithm considerably, since it wold have to recognize header field behaviors over time without any knowledge about the protocols. The reason we could implement timestamp and sequence number repairing for RTP with relative ease is that we knew which fields contained the information. Learning this, on the fly, without any knowledge about the protocols, while keeping the algorithm fast and accurate, seems to be a daunting task. However, this problem is smaller than it might seem at first: of all standard protocols used on the network and transport layer, only IPv4 and TCP use sequence numbers, and for IPv4, they only serve a purpose if the nowadays extremely exotic IP-layer fragmentation is used. For TCP, a protocol whose basic idea is reliable data transfer, the use with a tool whose basic idea is tolerance to errors is of questionable motivation.

Again, note that error-sensitive connections are out of scope of the question: repair never occurs in the first place, so problems with repairing those packets are irrelevant.

4.3.3 Protocol Adaptation

The classification algorithm needs to be able to distinguish between correct and corrupted packets, so that learner and predictor are only used on the appropriate packets. We can solve this problem like in Chapter 3, by adding a `checksumfailed` flag about whether a checksum has failed to the `sk_buff` structure which contains per-packet information in Linux. Typically, it makes sense to use the MAC layer checksum: it is available and has been checked by the time we need to decide whether a packet should be handed to the predictor, and MAC-layer checksums typically cover the whole frame (such as in Ethernet [IEEE12a][44] and WLAN [IEEE12b]). This, of course, requires changing the used MAC layer protocols to set this flag accordingly, in the same way as for Refector (cf. Section 3.2.3).

In fact, every protocol with checksums needs changes to its code to switch off packet drops on checksum mismatches and instead set the `checksumfailed` flag to true. Once the packet is about to be handed over to the application, the `sk_filter` then checks whether the flag is 0 or the `SO_BROKENOK` socket option is set, and otherwise drop the packet.

That every protocol using checksums (so, for example, there is no need to change the IPv6 handler) needs to have its checksum-checking code changed goes somewhat

[44] While the Ethernet II de-facto standard is used much more widely than IEEE 802.3, both use the same checksumming behavior and are therefore equivalent for these purposes.

against the original design of having a protocol-independent solution. However, in this case, we see no other choice but to compromise and do so. This is indubitably a weakness of the approach. However, while this means that, for every new protocol that should be supported, a human needs to check the code and change it accordingly, we argue that there is still a very significant difference in work involved compared to implementing protocol-specific heuristic header error repair. Checksum checking code is often very self-contained and generic. Conversely, creating an error-tolerance solution for a new protocol requires knowledge about the protocol, analysis, design, and a much larger change in the code. In this way, a protocol-independent classification approach still saves much time and work, and significantly eases the burden of supporting new protocols.

4.4 Evaluation

In the following, we will investigate the performance of our classification algorithm. We will answer three questions:

1. How many packets are correctly attributed, and how many are misattributed, for different cutoff threshold settings?
2. How fast is the implemented algorithm?
3. How fast does the classification algorithm converge, that is, how many correct packets does it take to produce high accuracy?

4.4.1 Experimental Setup

For the same reasons as given in Section 3.4.5.1, we opted for a simulative approach to test the algorithm's performance. To do so, we used a very similar setup as before: packets were fed into a corruptor, which introduced bit errors, and then handed over to the classification algorithm for learning or predicting, depending on whether the packet had had any errors introduced. To distinguish between the two cases, the corruptor signaled whether a packet had had errors introduced or not, emulating the behavior of a checksum algorithm which signals this information in a real system.

For the design of the algorithm, we had already created a data set of diverse applications. To further evaluate the algorithm's performance, we created a second data set, because testing on the same data set that was used to design the algorithm can lead to problems similar to overfitting: the algorithm and its learning procedures might end up being adjusted for high performance on only this specific data set, while not delivering the expected performance on others.

Similar to the first data set, this set comprised a large number of different connection types: web surfing, audio streaming via HTTP web radio, audio streaming via RTP, video streaming via HTTP, video streaming via RTP, a debian Linux `aptitude upgrade` to update the package database and upgrade outdated packages, ICMP

pings, as well resultant DNS lookups. While the use cases were similar to the ones used in data set 1 (because those are the ones we consider typical use cases of an Internet connection), we made sure to change specific parameters such as which radios or videos to stream. Overall, data set 2 comprised 214 connections with 48 268 packets, with many (especially HTML-over-HTTP) connections only having a single-digit number of packets, while others (especially the video streaming) received thousands of packets.

As in the RTP evaluation (cf. Section 3.4.5), we used a Bernoulli process to inject errors to model a binary symmetric channel, investigating the whole BER range from 0 to 0.5 in 0.01 increments. One problem is that at high BERs, virtually all packets are corrupted, so the classification algorithm cannot learn from correct packets and cannot produce any meaningful evaluation result. In the RTP case, we solved this by sending several initial packets error-free to collect information to subsequently predict fields such as sequence number and timestamp. In this evaluation, with an algorithm that does not know anything about the inner workings of the protocols it is expected to classify, two initial packets is not enough. We therefore opted for a scenario in which the error injection works in a two-step process: first, the corruptor decides whether to corrupt the packet at all, or to forward it unscathed. We set the chance of a packet being forwarded without errors to a static 30% in all our experiments. The remaining 70% then had errors injected according to the specified BER. We acknowledge that this means that, at very high BERs, we potentially overestimate the algorithm's performance, because at such BERs, we cannot expect to receive any correct packets. Conversely, this setup removes the constraint that "the first n packets are received correctly", as in the RTP case. Our setup, with high burstiness in errors can be seen as modeling a highly variant connection, in which factors such as strong fading, interference, or sub-par rate adaptation leads to a large number of corrupted packets, but with intermittent transmission successes. It could be argued that such a scenario models the real world example of home users better than an overall unvaryingly high BER, which would typically be the result of low transmission power or strong path loss due to high distances between sender and receiver.

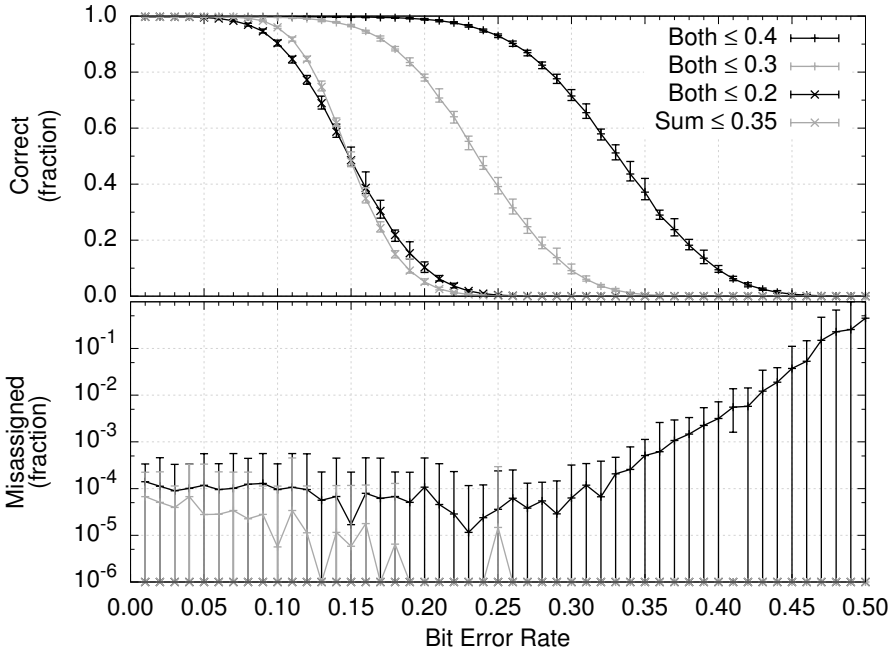
As mentioned in Section 4.2, we considered the first 40 bytes of each packet for classification. Unless otherwise specified, each data point in our evaluation was the result of 20 repetitions of the experiment. To further increase variance, we did not send the same monolithic block of 48 268 packets in the same order to the classification algorithm every time. Instead, we logically split the capture into 214 sub-captures, one for each connection, and varied the starting time for each of those captures during the experiments, so that the algorithm had to cope with different combinations of concurrently open connections at any given point during each of the experiments. Furthermore, we used the same approach with our original data set that was used during algorithm design (data set 1). However, in the following, unless otherwise specified, the results presented will be from evaluation on data set 2, for the reasons given above.

4.4.2 Classification Accuracy

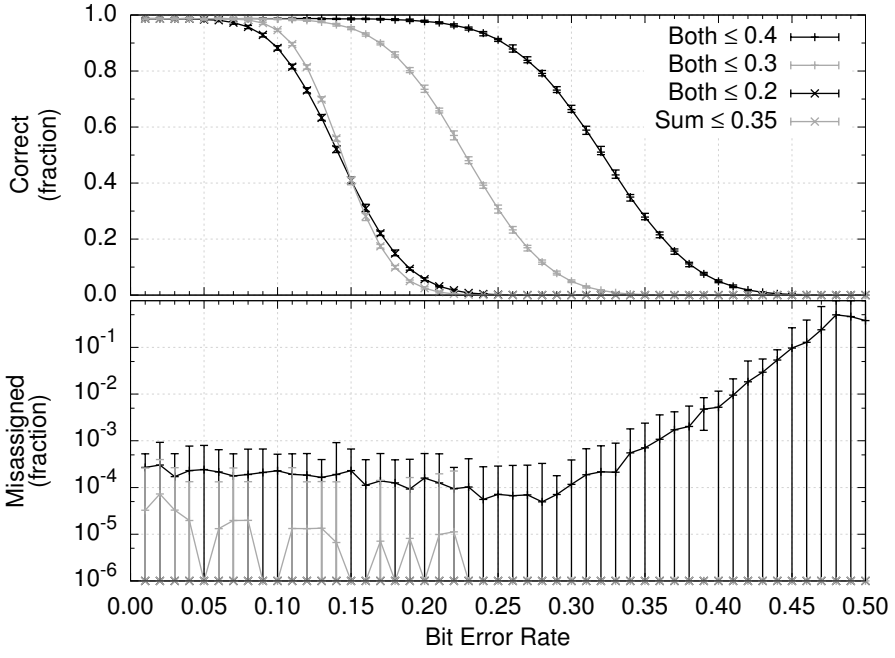
We will now examine the accuracy of the classification algorithm described in Section 4.2. Two questions will be the main focus of this section: (1) How many packets can be correctly assigned to the application they belong to, and (2) how many packets are incorrectly assigned? Note that due to the cutoff threshold, the two questions are not simply two sides of the same coin: a packet can be either assigned correctly, misassigned, or not assigned at all because the classification algorithm refused to come to an assignment decision. To further investigate especially this effect on the classification performance, we will investigate different settings for the algorithm's thresholds θ_s and θ_{cs}. When we present the results, we will focus on the performance for error-tolerant streams, which we define as the audio and video streams in the data sets. That means that our correct assignment and misassignment rates are calculated from the number of packets that belong to those connections, not from the total number of packets in the data set. The reason for this is that classification accuracy for error-sensitive streams is not important in itself, because error-sensitive streams will not have any corrupted packets assigned to them, neither correctly or incorrectly. Hence, misassignment rate is always 0 for those streams. On the other hand, those streams indirectly influence the performance of the algorithm, and as such, their presence is important: We need to create classifiers for each stream, error-tolerant or not, so that a corrupt error-sensitive packet has a high chance to be correctly recognized as such, reducing the misassignment rate. If classifiers for error-sensitive connections did not exist, every corrupted packet would only be matched against classifiers for error-tolerant connections, and corrupt error-sensitive packets would have no chance to be classified correctly. Conversely, the existence of a sizable number of error-sensitive connections increases the risk that an error-tolerant packet will be misassigned: the higher the number of managed classes, the larger the risk that a misassignment occurs. While a misassignment of an error-tolerant packet to an error-sensitive connection has no negative repercussions for that connection (corrupted packets assigned to error-sensitive applications are dropped), the result is a packet loss for the application it was destined for, and a reduction in accuracy.

Correct assignment rate is hence calculated as the number of packets from error-tolerant connections that were assigned to the correct application, while misassignment rate is calculated as the number of packets (both error-tolerant and error-sensitive) that were incorrectly assigned to an error-tolerant stream and therefore lead to a packet being sent to the wrong application. Note that this means that our misattribution rates are indeed higher than if we had considered packets from all streams: since error-sensitive connections, by design, cannot have any misassignments (because corrupted packets assigned to them are dropped), their misassignment rates are always 0 and would decrease the overall misassignment rate.

Figure 4.5 shows the classification algorithm's accuracy for both data sets. Error bars in the graphs denote minimum and maximum values for that data point during our evaluation, not confidence intervals. Presented are numbers for settings of $\theta_s = \theta_{cs} = 0.2, 0.3, 0.4$, as well as a "semi-dynamic" threshold setting of $\theta_s + \theta_{cs} = 0.35$. In this last case, the cutoff threshold in the combined scoring phase is more lenient if the score in the normal scoring phase was low and indicated a good match. The



(a) Data set 1 (also used during design phase)



(b) Data set 2 (newly created for evaluation phase)

Figure 4.5 Classification accuracy for all (error-tolerant) streams. Note the linear scale in the upper and the logarithmic scale in the lower subgraphs. Performance differences between the two data sets are negligible. Stricter cutoff thresholds θ_s and θ_{cs} effectively present misassignments. Conversely, they lead to earlier drop-offs in correct assignments, opting on the safe side and dropping many packets that would have been correctly assigned.

two data sets show only negligible performance differences: correct assignment is minimally lower and misassignment rate minimally higher in data set 2, but only to very small degrees. This shows that the algorithm, which was designed using data set 1, has not been overspecialized by mistake and also works well for other scenarios.

The cutoff thresholds, unsurprisingly, show a similar influence on performance as the ones used in the RTP library implementation. As thresholds become stricter, misattributions can be effectively prevented. At a threshold of 0.4, misattributions occur at rates between 10^{-4} to 10^{-3} until BER is in excess of 30%. A threshold of 0.3 already decreases the misassignment rate by 1–2 orders of magnitude, to the point where in some test series, we did not witness any misassignments at all. At a threshold of 0.2, misassignments are an exceedingly rare occurrence. In all experiments, data points, and repetitions, we did not witness a single misassignment at that threshold. Conversely, stricter thresholds reduce the rate of correct assignments earlier on (at lower BERs), because they err on the side of caution more often and drop packets that would have been assigned to the correct application. However, even at the strict cutoff of 0.2, packet drops only start occurring at about 6%. The semi-dynamic threshold, while also preventing misattributions, keeps the correct assignment rate at top performance up to about 8%.

One interesting effect that can be seen in the results, though it is not very clear and of rather small magnitude, is that, up to BERs of about 0.3, the misassignment rate in fact decreases very slowly, before picking up again. This is because, at higher BERs, scores increase on average, leading to more packets being dropped, which also means that more misattributed packets are dropped. When BERs reach extremely high values, the number of correctly assigned packets becomes so low that even a small number of remaining misattributions increases the fraction of misassignments and offsets this effect.

These results are very promising: without having any domain knowledge about the employed protocols, we were able to design a classification algorithm that correctly assigns virtually all packets up to a BER of about 8%, while preventing any misassignments. To further stress-test the algorithm, we then designed a scenario in which we took nine YouTube RTP streams, let them run concurrently, and investigated the classification performance. The idea is that the connections should be as hard to distinguish between themselves as possible, which is assured by the fact that all these streams are of a the same communication type, from the same provider. The results are shown in Figure 4.6, and comparison to Figure 4.5 shows a somewhat surprising result. While correct assignment rates are somewhat lower, showing a decrease from the perfect 100% at BERs between 4% and 7% the misassignment rates are, in fact, also lower. This, however, is mostly due to the fact that the maximum concurrent streams in this scenario was 9, while in the larger data sets, more concurrent streams were possible and regularly happened. With an increasing number of streams and hence classifiers, the risk of misassignments increases. Overall, however, it can be seen that even in such a somewhat arbitrary scenario (we do not expect users to watch 9 videos at the same time), the classification algorithm still performs very well.

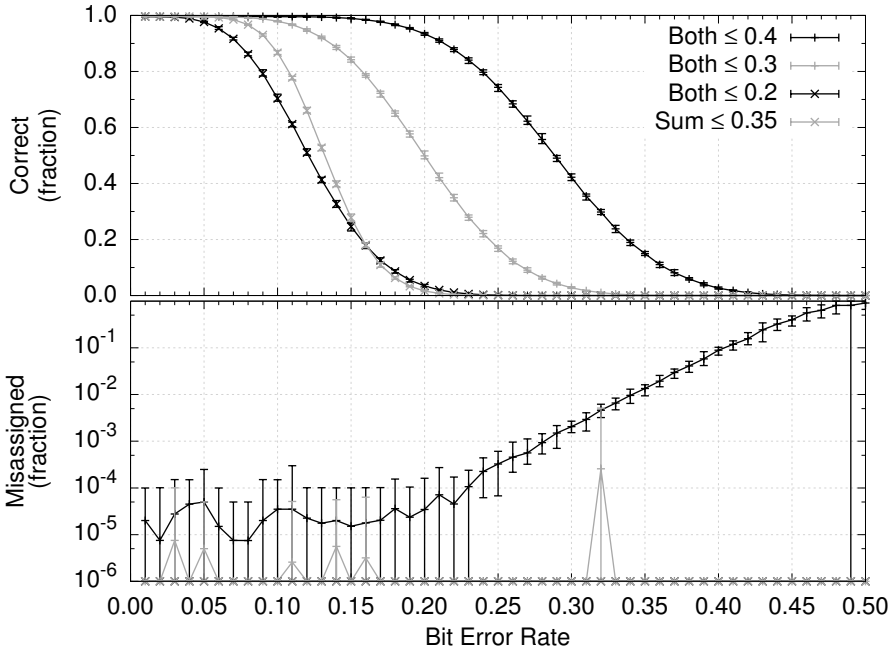


Figure 4.6 Classification for nine concurrent YouTube streams. Even in such a stress-test scenario with a large number of very similar streams, classification stays highly accurate and comparable to the less artificial case presented in Figure 4.5.

4.4.3 Classification Speed

We have shown that the classification algorithm is accurate to a very high degree, even under moderately high BERs. However, for the algorithm to be practical, we also need to show that its performance is able to keep up with packet rates typically expected in wireless networks, preferably without producing a large computation overhead. The lightweight design of the algorithm suggests that this should be the case; in this section, however, we will investigate the performance in a quantitative manner.

For this, we will distinguish between learner and predictor performance. There are two main reasons for this. First, every packet only traverses either the learner (if it had a correct checksum) or the predictor (if it had a checksum mismatch). At low BERs, most packets will pass the learner; as BER increases, the weight will shift over to the predictor. Second, learner and predictor have very different complexities: All the learner has to do (after the initial setup of a class for a connection) is to look up the class for the connection, and recalculate the mask. The predictor, on the other hand, has to match packet contents against each open connection, and in case combined scoring is done, also against combinations of connections.

For this evaluation, we ran the algorithm on an Intel Core 2 Duo CPU at 2.66 GHz. Note that the algorithm is single-threaded and only used one core. The times presented are the computation time of the `learn` and `predict` functions. Initialization, such as memory allocation for mask and value bitfields for new connections, is not accounted for. However, since such an initialization only occurs once per connection,

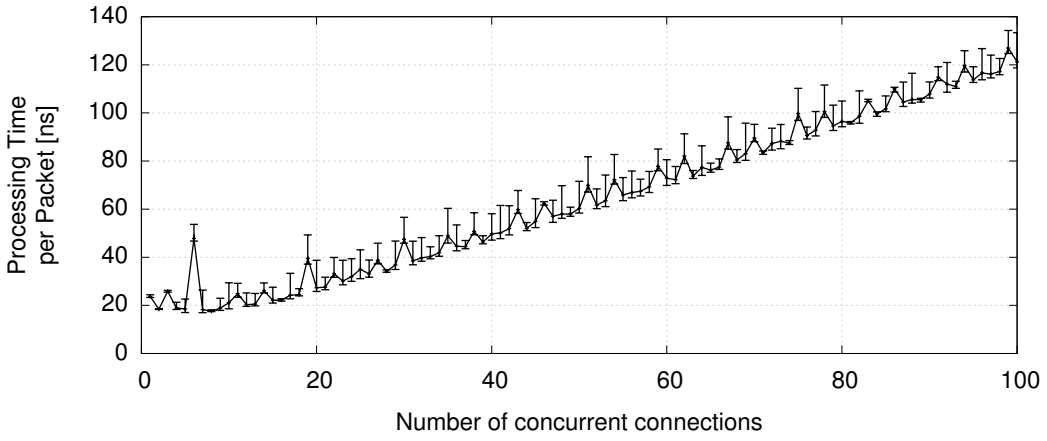


Figure 4.7 Processing time in the learner is on the order of nanoseconds and increases linearly with number of open connections. This linear increase is due to the use of linked lists in the implementation and could be further reduced to a static overhead by using a different data structure or lookup mechanism.

it is less relevant for the overall speed of the algorithm. The results presented here classified packets based on the first 40 bytes and used thresholds of $\theta_s = \theta_{cs} = 0.2$.

Figure 4.7 shows the performance of the learner. As can be seen, learning from a packet can be done within nanoseconds. The linear increase with increasing number of connections is due to the current implementation, in which the classes are kept in a linked list that takes $\mathcal{O}(n)$ to traverse. The implementation could be changed to have an $\mathcal{O}(1)$ lookup, for example, by using a hash table, or, even more efficiently, having a pointer in the kernel's `socket` structure that directly points to the class of that connection. Since learning happens immediately before the packet is handed over to the application, the socket is known and could hold this additional information, which would eliminate even the static hashing overhead. Nevertheless, the current implementation already shows that learning is so fast that this is not a pressing issue.

Prediction speed is, as expected, lower. Figure 4.8 shows processing times of the predictor; note that variance of the results was so low that error bars are practically indiscernible. The predictor's performance is governed by two influences – number of open connections and BER – and is significantly higher. Still, even at a high number of concurrent connections, processing time stays in the low microsecond range. This time, the linear increase in processing time with increasing number of connections is not chiefly caused by the lookup in a linked list. Since the predictor has to calculate scores for each packet for all open connections, this linear increase is expected. That the increase is linear and not quadratic proves our conjecture we discussed when we introduced combined scoring: given the right threshold, the $\mathcal{O}(n^2)$ complexity can be avoided. Interestingly, the processing time decreases with increasing BER. The reason is that, the higher the BER, the higher, on average, scores are (because there tend to be more non-matching bits). This means that more connections are ruled out as suitable candidates during phase 1 of the algorithm as described in Section 4.2.2, which in turn means that fewer connections have to be considered for combined

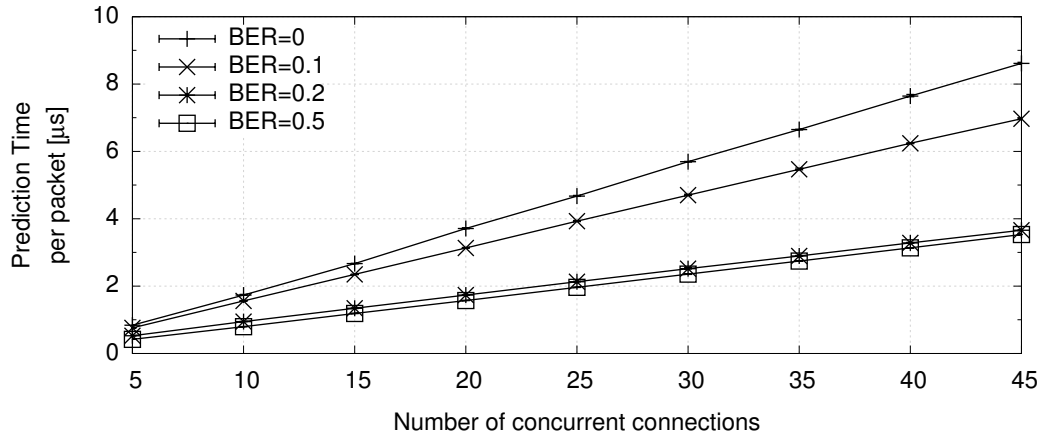


Figure 4.8 Processing time in the predictor is on the order of microseconds and increases linearly with number of open connections. Processing time decreases as BER increases because at high BERs, more streams are ruled out as candidates during step 1 of the classification algorithm.

scoring. Note that the data points for BER = 0 are a synthetic test: to create these results, we injected uncorrupted packets into the predictor after learning from them. However, predicting packets with a BER of 0 (that is, error-free packets) is, in itself, not an unrealistic case. Failing checksums, the sign that a packet is not to be trusted and needs to be predicted, only denote errors *somewhere* in the packet. Those errors, however, might be in areas not considered by the algorithm, for example, in the later parts of the packet containing the payload, while the considered areas are completely error-free.

To put these numbers into context, we provide some approximations on the average inter-arrival time in 802.11 when the channel is fully utilized. We do this by taking the channel throughput, deciding on two scenarios that differ by packet size, and dividing those sizes by the throughput. Table 4.1 shows some examples for 802.11n and 802.11g. Note that the nominal speeds of 600 Mbit/s and 54 Mbit/s are unreachable in practice: they merely denote the peak speed that can be reached while a data frame it sent, and do not take into account overhead due to interframe spaces and contention periods. The other speeds are examples taken from measurements by Pefkianakis et al. [PHW+10] in the case of 802.11n and our own measurements in the case of 802.11g. Even using 3x3 MIMO, channel bonding to increase the bandwidth, and frame aggregation to combine several packets as subframes into one aggregated frame, Pefkianakis et al. were only able to reach 180 Mbit/s under very favorable conditions. In most of their scenarios, they could barely reach 100 Mbit/s, often significantly less. The frame sizes that we chose stem from the sizes used in our own measurements: an overall packet size of 112 bytes is equivalent to a payload size of 50 bytes in Figure 2.8 on page 31, and 1532 equivalent to 1470 bytes. The difference of 62 bytes is due to packet headers, since Figure 2.8 denotes application-layer payload size, while Table 4.1 denotes MAC-layer frame sizes.

At first glance, Table 4.1 seems to suggest that an 802.11n network at full throughput can potentially overwhelm the predictor while sending many small frames. For

802.11 standard	Throughput	(Sub-)Frame size (on MAC layer)	Avg. time between (sub-)frames
802.11n	600 Mbit/s	1532 bytes	19.5 µs
		112 bytes	1.4 µs
	180 Mbit/s	1532 bytes	64.9 µs
		112 bytes	4.7 µs
	100 Mbit/s	1532 bytes	116.9 µs
		112 bytes	8.5 µs
802.11g	54 Mbit/s	1532 bytes	216.4 µs
		112 bytes	15.8 µs
	21 Mbit/s	1532 bytes	69.6 µs
	1.3 Mbit/s	112 bytes	657.3 µs

Table 4.1 Estimation of average time between two frames. The 802.11 numbers assume frame aggregation and hence give the average time between two subframes. 600 Mbit/s and 54 Mbit/s denote nominal maximum rate throughput, which is impossible to achieve in practice. The other 802.11n throughputs are suggested by results in [PHW+10], while the other 802.11g throughputs are taken from our own measurements, presented in Figure 2.8 on page 31.

example, at 180 Mbit/s, the average interframe time is 4.7 µs. The results presented in Figure 4.8 show that we can only support approximately 30 concurrent connections at a BER of 10%, and about 25 connection if the BER is lower. However, note that to even reach such high throughput and consequently low average inter-frame times, the frame error rate must be very low. Therefore very few packets will be processed by the predictor. Most will be processed by the learner, which has a much lower processing time. However, even if the error rate were to reach 25%, the average time between frames passed to the predictor would quadruple to, for example, 18.8 µs for the case of 100-byte frames at 180 Mbit/s.

In more challenging situations with higher error rates, rate adaptation will increase robustness, which will decrease throughput. Consequently, the average time between frames will increase. Therefore, under any but the most unfavorable circumstances, our classification algorithm will be able to keep up with 802.11n networks, and comfortably so when 802.11g rates are used.

4.4.4 Classifier Convergence Speed

So far, we have shown in the evaluation that our classification is fast and accurate. However, one requirement posed for our algorithm was that it is fast to adapt to changing conditions; most importantly, fast to create effective classifiers for newly-opened connections. The question of how soon a classifier for a new connection is to be fully trusted also influenced our decisions to introduce combined scoring, and when to use it. Hence, we now will have a deeper look into how fast classifiers converge, that is, how many packets it takes for a classifier to become stable and trustworthy.

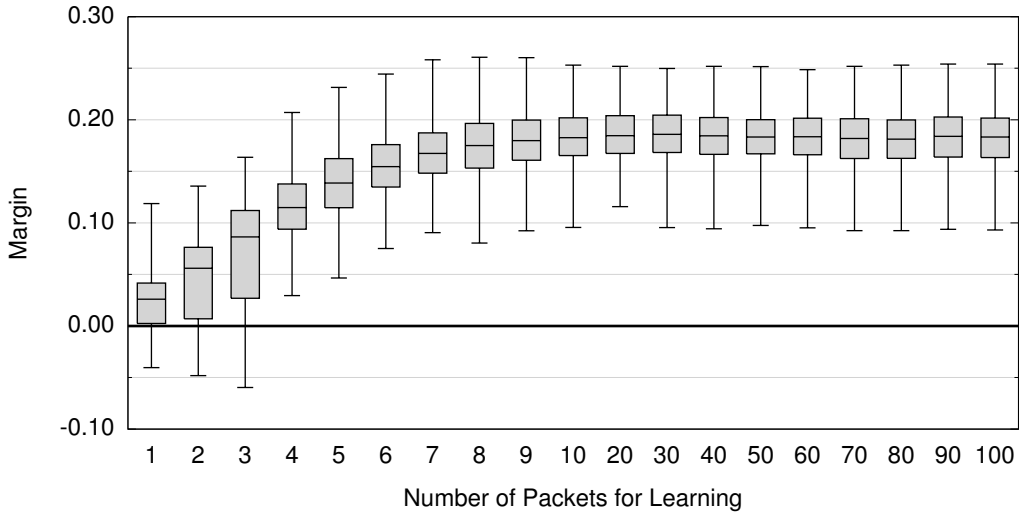


Figure 4.9 For all connections from data set 1 with more than 100 packets, a variable number of packets was used for learning, then all subsequent packets were corrupted with a BER of 0.1. Metric was the difference between the score to the correct connection and the best non-correct score. If this value was below 0, a misattribution had occurred. Whiskers denote minimum and maximum values. After only 4 packets, the classifiers had converged to the point that misattributions did not occur any more. After 7–10 packets, the results were virtually indistinguishable from setups with 100 learned packets.

For this evaluation, we took the first n packets from each connection in data set 1 that had at least 100 packets and used them to train the classifiers by injecting them into the learner. Afterwards, we took the remaining packets, corrupted them and injected them into the predictor. We then investigated how reliably the classifiers worked in identifying packets correctly. To get a deeper insight than the binary decision between correct and misassignment, we calculated for each packet the difference between the score for the correct class (that is, the one that packet needs to be assigned to for correct assignment) and the lowest score for an incorrect connection. If this difference is negative, then misassignment has occurred; if it is positive, then the assignment was correct. We investigated this metric for different BERs. As a representative example, we show the results for BER $= 0.1$ in Figure 4.9.

The relative behavior of the results does not change with BER. The main change is that the difference decreases roughly proportionally as BER increases, or, more figuratively speaking, the barrier "rides up": The classes still converge at the same speed (unsurprisingly, because the learner only uses uncorrupted packets), but it takes more learned packets before misattributions can be ruled out, until at some point, the BER is so high that misattributions occur regardless of number of learned packets, and can only be prevented by thresholds and combined scoring.

This might raise a question: if we do not see any misattributions at all after only 4 packets, why do we not simply make our classification algorithm drop packets for connections that have fewer than 4 learned packets and are unreliable? Would this not be a more reliable and more efficient way to prevent misattributions than using combined scoring, which, as can be seen in Figure 4.5, indeed does not fully prevent misattributions without cutoff thresholds that, conversely, lead to many packet drops

at high BERs? Sadly, this is not possible because new connections open regularly, and those connections can produce corrupted packets. Since we do not know what connection a corrupted packet belongs to when it enters the system (which is the reason why the prediction algorithm exists in the first place), we have to deal with corrupted packets that might belong to classifiers that are not yet fully trained. However, the results from this evaluation answer the question from Section 4.2.2 that we promised to answer later on: why the seemingly arbitrary number of 10 learned packets, below which we apply combined scoring? After 10 packets, the classifier has converged to a point that it can be considered so good to be almost indistinguishable from classifiers with more learned packets.

4.4.5 Summary

By now, we have shown that our algorithm fulfills the requirements laid out in Section 4.1. The algorithm was designed to work on streaming data (requirement 2) and has ways to set cutoff thresholds that have shown themselves to be effective at reigning in misattribution before (requirement 3). While we have not explicitly gone into the details of the implementation, it does fulfill requirement 6 and works without using floating-point arithmetic.

In the evaluation, we have further investigated the performance of the algorithm with regards to the remaining requirements. The algorithm is indeed effective (requirement 1). With the right parameterization, misattribution can be effectively prevented, while virtually all packets are assigned to the correct application up to BERs rates of 7%-10%. This means its effectiveness is similar to the one presented in Section 3.4, but it works protocol-independently and does not need any domain knowledge about the protocol headers that it uses for classification, learning all significant contents by itself. Furthermore, it does so very quickly (requirement 4). It only takes few packets to construct an effective classifier. After learning from 10 packets, the classifier has converged to a pattern that is as effective as after 100 packets and more. Finally, due to its use of binary operations during the classification process, it is also extremely fast (requirement 5) and can easily keep up with typical wireless connections. Overall, we consider this algorithm a very good match for the problem statement.

4.5 Classification via Extrinsic Factors: Size and Inter-Arrival Time

While the classification algorithm that we presented fulfills all the laid out requirements as it is, we considered some extensions to it during the design phase, especially while it was still unclear whether packet classification purely based on packet content, as presented in this chapter up to this point, would be able to perform as required.

In the following, we will present our investigation into the behavior of what we termed *extrinsic factors*. Those factors are extrinsic in the sense that they are not contained in the packets themselves; they are not part of the transmitted data itself. Two such factors, and the ones we focused on, are size and inter-arrival time.

The main motivation for this work was an observation that streaming data exposes a specific communication behavior different from other data transmissions. This is in its purest form seen in audio streaming scenarios, such as Internet live radios or VoIP communications. Applications, or the network stack for them, typically try to aggregate enough data to send large packets, because this reduces the overhead introduced by protocol headers. On the other hand, streaming data needs to be sent in a timely manner to not arrive at the receiver too late to be of use. This is especially a problem in live audio scenarios: audio data can be compressed well enough to be comparatively small, and as such, a single large packet could contain more than the 500 ms of audio, which are considered the upper limit to delay [ITU03], after which the delay renders the connections nearly unusable. Considering that quality starts to suffer at delays above 150 ms [ITU03], and these are introduced by intercontinental transmission delays alone, most audio codecs and applications send packets in much shorter time intervals to reduce delay due to the application. For example, AMR is designed to use 20 ms intervals [ETSI00]. This means that audio streaming connections should show very specific packet inter-arrival times: every 20 ms, a packet for a connection should arrive. Additionally, unless the used codec uses adaptive bit rates, all packets should also have the same size, because they each contain data for exactly 20 ms. The idea therefore was to learn a "footprint" of typical packet sizes and inter-arrival times for every connection.

This idea is not completely new. Statistical analysis of traffic to identify connection types has been investigated in recent years. However, there are some fundamental differences between those approaches and ours. First, such solutions typically try to recognize traffic belonging to a certain *type* of traffic, such as mail traffic, web traffic, or SSH traffic [MZ05, CDGS07], or to recognize unusual behavior patterns for connections of a certain type to, for example, recognize HTTP or SSH tunnels [DCGS09]. Furthermore, these classifiers generally require offline training on precollected datasets. In contrast, for our classification, recognizing the correct traffic type is not enough, it is also necessary to distinguish between potentially several flows of the same type occurring concurrently. We also have to learn new patterns on-the-fly, and cannot rely on offline learning.

Furthermore, we are much stricter in our classification accuracy requirements. Most traffic classification approaches are used to do traffic shaping or filtering at the backbone or edge router level; in those cases, occasional misclassifications are acceptable. For example, the above-cited works only produce hit ratios (correct classifications) of 80–99%. A misclassification of even 1%, however, means that such an algorithm would produce unacceptably high misassignments of packets, at least on its own. Nevertheless, previous work in the field gave us hope of devising a solution tailored to our needs.

In the end, we abandoned this classification approach, for two reasons: First, it turned out that the content-based classification algorithm already performs very

well in its designed form, and it was questionable whether the additional overhead introduced by extensions that take into account extrinsic factors would produce enough improvements to be a worthwhile tradeoff, especially with respect to increased processing time.

Second, the challenges presented in this section, while deceptively simple at first, turned out to be highly problematic. In fact, we were not able to produce convincing results in our endeavor.

Thus, in contrast to all other parts of this dissertation, this section documents an investigation that did not produce directly applicable results. However, the lessons from this investigation are, in our opinion, too valuable to be disregarded. At the least, they will serve to give additional insight into the problem faced when dealing with unreliable information and corrupted data in data communication scenarios, and are therefore a good match for the overall topic of this dissertation; at best, they will engender further research in this area and how extrinsic information can be used for packet classification.

In the following, we will show the problems that we faced with this approach.

Applicability to other types of traffic

From the beginning, it was clear that some connections would not show specific packet sizes and inter-arrival times. For example, file transfers, web browsing HTTP traffic, or remote shell connections could all be expected to not show such behavior. File transfers typically use maximum packet sizes to minimize overhead, but then send data as fast as the connection (and potentially protocol controls, such as TCP's congestion control) allows. The same would be the case for image transfers and larger HTML text pages. While recognizing size and inter-arrival time patterns in encrypted HTTP connections to identify which server a client is communicating with has been investigated in much detail [BLJL06, LL06, WCN+14] in security research, those patterns are neither regular nor periodic and can at best be used to recognize a connection that was footprinted before, not an ongoing connection as it progresses. For SSH connections, most packets would be small, but their timing unpredictable. Thankfully, none of these types of connections are error-tolerant streaming applications.

However, we noticed that the problem is almost as serious in streaming connections. In video streams, key frames are often large enough not to fit into a single packet and need to be fragmented. This leads to bursts in which several large packets arrive with extremely short intervals, finished by a non-maximum-size packet. This packet can have any size, since it just happens to contain the remainder of the key frame. There is therefore no way to define a connection-specific packet size. Even in audio streams, this can become an issue, typically due to variable-bitrate codecs or non-live audio content in which large amounts of data are sent in the initial buffering phase of the stream.

Figure 4.10 shows two sets of examples, one with three audio streams and one with three video streams. The two criteria (size and inter-arrival time) are used

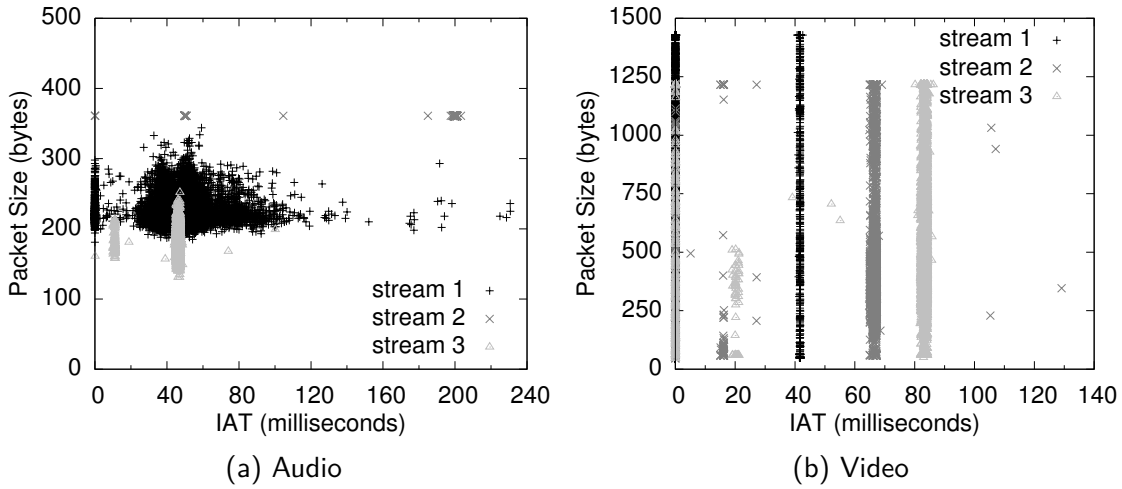


Figure 4.10 Two examples illustrate the problems faced when trying to exploit packet size and inter-arrival time (IAT) for classification of packets towards streams. The two criteria are plotted onto the axes. For reliable classification, each stream's packets should fall into a distinctive area. However, overlaps are readily visible (e.g., at 40 ms and 200 bytes in Figure 4.10a and at 0 ms in Figure 4.10b), illustrating problems in the separation.

as dimensions in these 2-dimensional graphs. To properly classify the packets to their streams, each stream would need to form a class that can be visualized as one or several two-dimensional areas within the graph, without any overlap with other stream's areas. The overlap between the different streams highlights the problem that reliable separation by these criteria alone is impossible.

This leads us to the second factor that is also visible in the figures, and exacerbates the problems.

Influence of Jitter

While streams are not completely separable from each other by means of size and inter-arrival time, such information from a packet can still give hints which connection a packet most likely does *not* belong to; it could be used to reduce the number of possible connections to consider. However, the figures already show that there are easily recognizable clusters, but also outliers in many locations. The outliers typically occur in the time domain, which makes sense: packet sizes stay unchanged during transmission. However, jitter occurs regularly during Internet transmissions, which is one of the reasons that streaming applications employ buffers to prevent choppiness. Note that these measurements were done over an Ethernet line, so the data was transmitted through the Internet over lines that do not have to deal with collisions. In a wireless scenario, where the channel medium is shared and collision avoidance with exponential backoffs is in place, jitter can be expected to be much worse. This further weakens the specificity of the classifiers.

While there are measures to detect and remove outliers from classes, this incurs additional and non-negligible overhead. Furthermore, there is another effect that produces even worse artifacts.

Learning under non-perfect conditions

The goal of our endeavor is to classify corrupted packets. However, we explained before that we only learn from correct packets, because otherwise we run the risk of tainting our classifier with wrong data resulting from misassignments. This reason still stands for the size/inter-arrival time classifier. However, if only some packets are used for learning, there is no reliable way any more to recognize the inter-arrival time. Consider a case in which a packet is sent every 20 ms. If a packet becomes corrupted and is not processed by the learner, the inter-arrival time suddenly appears to have become 40 ms. If more than one packet is corrupted in sequence, the time increases further. Even worse, during the initial learning phase, the learner might at first not even notice that the "typical" inter-arrival time is 20 ms, settling for a multiple of that value. If we indeed, as we have suggested above, only use the classifier to rule out impossible connections, we have thus ruled out the correct connection and set up the main classifier to produce either a misassignment or a packet drop.

Thus, to cope with the problem of several corrupted packets in a row, we might have no other choice than to keep track of all received packets, corrupted or not: We would need to keep track of when we assigned a corrupted packet to a connection, so that the timer for the inter-arrival time can be reset, and the next packet (provided it arrives at the correct time, see above) can be recognized as belonging to the connection. However, this again goes against our rule not to have corrupt packets influence future assignment decisions, and again, this rule exists for a good reason: if a misattribution occurred, the inter-arrival time timer will be reset at the wrong time, and until the next correct packet is received for a stream, resynchronizing it, all expected arrival times will be off.

Alternatively, we could not only consider the base inter-arrival time, but also multiples of it: so if the inter-arrival time is 20 ms, then $40, 60, 80, \ldots$ ms will also be acceptable. This, however, further waters down the already low specificity of the classifiers.

Separability of concurrent streams

Compared to all these issues, this one seems almost small in comparison. However, compared to the others, it is less practical and more conceptual, casting another problematic light on the approach. If the inter-arrival times for two streams are not relatively prime to each other (and, depending on the severity of jitter, even if they are), the expected reception times for a packet for both streams will overlap regularly. Unless packet sizes for the two streams are so specific and different that they can be separated by that alone, the algorithm will not be able to decide which packet a connection belongs to.

Again, if the size/inter-arrival-time classifier is only used as a prefilter to the content classifier, reducing the number of eligible classes, this effect is not too serious. However, using a potentially complicated algorithm (creation of classes from fuzzy input, outlier removal) as a prefilter to an algorithm that has shown to be fast and accurate raises the question of whether this endeavor is worthwhile.

4.6 Conclusion

In this chapter, we presented a novel solution that goes above and beyond what Refector has already achieved. We could show that it is possible to design an algorithm that can classify packets, assigning them to the correct connection, even under header errors, when the connection identification information is broken, without having any understanding about the protocols that are used to carry that information. Without any prior knowledge, the algorithm recognizes parts of packet headers that are significant to the connection assignment decision, learns these patterns online without any necessary off-line training, and can assign corrupted packets correctly with a very high probability, even under strong error conditions.

For example, given the right parameterization, almost all packets can be correctly assigned up to (extremely high) BERs of 7–9%, while completely preventing any incorrect assignments. Furthermore, the algorithm is very fast, both with respect to its runtime performance, which can easily keep up with current WLAN speeds, and with respect to convergence of newly learned patterns, where it only takes roughly 10 uncorrupted packets to produce a highly accurate classifier.

While we were less successful in creating an algorithm that takes into account extrinsic information (packet sizes and inter-arrival times), we can safely say that, given the mentioned results, such an algorithmic extension to our classifier is not even necessary. The results achieved by content classification produce such an accurate classification on their own that it is questionable whether the considerable overhead that we expect classification of extrinsic factors would introduce could offset the potential marginal benefits.

Overall, we consider the results of this chapter fascinating, and a very interesting culminating point of our research into error recovery for Internet protocols that surpassed our initial experiences significantly, to the point where we feel that we are at a point at which these results are well-rounded. For the last main chapter of this dissertation, we will therefore leave the field of *implementing* error recovery, and instead focus on *supporting* it, by solving a fundamental practical problem that stood in the way of applying header error recovery in WLAN, the wireless access technology that is by far most-used by end users.

5

一寸先は闇
(いっすんさきはやみ)

One inch ahead is darkness.

—Japanese Proverb

OFRA: Rate Adaptation for 802.11 Networks Without Acknowledgments

In the previous chapters, we focused on creating solutions to introduce error tolerance into the network stack to recover from errors in headers. We could show that such solutions are both feasible and highly effective. While the approaches were designed to be as independent of the underlying MAC and PHY as possible, we did use 802.11 as our application scenario, due to its widespread use for wireless communications.

However, we deferred a detailed discussion on how to use heuristic header error recovery in 802.11 until now. Previously, we simply mentioned that we would use the 802.11e [IEEE05] extension that allows us to send frames without acknowledgments. We also explained how to send frames with and without acknowledgments concurrently, if need be. However, we did not take into account the fundamental challenge that current state-of-the-art rate adaptation algorithms have with such No-ACK traffic. Since these algorithms rely on ACKs as feedback that informs them about current channel conditions, none of them will properly manage such traffic. Instead, they either react slower or not at all to changing channel conditions (depending on the amount of ACKs created by concurrent traffic that does not use No-ACK), or will interpret the lack of ACKs from No-ACK frames as frame losses and reduce the rate until the lowest rate is reached.

This is obviously undesirable. Not only does it massively reduce throughput, since only the slowest rate is used. It also increases the time every frame occupies the channel, and hence the latency of the transmission. This motivates our design of a novel rate adaptation algorithm that can properly adapt rates even when no ACKs are received. Instead of ACKs, we use special feedback frames that convey information about the channel, and which are only sent on-demand, that is, when channel conditions change significantly. One of our additional requirements for this

algorithm is that it should also perform at least on par with current state-of-the-art algorithms when it comes to standard, ACK-ed traffic, since all error-sensitive traffic is still expected to be sent with ACKs. We will see that we indeed reach both goals.

The rest of this chapter is structured as follows: We will first take a step back in Section 5.1 to describe and discuss the solution space. This motivates our decision to use the No-ACK scheme for error-tolerant transmissions, despite its lack of rate adaptation support. In Section 5.2, we will introduce the concept of on-demand feedback. We will explain in detail how modulation and coding influence error rates and throughput in wireless systems, with a focus on 802.11, and how we can use this knowledge to decide when to send feedback. Finally, we will explain how to send this feedback and introduce the concept of feedback frames. We will discuss related work in Section 5.3 and shortly explain the basic building blocks of our implementation of OFRA in the ns-3 [LH06, HRFR06, ns3] network simulator in Section 5.4. In Section 5.5, we will evaluate the performance of OFRA in detail against several state-of-the-art rate adaptation algorithms with respect to various metrics. We will discuss some possible extensions to OFRA in Section 5.6 before concluding in Section 5.7.

5.1 Introduction and Motivation

Before we focus on the problem of creating a rate adaptation algorithm that supports No-ACK traffic, let us take a step back and look at the bigger picture. Sending error-tolerant traffic with disabled ACKs by employing the 802.11e extension is not the only conceivable option. We will use this introduction to comprehensively discuss all sensible choices for ACK schemes. This gives us the possibility to check whether there is a more suitable scheme that can support the requirements of both error-sensitive and error-tolerant traffic well, and that is feasible to implement in 802.11. The results will show that there is no such scheme, and suggest that it is indeed best to use No-ACK for error-tolerant transmissions.

To do so, we will start with an elementary discussion on the role of ACKs in data and especially wireless communications. We will discuss the semantic meaning of acknowledgments, and show different ways ACK schemes can be set up. Afterwards, we will discuss the conceptual advantages and disadvantages of each scheme, as well as the practical and technical feasibility to implement each scheme in 802.11.

5.1.1 The Role of ACKs in Data Communications

As seen in the preceding chapters, our error tolerance concepts envision a use primarily in wireless networks. This is due to the comparatively high error rate in wireless networks compared to wired networks, in which bit errors are exceedingly rare, and packet loss is more commonly due to congestion of links. This high BER motivates the use of ACKs in IEEE 802.11 [IEEE12b] networks. The type of ACKs used there is the standard (non-negative) case: if a frame was correctly received,

an ACK is sent. Otherwise, the receiver stays silent. At closer inspection, such an acknowledgment scheme conflates three outcomes into two reactions (as do most ACK schemes). For every frame, there are three possible outcomes at the receiver:

Case 1: The frame is received and correct (checksum matches).

Case 2: The frame is received, but with errors (checksum does not match).

Case 3: The frame is not received at all.

802.11 conflates cases 2 and 3 into the behavior "do not send ACK". In normal operation, this makes sense for 802.11. Frames that are received with errors are considered to be as useless as lost frames. However, as soon as corrupted frames are assumed more useful than completely lost frames, this conflation is not sensible any more. It is therefore important to scrutinize whether this differentiation of the two cases also requires or at least strongly suggests differentiation in the acknowledgment behavior.

Looking at the three cases, the easiest one to analyze is indubitably Case 3. If a frame is completely lost, the receiver will not even be aware that a frame destined for it was sent. This means that it cannot behave in any special way.[45] This non-responsiveness in itself functions as a form of communication. In this special case, "one cannot not communicate" [WBBJ67], while aimed at human communication, also holds true for a network protocol. In this case it informs the sender of the unreliability of the communication: either the frame it sent did not arrive correctly, or the acknowledgment was sent in response did not arrive correctly (which, while possible, is much less likely, due the small size and robust encoding of ACK frames in 802.11).

The next easiest case is Case 1. On reception of a frame, the receiver calculates the checksum over the received frame and checks whether it matches the received checksum calculated by the sender before transmission. If the checksums match, the frame is correct and and ACK response is created.

Case 2 is somewhat more complicated. Note that Cases 1 and 3 are very specific: Either the frame was lost completely, that is, no correct data was received at all; or the frame was received correctly, that is, all data in the frame was received. Case 2 runs the gamut between theses cases. Single bit errors in large frames are covered by this case, as well as almost completely corrupted frames that were just barely received at all.

After looking at these three cases, we will now analyze several different acknowledgment scenarios and how effectively they work within the concept of error tolerance. Note that the four cases to be discussed cover all schemes that are both possible

[45] This is only true for packet-switched networks, in which contention for the channel is the primary way of access. In situations in which channel capacity is reserved for each communication partner, for example, in the case of Time Division Multiple Access (TDMA), a receiver will know when to expect data from a sender, and can consequently answer with a NACK if no data was received at all.

	Received correctly	Received with errors	Not received
No change	Send ACK	Do not send ACK	
Always ACK	Send ACK		Do not send ACK
Never ACK	Do not send ACK		
Special ACK	Send ACK	Send special ACK	Do not send ACK

Table 5.1 Overview of the behavior of the different ACK choices presented in this section. For each scheme, the behavior on the three cases "received correctly", "received with errors", and "not received" is shown. Note that because it is not possible to send an ACK when no frame was received, this covers all possible and sensible cases.

and sensible (also cf. Table 5.1). Because it is not possible to send ACKs when no frame was received at all, "Always ACK" covers the maximum possible amount of ACKed frames. Also, because the three possible outcomes are of increasing severity for communication, it does not make sense to change the order of behavior in the table (e.g., send no ACK for correct reception, but send one for a reception with errors); if a lower-severity case skips sending an ACK, a higher-severity case should not be signaled by sending an ACK.

In the following, we will discuss each of the four approaches with respect to both conceptual and practical benefits and drawbacks of each approach.

5.1.2 Conceptual and Practical Considerations

No change

This scenario does not change the standard 802.11 behavior. The advantages are obvious: no change means no additional work in developing a novel solution.

The sender will expect an ACK if the packet was received without any errors, and no ACK otherwise. This standard behavior is suited for error-sensitive traffic, in which only a completely correct transmission is of any use.

On the other hand, it is problematic for error-tolerant streams, and is the reason why we use No-ACK for Refector. Not receiving an ACK because the transmission was corrupted, the sender will retransmit the frame continuously (up to a sender-defined maximum). Thus, the channel will be occupied for an inordinate length of time, which massively reduces overall throughput. It also means that the receiver might be better off hoping for a correct retransmission instead of employing error tolerance. If retransmissions arrive regardless, and there is no way for the receiver to make them stop, waiting for a correct reception is a valid strategy if latency is not a major concern.

Always ACK

If not sending an ACK on reception of a corrupted frame leads to inefficient behavior, the next idea might be to stop those unwanted retransmissions. This can be done

by always sending an ACK when a frame is received, whether correct or corrupt. This effectively eliminates retransmissions. It has the benefit that throughput is increased, because instead of using the channel for retransmissions, more data can be sent.

This behavior provides a decent match for error-tolerant transmissions due to skipping retransmissions. However, this also entails practical problems. First, this mode is not useful for error-sensitive traffic. Since retransmissions are switched off and partially corrupted packets are discarded, packet loss rate will be high. The transmission will either be slow since higher-layers have to trigger retransmissions, or they will break completely.

Furthermore, as we mentioned at the beginning of this chapter, most rate adaptation algorithms for 802.11 use ACKs to gauge the current channel quality. If an ACK is received, the assumption is that the channel is able to support the currently chosen rate without producing errors at that point in time. Always sending ACKs therefore suggests a higher channel quality to the receiver than is available in reality. This will lead to overselection (choosing a higher data rate than sensible) by the sender, which is one of the primary causes of bit errors in the first place. In short, this scenario will further increase the bit error rate to potentially unacceptable amounts.

Finally, there is no support from the 802.11 standard for such a behavior, so it would need to be implemented. The main problem with this is that, while rate adaptation algorithms generally reside in software in the OS, checksum checking and ACK sending are typically part of the firmware, which is closed-source and hardware-specific, making it hard to change. In some cases, this functionality might even be implemented in the hardware itself.

This is done for reasons of efficiency and speed, to fulfill the strict timing requirements for ACK sending in the 802.11 standard. The 802.11 ACK scheme assumes that there is always only one frame outstanding.[46] When a frame is sent, the sender reserves the channel not only for the time it requires to send its frame over the channel. It reserves additional time comprising the time an ACK frame takes to be sent, plus a Short Interframe Space (SIFS). This short time (16 µs) accounts for radio latencies (such as switching from receiving to sending), as well as for calculation of the checksum itself. Because it generally is impractical to have the whole frame off-loaded to the OS, the checksum checked there, and the result signaled back to the network adapter, the checksumming and decision whether or not to send an ACK are done within the network adapter.

Changing the ACK scheme to also send ACKs if the checksum did not match therefore requires firmware rewrites. It also means that no general solution can be provided; each chipset requires changes to its own firmware. Combined with the fact that firmwares are rarely publicly available, at least not in source code, it makes this approach very time-consuming and hard to realize in practice.

[46] Exceptions from this rule are the Block ACK mechanism. Nevertheless, even Block ACKs have strict timing requirements, so the arguments given here for normal ACKs also hold true for Block ACKs.

Never ACK

Since neither keeping the current ACK scheme nor changing it to send ACKs also in the corrupted case produces desired outcomes for error-tolerant transmissions, a third idea is to go the opposite way and send fewer ACKs than in the standard case. This means never sending any ACKs at all, even if the frame was received correctly. This behavior is the one provided by the 802.11e No-ACK policy.

This behavior closely mirrors the Always-ACK behavior. For error-tolerant transmissions, Never-ACK has the advantage that it prevents retransmissions if a packet was received, but with errors. However, we need to make sure that this scheme is not used for error-sensitive transmissions, since there will be no support for retransmissions.

Not sending any ACKs will also lead to rate adaptation problems, albeit in the opposite direction. Since no ACKs are received at all, most rate adaptation algorithms will assume that frames could not be sent correctly at the currently chosen rate, and reduce the rate further and further, down to the base rate. The combination of lowest sending rate and large numbers of retransmissions for every single data frame leads to unacceptable performance.

The main advantage of Never-ACK over Always-ACK is its ease of use. The Never-ACK policy is easy to use and deploy since it is implemented via the 802.11e No-ACK policy. Support for No-ACK by the standard has an additional advantage. Because there is no need to send ACKs, the inter-frame timings are also tightened by the standard for such transmissions. Whereas in an ACK scenario, the sender reserves the channel for an additional SIFS and ACK, this is not done in a No-ACK situation. The channel is therefore available earlier for further transmissions. In good conditions, when frames are rarely lost, No-ACK therefore increases the capacity of the WLAN channel and the overall possible throughput.

Special ACK

From the scrutiny of these three approaches, it follows that none of them, in itself, can provide a satisfying solution for both error-sensitive and error-tolerant traffic. This is possible, however, with the Special-ACK solution, which extends the two-state signaling of current ACKs with a tri-state system, in which a special notification is sent if a frame was received, but with errors. Extensions such as this are not unheard of. For example, TCP HACK [BLK+01] introduces an additional TCP option that sends a special header-only checksum with the (TCP-layer) packet. This allows the receiver to signal corruption in the data portion to the sender. Maranello [HSG+10] introduces sub-checksums over blocks of the frame, the match or mismatch of which are signaled back to the sender. From a conceptual point of view, this seems like the most elegant solution, because it neatly separates the three cases into three different types of answers.

To support both error-sensitive and error-tolerant traffic with this policy, the following behavior could be implemented. If the special ACK that denotes a reception

	No change	Always ACK	Never ACK	Special ACK
Error-Sensitive Traffic	unsuited	suited	suited	suited
Error-Tolerant Traffic	suited	unsuited	unsuited	suited
Ease of Implementation	easy (default behavior)	impractical to implement	easy (support by standard)	impractical to implement
Rate Adaptation Support	full support	causes strong overselection	causes strong underselection	full support

Table 5.2 Tradeoff between the different choices of ACK scheme with regards to error-tolerant transmissions.

with errors is received for a packet form an error-sensitive connection, a retransmission is sent. If such a special ACK is received for an error-tolerant packet, no retransmission is triggered.

Furthermore, rate adaptation algorithms should work well with this scheme, since the "normal" ACK is received under exactly the same conditions as in the standard, no-change scenario.

However, this solution suffers from the same problems as the Always-ACK approach with respect to practical feasibility. Introducing additional ACK behavior that is not mandated by the standard requires rewriting network adapter firmware. This alone makes this approach impractical. In addition to that, the introduction of a new type of ACK also means defining a new frame type that is used and needs to be supported by both parties. This is, in itself, not an insurmountable problem; [47] however, in combination with the necessary firmware rewrite, this approach can be categorized as the least practical.

5.1.3 Summary

From the previous considerations, it becomes clear that no solution is a perfect fit, and that there are important tradeoffs. A short overview is given in Table 5.2.

While Special ACK is the most promising concept because it can support error-sensitive and error-tolerant traffic equally well, it is hampered by its incompatibility with the 802.11 standard and infeasible implementation overhead. Since all other concepts only support either one or the other, we will need a combined solution.

For error-sensitive traffic, the standard behavior is indeed the best fit, since the other two are not suited for such traffic.

With regard to error-tolerant traffic, we can decide between the Always-ACK and the Never-ACK concept. From the summary in Table 5.2, Never-ACK is clearly

[47] In fact, we will later see in Section 5.2.3 that such a novel frame type can be very useful and its implementation is feasible

better than Always-ACK, simply because of its ease of implementation. Hence, the decision in Chapter 3 to employ the No-ACK scheme for our heuristic header error recovery is indeed the best choice among the conceivable ACK schemes.

However, it is also apparent that, while the concept is suited in general, it suffers from bad performance because of its lack of rate adaptation support. No ACKs mean no working rate adaptation for virtually all rate adaptation solutions. Without a working rate adaption, however, the performance will suffer greatly. Hence, we need to provide a novel rate adaptation algorithm to unlock the full potential of error tolerance in 802.11.

Note that such a solution will not only benefits error tolerance concepts. It also, in addition, opens the door for practical use of the No-ACK scheme in 802.11 in general, which so far has been neglected for reasons of which no rate adaptation support is not the least.

5.2 Concept

After this decision to use the No-ACK concept for error-tolerant transmission, its practical downside has to be solved by implementing a novel rate adaptation algorithm. To do so, we have to consider the general challenges that rate adaptation algorithms face, and solve them within the framework of No-ACK transmissions.

5.2.1 Scarcity of Information and Provision of Feedback

First and foremost, we have to solve the fundamental rate adaptation challenge of *scarcity of information*. The adaptation of transmission rate is done by the sender, because it has to construct the packet and choose the bit rate on the PHY layer. However, the quality of reception as the deciding factor for transmission errors is only known by the receiver, because channel effects such as path loss that occur between sender and receiver are not apparent to the sender. Therefore, some sort of information transfer is necessary to close the feedback loop and facilitate rate adaptation.

In standard 802.11 transmissions, ACKs serve as this kind of feedback: if the sender receives an ACK, it can be sure that the transmission was successful; if it does not receive an ACK, it can assume that the transmission failed.[48] Rate adaptation schemes that rely on this information are classified as *frame-loss based* approaches. Their advantage lies in their simplicity: typical systems already employ ACKs, and the information whether or not the transmission succeeded is readily available (if only because it is needed to schedule potential retransmissions). Their disadvantage is the low amount of information gathered this way: all the receiver knows in an

[48]The information the sender receives this way is not perfect. On the one hand, it cannot distinguish between a frame that was received with one or more bit errors and a frame that was not received at all. On the other hand, it is also possible that the frame was received correctly, but the ACK was lost.

error case is that the transmission failed in one way or another. No fine-grained information about the reception quality is available. For our scheme, because we decided not to use any ACKs, we of course cannot use this way to transfer the necessary information.

To gain more detailed information about the reception quality, one popular approach in proposed rate adaptation schemes is to look at the SNR of received frames. These *SNR-based* approaches use the measured reception quality of frames the sender received from the receiver to estimate the reception quality of the opposite direction, that is, of frames that are to be sent from the sender to the receiver. These measurements can be done on ACKs received in response to sent frames, or on received data frames due to bidirectional traffic (or both). However, this is also not without problems.

First of all, these schemes assume channel reciprocity (that is, that the frames sent from A to B witness the same attenuation as frames sent from B to A), which is not a universally accepted concept. As an example, hardware effects [FZJJ06] can distort the theoretical reciprocity property [Tai92]. Multiple Input Multiple Output (MIMO) setups do not follow it at all and need careful calibration to reach near-reciprocity [BCK03].

Second, since channel conditions change over time, if the last reception was so long ago compared to the speed of channel quality changes that the conditions changed significantly (the so-called *coherence time*), the rate decision is based on stale information, and the chosen rate might not at all correspond to the currently optimal choice. This either leads to choosing a rate that is not robust enough and produces a high BER (*overselection*), or to choosing a rate that is far more robust than necessary, reducing throughput (*underselection*). This is a problem that occurs when extracting SNR information from both ACK and data frames. Even though ACK frames have to follow very strict timing rules, note that these only assure timely sending of ACKs *after* a data frame. For a sender, however, it is important to have information from a frame that arrived only very shortly *before* the rate for the current frame has to be chosen.

Third, size differences between received and sent frames hamper the comparability derived from channel reciprocity. In fast-fading situations, short frames can be affected very differently from long ones, in which some of those effects can average out.[49] Furthermore, relying on ACKs for SNR-based decisions is problematic in quickly deteriorating channels. In such situations, frames will more likely be received erroneously or not at all, in which case no ACKs are being sent that could inform the receiver (via their low SNR) that it should switch to a more robust bit rate, which can lead to a total breakdown of communication.

Comparing the two approaches, while SNR-based solutions are not without problems, they still have the substantial advantage that they can provide more fine-

[49]This averaging out chiefly relies on the method of calculating frame SNR. If the SNR is calculated over the whole frame, this effect can occur. If, however, only the preamble or preamble plus beginning of the frame are used to calculate the SNR, as is the case in some hardware (e.g., [MSA06]), this effect is much less prominent; nevertheless, comparability of SNR between frames is problematic in a fast-fading scenario.

grained information. While frame-based approaches are by design limited to very coarse information, SNR-based approaches merely need to find ways to improve timeliness and correctness of feedback. Therefore, we will consider such a solution for our rate adaptation scheme. There is also a practical consideration: one of the advantages of frame-based approaches is the simplicity of providing feedback, because standard ACK frames double as feedback. However, since we already decided to use an ACK-less scheme, this main advantage of frame-based approaches is void.

There is another effect that comes with eschewing ACKs: we cannot assume any more that we will receive at least some bidirectional traffic. In a standard acknowledged system, even if traffic is unidirectional, we can expect to receive ACK frames in the opposite direction. Thus, unless we require some amount of bidirectionality in traffic from the upper layers (a dangerous assumption, because it might not always hold, and in those cases render our rate adaptation scheme ineffective), we have to provide feedback without any strong coupling to data transmissions. Therefore, we cannot use any approach that relies on reciprocity, be it by employing received data frames or ACK frames.

Instead, we need to provide explicit feedback from the receiver to the sender. Such a receiver-based approach is not a completely new concept. However, we will see in Section 5.3 when we discuss related work that none of those rate adaptation algorithms fit our requirements, since most of them either piggyback their SNR feedback onto ACK frames, or produce impractical overhead due to constant RTS/CTS exchanges. Instead, we will explicitly send feedback from the receiver to the sender, when feedback is necessary because channel conditions change significantly and require a rate change. From this behavior stems the name of our approach, On-demand Feedback Rate Adaptation (OFRA).

5.2.2 When to Send Feedback: Choice of Optimal Rates

This "when feedback is necessary" can be quantified, and we will do so in this section. To do so, we will first analyze in detail how robust different rates are to errors. In Section 2.2, we explained how errors arise in wireless networks. In this section, we will quantify how strongly error conditions influence different rates in 802.11. This explanation follows a multi-step approach. First, we will consider the susceptibility to errors of those modulations used in WLAN. Next, we will add the influence convolutional coding at different code rates to these results, to give an understanding of how susceptible different bit rates – which are a combination of a modulation and a code rate and are hence also termed Modulation and Coding Scheme (MCS) – are to errors. Finally, we will investigate how those different bit rates influence throughput over bit error rates, at which point we will be able to answer our initial question, and arrive at an understanding of when feedback needs to be sent.

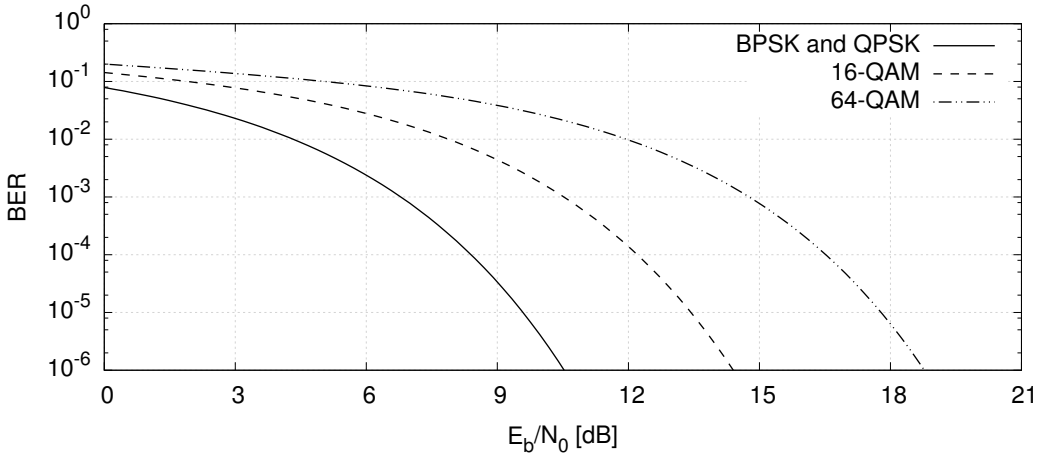


Figure 5.1 Bit error rate vs spectral density for several modulations, given as energy per bit to noise E_b/N_0.

5.2.2.1 Modulation

As a first step, we will look at the four modulations used in 802.11 (BPSK, QPSK, 16-QAM and 64-QAM) on their own, that is, with an implied code rate of 1. One basic way of measuring modulation performance is by looking at BER against E_b/N_0, the *energy-per-bit to noise power spectral density ratio*. The main advantage of this metric is that it is independent of the power spent on symbols, which encode one or several bits at the same time. In easy terms, if a symbol contains twice as many bits, it may use double the power and still provide the same efficiency.

Given a certain modulation and E_b/N_0, there exist formulas to calculate the BER, under the assumption of an Additive White Gaussian Noise (AWGN) channel. This has the added benefit that there is no need to distinguish between the additional modulations applied afterwards, such as DSSS (802.11b) or OFDM (802.11a/g): because AWGN is neither time variant nor frequency selective, the noise creates the same effects for both and keep them comparable. For the same reasons this also abstracts from the modulation to translate the signal to the carrier frequency, and therefore does not need to distinguish between different WLAN channels.

Because in BPSK, the binary signals are antipodal, the BER can be calculated as [Pro85, pp. 144–148, 168]

$$BER = \frac{1}{2}\operatorname{erfc}\left(\sqrt{\gamma}\right) \tag{5.1}$$

where γ is the E_b/N_0, and erfc is the *complementary error function*, defined as

$$\operatorname{erfc} = 1 - \frac{2}{\sqrt{\pi}}\int_0^\infty e^{-t^2}\,\mathrm{d}t \tag{5.2}$$

For QPSK,the general Phase-Shift Keying (PSK) formula for BER calculation can be used [Gar07, pp. 257–258]:

$$BER_{PSK} = \left(\frac{1}{\log_2 M}\right)\operatorname{erfc}\left(\sin\left(\frac{\pi}{M}\right)\sqrt{\gamma \log_2 M}\right) \tag{5.3}$$

where M is the number of distinct values a symbol can hold, and $\log_2 M$ therefore the number of distinct symbols in that modulation. Its result for QPSK shows that the BER is equal to BPSK's for the same E_b/N_0:

$$BER_{QPSK} = \frac{1}{2}\operatorname{erfc}\left(\sin\left(\frac{\pi}{4}\right)\sqrt{2\gamma}\right) = \frac{1}{2}\operatorname{erfc}\left(\sqrt{\gamma}\right) \tag{5.4}$$

This is because QPSK's two components are orthogonal to each other, and therefore provide the same BER as BPSK for equal E_b/N_0 [Pro85, pp 168–169].

For the more sophisticated QAM schemes, the formula has to take into account not only phase modulation, but also amplitude modulation. The BER is calculated as [CY02]:

$$\begin{aligned} BER_{QAM} = \frac{1}{\sqrt{M}\log_2\sqrt{M}} \sum_{k=1}^{\log_2\sqrt{M}} \Bigg\langle \sum_{i=0}^{(1-2^{-k})\sqrt{M}-1} \Bigg[(-1)^{\left\lfloor \frac{i\cdot 2^{k-1}}{\sqrt{M}} \right\rfloor} \\ \cdot \left(2^{k-1} - \left\lfloor \frac{i\cdot 2^{k-1}}{\sqrt{M}} + \frac{1}{2} \right\rfloor\right) \cdot \operatorname{erfc}\left((2i+1)\sqrt{\frac{3\gamma\log_2 M}{2(M-1)}}\right) \Bigg] \Bigg\rangle \end{aligned} \tag{5.5}$$

For the two cases used in 802.11, 16-QAM and 64-QAM, this yields

$$BER_{16QAM} = \frac{3}{8}\operatorname{erfc}\sqrt{\frac{2}{5}\gamma} + \frac{1}{2}\operatorname{erfc}\sqrt{\frac{18}{5}\gamma} - \frac{3}{8}\operatorname{erfc}\sqrt{10\gamma} \tag{5.6}$$

and

$$\begin{aligned} BER_{64QAM} = \frac{7}{24}\operatorname{erfc}\sqrt{\frac{\gamma}{7}} + \frac{1}{4}\operatorname{erfc}\sqrt{\frac{9}{7}\gamma} - \frac{1}{24}\operatorname{erfc}\sqrt{\frac{25}{7}\gamma} \\ + \frac{1}{24}\operatorname{erfc}\sqrt{\frac{81}{7}\gamma} - \frac{1}{24}\operatorname{erfc}\sqrt{\frac{169}{7}\gamma} \end{aligned} \tag{5.7}$$

respectively. The detailed calculations are given in Appendix B. Note that for $M = 4$, Equation 5.5 produces the same result as Equation 5.4, interpreting QPSK as 4-QAM.

However, as can be seen from the results (which are visualized in Figure 5.1), the value of these results for our use case is limited. This is because the point of view taken is one of high abstraction. To compare the performance of modulations under noise, each modulation is assumed to operate under its maximum efficiency η:[50]

$$\eta = \frac{R_b}{B} = \frac{M \cdot R_s}{B} \tag{5.8}$$

E_b/N_0 (denoted as γ) abstracts from both (frequency) bandwidth B and bit rate R_b (and symbol rate R_s):

$$\gamma = \frac{SNR}{\eta} = SNR \cdot \frac{B}{R_b} = SNR \cdot \frac{B}{M \cdot R_s} \tag{5.9}$$

[50] Or each modulation at least operates at the same η.

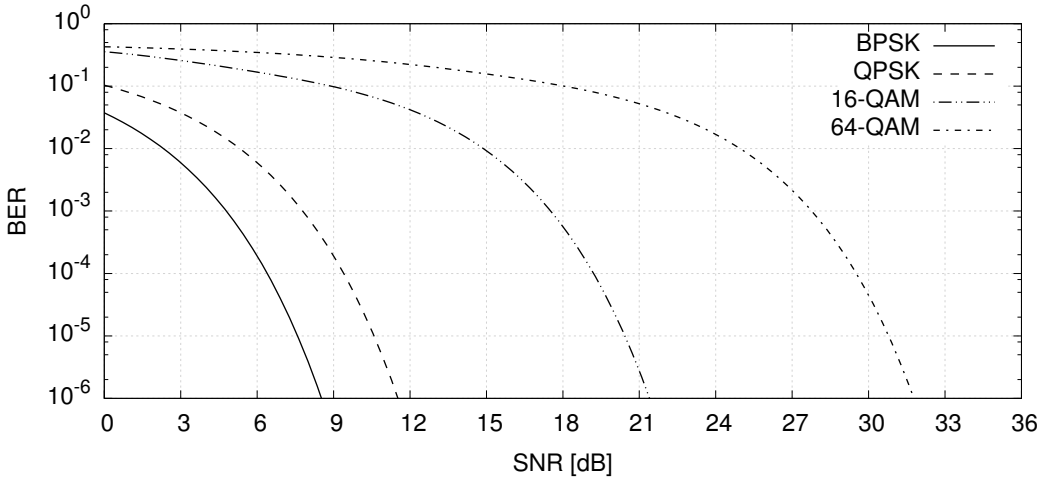


Figure 5.2 Bit error rate vs signal-to-noise ratio for several modulations.

In simple terms, E_b/N_0 gives us a metric of performance depending on power consumption. Given an energy budget of e joule and k bits to transmit, the energy budget per bit is $\frac{e}{k}$. With the same E_b/N_0, we can, for example, either send $\frac{k}{2}$ BPSK symbols spending $2\frac{e}{k}$ J on each symbol, or $\frac{k}{64}$ 64-QAM symbols spending $64\frac{e}{k}$ J per symbol. Because the symbols contain a different number of bits, more energy can be spent on each QAM symbol and still use the same energy budget. Likewise, the E_b/N_0 abstracts from bandwidth: With the same E_b/N_0, we can send a symbol at bandwidth B Hz, taking time t seconds, or at bandwidth $2B$ Hz, and, taking advantage of the increased bandwidth, doubling the rate and only spending $\frac{t}{2}$ s on the symbol.

In a practical rate adaptation scenario, both bandwidth and bit rate are fixed and cannot be changed at will. While there are several standardized bandwidths in 802.11 (from 5 MHz over the common 20 MHz in a/b/g to as much as 160 MHz in the new 802.11ac [IEEE13] standard), switching between bandwidths on-the-fly is not provisioned for. The standard defines a set of rates, which each have a fixed bandwidth and bit rate setting. Looking at the four modulations used by 802.11a/g, Table 5.3 shows that they operate at different bit rates, keeping symbol rate and bandwidth constant. Finally, while there are schemes to dynamically adapt the

Modulation	(Gross) bit rate	Bits per symbol	Symbol rate	Bandwidth
BPSK	12 Mbit/s	1	12 MBd	20 MHz
QPSK	24 Mbit/s	2	12 MBd	20 MHz
16-QAM	48 Mbit/s	4	12 MBd	20 MHz
64-QAM	72 Mbit/s	6	12 MBd	20 MHz

Table 5.3 Modulations as used by OFDM in 802.11a/g. The rates are set up so that the symbol rate stays constant. This table shows the settings for 20 MHz channels. For other bandwidths defined in the standard, the bit rate is scaled accordingly to keep the spectral efficiency η constant.

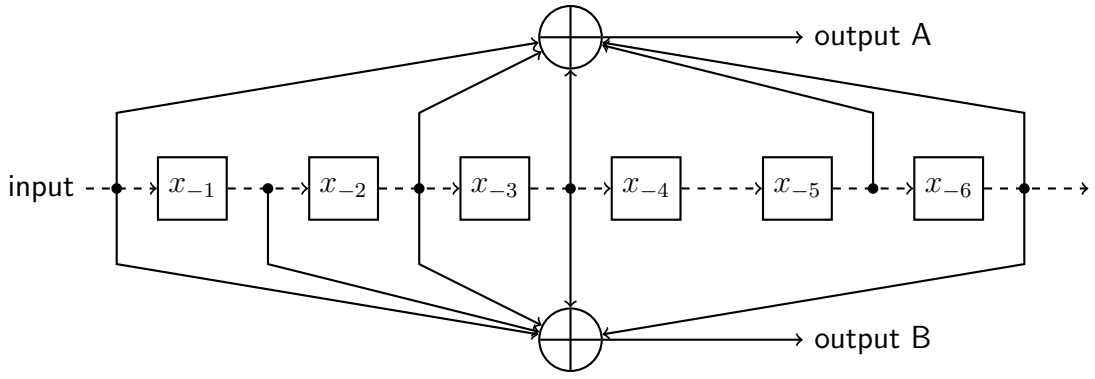


Figure 5.3 The standard $k = 7$ convolutional code, as used by 802.11. Dashed arrows denote shift direction of the shit register, solid arrows input to and output from the generator polynomials. $\oplus$ denotes exclusive-or, or addition modulo 2. Every time a bit is entered into the shift register, two encoded output bits are produced.

transmission power and thereby influencing the energy per bit and, consequently, E_b/N_0 [RKZG08,PEG12], this is not used by standard setups on consumer hardware. With all these parameters fixed, the deciding factor for correct or corrupt reception of a frame in an 802.11 network are therefore the channel conditions. These can be quantified by the SNR. By inserting the respective values from Table 5.3 into Equation 5.9, Equations 5.1, 5.4, 5.6 and 5.7 can be used to calculate the BER given the SNR. The results are shown in Figure 5.2.

This gives us an intuitive understanding of the strengths of the different modulations. As the number of bits per symbol increases, so does the susceptibility to errors (cf. Section 2.2). The higher potential throughput of those modulations is offset by their requirement for a better SNR to produce the same BER. By applying Equation 5.9, we can calculate that for the same BER, QPSK requires 3 dB more SNR than BPSK, 16-QAM 6 dB more than QPSK, and 64-QAM 6 dB more than 16-QAM.

5.2.2.2 Coding

However, this does not draw the complete picture. To increase robustness and to allow more fine-grained steps between performance levels, data is not directly modulated and sent. Instead, it is encoded in a way that increases robustness and allows recovery from a certain amount of bit errors. We will now add coding to our examination.

In 802.11, a type of convolutional coding is used. Convolutional codes transform inputs of m data bits into output of n code bits, with the overwhelmingly common case (which will be described here) setting $m = 1$. They comprise a shift register of length k, as well as n generator polynomials which define the *code rate* and robustness.

As an example, the convolutional code used in 802.11 is given in Figure 5.3. The length of the shift register is 6. Whenever a data bit is read, the new contents of the shift register are used as inputs to two generator polynomials, producing two 1-bit

Net bit rate	Code rate	Gross bit rate	Modulation	Bits per symbol	Symbol rate	Band-width
6 Mbit/s	1/2	12 Mbit/s	BPSK	1	12 MBd	20 MHz
9 Mbit/s	3/4	12 Mbit/s	BPSK	1	12 MBd	20 MHz
12 Mbit/s	1/2	24 Mbit/s	QPSK	2	12 MBd	20 MHz
18 Mbit/s	3/4	24 Mbit/s	QPSK	2	12 MBd	20 MHz
24 Mbit/s	1/2	48 Mbit/s	16-QAM	4	12 MBd	20 MHz
36 Mbit/s	3/4	48 Mbit/s	16-QAM	4	12 MBd	20 MHz
48 Mbit/s	2/3	72 Mbit/s	64-QAM	6	12 MBd	20 MHz
54 Mbit/s	3/4	72 Mbit/s	64-QAM	6	12 MBd	20 MHz

Table 5.4 The well-known bit rates of 802.11a/g at 20 MHz are a result of a combination of four modulations with three code rates.

outputs. For each input bit, two output bits are generated, that is, the code rate is 1/2. The generator polynomials each take a defined subset of the current input bit (0) and the register bits (1–6) and perform a binary XOR on them. In the case of 802.11, the first polynomial uses bits 0, 1, 2, 3, and 6; the second uses bits 0, 2, 3, 5, and 6. In binary representation, this is 1111001 and 1011011, respectively. In the definition of convolutions codes, the generator polynomials are typically given in octal notation, leading to $g_0 = 171_8$ and $g_1 = 133_8$.[51] This code has been is use since at least NASA's Voyager program, which used this code for redundancy [YV85]. To create more fine-grained steps between the code rates of 1/2 and uncoded data (which equals to a rate of 1) to offset robustness and throughput, a process called *puncturing* is used. This means that some of the encoder's output bits are deleted in a pattern that still allows decoding. These patterns are given in the form of puncturing matrices. For the two punctured rates used with 802.11, 2/3 and 3/4, the matrices are:

$$P_{2/3} = \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix} \qquad P_{3/4} = \begin{bmatrix} 1 & 0 & 1 \\ 1 & 1 & 0 \end{bmatrix} \tag{5.10}$$

This means that, for example, for the rate 3/4 code, for each 3 input bits, the second output of g_0 and the third output of g_1 are punctured and not transmitted. By combining the four modulation schemes presented in the last section with different code rates as shown in Table 5.4, we arrive at 802.11's well-known OFDM bit rates.

Due to its widespread use, the error-correcting capabilities of the above convolutional code and its most-used puncturing patterns are well-researched. The most important property to investigate the error-correcting capability of a convolutional code is the *free distance*, the minimal Hamming distance between encoded sequences. The

[51] Note that enumerating the polynomials implicitly defines the shift register length; the new bit is always assumed to be used by at least one generator polynomial, because otherwise the first element of the shift register would be unused and w.l.o.g., the register could be shortened by that element. Therefore, the number of digits in binary representation defines the shift register's length. The code rate is derived from the number of used generator polynomials.

output of an error-correcting code is chosen so that this free distance is maximized. During the decoding, the decoder will try to match the received encoded data to possible decoded sequences. As long as half or fewer bits are flipped, correct decoding is possible. If more than half of the bits are flipped, another sequence is closer to the received sequence than the correct one, leading to a decoding error. The worst-case error correcting capability is therefore $\left\lfloor \frac{d_{free}-1}{2} \right\rfloor$. This d_{free} is found by creating a state diagram of the convolutional coder, with the contents of the shift register being the vertices and the possible outputs $g_0 g_1 \dots g_n$ the edges [Vit71]. A code's state diagram therefore has 2^k vertices and $2 \cdot 2^k$ edges (because each shift register state can only change by shifting in either a 0 or a 1, and therefore only two outputs are possible from each state). Then, each edge is weighted by the Hamming weight (number of 1s) in the edge's label. Finally, the free distance is the minimum weight of a (simple) cycle through the graph with start and end vertex $\underbrace{0 \dots 0}_{k}$. This d_{free} has to be found, as well as the number of cycles $a_{d_{free}}$ that have a distance of d_{free}. This gives a good approximation of the error-correcting-performance of the convolutional code; however, for exact analysis, *all* distances d (and not only the free distance) have to be known, and their respective a_d.

In the general case, the probability that a decoding error occurs in the encoded stream at distance d and BER ρ is: [Vit71, PT87]

$$P(d,\rho) = \begin{cases} \sum\limits_{i=\frac{d+1}{2}}^{d} \binom{d}{i} \rho^i (1-\rho)^{1-i} & \text{if } d \text{ odd} \\ \frac{1}{2}\binom{d}{\frac{d}{2}} \rho^{\frac{d}{2}} (1-\rho)^{\frac{d}{2}} + \sum\limits_{i=\frac{d}{2}+1}^{d} \binom{d}{i} \rho^i (1-\rho)^{d-i} & \text{if } d \text{ even} \end{cases} \tag{5.11}$$

where in the even case, it is assumed that the decoder guesses correctly with a 50% chance if there are $\frac{d}{2}$ errors. Given a certain input code word, the probability that any uncorrectable error occurs is therefore [Vit71]

$$P_e(\rho) = \sum_{d=d_{free}}^{\infty} a_d \cdot P(d,\rho) \tag{5.12}$$

However, in most cases, the first few terms dominate the equation to such an extent that only d_{free} and possibly $d_{free}+1$ are considered. Such a modeling was used to calculate the BERs shown in Figure 5.4. Again, as in the case of considering modulation without coding (cf. Figure 5.2), the results match the intuitive understanding of the tradeoff between speed and robustness. Higher data rates requires less robust modulation and/or less robust coding. They therefore require a higher SNR to produce the same BER as more robust rates. There are a few peculiarities though: first, at very high error rates, the analytical results show crossovers between 18 Mbit/s (QPSK and rate 3/4) and 24 Mbit/s (16-QAM and rate 1/2) and, to a lesser degree, between 36 Mbit/s (16-QAM and rate 3/4) and 48 Mbit/s (64-QAM and rate 2/3), with the generally less robust modulation and coding producing a lower BER. However, this crossover disappears at larger chunk sizes than the 36 bits assumed in the figure; even at such a low chunk size, it only appears at error rates that are practically useless for transmission of data (beyond 20% BER).

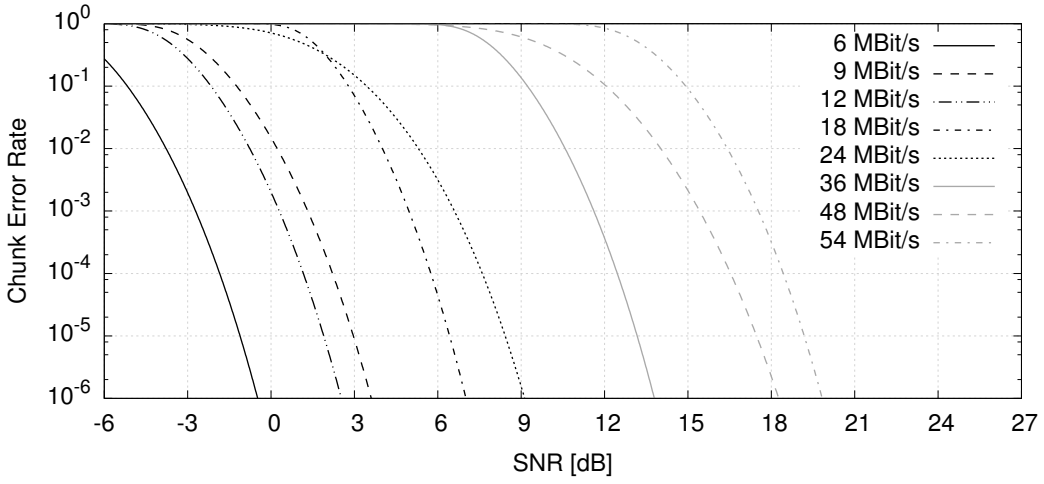


Figure 5.4 Chunk error rate, that is, the probability that at least one bit error occurs in a chunk after decoding, for the 802.11's OFDM bit rates. Chunk size is 36 bits, the smallest size that ends at a symbol boundary after modulation and coding for all rates. Note how 12 Mbit/s always outperforms 9 Mbit/s. The crossover between 18 Mbit/s and 24 Mbit/s disappears at (more realistic) larger chunk sizes.

5.2.2.3 Throughput

However, we still are not done with our observation. BER is an important metric of bit rates, but while an error-less transmission is desirable, it is not the only goal. We have seen that slower rates with more robust modulation and coding reduce error rate and increase robustness in the same channel conditions. However, if we always exclusively used these rate, communication would be very slow and introduce considerable additional delay (since frame transmissions take more time). In most scenarios, a tradeoff between robustness and speed has to be found. Throughput[52] is the typical metric employed in this case, and indeed is generally presented as the main performance metric in literature concerning rate adaptation.

BER is an important factor in the performance of a transmission, but the main deciding factor is typically considered to be the throughput that is achievable by the system. Throughput is closely related to the BER, but it also depends on the size of the data units sent that form a common correctness unit and are considered corrupted when at least one data bit is flipped (after demodulation and decoding). These data units are typically called "frames" or "packets", and their PER depends on the BER:

$$PER = 1 - (1 - BER)^n \tag{5.13}$$

[52]Some literature makes the more detailed distinction between raw throughput (amount of data that can be sent per time unit) and goodput (amount of data that can be *correctly* received per time unit). In the following, we will use the more common definition that argues that "throughput" in the preceding sense is of little use (because sending faster without any regard to receivability is always possible by using less robust modulation or coding), and use "throughput" and "goodput" interchangeably.

where n is the length of the packet in bits. In standard ARQ systems, one or more bit errors in a packet lead to a packet error, which leads to the complete packet being dropped.

From a point of error-tolerant transmissions, this does not seem to be overly relevant; after all, the whole point of error tolerance is to not discard a packet in the case of bit errors. However, error-tolerant traffic is generally expected to coexist with error-sensitive traffic. Furthermore, our rate adaptation scheme should be designed to be usable and useful even for standard traffic; thus, it will not only be an extension to Refector, but a stand-alone solution that can be used to great effect in all WLAN systems. Hence, throughput as a performance metric for rate adaptation is also important to our approach.

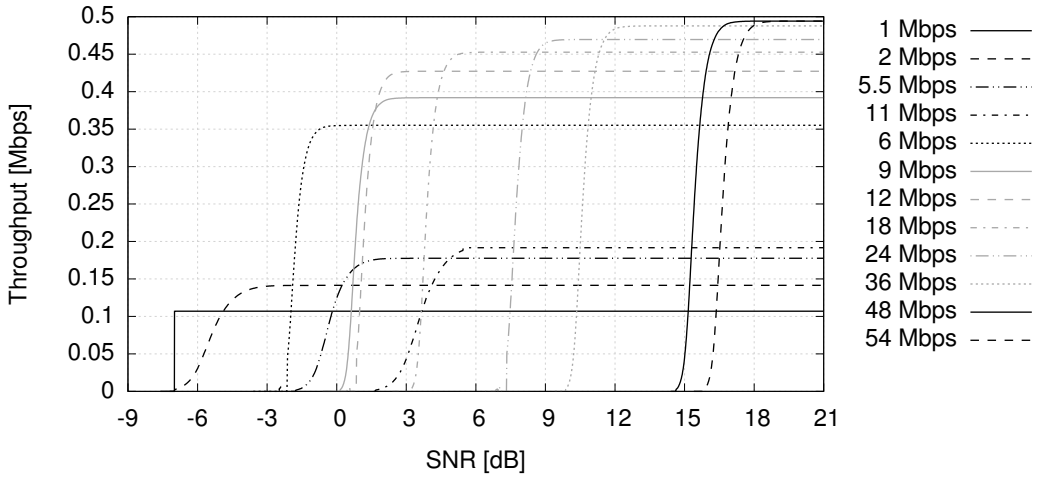
However, it is very hard to analytically derive throughput for certain MCSs, BERs or PERs, and packet sizes. A naïve approach might make it seem as if the throughput T were proportional to the PER,

$$T \propto PER \tag{5.14}$$

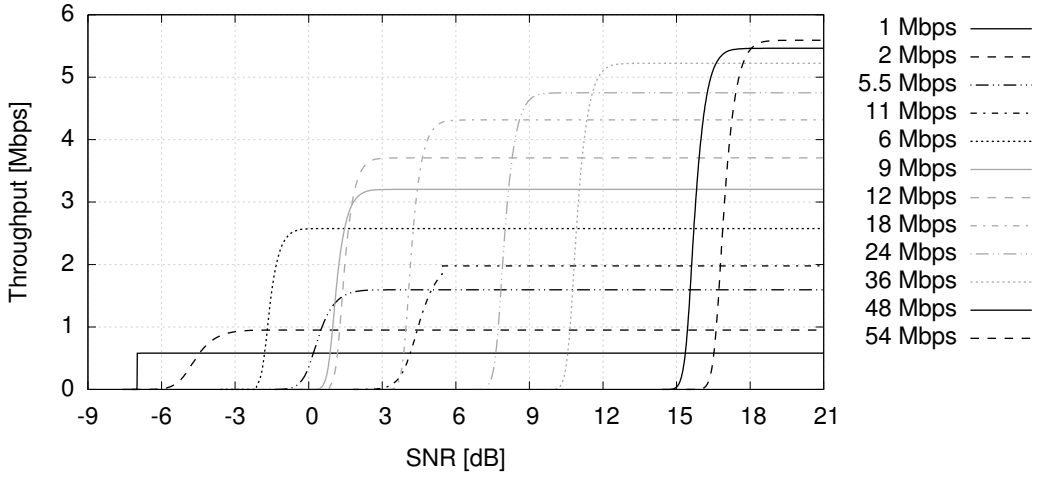
which in turn would suggest that packetization is altogether harmful to throughput, because a packet size of 1 bit produces the lowest PER, because no bits share a common fate. However, this is not the case, because the sending of each packet incurs significant overhead that is unrelated to its size. This overhead is both due to headers that are sent with each frame, and due to the fact that, for each frame, the wireless channel has to be contended for, losing additional time. There exists at least one analytic model that tries to calculate the exact interplay between packet sizes, bit rates and throughput [QCJS03]. However, it is quite complicated, and in practical experiments, the results were not completely satisfactory [Fan12], though we could not conclude whether this was due to the model or the measurements used as its input.

Instead, we will now move on to using simulation instead. As a simulator, we use ns-3 [LH06, HRFR06, ns3], a well-known and -tested simulation framework in wide use. One advantage of ns-3 in this specific use case is that its wireless models have been under close scrutiny [PH09, PH10, BRENM+10] and shown that they map real-world results very well. To produce the throughput measurements presented in the following, we created a very simple simulation setup:

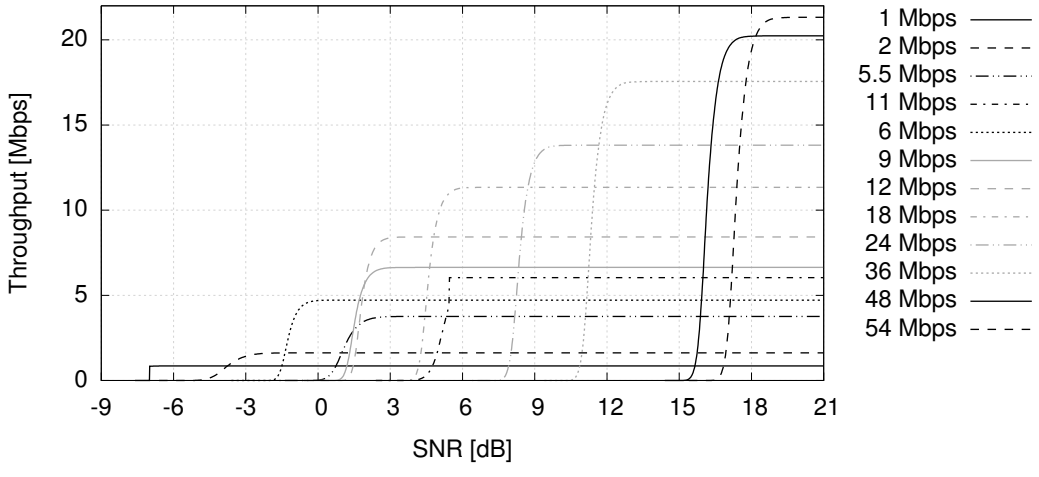
1. Two nodes, an AP sending data and a STA receiving it.
2. An application creating packets at a speed that is guaranteed to saturate the channel.
3. Use of the `FixedRssLoss` propagation loss model, which sets the channel SNR to a fixed value.
4. Use of the `ConstantRateWifiManager`, which disables all rate adaptation and sends all frames at a specified data rate.
5. 60 seconds of simulation time.



(a) Throughput in Mbps for an application-layer packet size of 20 bytes.

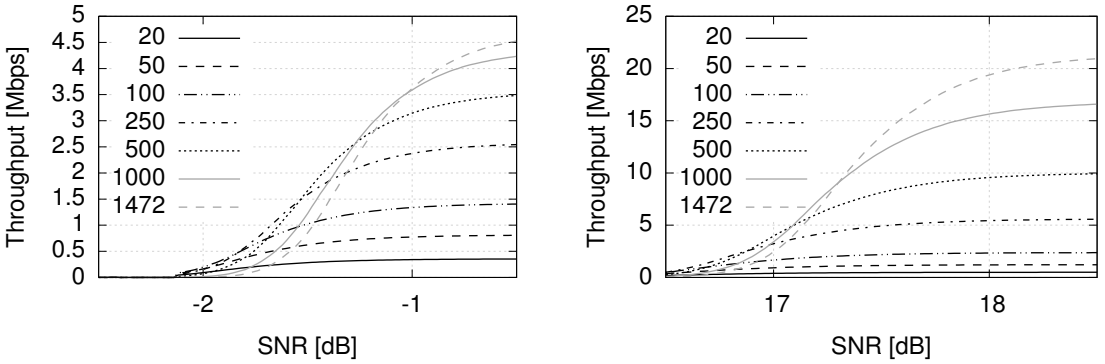


(b) Throughput in Mbps for an application-layer packet size of 250 bytes.



(c) Throughput in Mbps for an application-layer packet size of 1472 bytes.

Figure 5.5 Simulated (application-level) throughput for all 802.11g rates at different packet sizes. Crossover points between the rates are clearly visible. The two faster DSSS rates (5.5 and 11 Mbit/s) are not competitive and outperformed by OFDM rates in every scenario. 9 Mbit/s is practically useless due to its extremely small area of top performance. The sharp cutoff of 1 Mbit/s at -7 dB is due to the energy detection threshold of the simulated network controller.



(a) Throughput at 6 Mbit/s (b) Throughput at 54 Mbit/s

Figure 5.6 Simulated (application-level) throughput for two 802.11g rates at different packet sizes. While the absolute throughput strongly depends on packet size, the SNR region in which a rate switches from unusable to top performance stays nearly the same.

6. Several simulation runs with different application-layer packet sizes: 20, 50, 100, 250, 500, 1000, and 1472 (a typical Maximum Segment Size (MSS) for Internet traffic) bytes. Several runs for each setup were not necessary, because results varied insignificantly over several repetitions.

Results for some packet sizes are shown in Figure 5.5; the omitted results can be approximated from the presented results. These results show that throughput strongly depends on packet size: At a payload of 20 bytes, throughput caps out at 0.5 Mbit/s even for the fastest rates under very good conditions, while at 1472 bytes, throughput exceeds 20 Mbit/s. The large difference between these numbers and the nominal bit rates is due to various kinds of overheads, as mentioned above: inter-frame spaces, preambles, and protocol headers. In the investigated case, the latter added 30 bytes for 802.11 MAC (26 Bytes header including 802.11e QoS extensions, 4 bytes CRC footer), 8 bytes for SNAP, 20 bytes for IPv4 and 8 bytes for UDP, which on its own amounts to 330% overhead to each 20-byte packet.

Each of the graphs in Figure 5.5 shows a characteristic pattern over S-shaped curves that ”cross over” at certain points. These “crossover” points between rates define where a switch from one rate to another is beneficial. In better channel conditions, the faster rate can show its advantages of transmitting data faster; in worse channel conditions, this advantage is overshadowed by the higher BER, which offsets the speed advantages, and makes a slower but more robust rate more beneficial.

Careful comparison of the crossover points in 5.5 shows that, while absolute throughput varies greatly depending on packet size, the SNR value at which one rate starts to outperform another varies little. For example, the crossover between 12 Mbit/s and 18 Mbit/s is at 4.64 dB for 20-byte packets and at 4.82 dB for 1472-byte packets. This can also be seen in Figure 5.6. Instead of showing all rates for a certain packet size in one graph as in Figure 5.5, each graph shows the performance of all packet sizes for a certain rate. The difference between full-performance throughput and no throughput at all always falls into the same SNR region and and spreads less than a 2 dB interval. This observation is important because it means that packet size is

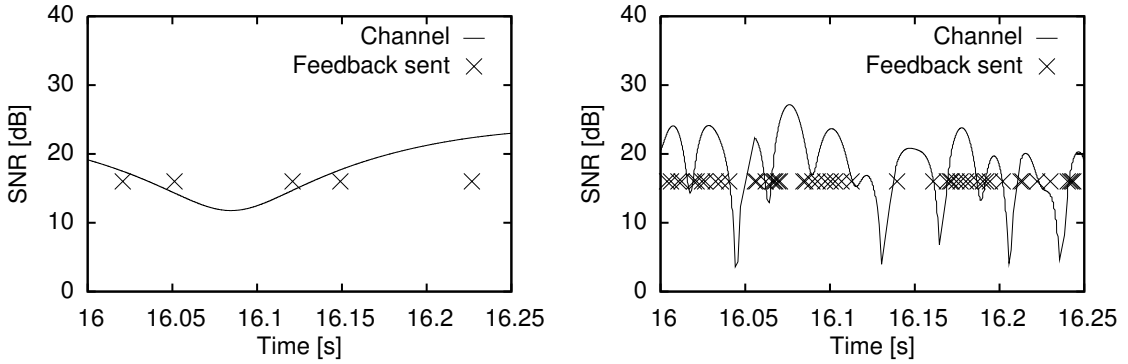


Figure 5.7 Feedback frequency adapts to the speed of the channel. In a slow channel, few rate changes are required and hence little feedback is. In a fast channel, feedback is sent much more frequently to constantly adapt rates to the strongly varying channel conditions.

negligible as a factor, and we can define a single set of crossover points that will produce good performance for all packet sizes.

We have now finally answered our question of when to send our on-demand rate adaptation feedback. Feedback needs to be sent when channel conditions change enough to cross a crossover point. At this time, the receiver needs to inform the sender that another rate than the currently used one provides higher performance (throughput). At other times, the receiver can abstain from sending feedback and leave the channel open for others to use (e.g., for the sender to send more data, or for other participants). This has the advantage that the overhead adapts to the channel speed. In stable channel conditions that do not require rate changes, no feedback is sent, and the time saved can be used for sending more data. In strongly varying conditions, many feedback frames will be sent, so that the sender can adapt to the optimum rate at any given time. Figure 5.7 illustrates this via a quarter-second excerpt taken from the simulative evaluation we will present later in Section 5.5.

5.2.3 How to Send Feedback: A New MAC Frame Type

After describing and answering the question of when to sent feedback, we still have to solve the problem of how to send it. As mentioned before, we will not use acknowledged traffic for error-tolerant transmissions. We also do not want to rely on sufficient error-sensitive (and thus acknowledged) traffic to be concurrently present at any given time, which could be used to adapt rates correctly for both acknowledged and non-acknowledged traffic.

We therefore have to transfer rate adaptation information in another way. To do this, we extend the 802.11 standard by defining a new frame type called a *feedback frame*. The 802.11 standard already defines a number of frame types, which are identified in the MAC header via two fields, the *type* and the *subtype* fields. The former is 2 bytes long and defines the well-known classes of frames: data frames, management frames and control frames. The latter is 4 bytes long and defines a

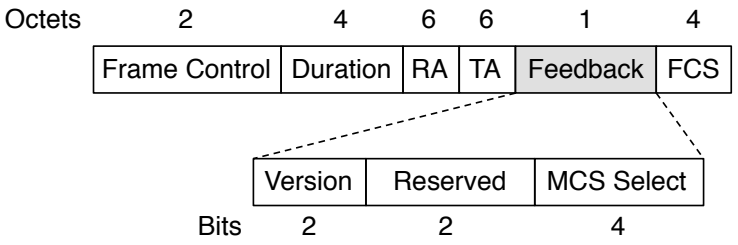


Figure 5.8 OFRA's feedback frame follows the layout of a standard 802.11 frame. The actual feedback information is encoded in the four bits denoted "MCS Select".

number of different frames within each class. Some examples of management frames (type = 00) are association requests (subtype = 0000) and responses (0001), or beacons (1000). Examples for control frames (type = 01) are RTS (subtype = 1011) and CTS (1100). Data frames also come in different subtypes, especially for data frames with piggybacked control information.

The difference between management and control frames is that management frames deal with the connection of a participant to an 802.11 network. Frames that contain information pertaining to connection setup and teardown are management frames. Control frames, on the other hand, facilitate data transfer between participants of the network. As such, it is clear that our on-demand feedback belongs to the category of control frames, because it ensures that data transfers between participants are possible and efficient.

Adding a new control frame subtype is possible because the 4-bit address spaces allows 16 such subtypes to be defined, but only 9 are used (in the current, 2012 version of the 802.11 standard [IEEE12b]). Thus, we can define our feedback frame to be one of the remaining 7 options which are reserved for future use. This has the advantage that participants supporting OFRA can decode the frame, while participants that do not support OFRA will discard the unknown frame type. Creating and handling such a new frame type is not only theoretically possible, but also practically implementable, even on typical commodity hardware [Göt13].

Figure 5.8 shows the layout of our feedback frame. The upper row contains fields requested by the standard to create a valid frame. The frame control field, among other information, denotes the frame as a control frame of the feedback frame type. The duration field informs all listeners about the time the channel will be occupied by this frame. RA and TA denote the receiver's and the transmitter's address. FCS is the frame check sequence, a 32-bit CRC. The actual information that we feed back is contained within a single byte. Its content fields are shown in the lower row of the figure. The version field was added to allow future extension of the feedback frame format, for example, to extend the presented scheme for full 802.11n [IEEE09] and 802.11ac [IEEE13] support, which provides more rates to choose from, as well as schemes such as MIMO and channel bonding. In its current, initial version, it is set to 0. The next two bits are not used to transmit any information and are defined

as "reserved for future use".[53] Finally, the choice of rate is encoded in four bits. The 16 possible values in this field allow us to encode all 802.11g rates.

On-demand feedback is thus implemented by the receiver checking the SNR of frames received from another station, as well as the bit rate at which these frames were sent. If the SNR suggests that a different rate than the one used (by comparing it to the crossover points identified in Section 5.2.2), the receiver sends a feedback frame to the sender of the data, denoting the new optimal rate in the MCS Select field. When a sender receives a feedback frame, it will remember the rate choice signaled in that frame, and use that rate for future communication with that station. Note that feedback frames are not acknowledged. Correct reception of a feedback frame is implicitly signaled by a change in bit rate. If the receiver notes that rates have not changed, it will send another feedback frame. To increase the robustness of feedback frames, they are always sent at the lowest basic rate (typically 6 Mbit/s for OFDM transmissions). For such a small frame, using higher rates provides very little absolute speed benefit (cf. Figure 5.5a, and keep in mind that 20 bytes of application payload produce frames of 86 bytes, or almost 4 times larger than a feedback frame), so the robustness provided by using the lowest basic rate comes at a small price, and virtually guarantees correct reception of the frame at medium and high SNRs.

Note that we send information about the preferred rate in our feedback frames. Another possibility would have been to send the sensed SNR value to the sender. It is mainly for practical reasons that we decided against this: while we have SNR values easily available in simulation, many hardware devices do not provide this metric, for the problems that come with its measurement. Theoretically, a correct SNR measurement would need to measure both signal and noise at the same point in time. In simulation, this is trivial, because both values are calculated by the simulator, and only afterwards combined into an SNR value. Real hardware, however, will have to deduce these values from the received signals. In the best case, it can try to deduce them from the known preamble pattern of frames. Other than that, it can only measure the received power of the signal, and compare it to power measurements during earlier times in which no frame reception was in progress. As a result of these problems, most hardware only provides signal quality metrics in the form of Received Signal Strength (RSS). Worse, especially on older hardware, this RSS is often presented in an arbitrary metric. For example, while Atheros cards typically report the RSS in dBm, older cards often used an arbitrary so-called Received Signal Strength Indicator (RSSI) scale that might, for example, give values on a scale of 0 to 100. Therefore, taking the reception quality indicator (SNR or RSSI) and sending it back from the receiver to the sender might not help the sender at all. Unless both sender and receiver are of the same hardware type, there is no guarantee that their interpretation of the numbers is identical. Thus, sensible rate adaptation cannot be ensured anymore. This is why the receiver sends a rate instruction in the feedback frame, instead of informing the sender about the measured signal quality.

Feedback obviously requires high timeliness. The longer it takes for the sender to correctly adapt their rate to the optimum one, the more the performance suffers.

[53] This is done to pad the feedback information to a full byte, because there are no provisions in 802.11 to send partial octets.

Either data is sent at a lower rate than it should be, or (potentially even worse) at a higher rate, which increases the risk of bit errors. We therefore need to make sure that feedback is created and sent, as well as received and interpreted in a timely fashion. The fast creation of feedback is guaranteed by the fact that the receiver checks every received frame for SNR and bit rate, and immediately creates a feedback frame if necessary. The sender will react to a received feedback frame immediately by changing the rate to the one instructed by the receiver. The main delay in this scheme is the time the feedback frame potentially spends in the network adapter's send queue on the (data) receiver's side. Especially in the case of bidirectional or concurrent traffic, the queue might be filled with data frames. If the feedback frame were to be appended to that queue, this might lead to a considerable delay in feedback sending. However, because we decided to use No-ACK for our data transmissions, and because this scheme was introduced by the 802.11e extension [IEEE05], we can make use of all the other functionalities introduced by this extension. One of those is the concept of several queues with different priorities to better support QoS. Instead of one single queue, 802.11e defines four queues (cf. Section 2.5), with the high-priority queues having a higher chance to contend for the channel. By putting the feedback frames into the highest-priority queue, we create a fast track that allows our feedback frames to overtake standard data frames that are added to the normal-priority queues, and increase the speed at which our feedback frames are sent out.

To summarize, we have explained the problem of scarcity of information, and how our feedback-based system can circumvent it (Section 5.2.1). We showed when it is beneficial to switch between rates, and that feedback should be sent on demand when such a crossover point is reached (Section 5.2.2). Finally, we explained how feedback can be encoded for transport between receiver and sender, and how such transport can be done in a timely fashion (Section 5.2.3).

5.3 Related Work

Because practical usage of IEEE 802.11 requires the use of rate adaptation, but the standards do not define a rate adaptation algorithm, such algorithms have been proposed in literature in large numbers. Thus, it is impossible to give a complete overview of the whole field of rate adaptation in this section. However, typical examples of different types of algorithms will be presented. They can be categorized in a myriad of ways, but for this work, a useful way is to differentiate between sender- vs. receiver-based as well as frame- vs. SNR-based approaches. These can be combined to give us a two-dimensional space. Table 5.5 categorizes related work amongst those lines.

Frame-based Approaches

Virtually all algorithms currently used in practice fall into the frame-based, sender-based category. One reason for this is the ease of implementation and guaranteed

	Frame-based	SNR-based
Sender-based	ARF [KM97] AARF [LMT04] CARA [KKCQ06] RRAA [WYLB06] minstrel [minstrel]	CHARM [JWS08] BRAVE [DD12]
Receiver-based		RBAR [HVB01] OAR [SKSK02] SoftRate [VBJ09] AccuRate [SSCN10] RAM [CGQ12] **OFRA [SHP+12]**

Table 5.5 Rate adaptation algorithms can be distinguished along the two dimensions of sender- vs. receiver-based and frame- vs. SNR-based approaches. This table shows how OFRA and related work fall into those categories.

hardware support: all the algorithm needs to do is to keep track of whether data transmission were successful, something that all commodity cards signal back to the operating system. It also does not need to rely on signal quality measurements and the various ways these can be acquired and presented (as discussed in the previous section).

Typical examples of frame- and sender-based approaches include ARF [KM97], one of the first widely used rate adaptation algorithms; its extension AARF [LMT04]; CARA [KKCQ06]; RRRA [WYLB06]; and minstrel [minstrel], the current standard algorithm used by the Linux kernel. Because these algorithms are used as comparison algorithms for OFRA in our evaluation, we defer explanation of their behavior to Section 5.5.3. As a general rule, these algorithms count transmission successes and failures at different bit rates and decide, based on this, which rate to use for the next frame that is to be sent. They differ in details such as how aggressively they age their information, and whether they use RTS/CTS in certain situations.

One behavior that all these frame-and-sender-based algorithms share is *probing*. Since they do not receive any explicit quality information from the receiver, they need to infer quality from transmission successes or failures. However, this only allows the recognition of degrading channel conditions: a certain rate will produce increasing numbers of transmission errors, so the algorithm will choose a lower rate. Conversely, if channel conditions improve, there is no direct way to recognize this: transmission failures will become increasingly uncommon, but this on its own does not guarantee that the next higher rate will produce satisfactory results. Compare Figure 5.5c, where a rate of 36 MBps can produce near-perfect results for approximately 5 dB before the next higher rate of 48 MBps becomes usable. Therefore, such algorithms need to occasionally send frames at a higher rate to probe whether this rate produces satisfactory results and should be switched to. Especially in slowly-changing channel conditions, this can lead to a comparatively high error rate, because regular probing leads to a number of transmission failures. Conversely, in fast-changing channels,

the probing of such algorithms is susceptible to being too slow in its reaction: unless the algorithm probes aggressively and risks switching to higher rates early, channel conditions will have changed significantly by the time probing suggests a higher rate to be used. This can lead to either high error rates, or to being too conservative and constantly sticking to low rates.

The fact that Table 5.5 lists no receiver-based, frame-based algorithms is because such a combination makes little sense. The idea of a receiver-based algorithm is to feed back information to the receiver that it does not have yet. However, at least in ACKed traffic, the information whether frames were received or not is already conferred by acknowledgments. Simply feeding back this information, or information solely based on it, would therefore be tantamount to useless duplication of information. Furthermore, on the receiver's side, the SNR of received frames is known. This information gives much more detailed insight into channel conditions than the simple binary received—not received. It would make little sense to disregard this richer information.

SNR- and Sender-Based Approaches

SNR-based algorithms that do their rate decision at the sender's side rely on channel reciprocity, that is, that the channels exhibits symmetric properties: if a node receives frames at a certain SNR from another node, it expects its own frames to show the same SNR when sending it to that node. These algorithms combine several advantages: first, they use the richer information available from SNR. Second, they do not have to feed back information from the receiver to the sender. This means that they are self-contained and can work on their own without any support from the receiver.

However, there are also several downsides to this approach. Most importantly, the assumption that channels are always symmetric does not necessarily hold in practice. It has been shown [CGQ09] that, especially in mobile scenarios, channels can exhibit asymmetric behavior, in which case this class of algorithms cannot adapt the rate properly. Furthermore, like frame-based approaches, these algorithms, by design, cannot properly adapt No-ACK traffic, especially if they rely on measuring the SNR from ACK frames. If they rely on data frames from other stations (solely, or in addition to ACKs), they will be able to collect SNR information and switch No-ACK data traffic; however, unidirectional such traffic will not provide any SNR feedback to the sender, at which point the algorithms cannot adapt the rate any more.

Two popular examples of this type of rate adaptation are CHARM and BRAVE. Like all SNR-based algorithms, CHARM [JWS08] uses a lookup table to decide which rate to use for a certain SNR. This SNR is the weighted average of the last received SNR a history of previous SNR values. One idiosyncrasy of CHARM is that it allows online calibration of its lookup tables by keeping track of how often frame transmissions failed at a predicted rate. However, while it is clear how crossover points in CHARM's lookup table can be adjusted conservatively (reducing them to choose lower rates), it is not clear how the algorithm recovers from them and can

adjust them in the opposite direction: since it does not employ probing, higher rates are not chosen optimistically, so there is the risk that the algorithms "digs itself into a hole" of underselection.

BRAVE [DD12] similarly uses a lookup table, but in addition comes with two variants of its rate decision algorithm, a conservative (termed "SAFE") and an optimistic (termed "AGGRO" for aggressive) version. It switches between these two variants by counting frame successes, which in effect makes it a hybrid algorithm that mostly relies on SNR, but also uses frame-based concepts. BRAVE's lookup table itself is extremely simple, only using three entries instead of one for every rate. This is offset by specifying several rates in each table entry, which are tried successively on frame errors. This means that BRAVE, more so than other sender-based, SNR-based algorithms, depends on acknowledged traffic. Using it to send No-ACK traffic will either lead to sending out each frame multiple times, or always choosing the first (highest) rate in each of its three lookup table entries, leading to strong overselection.

SNR- and Receiver-Based Approaches

Receiver-based rate adaptation algorithms are based on the observation that the sender knows the current channel conditions at its position, and therefore which rate is optimal at any given point in time. However, it needs to feed back this information, preferably shortly before a data frame is sent by the sender. This shows two problems that every receiver-based algorithm needs to tackle, and which were described for OFRA in the previous sections (Sections 5.2.2 and 5.2.3): when should feedback be sent so that its information is available and not outdated, and how should it be sent?

Early receiver-based algorithms used RTS/CTS exchanges to solve the problem of timely and accurate feedback. For example, in RBAR [HVB01], the sender sends an RTS, to which the receiver answers with a CTS. This CTS contains information fro the sender about the desired rate, which is done by changing the contents of the CTS frame from the standard-mandated layout. However, the authors claim that these changes are backwards-compatible to standard RTS/CTS frames. The RTS/CTS exchange is done before each data frame. Therefore, while the feedback information is very current, the overhead introduced by RBAR is very large. In addition to the large overhead, there is also the problem of channel reservation. The sender already reserves the channel for a certain amount of time in its RTS frame. However, at this point in time, it does not know yet how much time it will take to transmit the data frame, because it does not know which rate to use, because that is only signaled in the CTS frame. It therefore needs to conservatively reserve as much time as the frame would take at the lowest rate. In many cases, the channel will therefore stay reserved, but unused if a higher rate is chosen, compounding the efficiency problems of RBAR.

OAR [SKSK02] ameliorates this latter problem by adding a burst sending mode. If an RTS reservation is long enough that, at the chosen rate, more than one data frame fits into the reservation, several are sent during that time. Note that this is different from Block ACKs as defined in the IEEE 802.11e extension [IEEE05] years

later, in that every frame within this burst is acknowledged independently. However, it still requires the overhead of the RTS/CTS exchange.

RAM [CGQ12] solves the problem of high RTS/CTS overhead by signaling its rate decisions in the ACK frame following a data frame. This is done by sending the acknowledgment at the desired data frame rate, instead of at the standard-mandated base rate. While this is less accurate than the RTS/CTS approach (because the time between an ACK and a subsequent frame is larger than between a CTS and data frame following it), it solves the problem of the massive overhead which is introduced by the earlier schemes. This is all the more important because, with increasing data rates, the static RTS/CTS overhead leads to an increasing relative overhead. When those algorithms were proposed in the early 2000s, original 802.11 products (which only supported rates of 1 Mbit/s and 2 Mbit/s) were still widespread and the newer 802.11a and 802.11b standards had just been ratified. With higher rates, the static RTS/CTS meant higher and higher potential throughput losses. Since RAM does not use its own feedback frames, but instead encodes the feedback information into the data rate at which the ACK is sent, it has a lower overhead than OFRA. However, sending ACKs at high data rates increases the risk that these become corrupted or lost, leading to unnecessary retransmissions. Furthermore, such a scheme is obviously incompatible with No-ACK transmissions, in which no ACKs are sent that could signal the feedback.

SoftRate [VBJ09] follows an approach similar to OFRA in how it sends feedback: it also employs specially crafted feedback frames. However, the rate adaptation algorithm goes one step further in how it collects frame quality information. Instead of relying on an SNR or RSS metric, it uses detailed physical layer per-bit information, so called soft information, that it receives from SoftPHY [JB07], which SoftRate's authors presented in an earlier paper. Such soft information does not only signal which bit the PHY decoded and demodulated from the received wave pattern, but in addition a value that denotes the confidence of the PHY in the bit's correctness, that is, the probability with which the bit is 0 or 1.[54] This high amount of information leads to potentially better rate adaptation choices, especially since SoftPHY also employs frame postambles in addition to the standard preambles to aid in detection of interference and collisions (as opposed to low-qality channels). However, this also means that SoftRate is not directly compatible to standard 802.11. In addition, the used soft information is not provided by PHY implementations on commodity cards; in fact, the authors had to implement their algorithm on a USRP [USRP] with GNU Radio [gnuradio], a software-defined radio, and then struggled with its performance.

AccuRate [SSCN10] follows a very similar approach to SoftRate, but instead of using per-bit soft information, it goes one step further and derives its quality metric from per-(PHY-)symbol information about In-Phase and Quadrature deviation from the optimum values. As a result of the similarity to SoftRate, AccuRate shows the same disadvantages: it cannot be used on commodity hardware and loses full compatibility to standard 802.11.

[54] We explained the concept of soft information in more detail in Section 3.3.1 when we presented ISCD as use case for error tolerance.

Finally, one conceptual downside of receiver-based rate adaptation algorithms is that they require support on both sides of the communication. Sender-based approaches collect all required information locally, for example, by counting ACKs and their absence, or by measuring the SNR of received frames. Receiver-based approaches need logic on the receiver side to measure and feedback this information, and on the sender side to interpret this feedback and act accordingly. However, in our use case scenario, this downside is not problematic. Since we assume a deployment in which both sides of the communication are under control of the user (home or small business scenarios), deploying a receiver-based scheme is not as problematic as in other scenarios. In fact, we already require some changes to the WLAN AP to properly support coexistence of error-tolerant and error-sensitive traffic (cf. Section 3.5). Hence, we can deploy a receiver-based rate adaptation algorithm such as OFRA without introducing any further limitations on the applicability of our error-tolerance scheme.

5.4 Implementation

After the conceptual designing of OFRA, we decided to implement and test it in a network simulator. This has numerous advantages, among them the ease of creating network topologies and scenarios without cumbersome deployment of real hardware, and full control over the wireless channel to investigate OFRA's behavior over a wide range of environmental settings.

The decision to use ns-3 was based on several properties of that simulation framework. Besides the fact that a great number of rate adaptation algorithms had already been implemented for ns-3, which would ease our comparative evaluation, and that, as mentioned in Section 5.2.2.3, the wireless models have been scrutinized heavily [PH09, PH10, BRENM+10], one main advantage of ns-3 over many other simulators is that its simulation models closely approximate implementations in real systems. An implementation in ns-3 therefore already gives hints towards potential implementation problems in real systems. ns-3 also comes with a detailed implementation of the 802.11 MAC and PHY, which has all parts that are required for this work completely modeled.

802.11 rate adaptation algorithms are implemented in ns-3 by deriving from the abstract base class `WifiRemoteStationManager`. It provides all necessary event methods that are called by the lower and upper layers to trigger rate adaptation; for example, cases of successful or failed receptions of Data frames. Whenever a node is created in the simulation, the creator decides which rate adaptation scheme a node will use. Afterwards, whenever the node communicates with another network participant for the first time, an instance of the rate adaptation class will be created. In the typical case of infrastructure WLAN (which we investigated), STAs will only have one communication partner, the AP. Only the AP will have to manage multiple `WifiRemoteStationManager` instances.

The `WifiRemoteStationManager` is embedded into a framework of classes that form ns-3's implementation of the 802.11 MAC, as is shown in Figure 5.9. `WifiMac`

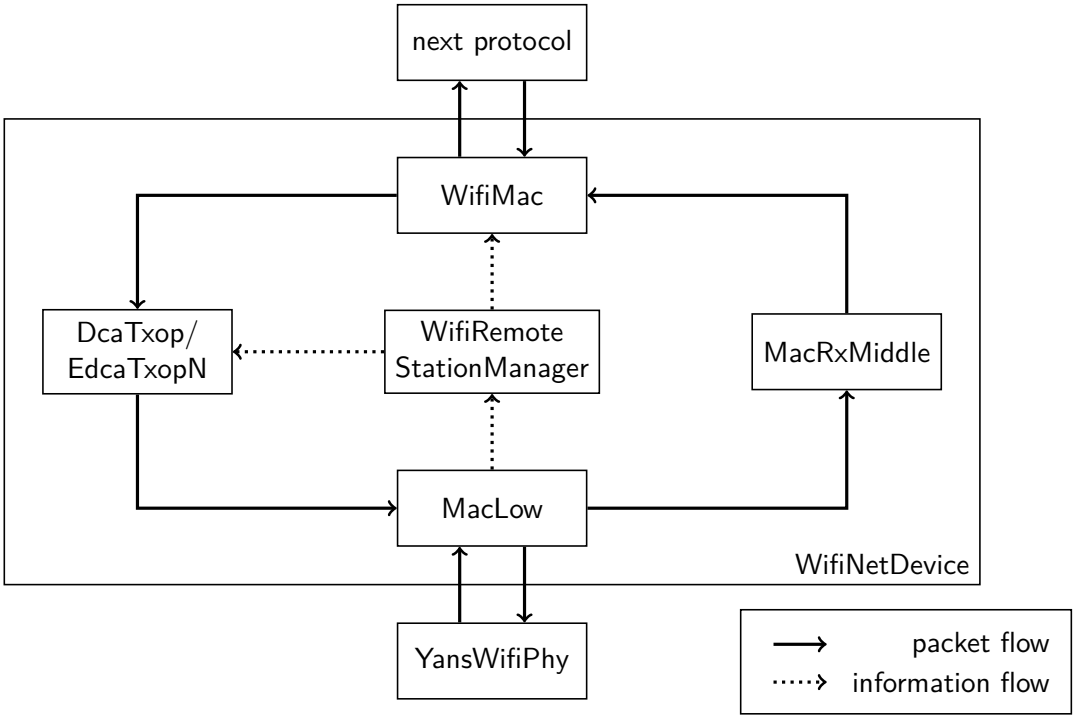


Figure 5.9 Packet and information flow through ns3's 802.11 MAC implementation. `MacLow` checks for transmission successes or failures and informs `WifiRemoteStationManager`'s rate adaptation, which in turn instructs `DcaTxop` or `EdcaTxopN` on which rate to use. We add an additional information stream to `WifiMac` to instruct it to send feedback frames when appropriate.

implements high-level management such as network association, and is the interface to the network layer. Frames are then sent to `DcaTxop` or `EdcaTxopN`, depending on whether the 802.11 extensions are used. This class deals with enqueueing frames as well as correct contention and backoff behavior. Once the channel has been claimed, frames are forwarded to `MacLow`, which forms the interface to the PHY and deals with transmission and reception of frames within a contention-free period, that is, ACKs for Data frames or RTS–CTS–Data–ACK exchanges. Received Data frames are passed on to `MacRxMiddle`, whose main job is filtering of retransmissions (in case an ACK got lost and a frame was received multiple times) and reassembly of fragmented frames.

In its standard implementation, `WifiRemoteStationManager` is purely about information management; `MacLow` informs it about transmissions successes or failures, and `DcaTxop/EdcaTxopN` will request a rate decision whenever a frame is prepared for sending. We extended this so that `OfraRemoteStationManager` can trigger the sending of feedback frames by instructing `WifiMac` to do so. Furthermore, `MacLow` was extended to recognize feedback frames and pass the contained information to `OfraRemoteStationManager`.

The actual decision when to send feedback is done by monitoring the SNR of received frames, and by comparing those SNRs to the crossover points as defined in Section 5.2.2.3.

5.5 Evaluation

Before we present results from our simulation, we need to describe the setup that we used to parameterize our simulation. Because the performance of rate adaptation algorithms is strongly dependent on channel conditions, it is especially important to note the choices made to assess the significance and validity of the presented results.

5.5.1 Simulation Model

In ns-3, channel conditions can be influenced by several models at the same time. This approach is typically used to combine (static) path loss models with (dynamic) fading models. For our simulations, we used the simple log-distance model to account for path loss:

$$PL = PL_{d_0} + 10\gamma \log_{10} \frac{d}{d_0}$$

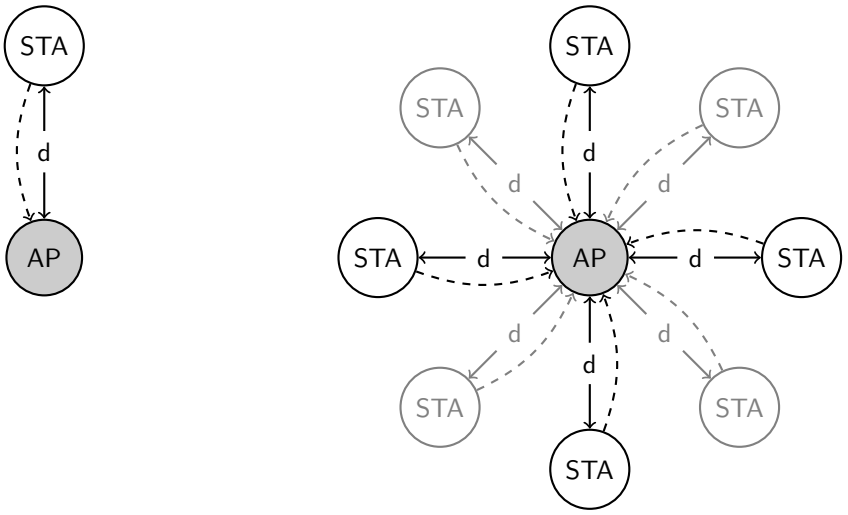
with PL being the path loss in dB, PL_{d_0} the path loss at a reference distance d_0, d the actual distance between sender and receiver, and γ the path loss exponent that governs the amount of loss over distance. For example, for $\gamma = 2$, the log-distance model produces the basic free-space path loss model with its loss proportional to the square of the distance. While this model is very simple, it is sufficient for our purposes. Since we are mainly interested in the channel dynamics, the log-distance model merely gives us baseline SNR number depending on distance, while the main influence on rate adaptation performance is given by the fading model.

To model fading, we employed a stochastic sum-of-sinusoids model [WPY07]. Such a model is especially suited to model OFDM behavior due to its use of correlated Rayleigh fading channels to simulate frequency-selective fading [SSGA10], which otherwise is not modeled in ns-3.

5.5.2 Simulation Setup and Topology

The focus of all our work, which is on small deployments, such as in homes or small companies, is also reflected in the topologies we investigated. In all our setups, one AP was the designated receiver for data sent from 1, 4, or 8 STAs (upstream scenario). We investigated this scenario because, as opposed to the AP sending data to the STAs (downstream), the upstream scenario encounters hidden station problems if the STAs are sufficiently distant from one another. The third option, communication between STAs in the same infrastructure network, is merely a combination of the two previous scenarios, because STAs never communicate directly with each other, instead sending their data to the AP which then relays it. There is hence little additional insight in this scenario.

The topology setups are shown in Figure 5.10. The STAs were equidistantly distributed on the circumference of a disc with radius d and origin on the AP. The



(a) single-node (b) multi-node (4- and 8-node) topology

Figure 5.10 In the single-node topology, a single STA sent traffic to the AP. In the multi-node topology, 4 or 8 STAs were equidistantly spaced along a ring of radius d, so that each STA had the same distance to the AP, and each had the same neighborhood topology of STAs. Dashed lines denoted data traffic flow direction (direction of DATA frames), gray STAs those that are added in addition to the black STAs in the 8-node topology.

setup was evaluated for radii d of 10, 20, 30, 40, 50, 60, 70 and 80 m. However, for reasons of lucidity, we will only present results for 10, 30, 50 and 70 m; there is no irregular behavior for the intermediate distances, while at 80 m, intermittent association losses of the STAs to the AP produced results that were skewed and provided little insight.

To model different channels, we initialized the sum-of-sinusoids with three different Doppler shifts that model environmental speeds. While the stations are immovable within the simulation (and with respect to the log-distance path loss model), we can interpret the environmental speeds as constant-speed mobility of the stations, or simply as more challenging channel conditions. In the former case, our Doppler shift values correspond to environmental speeds of 0.72, 4.32 and 14.1 km/h. Just as in the distance case, we will limit ourselves to presenting only the case of 0.72 km/h (from here on termed "slow" channel) and 14.1 km/h (termed "fast" channel).

Results presented in this evaluation section were aggregated from 20 repetitions of each scenario. Following the advised practice, ns-3's MRG32k3a pseudo-random number generator [L'E99] was initialized with its standard seed and incrementing run numbers, which change the RNG state in a way that guarantees no overlaps in the generated random number streams [ns3man]. Confidence intervals denote 95% confidence.

5.5.3 Comparison Algorithms

For our performance evaluation, we compared OFRA with several well-known rate adaptation algorithms:

- **ARF** [KM97] (Auto Rate Fallback) was one of the first widely used rate adaptation algorithms. The algorithm increases the rate after 10 consecutive successful transmissions and decreases it after 3 consecutive transmission failures (or immediately falls back if the first "probe" frame at the higher rate fails). This leads to slow adaptation in fast channels and sawtooth behavior in slow and stable channels: after the optimum rate is found, ARF will repeatedly try the higher rate, which leads to transmission failures and a drop back to the original rate. Nevertheless, as a baseline algorithm, it is popular for comparisons.

- **AARF** [LMT04] (Adaptive ARF) is an extension of ARF that dynamically adapts the numbers of required consecutive transmission successes or failures. In a stable channel, AARF displays a much subdued sawtooth pattern, by requiring a high number of transmission successes before testing the next higher rate.

- **CARA** [KKCQ06] (Collision-Aware Rate Adaptation) The main difference of CARA over previous ARF rate adaptation algorithms is the introduction of a dynamical switching scheme and enables or disables RTS/CTS for data transmissions based on previous transmission errors. Thus CARA aims to differentiate transmission errors due to channel effects from transmission errors due to collisions.

- **RRAA** [WYLB06] (Robust Rate Adaptation Algorithm) switches between rates by calculating a loss ratio over a sliding window whose size depends on the currently used rate. If loss ratio is above an upper or below a lower threshold (which are also rate-dependent), the rate is changed to the next lower or higher rate, respectively.

- **Minstrel** [minstrel] is the standard rate adaptation algorithm used by the Linux kernel. It uses 10% of all sent frames as probe frames, which are sent with rates that can be either higher or lower than the currently used rate. Internally, it keeps track of the performance of each rate with respect to throughput and transmission reliability, and switches to whichever rate it considers best for the current conditions. Specifically, it can also skip rates, while the previously described algorithms can only step up and down one rate at a time.

In addition, we investigated two OFRA setups:

- **OFRA-ACK** is OFRA used in a standard setup in which all frames are acknowledged. This means that OFRA cannot show its full potential, because, while the comparison algorithms use ACKs for rate decisions, OFRA does not, and has to send its feedback frames in addition to acknowledgments. However, this is not only a realistic use case for a wireless network that employs OFRA, but currently has no error-tolerant connections; it is also a fairer comparison case because the additional throughput that a No-ACK system can reach under good conditions (cf. Section 2.5 and Figure 2.8 on page 31) might otherwise overestimate OFRA's performance in scenarios comparable to its competitors.

- **OFRA-NoACK** uses OFRA for rate adaptation, and the sent traffic is not acknowledged on the MAC layer. Other rate adaptation algorithms cannot produce sensible adaptation in this scenario: they interpret the absence of ACKs as transmission failures, and reduce the rate for all frames to the base rate, the slowest rate available. This setup therefore especially showcases OFRA's ability to enable rate adaptation for transmissions without acknowledgments.

5.5.4 Evaluation results

In the following, we will present evaluation results from our simulation. We will highlight several performance metrics. Specifically, we will investigate the following behaviors:

- We will first present throughput-related results: goodput (data rate of correctly received data), error rate (fraction of failed transmissions), and overhead (from non-data frames). While goodput and error rate are intuitively understandable, overhead requires a definition. For this evaluation, we defined overhead as the fraction of simulation time that was spent on sending control frames that are related to data transmissions:

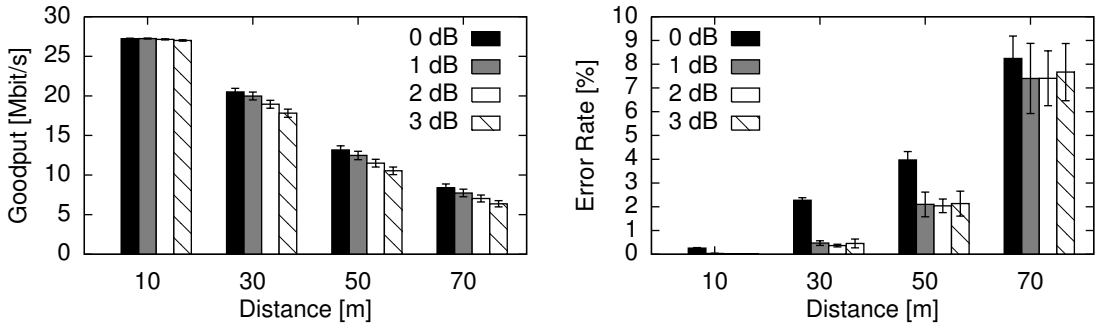
$$t_{overhead} = \frac{t_{ACK} + t_{RTS} + t_{CTS} + t_{feedback}}{t_{sim}}$$

- Next, we will analyze rate selection accuracy, that is, how often the rate adaptation chose the rate that, given the receiver's SNR for the frame, turned out to be the optimum rate.

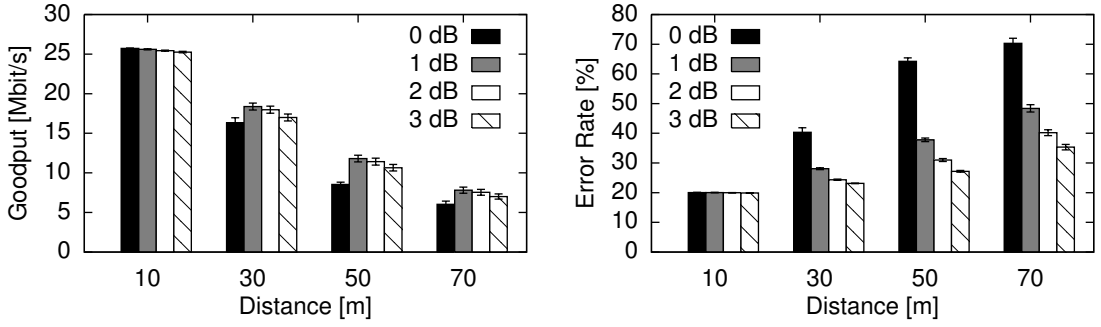
- Finally, we will look at error burst length, that is, the relative occurrence of subsequent transmission failures.

However, before we look into these metrics to compare OFRA with other rate adaptation algorithms, we need to address an effect that we witnessed during early evaluation results and that lead us to slightly modify our approach. Early results showed OFRA to have a problematically high error rate. After investigation, we noticed that basing our rate change decisions exactly on the optimum SNR resulting from the crossover points as shown in Figure 5.5 led to a high number of overselections (i.e., selecting a rate that was higher than optimum). This was due to the fact that, like every rate adaptation algorithm, OFRA has to predict the optimum rate of the next frame from witnessing previous frames. In a fast channel, it would happen regularly that we switched to a rate that turned out to be too optimistic for the next frame, a behavior that has been witnessed and described previously [CK10] for other SNR-based rate adaptation algorithms.

To assuage this behavior, we investigated shifting the SNR thresholds at which to switch by 1, 2 and 3 dB. This gives us a safety margin: the rate is not shifted unless the previous frame has been well above the crossover point. Figure 5.11 shows results for no safety margin and safety margins of 1, 2 and 3 dB. For example, if the



(a) 1 node, slow channel



(b) 8 nodes, fast channel

Figure 5.11 Switching rates exactly at the crossover points leads to undesirably high error rates. Introducing safety margins that shift rate switching to a higher SNR strongly reduces error rate, has little detrimental effect on goodput (and can even increase it).

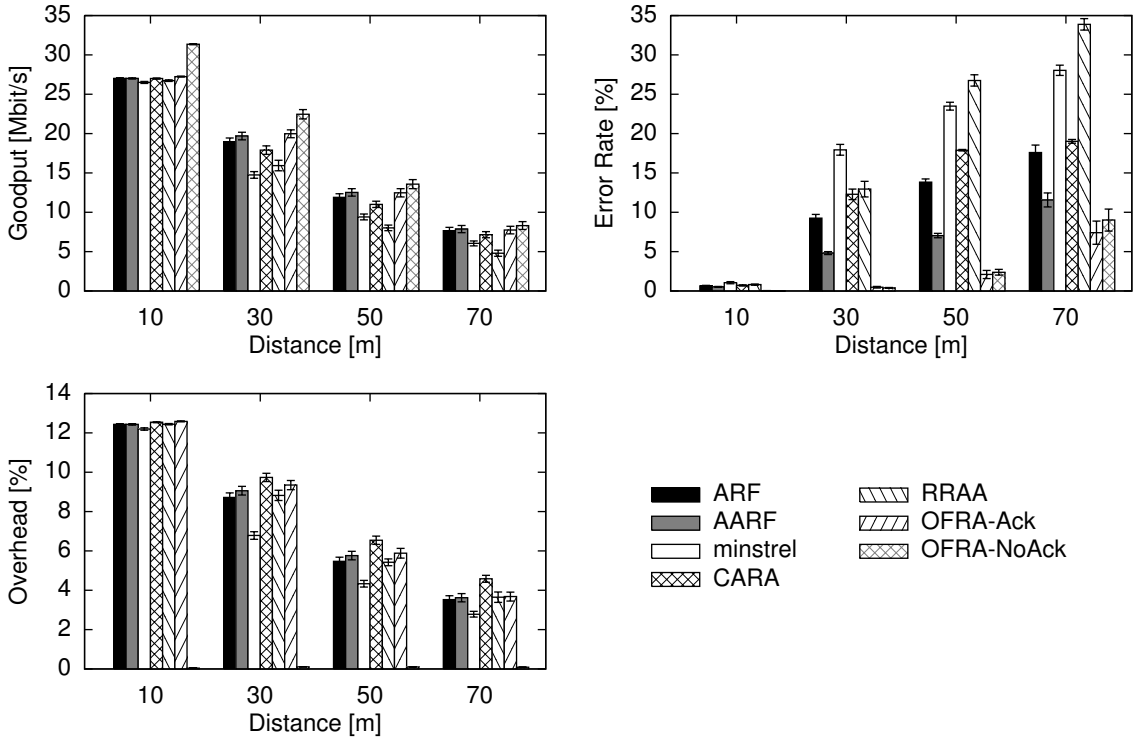
crossover point between two rates was 11 dB, the faster rate would not be chosen unless the expected SNR was at least 12 dB in the case of a 1 db safety margin. For brevity reasons, we do not show all results; rather, we present results from the two extremes of our evaluation, the 1-node setup in a slow channel and the 8-node setup in a fast channel.

As can be seen in the figure, a safety margin of 1 dB produces a significant reduction of error rate in all cases. Results from higher safety margins are not as clear: in the simple 1-node, slow-channel case, they do not further decrease error rate, but are detrimental to goodput. In the challenging 8-node, fast-channel case, they help decrease the error rate further, but their effect per dB is much lower; in addition, highest throughput is achieved at the 1 dB safety margin setting.

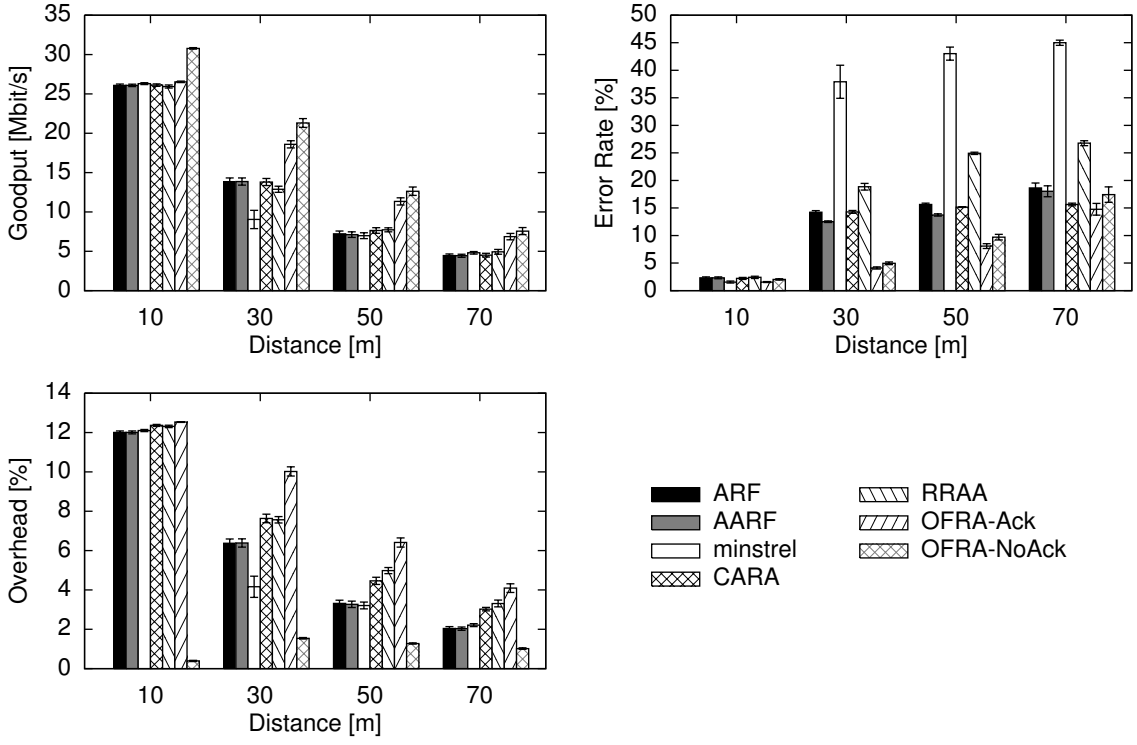
Based on these results, we decided to configure OFRA with a safety margin of 1 dB for comparison with other rate adaptation algorithms. For the remainder of this evaluation section, when we use OFRA, it is used with this safety margin of 1 dB.

5.5.4.1 Throughput-Related Metrics

We will now present comparison results with the rate adaptation algorithms listed in Section 5.5.3. Figures 5.12 and 5.13 present results from our simulations for the

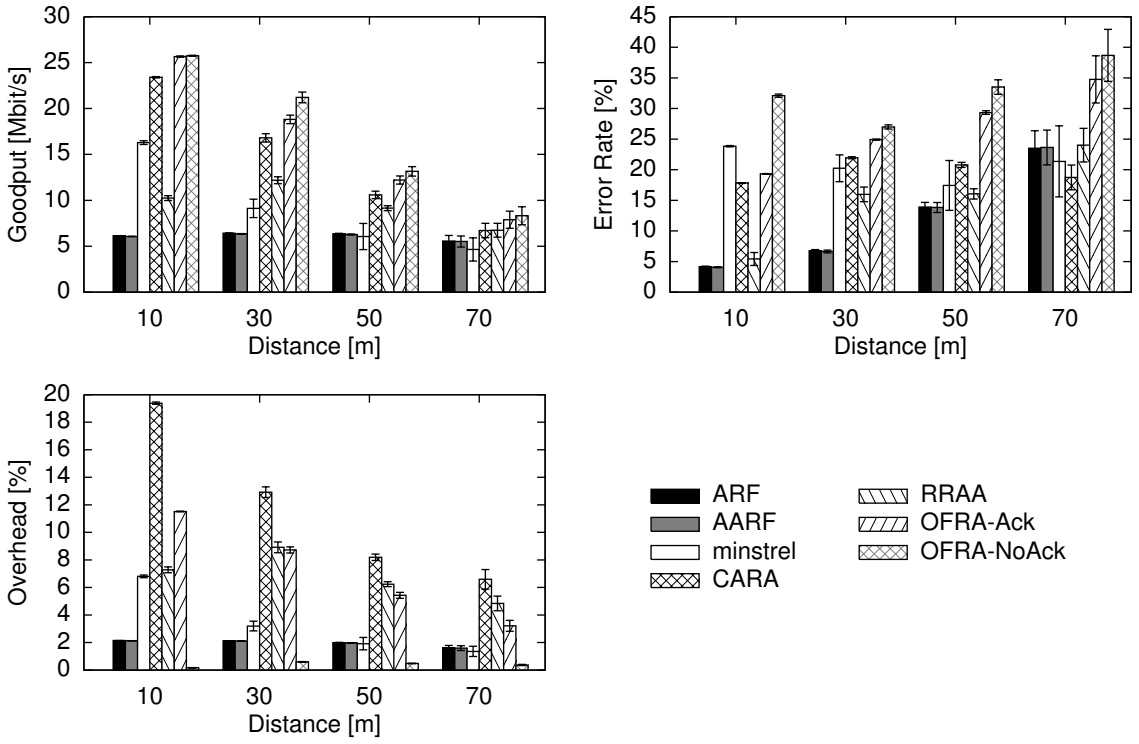


(a) slow channel

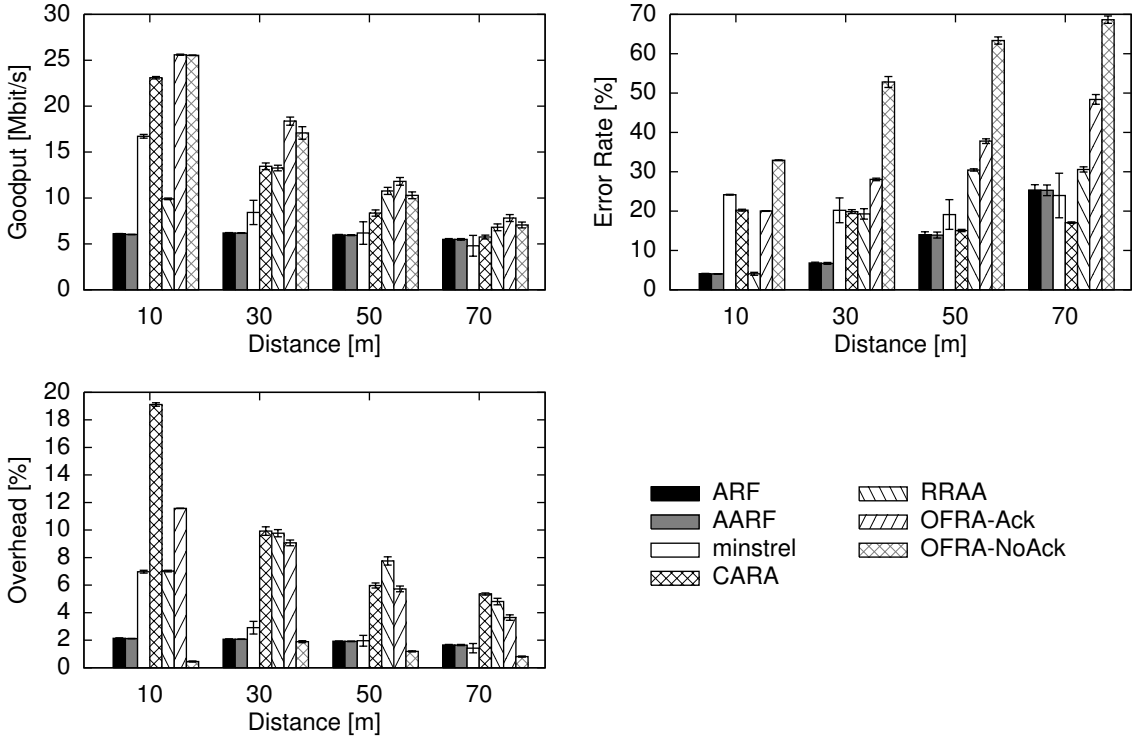


(b) fast channel

Figure 5.12 Goodput, error rate and overhead in the single-node topology. In both slow and fast channels, OFRA-Ack performs as good or better than all comparison algorithms for all metrics except overhead. OFRA-NoAck outperforms all other algorithms in overhead (due to not sending ACKs) and goodput, at similar error rates to them.



(a) slow channel



(b) fast channel

Figure 5.13 Aggregated goodput of all 8 STAs, error rate and overhead in the 8-node topology. OFRA-Ack provides higher goodput than comparison algorithms, at the cost of higher overhead (due to sending both ACKs and feedback frames). Error rates vary strongly, but only at the highest distances does OFRA-Ack's error rate significantly deteriorate compared to the other algorithms. Regarding OFRA-Noack, again, overhead is extremely low (due to saving ACKs), goodput is at least as high (and generally higher) than for comparison algorithms, but error rates increase sharply, especially in fast channels at large distances.

metrics of goodput, error rate and overhead at several distances, number of STAs and channel speeds, as explained in Section 5.5.2.

For the less challenging single-node topology (Figure 5.12), the very simple baseline algorithm ARF already produces good results. The higher sophistication of RRAA in fact works against it in this setup. To a lesser extent, this also holds true for CARA: in a single-node topology without collisions or hidden stations, its collision detection does not help. Minstrel's aggressive probing for higher data rates leads to a high error rate and a somewhat reduced goodput.

Two results regarding OFRA are of special note in these setups. First, especially in slow channels (Figure 5.12a), both OFRA versions (Ack and NoAck) produce a much low error rate than the comparison algorithms. Due to their receiver-based nature, the sender can rely on getting feedback from the receiver whenever it needs to change the rate. It therefore does not need to employ any kind of probing for higher rates, which can lead to harmful overselection, which often leads to frame errors. Second, OFRA-NoAck's overhead is extremely small compared to all other algorithms, to the point where the results are almost invisible in the graph in the slow channel case. This is due to the fact that in this scenario, no ACKs were sent. The overhead is therefore merely due to feedback frames, which are sent only when required—that is, very rarely in a slow channel.

The fact that overhead decreases with distance (that is, lower SNR values) is due to the fact that in those situations, data frames were sent at lower rates. This means that fewer data frames could be sent within a simulation run, which decreased the number of ACK frames as well, which means less time was spent on frames categorized as overhead.

The 8-node topology (Figure 5.13), as a more challenging scenario, shows more pronounced differences between the investigated rate adaptation algorithms. ARF and AARF clearly produce the overall worst performance.[55] They produce almost no adaptation to speak of, falling back to the base OFDM rate of 6 Mbit/s and keeping it for all distances. Minstrel, and even more so CARA, show high goodput performance at small distances, but strongly degrade with distances. Note that both versions of OFRA outperform CARA, even though the latter tries to protect against collisions, a problem that OFRA does not target specifically. This protection against collisions via RTS/CTS is clearly visible in the high overhead of CARA in this simulation setup. Overall, OFRA provides very stable goodput performance: differences between 1-node and 8-node setups, as well as between slow and fast channel setups, are minimal.

Even though OFRA produces high goodput and low overhead in all scenarios, there is one downside visible in the 8-node case, especially with a fast channel. Error rates are higher than for the comparison algorithms, especially for OFRA-NoAck. This is due to a combination of several factors. (1) The more stations send concurrently, the longer the average time between consecutive frames of one connection becomes. Together with increasingly fast changes in channel conditions (i.e., decreased coherence

[55] In fact, taking both scenarios into account, it can be seen that the performance differences between ARF and AARF are negligible in most cases.

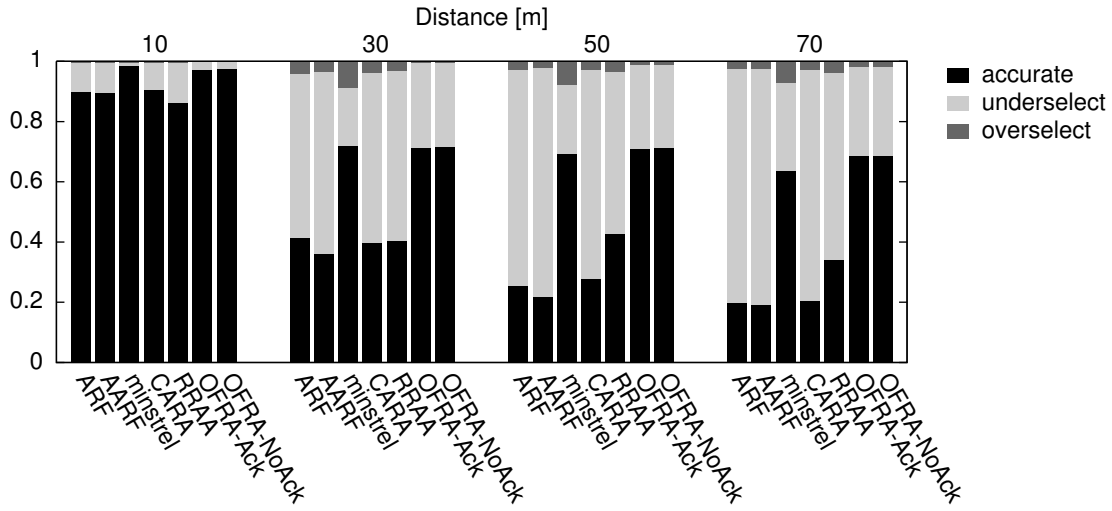
time), correct rate adaptation becomes harder for all algorithms. (2) Since OFRA signals its feedback via special frames, those have to contend for channel access, as opposed to ACK frames, which are sent immediately upon reception because the sender already reserved the channel for the subsequent ACK. With more and more concurrently sending nodes, the chance of successfully receiving channel access is lowered, and feedback sending delayed. (3) Especially for the case of OFRA-NoAck (which shows even high error rates than OFRA-Ack), the interplay between No-ACK traffic and contention becomes a problem. Under normal (ACKed) circumstances, if a frame is not transmitted correctly, the size of the contention window (i.e., the maximum time a sender may randomly wait before sending the frame) is increased to assuage the risk of collisions, which in effect leads to a lower number of frame transmission during a period of low channel quality that was not yet adapted to. The contention window limits are reduced upon successful transmission. Because with No-ACK traffic, the feedback inherent to the ACK whether transmission was successful is missing, no such contention window adaptation is done. Hence, even after frame losses, frames are sent as often as before. This means that more frames are sent when conditions are especially problematic, increasing the overall error rate.

However, OFRA is still competitive, providing the best results of the field in the important goodput metric. Furthermore, it should be noted again that it is the only rate adaptation algorithm capable of adapting No-ACK traffic. All other rate adaptation algorithms will fall back to their base rate, providing no sensible adaptation at all.

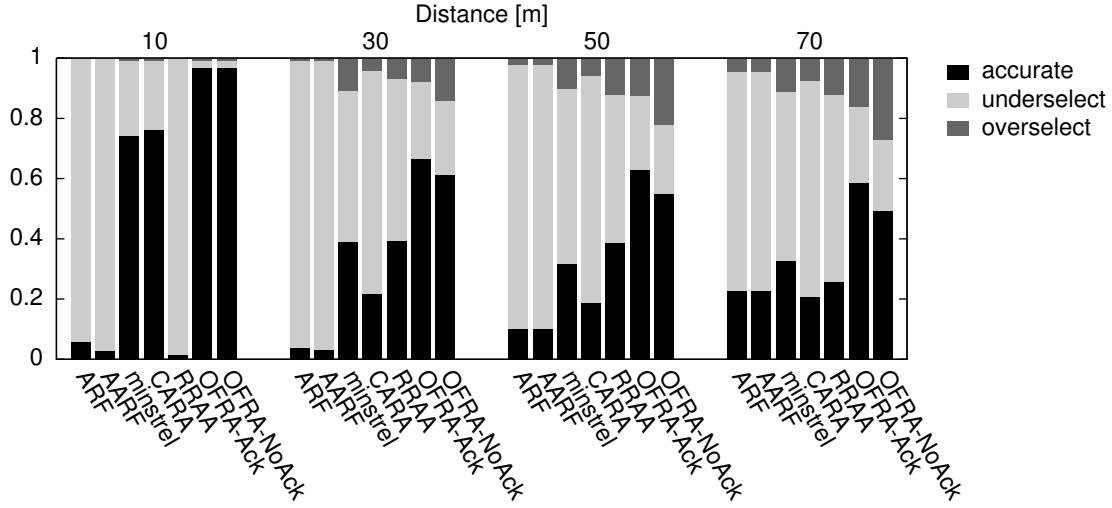
5.5.4.2 Rate Selection Accuracy

As a second step of our evaluation, we investigated the accuracy of rate adaptation algorithms in choosing the optimum rate for each data frame. For each frame, the chosen rate is compared to the SNR that was measured for that frame. We then compared whether the rate was the optimum rate for that SNR. This information is directly inferred from the crossover points that were extracted from the throughput measurements presented in Figure 5.5c. Since it is impossible to exactly predict future channel conditions, no algorithm can choose the optimum rate every time. How often it chooses this optimum rate, however, is an important performance metric: the more often, the higher the potential goodput. Choosing a rate lower than what the channel would have supported, that is, *underselection*, leads to less-than-optimum use of the channel and reduced goodput. The opposite, that is, *overselection*, can potentially increase goodput beyond that of accurate rate decision. However, it incurs a high risk of transmission errors and consequently dropping of the frame. This more than offsets the higher transmission rate (and is, in fact, the reason for the position of the crossover points between rates at certain SNRs). It is therefore potentially harmful to overselect the rate.

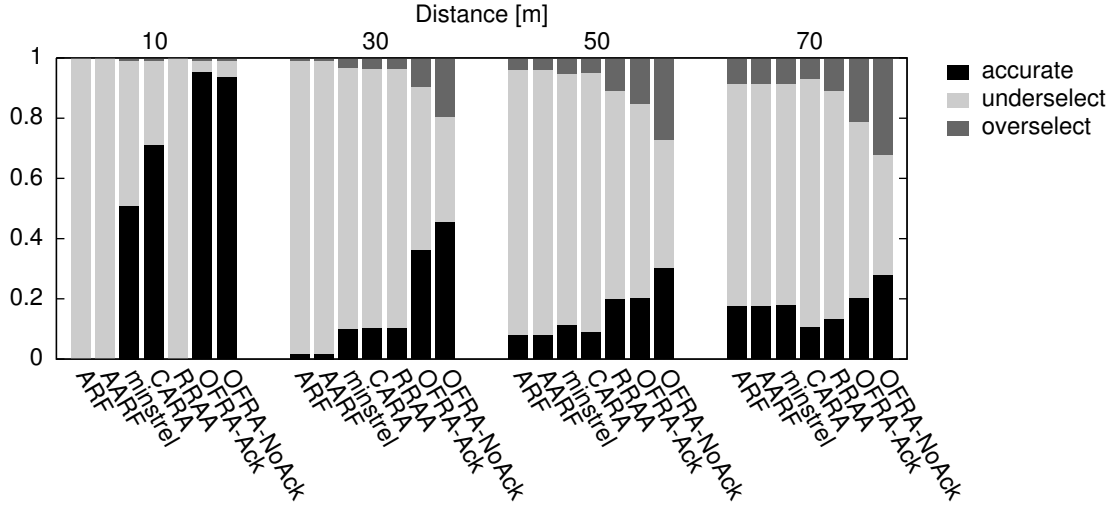
Figure 5.14 shows results from 3 scenarios. Since all slow-channel scenarios were very similar and followed the same behavior as the 1-node, fast-channel scenario presented in Figure 5.14a, we opted to present three different fast-channel scenarios for 1, 4 and 8 nodes.



(a) 1 node



(b) 4 nodes



(c) 8 nodes

Figure 5.14 Rate selection accuracy for different algorithms, distances and node numbers. All presented results are from fast-channel scenarios. OFRA consistently shows the highest amount of accurate rate selection, and in the single-node scenario (as well as all slow-channel scenarios, which are not presented here) the lowest amount of harmful overselection. however, with increasing number of nodes, OFRA's overselection increases.

The 1-node case poses little challenge to the rate algorithms. There are two distinct classes of algorithms with respect to accurate selection: ARF, AARF, CARA and (to a certain extent) RRAA show almost equal selection accuracy, while minstrel and both OFRAs form another class that fares better. Overselection is low for all algorithms, with minstrel producing the largest overselection due to its aggressive probing scheme.

The 4-node case shows that the baseline algorithms ARF and AARF become almost useless at adaptation. In fact, they spend most of their time sending frames at the base rate of 6 Mbit/s or marginally higher. The fact that their accuracy increases with distance is solely to the fact that at larger distances, this choice sometimes happens to be the right one. RRAA, while producing satisfactory results at higher distances, seems to eschew the highest rates completely, which leads to immense underselection at the 10 m distance. CARA and minstrel both provide high accuracy at small distances, but deteriorate with distance, by both over- and underselecting more often. Both OFRAs again provide the highest amount of accurate selection. However, they are prone to overselection, with OFRA-Ack producing similar amounts as the comparison algorithms, while OFRA-NoAck shows the highest amount of overselection of all algorithms.

Finally, the 8-node case is mostly similar to the 4-node case, with somewhat more extreme results. ARF and AARF still perform badly, minstrel, CARA and RRRA accurately select somewhat better (with, again, the except of RRAA at 10 m), and both OFRAs provide the highest accurate selection, at the cost of the highest overselection.

These results confirm our results from the previous section, where both OFRAs produced the highest goodput, but showed high error rates in fast-channel, 8-node scenarios. We can now infer that these are indeed due to high accurate selection at the cost of high overselection.

This also ties in with our results with regard to dB safety margins. The effect of such margins is to shift rate selection from overselection towards underselection. It therefore stands to reason that a dynamic adaptation of safety margins might be beneficial, with a 1 dB (or even 0 dB) margin for simple scenarios with few nodes and a slow channel, and higher margins for more challenging situations. We will discuss this idea in Section 5.6.

5.5.4.3 Error Burst Lengths

As a last step in our evaluation, we investigated the length of error bursts, that is, how often several frames in a row are lost or received with errors. This can give us some insight into the speed at which a rate adaptation algorithm can adapt to sudden drops in channel quality. Since we already investigated the overall error rate in Section 5.5.4.1, we decided to normalize the data for this step of the evaluation: We investigated how many 1-bursts (i.e., single frame errors), 2-bursts (two frames corrupted back-to-back), ..., occurred for each STA–AP connection. We then aggregated, for each rate adaptation algorithm, the results from all runs at all distances

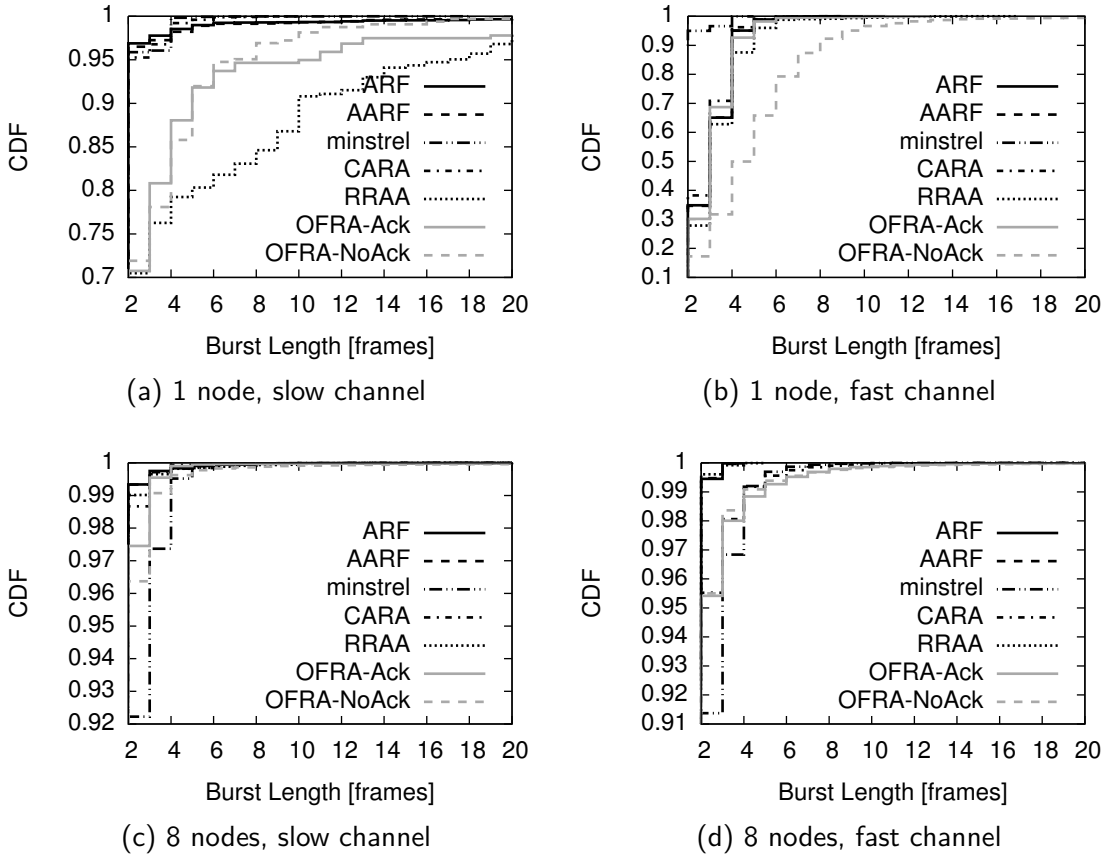


Figure 5.15 Relative occurrence of different lengths of error bursts over all error occurrences. OFRA tends to produce longer a larger fraction of long error bursts than comparison algorithms only in situations in which its absolute number of errors is very low. In all other situations, there are no significant differences.

(but did not aggregate over node numbers and channel speeds). Finally, we created CDF graphs out of the results by showing the relative occurrence of different burst lengths.

The results are shown in Figure 5.15. Note that the CDF axes do not start at 0, and each at different value, to improve lucidity. At first glance, these results seem counter-intuitive: as conditions become more challenging (more nodes, faster channel), the number of longer error bursts seems to decrease. This should not be interpreted as an increase in quality: rather, as the amount of errors increases in such conditions, the number of single frame errors increases compared to those of longer bursts. Hence, there are not fewer long error bursts in challenging conditions, but the number of single frame errors increases even more strongly, shifting the CDF values.

The 8-node graphs do not show any strong differences between the investigated rate adaptation algorithms: ARF, AARF, and RRAA produce slightly fewer bursts, which correlates with their relatively low throughput performance (cf. Figure 5.13). Both versions of OFRA show a behavior that closely resembles the other comparison algorithms.

The 1-node graph, interestingly, shows larger differences. Especially under very non-challenging conditions (1 node, slow channel), OFRA produces significantly longer error bursts than most comparison algorithms. However, this again is an artifact of the presentation of these results as relative values: in these experiments, both OFRAs' error rates were extremely low (cf. Figure 5.12), so this CDF shows the opposite effect as the one described above: just as single frame error rates increase overproportionally as error rate increases, those single bit errors are also the ones that disappear more strongly as error rate decreases. In absolute numbers, OFRA in fact produced fewer long error bursts, but the number of single bit errors decreased even more strongly (e.g., by a factor of almost 50 when comparing ARF and OFRA in the single-node, slow-channel scenario). The obvious outlier is RRAA, which has an even higher tendency for long burst errors; additionally, this one cannot be explained by an overall lower error rate. We can only conject that RRAA's sliding windows makes it slow to adapt to channel changes, especially since, once the rate is chosen, it is kept for a certain amount of frames to fill the window before another rate decision is started. Why this is so much more obvious in this specific scenario, however, is a question that might warrant further scrutiny in the future.

Overall, we can derive from these numbers that, especially in challenging conditions, OFRA's burst error behavior is similar to those of other rate adaptation algorithms. In less challenging conditions, it tends to produce more and longer burst errors relative to single errors, but the overall error rate numbers, as presented before, are so low that this is no reason for concern.

5.5.4.4 Summary

Summarizing the evaluation results, we see that OFRA is a very competitive algorithm. Not only was it designed to support No-ACK traffic, which none of the comparison algorithms do. Even when ACKs are used, OFRA outperforms or at least performs on par with the comparison algorithms. This is even more noteworthy since OFRA does not use those ACKs for its rate adaptation decisions, and has to send its feedback frames in addition, which puts it at a disadvantage. That it can nevertheless compete favorably shows the strength of the concept.

Regarding throughput, both OFRA versions perform better than comparison algorithms in all tested scenarios. Error rate results are more varied, with error rates lower in single-node topologies, but higher in 8-node scenario. This directly correlates with rate selection accuracy: in single-node topologies, both OFRAs show the highest fraction of accurate selections (with only minstrel reaching the same accuracy) and lowest fraction of underselection. In the 4- and 8-node scenarios, accurate selection stays the highest in the field, but overselection also increases. With respect to the distribution of burst error lengths over all error events, there are few significant differences between OFRA and its competitors; these only appear in situations in which OFRA's extremely low error rate makes the differences inconsequential.

We consider OFRA, even in its version presented here, a strong contender for high-performance rate adaptation. This holds true even more because OFRA opens up a new field of rate adaptation possibilities, by allowing to adapt rates for No-ACK

traffic. Nevertheless, we will present some extensions in the following to further improve OFRA's behavior, for example, in challenging channel conditions.

5.6 Extensions and Future Works

OFRA, as presented so far, has shown to be highly effective at rate adaptation in many scenarios. However, there are still some open issues and options for improvement. Specifically, we identified the following problems:

SNR Calibration

The crossover points that we calculated in the ns-3 simulation are not directly transferable to real hardware. In fact, these points are not even transferable between different types of network adapters, because they depend on several factors, such as the signal gain due to the receiver hardware and antenna. Furthermore, most consumer hardware does not provide SNR values directly, because these are problematic to calculate (they would need to measure the background noise and the signal separately). Instead, most hardware will provide the received signal strength in dBm. Some older hardware might even only provide RSSI values, on an arbitrary scale that might, for example, run from 0 to 100.

In its current version, OFRA hence requires an offline training phase in which the crossover points are determined by long measurements that try to produce a wide range of SNR values during the measurement. In practice, this can be done by circumventing the irregularities of the wireless channel by connecting two network adapters with a coaxial cable and inserting different attenuators in between. While this works, it is cumbersome and not very practical. We hence need some way to calibrate SNR values online, while the card is running, in a bootstrap fashion that allows usage of the card (albeit potentially at reduced performance) during calibration.

Dynamic Safety Margins

We have seen in our evaluation that using the exact crossover points as switching points between rates produces harmful overselection. As a remedy, we decided to introduce a safety margin of 1 dB: only if the expected SNR was at least 1 dB above the crossover point, the higher rate would be selected. This worked very well in the 1-node scenario. However, in the 8-node scenario, especially at high channel speeds, error rates for OFRA were high due to high overselection. In such scenarios, a larger safety margin would be beneficial. Conversely, using a larger safety margin in less challenging scenarios where error rates are already very low would reduce throughput for no significant gains.

Therefore, it would be helpful to be able to set the safety margin dynamically, depending on the environmental conditions, such as channel speed and number of other stations.

As it turns out, both of these can be solved in the same way. By creating a system that changes the crossover points at run-time and/or influences the rate choice in other ways, both on-line calibration and dynamic safety margins can be realized. We will concisely explain two approaches, both with their specific advantages and disadvantages, and refer to [Göt13] for more in-depth explanations.

The Binning Method

This approach is inspired by the on-line calibration method employed by charm [JWS08]. If, within a window of frame receptions, the average SNR is lower than the crossover point, but the PER is below a preset threshold, then the crossover point is moved so that the rate is also used at the lower SNR. Conversely, if the average SNR over the window was higher than the crossover point, but PER was above a threshold, then the point is moved so the rate will only be used at higher SNR.

This approach allows on-line calibration as long as provisions are made that no rate's SNR range can completely overlap another's. Otherwise, some rates could stop to be chosen completely. This can happen if a crossover point is moved so much that it falls on top of another crossover point. The reason this can happen at all is that crossover points are only moved when a rate is in use, because only by using the rate, calibration data is created. If one rate is constantly in use, and a large correction is necessary, it might end up overtaking an adjacent crossover point. Thus, for example, rate n would stop being used, because it seems to the rate adaptation algorithm that below the crossover point, rate $n-1$ would be a better choice, and above the point rate $n+1$. This can be easily prevented by ensuring strict monotony, by moving crossover points that otherwise would be overtaken, by a certain factor. For example, the calibration algorithm can make sure that adjacent crossover points have at least a distance of 0.5 dB, moving them accordingly otherwise. Ensuring this monotony makes sense: at least within a modulation class (DSSS, OFDM), every rate has an SNR range in which it produced the best performance.

The second problem with this approach is that it can cause a significant increase in feedback messages. Because the calibration sets a crossover point to the average of recent SNR measurements, it is very likely that SNR measurements in the near future will fall into the same area, slightly above and below the new crossover point. Every time the SNR now switches between slightly below and slightly above the new point, a feedback frame is created.

The Marking Method

This approach avoids the problem that causes the large amount of feedback frames being generated. Instead of changing the crossover points directly, it instead marks rates as temporarily unreliable. If a rate, despite the fact that the SNR suggests it should perform well, produces higher-than-expected PERs, it is marked as unreliable. While it is marked as such, it will not be used by the rate adaptation.

Because the crossover points themselves are not changed, this approach does not trigger excessive feedback as the binning method does. However, for the same reason, it cannot be used for on-line calibration of new network adapters without a set of crossover points that was pre-measured off-line. For this initial SNR calibration, it is necessary to create sensible crossover points from measurements. Additionally, to realize whether a marked rate is reliable again, the marking method needs to occasionally probe the channel with frames sent at those rates, in a way similar to how frame-based approaches, such as ARF or minstrel, do it. This increases the risk of frame loss somewhat.

In practice, a combination of the two approaches seems sensible: the binning method should be used to create initial SNR crossover points for new, unknown hardware, for which no such points are available; afterwards, the marking method can be used. Alternatively, the binning method can be extended by a method that limits the rate of feedback being sent, especially when the differences between the measured SNR and the crossover point are small. Concepts for such rate-limiting have been proposed in [Hit11].

Influence of Interference

The marking method has another advantage that can be highly relevant, depending on the environment. In our simulated environment, we used SNR values as quality metrics. These SNR values were calculated over the complete length of the frame. If interference occurred, this lead to a drop in SNR, which allowed the rate adaptation to recognize the problem. On the other hand, in real hardware, no such exact SNR is calculated. The calculation is generally only done over the preamble, which is easy to do: since the preamble comprises well-known patterns that are always the same, matching the received signals against the expected ones allows for easy SNR calculation. However, if interference only occurs in latter parts of the frame, this is not reflected in the SNR any more. Additionally, the vast majority of systems, which only provide Received Signal Strength (RSS) measurements, cannot recognize interference this way at all, regardless of whether they calculate RSS over the full frame or only the preamble, since interference introduces additional (but destructive) power into the received signal.

In such a scenario, the binning method changes crossover points incorrectly, because it cannot distinguish between high RSS due to a good reception or due to interference. The marking method, however, can mark our rates at which interference leads to frame loss. Since it can be beneficial to reduce rate under interference (such that the more robust signals can resist it), the marking method provides an efficient way of coping with interference, which otherwise OFRA is susceptible to.

5.7 Conclusion

In this chapter, we presented OFRA, an On-Demand Feedback Rate Adaptation algorithm, whose main novel contribution is the ability to adapt rate for traffic without

acknowledgments (No-ACK). Since state-of-the-art rate adaptation algorithms, as used in today's systems, rely on ACKs for rate adaptation decision, and therefore fall back to the lowest speed available, we introduced explicit feedback in the form of specially-crafted feedback frames.

These feedback frames have the added advantage that they can be sent on-demand, that is, only when channel conditions change significantly and warrant a change in rate. This allows timely reacting to changes in the channel state, while keeping the amount of feedback low in slow-channel conditions. As a result, we could show in Section 5.5 that the overhead of our feedback scheme is much lower compared to the constant stream of ACKs in 802.11. Consequently, OFRA realizes higher throughput than other rate adaptation algorithms when sending No-ACK traffic, being able to utilize the freed time on the channel otherwise spent on ACKs effectively. Furthermore, even with ACKed traffic, OFRA generally performs better than state-of-the-art algorithms: the additional overhead of the feedback frames is offset by its faster and more accurate rate adaptation.

In its current version, the algorithm loses performance in fast-channel scenarios with many concurrent users, even though this performance loss mostly manifests itself in losing its superior performance and only performing on par with comparison algorithms. This is mostly due to increased levels of harmful overselection in those case. However, we suggest solutions to this problem and some other practical downsides of OFRA in Section 5.6. Preliminary results [Göt13] have shown that these extensions are effective in solving, for example, the problem of OFRA's performance in interference scenarios.

Overall, we consider OFRA a good match to complement error tolerance, both heuristic header tolerance as presented in this dissertation, and previous solutions such as UDP-Lite, which also suffered from lacking support on the MAC layer and by rate adaptation. Additionally, OFRA's performance makes it a rate adaptation algorithm that can improve the throughput of small and medium size WLAN deployments, even when error tolerance is of no concern. Thus, OFRA can be seen both as an important and effective building block to deploy error tolerance in WLAN, as well as a solution whose performance allow it to stand on its own.

6

Before him stood the Tree, his Tree, finished. If you could say that of a Tree that was alive, its leaves opening, its branches growing and bending in the wind that Niggle had so often felt or guessed, and had so often failed to catch.

He gazed at the Tree, and slowly he lifted his arms and opened them wide. 'It's a gift!', he said.

—J. R. R. Tolkien, *Leaf by Niggle*

Conclusion

Wireless communications are becoming more and more important. However, compared to wired communications, they also pose additional challenges, the most prominent among them is a much higher error rate. Such errors lead to corruption of packets, which are so far consequently dropped at the receiver, and potentially then retransmitted. This behavior is very inefficient, especially for error tolerant applications, and even potentially harmful if these applications have strict timing requirements, as in the case of live streaming of VoIP.

This behavior motivated the desire to improve this inefficient scheme of packet drops and potential retransmissions, and stimulated our research that is presented in this thesis. By focusing on error-tolerant applications, we envisioned an approach in which errors are tolerated, and packets processed regardless. Realizing that techniques to support error-tolerance for application-level payloads already existed, we focused on the more comprehensive problem of how to support processing of packets with errors anywhere in the packet. This approach posed the fundamental challenge that errors in headers can lead to unexpected behavior during packet processing, the most notable and harmful of them being misattribution: The assignment of a packet to the wrong application, because the demultiplexing information that identifies which connection a packet belongs to was corrupted.

6.1 Contributions and Results

To solve this problem, we devised a system in which header fields are categorized by their importance to the identification decision. Unimportant fields can be ignored, while important fields have to be repaired. We solved this repair challenge by the observation that the network stack inside a host is aware of what connections are open at any given point in time, and what the contents of those header fields that

carry identification information have to be, because those contents are exactly the information that is used to demultiplex packets to their correct application even when no errors occurred. By using a simple Hamming distance metric, we identified which connection a packet most likely belonged to. Despite the simplicity of this metric, we could show that Refector, as we termed the approach, is highly effective at recovering from header errors, and can significantly reduce the packet loss rate even compared to UDP-Lite, the state-of-the-art algorithm that supports payload-only error tolerance. Furthermore, misattribution occurred extremely rarely. Finally, we could show that even encryption of packets does not pose any insurmountable challenges to Refector.

However, while Refector worked very well, we were not completely satisfied with it having the downside that, to extend support to additional protocols, it required in-depth knowledge about both error tolerance concepts and the inner workings of those new protocols. Hence, we further extended the Refector concept by a solution that does not require such knowledge to provide header error tolerance. We designed a classification algorithm that allows to learn packet content patterns that are specific to each connection, and use those patterns to match incoming packets to ongoing connections. An important property of this algorithm is that it does not require off-line learning: it learns new patterns as new connections are opened, and does so within a few (correct) packets within opening of the connection. Again, we could show that this classification algorithm can correctly identify which connection a packet belongs to, even under very high BERs, and that misattribution is an exceedingly rare occurrence.

The final main contribution presented in this dissertation is a novel rate adaptation algorithm for 802.11 (WLAN). This contribution stems from the desire to have an effective rate adaptation algorithm available that supports unacknowledged (No-ACK) traffic in 802.11 networks. The desire, in turn, was motivated by the fact that we needed to send error-tolerant traffic without ACKs in WLAN networks due to the idiosyncrasies of the 802.11 standard, namely the extremely strict timing requirements on sending ACKs, which our error recovery scheme could not fulfill. Since state-of-the-art rate adaptation algorithms for 802.11 depend on ACKs as feedback on transmission success or failure, those algorithms could not properly select rates for No-ACK traffic. We hence designed a novel rate adaptation scheme in which feedback is explicitly sent whenever channel conditions changed significantly and warrant a rate change at the sender. As a side effect, the amount of feedback adapts to the speed at which channel conditions change, so that less feedback is sent when the channel is stable than when it is volatile. As a result, we designed a rate adaptation algorithm that not only works with No-ACK traffic, but also provides higher performance than state-of-the-art algorithms even when standard traffic is used.

6.2 Future Work

While working on the topics of this dissertation, we encountered a number of questions and ideas for extensions. While we did not have the chance to pursue them in

detail, we will give an overview here in the hope that others might find it worthwhile to investigate them in the future. At the same time, discussing these open points will show current limitations of our approaches.

We already discussed several specific points relating to future work for Refector and for OFRA at the end of the respective chapters (Section 3.5 and Section 5.6, respectively). To keep this section concise, we will not repeat the points given there, and instead focus on more basic and general fields in which heuristic header error recovery could be extended to broaden its applicability further.

Extensions to the Machine-Learning Algorithms

In Chapter 4, we presented a machine-learning approach to recover from packet errors by creating connection-specific patterns on-line, as new connections are opened, based on characteristic content in packet headers, and then matching incoming corrupted packets against those patterns. We then presented preliminary work in Section 4.5 that we conducted to provide insight into recovery by matching on extrinsic factors, such as packet sizes and inter-arrival times. In the end, we abandoned our investigation of this sub-topic.

However, we are still convinced that this idea and work has merit, and that it might be possible to construct such an algorithm. This would be especially beneficial for encrypted transmissions, in which also all or parts of the headers are encrypted. While we showed in Section 3.2.4.5 that Refector can be used over encrypted links, this assumed that the packet headers have been decrypted before heuristic recovery occurs and hence their contents can be used for classification. This is the case in link-layer encryption, as used in WLAN, but not if VPN connections that employ IPsec tunneling are used, where decryption will only occur during IP header processing. This is a special challenge for protocol-independent recovery, because recovery is not done protocol-by-protocol (in which case decryption could be performed at the appropriate time, and the higher-layer headers then recovered afterwards), but all at once immediately after packet reception from the MAC layer. Such a work might provide enough work for a dissertation on its own, or at least provide a starting point for one. Such a dissertation would most probably straddle the areas of traffic classification, which has produced much work for classification of traffic on backbone routers (e.g., [MZ05, CDGS07]), and security, where works exist that investigate possible attack vectors on encrypted traffic by observing factors such as packet sizes and inter-arrival times (e.g., [SWT01, BLJL06]).

Error Tolerance for Low-Power Wireless Networks

In this dissertation, we presented approaches to recover from header errors in networks that employ Internet-style headers. While this encompasses the vast majority of all networks in current use, there are specialized environments in which other communication protocols are used, often because their special use cases do not require all capabilities provided by the Internet stack, and the resulting overhead of using it is considered disadvantageous.

One example of such an area is the field of sensor networks and fields which developed from it (e.g., Cyber-Physical Networks and the Internet of Things), in which low-power devices with wireless radios are used that generally employ the 802.15.4 [IEEE11] standard. One challenge in this area is that, to keep overhead down, sensor network deployments often used specially crafted protocols to communicate. Refector, as a protocol-specific approach, therefore would require adaptation to each such protocol, if they are used in a deployment. Protocol-independent recovery hence lends itself to this use case. Since our classification algorithm performs very fast and requires little memory, it might be usable even on such constrained devices, though we consider this a question that warrants further scrutiny.

Furthermore, since specially crafted protocols are often designed for a single specific purpose, the redundancy in headers due to unused or larger-than-necessary fields can be expected to be severely reduced. It is an open question how well heuristic header error recovery would perform in such a scenario. To improve performance, it might be beneficial to more closely investigate error patterns that occur in sensor networks, a topic that we investigated in some detail [SCW13,SCHW14], but did not discuss in this dissertation because of scope considerations. While we consciously did not use such error pattern information in the approaches presented in this dissertation because of the loss of generality incurred by using MAC- and PHY-specific information, in such a challenging scenario, this might be an option to consider.

Upstream Traffic

We pointed out from the beginning of this dissertation that our envisioned usage scenario for heuristic header error recovery was an end user on a wireless network connected to the Internet. The heuristic recovery approaches presented in the dissertation leverage the fact that in this scenario, an end host knows which connections it has open at any given time, and what values certain header fields have to contain to match each connection. It is therefore only possible to apply these solutions for downstream traffic, that is, traffic from the access point to the end host.

In contrast, to enable this solution for upstream traffic, the AP (or more generally speaking, the gateway node that connects the wireless network with the Internet) would need all the information that the end host uses in our scheme, for each end host that is associated with AP. Not only is this a potentially much larger set of connections to match each packet to; it also requires an additional communication scheme between end hosts and access point to exchange and continuously update that information, which itself uses channel capacity and is susceptible to errors.

Furthermore, the access point would need more stringent recovery rules, because the next hops on the Internet backbone cannot be expected to be tolerant to any kind of errors. On an end host, fields that are not important any more (for example, the packet's TTL) can be ignored. In contrast, an access point would have to repair them. Finally, to prevent the packet from being discarded on the next hop, the access point would also need to recompute checksums, which requires deep packet inspection and modification.

Multi-Hop Support

In theory, it is possible to extend a complete wireless multi-hop network, such as a mesh network, with heuristic error recovery. However, the problem mentioned above that an AP needs information about all connections of all end hosts is further compounded: in a wireless multi-hop network with heuristic error recovery on each hop, each forwarding node needs all connection information from all other nodes in the network. At this point, keeping this information up to date becomes problematic and similar to the overhead problems of proactive routing schemes.

Even if this large overhead can be tolerated, the fundamental problem of error propagation and amplification should not be underestimated. As mentioned above, the current heuristics are designed to identify the correct connection, but not to completely repair all header fields, much less the payload. Over several wireless hops, errors will therefore accumulate. Given enough hops, this can significantly decrease decoding quality of the streamed data.

Nevertheless, it is not impossible to design such a scheme. Especially if deadline requirements are strict, even a packet that might suffer from multiple error events over several hops that arrives in time might be more beneficial than waiting for retransmission, especially since the incurred delay is also additive if corruptions (and hence retransmissions) occur on multiple hops. In fact, Alizai et al. recently showed that header error recovery is possible in a multi-hop sensor network when using the Collection Tree Protocol (CTP) [AKH+15]. Their work leverages the fact that routing is predetermined, and as such, it is not necessary to keep track of different connections with different receivers to recover from header errors: the destination will always be the same, and routing will therefore not break if header errors occur. While this work shows the feasibility of multi-hop heuristic header error recovery in special scenarios, the greater question of the general feasibility of such an approach is still unanswered.

6.3 Final Remarks

This dissertation presents a relatively unorthodox approach to error handling in computer networks, which is already apparent form the fact that similar work is few and far between. As to the best of our knowledge, this dissertation is the first to take a comprehensive and rigorous look at the concept of heuristic header error recovery, considering different approaches, the interrelation between layers 1 and 2 on the one side and layer 3 and up on the other side, and how to effectively prevent misattributions.

At the beginning of this work, it was not clear to us whether such a heuristic recovery scheme would even be feasible. Much skepticism stemmed from the fact that we have to deal with unreliable information in areas of network communication were information is, under normal circumstances, assumed to be perfectly correct, and that today's network protocols, in design as well as in implementation, are not constructed with such a kind of error handling in mind.

However, not only were we able to show that this skepticism, while clearly not unfounded, turned out to be over-cautious, the results we reached exceeded our expectations at the beginning of this work by far. We could show that it is not only possible to identify which connection a packet belongs to if we do not know the exact contents of a packet's header, we furthermore showed that it is not even necessary to know anything about a protocol header's layout and semantics, and still be able to properly identify the connection with an extremely high confidence.

While it is clear from the aforementioned areas of future work that our approach is still far from perfect, we are nevertheless very satisfied with the extent to which we could show that heuristic header error recovery is possible, feasible, and practically implementable. We are interested to see whether this dissertation, and the works presented within, will stimulate future investigations in this field of research.

A

Analytical Approximation for Port Choice Misattributions

In Section 3.2.2.3, we presented an analytical approximation of the worst-case misattribution rate of a standard port choice without respect to maximizing Hamming distances, and our port selection scheme which guarantees a minimum Hamming distance to any other port of at least 5. In this appendix, we will give a more detailed calculation of how we derived the two expressions for worst-case misattributions. Note that these calculations assume a binary symmetric channel, that is, every bit's probability to be corrupted is solely defined by a fixed BER, and independent of errors in other bits.

Standard Port Choice

In the standard port choice, the worst case is a situation in which all 1-bit neighbors of a port are used, and therefore any bit error in the port field leads to a misattribution. We assume a port field size of 15 bits here because, while the port field is actually 16 bits wide, we only consider ports in a private port range starting from 32768, fixing the first bit at 1.

Given a certain BER and a number n of bits, the chance that no errors occurs, that is, that all bits are correct, can be calculated as

$$\prod_{1}^{n}(1 - BER) = (1 - BER)^n$$

since bit errors are considered independent event. The chance that at least one bit error occurs is the complementary event, and can therefore is

$$1 - \Big((1 - BER)^n \Big)$$

which is exactly the term given in Equation 3.1 for $n = 15$.

Refector's Port Choice

With Refector's port choice, the worst can was approximated as a misattribution occurring as soon as 3 or more arbitrarily selected bits out of the 15 are flipped. The probability that *exactly* k bits out of n are corrupted can be calculated using the probability mass function of the binomial distribution:

$$P(X = k) = \binom{n}{k} BER^k \, (1 - BER)^{n-k}$$

This models the urn problem, with the equivalent combinatorial problem of the probability to to draw exactly k white balls from the urn in n draws, with the drawn ball placed back into the urn after each draw, and the overall number of balls chosen so that the probability to draw a white ball is BER.

The probability that 3 or more out of 15 bits are flipped can now be calculated by the sum of the probability that exactly 3 bits were flipped, plus the probability that exactly 4 bits were flipped, ..., plus the probability that exactly 15 bits were flipped:

$$\begin{aligned} &\binom{15}{3} BER^3 \, (1 - BER)^{12} + \binom{15}{4} BER^4 \, (1 - BER)^{11} + \dots \\ \dots + &\binom{15}{14} BER^{14} \, (1 - BER)^1 + \binom{15}{15} BER^{15} \, (1 - BER)^0 \end{aligned}$$

This can be summed up to create the term in Equation 3.2:

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

B

BER Calculation for 16- and 64-QAM

In Section 5.2.2.1, we gave the analytical expressions for the bit error probability of 16-QAM and 64-QAM modulation on an AWGN channel. This appendix contains the detailed calculation to derive these expressions from the general formula.

The average bit error probability of M-ary (square) QAM is given by [CY02] as

$$P_b = \frac{1}{\log_2 \sqrt{M}} \sum_{k=1}^{\log_2 \sqrt{M}} P_b(k)$$

with $P_b(k)$, the k-th bit error probability, given as

$$P_b(k) = \frac{1}{\sqrt{M}} \sum_{i=0}^{(1-2^{-k})\sqrt{M}-1} \left[(-1)^{\left\lfloor \frac{i \cdot 2^{k-1}}{\sqrt{M}} \right\rfloor} \cdot \left(2^{k-1} - \left\lfloor \frac{i \cdot 2^{k-1}}{\sqrt{M}} + \frac{1}{2} \right\rfloor \right) \cdot \operatorname{erfc}\left((2i+1) \sqrt{\frac{3\gamma \log_2 M}{2(M-1)}} \right) \right]$$

which gives us a combined form of

$$P_b = \frac{1}{\sqrt{M} \log_2 \sqrt{M}} \sum_{k=1}^{\log_2 \sqrt{M}} \left\langle \sum_{i=0}^{(1-2^{-k})\sqrt{M}-1} \left[(-1)^{\left\lfloor \frac{i \cdot 2^{k-1}}{\sqrt{M}} \right\rfloor} \cdot \left(2^{k-1} - \left\lfloor \frac{i \cdot 2^{k-1}}{\sqrt{M}} + \frac{1}{2} \right\rfloor \right) \cdot \operatorname{erfc}\left((2i+1) \sqrt{\frac{3\gamma \log_2 M}{2(M-1)}} \right) \right] \right\rangle$$

We can now solve these equation for the required cases of 16-QAM and 64-QAM.

16-QAM

For $M = 16$, the first summation runs over only two terms:

$$P_b = \frac{1}{8} \cdot \left\langle \sum_{i=0}^{1} \left[(-1)^{\lfloor \frac{i}{4} \rfloor} \cdot \left(1 - \left\lfloor \frac{i}{4} + \frac{1}{2} \right\rfloor\right) \cdot \operatorname{erfc}\left((2i+1)\sqrt{\frac{2}{5}\gamma}\right) \right] + \sum_{i=0}^{2} \left[(-1)^{\lfloor \frac{i}{2} \rfloor} \cdot \left(2 - \left\lfloor \frac{i}{2} + \frac{1}{2} \right\rfloor\right) \cdot \operatorname{erfc}\left((2i+1)\sqrt{\frac{2}{5}\gamma}\right) \right] \right\rangle$$

This shows that the inner summation(s) also only run over few terms:

$$\begin{aligned} P_b &= \frac{1}{8} \cdot \left\langle \left[1 \cdot 1 \cdot \operatorname{erfc}\sqrt{\frac{2}{5}\gamma}\right] + \left[1 \cdot 1 \cdot \operatorname{erfc}\sqrt{\frac{18}{5}\gamma}\right] \right. \\ &\quad \left. + \left[1 \cdot 2 \cdot \operatorname{erfc}\sqrt{\frac{2}{5}\gamma}\right] + \left[1 \cdot 3 \cdot \operatorname{erfc}\sqrt{\frac{18}{5}\gamma}\right] + \left[-1 \cdot 3 \cdot \operatorname{erfc}\sqrt{10\gamma}\right] \right\rangle \\ &= \frac{3}{8}\operatorname{erfc}\sqrt{\frac{2}{5}\gamma} + \frac{1}{2}\operatorname{erfc}\sqrt{\frac{18}{5}\gamma} - \frac{3}{8}\operatorname{erfc}\sqrt{10\gamma} \end{aligned}$$

64-QAM

In this case, the first summation runs over three terms:

$$P_b = \frac{1}{24} \cdot \left\langle \sum_{i=0}^{3} \left[(-1)^{\lfloor \frac{i}{8} \rfloor} \cdot \left(1 - \left\lfloor \frac{i}{8} + \frac{1}{2} \right\rfloor\right) \cdot \operatorname{erfc}\left((2i+1)\sqrt{\frac{\gamma}{7}}\right) \right] + \sum_{i=0}^{5} \left[(-1)^{\lfloor \frac{i}{4} \rfloor} \cdot \left(2 - \left\lfloor \frac{i}{4} + \frac{1}{2} \right\rfloor\right) \cdot \operatorname{erfc}\left((2i+1)\sqrt{\frac{\gamma}{7}}\right) \right] + \sum_{i=0}^{6} \left[(-1)^{\lfloor \frac{i}{2} \rfloor} \cdot \left(4 - \left\lfloor \frac{i}{2} + \frac{1}{2} \right\rfloor\right) \cdot \operatorname{erfc}\left((2i+1)\sqrt{\frac{\gamma}{7}}\right) \right] \right\rangle$$

This time, we get a total of 17 inner summations instead of 5, as in the case of 16-QAM:

$$
\begin{aligned}
P_b = \frac{1}{24} \cdot \Bigg\langle & \left[1 \cdot 1 \cdot \operatorname{erfc}\sqrt{\frac{\gamma}{7}}\right] + \left[1 \cdot 1 \cdot \operatorname{erfc}\sqrt{\frac{9}{7}\gamma}\right] + \left[1 \cdot 1 \cdot \operatorname{erfc}\sqrt{\frac{25}{7}\gamma}\right] \\
& + \left[1 \cdot 1 \cdot \operatorname{erfc}\sqrt{7\gamma}\right] + \left[1 \cdot 2 \cdot \operatorname{erfc}\sqrt{\frac{\gamma}{7}}\right] + \left[1 \cdot 2 \cdot \operatorname{erfc}\sqrt{\frac{9}{7}\gamma}\right] \\
& + \left[1 \cdot 1 \cdot \operatorname{erfc}\sqrt{\frac{25}{7}\gamma}\right] + \left[1 \cdot 1 \cdot \operatorname{erfc}\sqrt{7\gamma}\right] + \left[-1 \cdot 1 \cdot \operatorname{erfc}\sqrt{\frac{81}{7}\gamma}\right] \\
& + \left[-1 \cdot 1 \cdot \operatorname{erfc}\sqrt{\frac{121}{7}\gamma}\right] + \left[1 \cdot 4 \cdot \operatorname{erfc}\sqrt{\frac{\gamma}{7}}\right] + \left[1 \cdot 3 \cdot \operatorname{erfc}\sqrt{\frac{9}{7}\gamma}\right] \\
& + \left[-1 \cdot 3 \cdot \operatorname{erfc}\sqrt{\frac{25}{7}\gamma}\right] + \left[-1 \cdot 2 \cdot \operatorname{erfc}\sqrt{7\gamma}\right] + \left[1 \cdot 2 \cdot \operatorname{erfc}\sqrt{\frac{81}{7}\gamma}\right] \\
& + \left[1 \cdot 1 \cdot \operatorname{erfc}\sqrt{\frac{121}{7}\gamma}\right] + \left[-1 \cdot 1 \cdot \operatorname{erfc}\sqrt{\frac{169}{7}\gamma}\right] \Bigg\rangle \\
= \frac{7}{24} & \operatorname{erfc}\sqrt{\frac{\gamma}{7}} + \frac{1}{4} \operatorname{erfc}\sqrt{\frac{9}{7}\gamma} - \frac{1}{24} \operatorname{erfc}\sqrt{\frac{25}{7}\gamma} + \frac{1}{24} \operatorname{erfc}\sqrt{\frac{81}{7}\gamma} - \frac{1}{24} \operatorname{erfc}\sqrt{\frac{169}{7}\gamma}
\end{aligned}
$$

Abbreviations and Acronyms

AC	Access Category
ACK	Acknowledgment
AIFS	Arbitration Interframe Space
AP	Access Point
ARQ	Automatic Repeat reQuest
AWGN	Additive White Gaussian Noise
BER	Bit Error Rate
BPSK	Binary Phase-Shift Keying
BSSID	Basic Service Set Identification
CBC-MAC	Cipher Block Chaining Message Authentication Code
CCM	Counter with CBC-MAC
CCMP	CCM mode Protocol
CRC	Cyclic Redundancy Check
CSMA/CA	Carrier Sense Multiple Access with Collision Avoidance
CSMA/CD	Carrier Sense Multiple Access with Collision Detection
CSRC	Contributing Source Identifier
CTP	Collection Tree Protocol
CTS	Clear To Send
DSSS	Direct Sequence Spread Spectrum
ETSI	European Telecommunications Standards Institute
EVM	Error Vector Magnitude
FEC	Forward Error Correction

IANA	Internet Assigned Numbers Authority
ISCD	Iterative Source–Channel Decoding
ITU	International Telecommunication Union
IV	Initialization Vector
MCS	Modulation and Coding Scheme
MAC	Medium Access Control
MIC	Message Integrity Code
MIMO	Multiple Input Multiple Output
MOS	Mean Opinion Score
MSS	Maximum Segment Size
MTU	Maximum Transmission Unit
NACK	Negative Acknowledgment
NAT	Network Address Translation
NAT-PMP	NAT Port Mapping Protocol
OFDM	Orthogonal Frequency-Division Multiplexing
OFRA	On-demand Feedback Rate Adaptation
OS	Operating System
PCP	Port Control Protocol
PDR	Packet Delivery Rate
PER	Packet Error Rate
PESQ	Perceptual Evaluation of Speech Quality
PHY	PHYsical layer
PLCP	Physical Layer Convergence Protocol
PLR	Packet Loss Rate
PSK	Phase-Shift Keying
QAM	Quadrature Amplitude Modulation
QoS	Quality of Service
QPSK	Quadrature Phase-Shift Keying

RSS	Received Signal Strength
RSSI	Received Signal Strength Indicator
RTCP	RTP Control Protocol
RTP	Real-Time Transport Protocol
RTS	Request To Send
SIFS	Short Interframe Space
SNR	Signal-to-Noise Ratio
SSRC	Synchronization Source Identifier
STA	Station
TDMA	Time Division Multiple Access
TID	Traffic Identifier
TKIP	Temporary Key Integrity Protocol
ToS	Type of Service
TTL	Time to Live
UPnP	Universal Plug and Play
VoIP	Voice over IP
WLAN	Wireless LAN
XOR	eXclusive OR

Bibliography

[3GPP03] Technical Specification Group Services and System Aspects—3G Security—Specification of the A5/3 Encryption Algorithms for GSM and ECSD, and the GEA3 Encryption Algorithm for GPRS, 3rd Generation Partnership Project Standard 3GPP TS 55.216, Revision 6.2.0, September 2003.

[3GPP06] Specification of the 3GPP Confidentiality and Integrity Algorithms UEA2 & UIA2—Document 2: SNOW 3G Specification, 3rd Generation Partnership Project Standard, Revision 1.1, September 2006.

[3GPP09] Technical Specification Group Services and System Aspects—3G Security—Specification of the A5/4 Encryption Algorithms for GSM and ECSD, and the GEA4 Encryption Algorithm for GPRS, 3rd Generation Partnership Project Standard 3GPP TS 55.226, Revision 9.0.0, September 2009.

[3GPP11] Specification of the 3GPP Confidentiality and Integrity Algorithms 128-EEA3 & 128-EIA3—Document 2: ZUC Specification, 3rd Generation Partnership Project Standard, Revision 1.6, June 2011.

[AB03] Stefan Alfredsson and Anna Brunstrom. TCP-L: Allowing Bit Errors in Wireless TCP. In *Proceedings of the Twelfth IST Summit on Mobile and Wireless Communications*, IST-MWC '03. IST, June 2003.

[ACS08] Marc Adrat, Thorsten Clevorn, and Laurent Schmalen. Iterative Source-Channel Decoding & Turbo DeCodulation. In Rainer Martin, Ulrich Heute, and Christiane Antweiler, editors, *Advances in Digital Speech Transmission*, chapter 13, pages 365–398. Wiley Online Library, 2008.

[AKH+15] Muhammad Hamad Alizai, Muhammad Moosa Khattak, Dong Han, Omprakash Gnawali, and Affan A. Syed. Recycling Corrupt Packets ofer Multiple Hops. In Tarik Abdelzaher, Nuno Pereira, and Eduardo Tovar, editors, *Wireless Sensor Networks*, volume 8965 of *Lecture Notes in Computer Science*, pages 242–249. Springer International Publishing, 2015.

[ASC14] Muhammad Naveed Aman, Biplab Sikdar, and Wai Kin Chan. Efficient Packet Recovery in Wireless Networks. In *Proceedings of the*

Twelfth IEEE Wireless Communications and Networking Conference, WCNC '14, pages 1791–1796. IEEE, April 2014.

[ath5k] Linux Wireless ath5k driver. `https://wireless.wiki.kernel.org/en/users/Drivers/ath5k`. [Online, accessed 2015-04-14].

[AVC05] Marc Adrat, Peter Vary, and Thorsten Clevorn. Optimized Bit Rate Allocation for Iterative Source-Channel Decoding and its Extension towards Multi-Mode Transmission. In *Proceedings of the 14th IST Mobile & Communications Summit*. EURASIP, June 2005.

[b43] Linux Wireless b43 driver. `https://wireless.kernel.org/en/users/Drivers/b43`. [Online, accessed 2015-04-14].

[BBC+98] Steven Blake, David L. Black, Mark A. Carlson, Elwyn Davies, Zheng Wang, and Walter Weiss. An Architecture for Differentiated Services. Request for Comments 2475, Internet Engineering Task Force, December 1998.

[BBC+04] Herbert Bos, Willem De Bruijn, Mihai Cristea, Trung Nguyen, and Georgios Portokalidis. FFPF: Fairly Fast Packet Filters. In *Proceedings of the Sixth USENIX symposium on Operating Systems Design & Implementation*, OSDI'04, pages 347–363. USENIX Association, December 2004.

[BBD+01] Carsten Bormann, Carsten Burmeister, Mikael Degermark, Hideaki Fukushima, Hans Hannu, Lars-Erik Jonsson, Rolf Hakenberg, Tmima Koren, Khiem Le, Zhigang Liu, Anton Martensson, Akihiro Miyazaki, Krister Svanbro, Thomas Wiebke, Takeshi Yoshimura, and Haihong Zheng. RObust Header Compression (ROHC): Framework and four profiles: RTP, UDP, ESP, and uncompressed. Request for Comments 3095, Internet Engineering Task Force, July 2001.

[BBD+02] Brian Babcock, Shivnath Babu, Mayur Datar, Rajeev Motwani, and Jennifer Widom. Models and Issues in Data Stream Systems. In *Proceedings of the Twenty-first ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, PODS '02, pages 1–16, New York, NY, USA, June 2002. ACM.

[BCK03] André Bourdoux, Boris Come, and Nadia Khaled. Non-reciprocal Transceivers in OFDM/SDMA Systems: Impact and Mitigation. In *Proceedings of the 2003 Radio and Wireless Conference*, RAWCON '03, pages 183–186. IEEE, August 2003.

[BCS94] Robert Braden, David Clark, and Scott Shenker. Integrated Services Architecture. Request for Comments 1633, Internet Engineering Task Force, June 1994.

[BG96] Claude Berrou and Alain Glavieux. Near Optimum Error Correcting Coding And Decoding: Turbo-Codes. *IEEE Transactions on Communications*, 44(10):1261–1271, October 1996.

[BKH+11] Anirudh Badam, Michael Kaminsky, Dongsu Han, Konstantina Papagiannaki, David G. Andersen, and Srinivasan Seshan. The Hare and the Tortoise: Taming Wireless Losses by Exploiting Wired Reliability. In *Proceedings of the Twelfth ACM International Symposium on Mobile Ad Hoc Networking and Computing*, MobiHoc '11, pages 7:1–7:11, New York, NY, USA, May 2011. ACM.

[BLJL06] George Dean Bissias, Marc Liberatore, David Jensen, and Brian Neil Levine. Privacy Vulnerabilities in Encrypted HTTP Streams. In George Danezis and David Martin, editors, *Privacy Enhancing Technologies*, volume 3856 of *Lecture Notes in Computer Science*, pages 1–11. Springer Berlin Heidelberg, 2006.

[BLK+01] Rajesh Krishna Balan, Boon Peng Lee, K. Renjish R. Kumar, Lillykutty Jacob, Winston Khoon Guan Seah, and Akkihebbal L. Ananda. TCP HACK: TCP Header Checksum Option to Improve Performance over Lossy Links. In *Proceedings of the Twentieth Annual Joint Conference of the IEEE Computer and Communications Societies*, volume 1 of *INFOCOM '01*, pages 309–318. IEEE, April 2001.

[BLV+10] Tobias Breddermann, Helge Lüders, Peter Vary, Ismet Aktaş, and Florian Schmidt. Iterative Source-Channel Decoding with Cross-Layer Support for Wireless VoIP. In Rudolf Mathar and Christoph Ruland, editors, *Proceedings of International ITG Conference on Source and Channel Coding*, SCC '10, Berlin, Germany, January 2010. ITG, VDE Verlag.

[Bor14] Carsten Bormann. 6LoWPAN-GHC: Generic Header Compression for IPv6 over Low-Power Wireless Personal Area Networks (6LoWPANs). Request for Comments 7400, Internet Engineering Task Force, November 2014.

[Bra89] Richard Braden. Requirements for Internet Hosts – Communication Layers. Request for Comments 1122, Internet Engineering Task Force, October 1989.

[BRC60] Raj Chandra Bose and Dwijendra Kumar Ray-Chaudhuri. On A Class of Error Correcting Binary Group Codes. *Information and Control*, 3(1):68–79, March 1960.

[Bre01] Leo Breiman. Random Forests. *Machine Learning*, 45(1):5–32, October 2001.

[BRENM+10] Nicola Baldo, Manuel Requena-Esteso, José Núñez-Martínez, Marc Portolès-Comeras, Jaume Nin-Guerrero, Paolo Dini, and Josep Mangues-Bafalluy. Validation of the IEEE 802.11 MAC Model in the ns3 Simulator Using the EXTREME Testbed. In *Proceedings of the 3rd International ICST Conference on Simulation Tools and*

Techniques, SIMUTools '10, pages 1–9, Brussels, Belgium, March 2010. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering).

[BTS05] Giuseppe Bianchi, Ilenia Tinnirello, and Luca Scalia. Understanding 802.11e Contention-Based Prioritization Mechanisms and Their Coexistence with Legacy 802.11 Stations. *IEEE Network*, 19(4):28–34, July 2005.

[BV09] Roohi Banu and Tanya Vladimirova. Fault-Tolerant Encryption for Space Applications. *IEEE Transactions on Aerospace and Electronic Systems*, 45(1):266–279, January 2009.

[Cas07] Stephen L. Casner. Media Type Registration of Payload Formats in the RTP Profile for Audio and Video Conferences. Request for Comments 4856, Internet Engineering Task Force, March 2007.

[CDGS07] Manuel Crotti, Maurizio Dusi, Francesco Gringoli, and Luca Salgarelli. Traffic Classification Through Simple Statistical Fingerprinting. *ACM SIGCOMM Computer Communication Review*, 37(1):5–16, January 2007.

[CET+11] Michelle Cotton, Lars Eggert, Joseph Touch, Magnus Westerlund, and Stuart Cheshire. Internet Assigned Numbers Authority (IANA) Procedures for the Management of the Service Name and Transport Protocol Port Number Registry. Request for Comments 6335, Internet Engineering Task Force, August 2011.

[CGQ09] Xi Chen, Prateek Gangwal, and Daji Qiao. Practical Rate Adaptation in Mobile Environments. In *IEEE International Conference on Pervasive Computing and Communications*, PerCom '09, pages 204–213. IEEE, March 2009.

[CGQ12] Xi Chen, Prateek Gangwal, and Daji Qiao. RAM: Rate Adaptation in Mobile Environments. *IEEE Transactions on Mobile Computing*, 11(3):464–477, March 2012.

[CJ99] Stephen L. Casner and Van Jacobson. Compressing IP/UDP/RTP Headers for Low-Speed Serial Links. Request for Comments 2508, Internet Engineering Task Force, February 1999.

[CK10] Joseph Camp and Edward Knightly. Modulation Rate Adaptation in Urban and Vehicular Environments: Cross-Layer Implementation and Experimental Evaluation. *IEEE/ACM Transactions on Networking*, 18(6):1949–1962, December 2010.

[CK13] Stuart Cheshire and Marc Krochmal. NAT Port Mapping Protocol (NAT-PMP). Request for Comments 6886, Internet Engineering Task Force, August 2013.

[CV95] Corinna Cortes and Vladimir Vapnik. Support-Vector Networks. *Machine Learning*, 20(3):273–297, September 1995.

[CY02] Kyongkuk Cho and Dongweon Yoon. On the General BER Expression of One- and Two-Dimensional Amplitude Modulations. *IEEE Transactions on Communications*, 50(7):1074–1080, July 2002.

[DCGS09] Maurizio Dusi, Manuel Crotti, Francesco Gringoli, and Luca Salgarelli. Tunnel Hunter: Detecting application-layer tunnels with statistical fingerprinting. *Computer Networks*, 53(1):81–97, January 2009.

[DD12] Pralhad Deshpande and Samir R. Das. BRAVE: Bit-rate Adaptation in Vehicular Environments. In *Proceedings of the Ninth ACM International Workshop on Vehicular Inter-networking, Systems, and Applications*, VANET '12, pages 33–42, New York, NY, USA, June 2012. ACM.

[DNP99] Mikael Degermark, Bjorn Nordgren, and Stephen Pink. IP Header Compression. Request for Comments 2507, Internet Engineering Task Force, February 1999.

[ETSI00] Digital cellular telecommunications system (Phase 2+) (GSM)—Adaptive Multi-Rate (AMR) speech transcoding (GSM 06.90 version 7.2.1 Release 1998), European Telecommunication Standards Institute Standard EN 301 704, Revision 7.2.1, April 2000.

[ETSI06] Telecommunications and Internet converged Services and Protocols for Advanced Networking (TISPAN)—Review of available material on QoS requirements of Multimedia Services, European Telecommunications Standards Institute Standard TR 102 479, Revision 1.1.1, February 2006.

[Fan12] Erwin Fang. Implementing On-Demand Rate Adaptation for IEEE 802.11. Bachelor's thesis, Rheinisch-Westfälische Technische Hochschule Aachen, August 2012.

[FDC84] David J. Farber, Gary S. Delp, and Thomas M. Conte. A Thinwire Protocol for connecting personal computers to the INTERNET. Request for Comments 914, Internet Engineering Task Force, September 1984.

[FS97] Yoav Freund and Robert E Schapire. A Decision-Theoretic Generalization of On-Line Learning and an Application to Boosting. *Journal of Computer and System Sciences*, 55(1):119–139, August 1997.

[FZJJ06] Gerhard Fettweis, Ernesto Zimmermann, Volker Jungnickel, and Eduard A. Jorswieck. Challenges in Future Short Range Wireless Systems. *IEEE Vehicular Technology Magazine*, 1(2):24–31, June 2006.

[Gar07] Vijay K. Garg. *Wireless Communications and Networking.* Morgan Kaufmann, 1 edition, 2007.

[GDFM+12] José Luis García-Dorado, Alessandro Finamore, Marco Mellia, Michela Meo, and Maurizio M. Munafo. Characterization of ISP Traffic: Trends, User Habits, and Access Technology Impact. *IEEE Transactions on Network and Service Management*, 9(2):142–155, June 2012.

[GK08] Shyamnath Gollakota and Dina Katabi. ZigZag decoding: Combating Hidden Terminals in Wireless Networks. In *Proceedings of the 36th ACM SIGCOMM Conference*, SIGCOMM '08, pages 159–170, New York, NY, USA, August 2008. ACM.

[gnuradio] The GNU Software Radio. `http://www.gnuradio.org`. [Online, accessed 2015-04-14].

[Gör00] Norbert Görtz. Iterative source-channel decoding using soft-in/soft-out decoders. In *Proceedings of the IEEE International Symposium on Information Theory*, ISIT '00, page 173. IEEE, June 2000.

[Göt11] Mario Göttgens. Heuristic Packet Repair for UDP/IP in the Linux Network Stack. Bachelor's thesis, Rheinisch-Westfälische Technische Hochschule Aachen, March 2011.

[Göt13] Mario Göttgens. On-Demand Feedback Rate Adaptation in the Linux Network Stack. Master's thesis, Rheinisch-Westfälische Technische Hochschule Aachen, June 2013.

[GZK05] Mohamed Medhat Gaber, Arkady Zaslavsky, and Shonali Krishnaswamy. Mining Data Streams: A Review. *SIGMOD Record*, 34(2):18–26, June 2005.

[Hen11] Martin Henze. A Machine-Learning Packet-Classification Tool for Processing Corrupted Packets on End Hosts. Diploma thesis, Rheinisch-Westfälische Technische Hochschule Aachen, March 2011.

[HGC10] Bo Han, Francesco Gringoli, and Luca Cominardi. Bologna: Block-based 802.11 Transmission Recovery. In *Proceedings of the 2010 ACM Workshop on Wireless of the Students, by the Students, for the Students*, S3 '10, pages 45–48, New York, NY, USA, September 2010. ACM.

[Hit11] Anwar Hithnawi. An On-Demand Rate-Adaptation Mechanism for IEEE 802.11 Networks. Master's thesis, Rheinisch-Westfälische Technische Hochschule Aachen, December 2011.

[HJL+09] Bo Han, Lusheng Ji, Seungjoon Lee, Bobby. Bhattacharjee, and Robert R. Miller. All Bits Are Not Equal – A Study of IEEE 802.11 Communication Bit Errors. In *Proceedings of the Twenty-Eigth IEEE Conference on Computer Communications*, INFOCOM '09, pages 1602–1610. IEEE, April 2009.

[HJL+12] Bo Han, Lusheng Ji, Seungjoon Lee, Bobby Bhattacharjee, and Robert R. Miller. Are All Bits Equal?: Experimental Study of IEEE 802.11 Communication Bit Errors. *IEEE/ACM Transactions on Networking*, 20(6):1695–1706, December 2012.

[HOP96] Joachim Hagenauer, Elke Offer, and Lutz Papke. Iterative Decoding of Binary Block and Convolutional cCodes. *IEEE Transactions on Information Theory*, 42(2):429–445, March 1996.

[hostapd] hostapd, a User-Space Daemon for Access Point and Authentication Servers. `http://w1.fi/hostapd/`. [Online, accessed 2015-04-14].

[HRFR06] Thomas R. Henderson, Sumit Roy, Sally Floyd, and George F. Riley. ns-3 Project Goals. In *Proceeding from the 2006 Workshop on Ns-2: The IP Network Simulator*, WNS2 '06, New York, NY, USA, October 2006. ACM.

[HRNK04] Florian Hammer, Peter Reichl, Tomas Nordström, and Gernot Kubin. Corrupted Speech Data Considered Useful: Improving Perceived Speech Quality of VoIP over Error-Prone Channels. *Acta acustica*, 90(6):1052–1060, December 2004.

[HSG+10] Bo Han, Aaron Schulman, Francesco Gringoli, Neil Spring, Bobby Bhattacharjee, Lorenzo Nava, Lusheng Ji, Seungjoon Lee, and Robert Miller. Maranello: Practical Partial Packet Recovery for 802.11. In *Proceedings of the Seventh USENIX conference on Networked systems design and implementation*, NSDI '10, Berkeley, CA, USA, April 2010. USENIX Association.

[HT11] Jonathan W. Hui and Pascal Thubert. Compression Format for IPv6 Datagrams over IEEE 802.15.4-Based Networks. Request for Comments 6282, Internet Engineering Task Force, September 2011.

[HVB01] Gavin Holland, Nitin Vaidya, and Paramvir Bahl. A Rate-Adaptive MAC Protocol for Multi-Hop Wireless Networks. In *Proceedings of the 7th Annual International Conference on Mobile Computing and Networking*, MobiCom '01, pages 236–251, New York, NY, USA, July 2001. ACM.

[HWM+14] Frederik Hermans, Hjalmar Wennerström, Liam McNamara, Christian Rohner, and Per Gunningberg. All Is Not Lost: Understanding and Exploiting Packet Corruption in Outdoor Sensor Networks. In Bhaskar Krishnamachari, AmyL. Murphy, and Niki Trigoni, editors, *Wireless Sensor Networks*, volume 8354 of *Lecture Notes in Computer Science*, pages 116–132. Springer International Publishing, 2014.

[IANA14] Internet Assigned Numbers Authority. Real-Time Transport Protocol (RTP) Parameters. `http://www.iana.org/assignments/rtp-parameters/rtp-parameters.xhtml`, July 2014. [Online, accessed 2015-04-14].

[IANA15] Internet Assigned Numbers Authority. Assigned Internet Protocol Numbers. http://www.iana.org/assignments/protocol-numbers/protocol-numbers.xhtml, April 2015. [Online, accessed 2015-04-14].

[IEEE05] IEEE Standard for Information technology—Telecommunications and information exchange between systems—Local and metropolitan area networks—Specific requirements, Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications, Amendment 8: Medium Access Control (MAC) Quality of Service Enhancements, Institute of Electrical and Electronics Engineers Standard 802.11e, Revision 2005, November 2005.

[IEEE09] IEEE Standard for Information technology—Telecommunications and information exchange between systems—Local and metropolitan area networks—Specific requirements, Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications, Amendment 5: Enhancements for Higher Throughput, Institute of Electrical and Electronics Engineers Standard 802.11n, Revision 2009, October 2009.

[IEEE11] IEEE Standard for Local and metropolitan area networks—Part 15.4: Low-Rate Wireless Personal Area Networks (LR-WPANs), Institute of Electrical and Electronics Engineers Standard 802.15.4, Revision 2011, June 2011.

[IEEE12a] IEEE Standard for Ethernet, Institute of Electrical and Electronics Engineers Standard 802.3, Revision 2012, December 2012.

[IEEE12b] IEEE Standard for Information technology—Telecommunications and information exchange between systems—Local and metropolitan area networks—Specific requirements, Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications, Institute of Electrical and Electronics Engineers Standard 802.11, Revision 2012, March 2012.

[IEEE13] IEEE Standard for Information technology—Telecommunications and information exchange between systems—Local and metropolitan area networks—Specific requirements, Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications, Amendment 4: Enhancements for Very High Throughput for Operation in Bands below 6 GHz, Institute of Electrical and Electronics Engineers Standard 802.11ac, Revision 2013, December 2013.

[IEEE14] IEEE Standard for Local and Metropolitan Area Networks—Bridges and Bridged Networks, Institute of Electrical and Electronics Engineers Standard 802.1Q, Revision 2014, November 2014.

[ISO11] Information technology – UPnP Device Architecture, International Organization for Standardization Standard 23941-1, Revision 2, September 2011.

[ITU01] ITU-T Recommendation P.862: Perceptual evaluation of speech quality (PESQ): An objective method for end-to-end speech quality assessment of narrow-band telephone networks and speech codecs, International Telecommunication Union Standard, February 2001.

[ITU03] ITU-T Recommendation G.114: One-way transmission time, International Telecommunication Union Standard, May 2003.

[Jac90] Van Jacobson. Compressing TCP/IP Headers. Request for Comments 1144, Internet Engineering Task Force, February 1990.

[JB07] Kyle Jamieson and Hari Balakrishnan. PPR: Partial Packet Recovery for Wireless Networks. In *Proceedings of the 35th ACM SIGCOMM Conference*, SIGCOMM '07, pages 409–420, New York, NY, USA, August 2007. ACM.

[Jia06] Wenyu Jiang. Bit Error Correction without Redundant Data: a MAC Layer Technique for 802.11 Networks. In *Proceedings of the 4th International Symposium on Modeling and Optimization in Mobile, Ad Hoc and Wireless Networks*, WiOPT '06, pages 1–8. IEEE, April 2006.

[JK09] Szymon Jakubczak and Dina Katabi. One-Size-Fits-All Wireless Video. In *Proceedings of the Eigth Workshop on Hot Topics in Networks*, HotNets-VIII, pages 449–450, New York, NY, USA, August 2009. ACM.

[JP04] Lars-Erik Johnsson and Ghyslain Pelletier. RObust Header Compression (ROHC): A Compression Profile for IP. Request for Comments 3843, Internet Engineering Task Force, June 2004.

[JWS08] Glenn Judd, Xiaohui Wang, and Peter Steenkiste. Efficient Channel-aware Rate Adaptation in Dynamic Environments. In *Proceedings of the Sixth International Conference on Mobile systems, applications, and services*, MobiSys '08, pages 118–131, New York, NY, USA, June 2008. ACM.

[KKCQ06] Jongseok Kim, Seongkwan Kim, Sunghyun Choi, and Daji Qiao. CARA: Collision-Aware Rate Adaptation for IEEE 802.11 WLANs. In *Proceedings of the Twenty-Fifth Annual Joint Conference of the IEEE Computer and Communications Societies*, INFOCOM '06, pages 1–11. IEEE, April 2006.

[KM97] Ad Kamerman and Leo Monteban. WaveLAN®-II: A High-Performance Wireless LAN for the Unlicensed Band. *Bell Labs Technical Journal*, 2(3):118–133, 1997.

[Koo02] Philip Koopman. 32-Bit Cyclic Redundancy Codes For Internet Applications. In *Proceedings of the 32nd International Conference on Dependable Systems and Networks*, DSN '02, pages 459–468. IEEE Computer Society, June 2002.

[KR07] Syed Ali Khayam and Hayder Radha. Maximum-Likelihood Header Estimation: A Cross-Layer Methodology for Wireless Multimedia. *IEEE Transactions on Wireless Communications*, 6(11):3946–3954, November 2007.

[KR13] James F. Kurose and Keith W. Ross. *Computer Networking*. Pearson Education, 6th edition, 2013. International Edition.

[LAN+12] Jie Li, Andreas Aurelius, Viktor Nordell, Manxing Du, Åke Arvidsson, and Maria Kihl. A Five Year Perspective of Traffic Pattern Evolution in a Residential Broadband Access Network. In *Proceedings of the Twenty-First Future Network & Mobile Summit*, FutureNet '12. IEEE, July 2012.

[LDP99] Lars-Åke Larzon, Mikael Degermark, and Stephen Pink. UDP Lite for Real Time Multimedia Applications. Technical Report HPL-IRI-1999-001, Hewlett-Packard Laboratories, April 1999.

[LDP+04] Lars-Åke Larzon, Mikael Degermark, Stephen Pink, Lars-Erik Jonsson, and Godred Fairhurst. The Lightweight User Datagram Protocol (UDP-Lite). Request for Comments 3828, Internet Engineering Task Force, July 2004.

[L'E99] Pierre L'Ecuyer. Good Parameters and Implementations for Combined Multiple Recursive Random Number Generators. *Operations Research*, 47(1):159–164, January 1999.

[Led12] Matthias Lederhofer. Classifying Corrupted Network Packets for Error-Tolerant Streaming Applications. Diploma thesis, Rheinisch-Westfälische Technische Hochschule Aachen, August 2012.

[LH06] Mathieu Lacage and Thomas R Henderson. Yet Another Network Simulator. In *Proceedings of the 2006 Workshop on ns-2, the IP Network Simulator*, WNS2 '06, New York, NY, USA, October 2006. ACM.

[linphone] Linphone, an open-source VoIP software. `http://www.linphone.org`. [Online, accessed 2015-04-14].

[LKK08] Kate Ching-Ju Lin, Nate Kushman, and Dina Katabi. ZipTx: Harnessing Partial Packets in 802.11 Networks. In *Proceedings of the 14th Annual International Conference on Mobile Computing and Networking*, MobiCom '08, pages 351–362, New York, NY, USA, September 2008. ACM.

[LL04] Patrick Pak-kit Lam and Soung C. Liew. UDP-Liter: An Improved UDP Protocol for Real-Time Multimedia Applications over Wireless Links. In *Proceedings of the 1st International Symposium on Wireless Communication Systems*, ISWCS '04, pages 314–318. IEEE, September 2004.

[LL06] Marc Liberatore and Brian Neil Levine. Inferring the Source of Encrypted HTTP Connections. In *Proceedings of the 13th ACM Conference on Computer and Communications Security*, CCS '06, pages 255–263, New York, NY, USA, November 2006. ACM.

[LMT04] Mathieu Lacage, Mohammad Hossein Manshaei, and Thierry Turletti. IEEE 802.11 Rate Adaptation: A Practical Approach. In *Proceedings of the 7th ACM International Symposium on Modeling, analysis and Simulation of Wireless and Mobile systems*, MSWiM '04, pages 126–134, New York, NY, USA, October 2004. ACM.

[MBK05] Allen Miu, Hari Balakrishnan, and Can Emre Koksal. Improving Loss Resilience with Multi-Radio Diversity in Wireless Networks. In *Proceedings of the 11th Annual International Conference on Mobile Computing and Networking*, MobiCom '05, pages 16–30, New York, NY, USA, August 2005. ACM.

[MD90] Jeffrey Mogul and Steve Deering. Path MTU Discovery. Request for Comments 1191, Internet Engineering Task Force, November 1990.

[MH07] Matt Mathis and John W. Heffner. Packetization Layer Path MTU Discovery. Request for Comments 4821, Internet Engineering Task Force, March 2007.

[minstrel] The Minstrel Rate Control Algorithm. `http://linuxwireless.org/en/developers/Documentation/mac80211/RateControl/minstrel`. [Online, accessed 2015-04-14].

[MJ93] Steven McCanne and Van Jacobson. The BSD Packet Filter: A New Architecture for User-Level Packet Capture. In *Proceedings of the USENIX Winter 1993 Technical Conference*, USENIX '93. USENIX Association, January 1993.

[MLKD10] Cédric Marin, Yann Leprovost, Michael Kieffer, and Pierre Duhamel. Robust MAC-Lite and Soft Header Recovery for Packetized Multimedia Transmission. *IEEE Transactions on Communications*, 58(3):775–784, March 2010.

[MM13] Travis Mandel and Jens Mache. Practical Error Correction for Resource-Constrained Wireless Networks: Unlocking the Full Power of the CRC. In *Proceedings of the 11th ACM Conference on Embedded Networked Sensor Systems*, SenSys '13, pages 3:1–3:14, New York, NY, USA, November 2013. ACM.

[MPC+10] Luca Mottola, Gian Pietro Picco, Matteo Ceriotti, Ştefan Gună, and Amy L. Murphy. Not All Wireless Sensor Networks Are Created Equal: A Comparative Study on Tunnels. *ACM Transactions on Sensor Networks*, 7(2):15:1–15:33, September 2010.

[MSA06] Patrick Murphy, Ashu Sabharwal, and Behnaam Aazhang. Design of WARP: A Flexible Wireless Open-Access Research Platform. In *Proceedings of the 14th European Signal Processing Conference*, EUSIPCO '06. European Association for Signal Processing (EURASIP), September 2006.

[MZ05] Andrew W. Moore and Denis Zuev. Internet Traffic Classification Using Bayesian Analysis Techniques. *SIGMETRICS Performance Evaluation Review*, 33(1):50–60, June 2005.

[NBBB98] Kathleen Nichols, Steven Blake, Fred Baker, and David L. Black. Definition of the Differentiated Services Field (DS Field) in the IPv4 and IPv6 Headers. Request for Comments 2474, Internet Engineering Task Force, December 1998.

[netem] The Linux Foundation. netem, a Network Emulator. `http://www.linuxfoundation.org/collaborate/workgroups/networking/netem`. [Online, accessed 2015-04-14].

[NOCW07] Sam Nguyen, Clayton Okino, Loren Clare, and William Walsh. Space-Based Voice over IP Networks. In *Proceedings of the Twenty-Eighth IEEE Aerospace Conference*, Aeroconf '07, pages 1–11. IEEE, March 2007.

[ns3] ns-3, a discrete-event network simulator for Internet systems. `http://www.nsnam.org/`. [Online, accessed 2015-04-14].

[ns3man] ns-3.9 Manual. `http://www.nsnam.org/docs/release/3.9/manual/manual.html`. [Online, accessed 2015-04-14].

[Orl12] David Orlea. Error Tolerance for the Real-Time Transport Protocol (RTP). Bachelor's thesis, Rheinisch-Westfälische Technische Hochschule Aachen, March 2012.

[ortp] oRTP, A Real-Time Transport Protocol (RTP, RFC3550) library. `http://www.linphone.org/technical-corner/ortp/`. [Online, accessed 2015-04-14].

[OZL14] Jiajue Ou, Yuanqing Zheng, and Mo Li. MISC: Merging Incorrect Symbols using Constellation Diversity for 802.11 Retransmission. In *Proceedings of the Thirty-Third IEEE Conference on Computer Communications*, INFOCOM '14, pages 2472–2480. IEEE, April 2014.

[PB61] William W. Peterson and David T. Brown. Cyclic Codes for Error Detection. *Proceedings of the IRE*, 49(1):228–235, January 1961.

[PBB05] Michal Przybylski, Bartosz Belter, and Artur Binczewski. Shall we worry about Packet Reordering? *Computational Methods in Science and Technology*, 11(2):141–146, 2005.

[PEG12] Oscar Puñal, Humberto Escudero, and James Gross. Power Loading: Candidate for Future WLANs? In *Proceedings of the Thirteenth International Symposium on a World of Wireless, Mobile and Multimedia Networks*, WoWMoM '12. IEEE, June 2012.

[Pel05] Ghyslain Pelletier. RObust Header Compression (ROHC): Profiles for User Datagram Protocol (UDP) Lite. Request for Comments 4019, Internet Engineering Task Force, April 2005.

[PG12] Carlos Pignataro and Fernando Gont. Formally Deprecating Some IPv4 Options. Request for Comments 6814, Internet Engineering Task Force, November 2012.

[PH09] Guangyu Pei and Tom Henderson. Validation of the ns-3 802.11b PHY model. Technical report, Boeing Research & Technology, Seattle, WA, USA, May 2009.

[PH10] Guangyu Pei and Tom Henderson. Validation of OFDM error rate model in ns-3. Technical report, Boeing Research & Technology, Seattle, WA, USA, 2010.

[PHW+10] Ioannis Pefkianakis, Yun Hu, Starsky H.Y. Wong, Hao Yang, and Songwu Lu. MIMO Rate Adaptation in 802.11n Wireless Networks. In *Proceedings of the Sixteenth Annual International Conference on Mobile Computing and Networking*, MobiCom '10, pages 257–268, New York, NY, USA, September 2010. ACM.

[Pos81] Jon Postel. Internet Protocol. Request for Comments 791, Internet Engineering Task Force, September 1981.

[Pro85] John Proakis. *Digital Communications*. McGraw-Hill, international student edition, 1985.

[PSJW13] Ghyslain Pelletier, Kristofer Sandlund, Lars-Erik Jonsson, and Mark A. West. RObust Header Compression (ROHC): A Profile for TCP/IP (ROHC-TCP). Request for Comments 6846, Internet Engineering Task Force, January 2013.

[PT87] Michael B. Pursley and D. J. Taipale. Error Probabilities for Spread-Spectrum Packet Radio with Convolutional Codes and Viterbi Decoding. *IEEE Transactions on Communications*, 35(1):1–12, January 1987.

[QCJS03] Daji Qiao, Sunghyun Choi, Amit Jain, and Kang G. Shin. MiSer: An Optimal Low-energy Transmission Strategy for IEEE 802.11a/h. In *Proceedings of the 9th Annual International Conference on Mobile*

Computing and Networking, MobiCom '03, pages 161–175, New York, NY, USA, September 2003. ACM.

[RFB01] K. K. Ramakrishnan, Sally Floyd, and David L. Black. The Addition of Explicit Congestion Notification (ECN) to IP. Request for Comments 3168, Internet Engineering Task Force, September 2001.

[Rij94] Anil Rijsinghani. Computation of the Internet Checksum via Incremental Update. Request for Comments 1624, Internet Engineering Task Force, May 1994.

[Riv97] Ronald L. Rivest. All-or-Nothing Encryption and the Package Transform. In Eli Biham, editor, *Fast Software Encryption*, volume 1267 of *Lecture Notes in Computer Science*, pages 210–218. Springer Berlin Heidelberg, 1997.

[RKZG08] Kishore Ramachandran, Ravi Kokku, Honghai Zhang, and Marco Gruteser. Symphony: Synchronous Two-phase Rate and Power Control in 802.11 WLANs. In *Proceedings of the Sixth International Conference on Mobile Systems, Applications, and Services*, MobiSys '08, pages 132–145, New York, NY, USA, June 2008. ACM.

[Ros08] Jonathan Rosenberg. UDP and TCP as the New Waist of the Internet Hourglass. Internet-draft, Internet Engineering Task Force, February 2008.

[SAAW11] Florian Schmidt, Muhammad Hamad Alizai, Ismet Aktaş, and Klaus Wehrle. Refector: Heuristic Header Error Recovery for Error-Tolerant Transmissions. In *Proceedings of the Seventh COnference on emerging Networking EXperiments and Technologies*, CoNEXT '11, pages 22:1–22:12, New York, NY, USA, December 2011. ACM.

[SC03] Henning Schulzrinne and Stephen L. Casner. RTP Profile for Audio and Video Conferences with Minimal Control. Request for Comments 3551, Internet Engineering Task Force, July 2003.

[SCFJ03] Henning Schulzrinne, Stephen L. Casner, Ron Frederick, and Van Jacobson. RTP: A Transport Protocol for Real-Time Applications. Request for Comments 3550, Internet Engineering Task Force, July 2003.

[Sch96] Bruce Schneier. *Applied Cryptography, Second Edition: Protocols, Algorithms, and Source Code in C*. John Wiley & Sons, 2nd edition, 1996.

[SCHW14] Florian Schmidt, Matteo Ceriotti, Niklas Hauser, and Klaus Wehrle. HotBox: Testing Temperature Effects in Sensor Networks. Technical Report AIB-2014-14, Rheinisch-Westfälische Technische Hochschule Aachen, Department of Computer Science, December 2014.

[SCW13] Florian Schmidt, Matteo Ceriotti, and Klaus Wehrle. Bit Error Distribution and Mutation Patterns of Corrupted Packets in Low-power Wireless Networks. In *Proceedings of the 8th ACM International Workshop on Wireless Network Testbeds, Experimental Evaluation & Characterization*, WiNTECH '13, pages 49–56, New York, NY, USA, September 2013. ACM.

[SHP+12] Florian Schmidt, Anwar Hithnawi, Oscar Puñal, James Gross, and Klaus Wehrle. A Receiver-Based 802.11 Rate Adaptation Scheme with On-Demand Feedback. In *Proceedings of the 23rd International Symposium on Personal Indoor and Mobile Radio Communications*, PIMRC '12, pages 399–405. IEEE, August 2012.

[SHW14] Florian Schmidt, Martin Henze, and Klaus Wehrle. Piccett: Protocol-Independent Classification of Corrupted Error-Tolerant Traffic. In *Proceedings of the Nineteenth IEEE Symposium on Computers and Communications*, ISCC '14, pages 1–7. IEEE, June 2014.

[SKSK02] Bahareh Sadeghi, Vikram Kanodia, Ashutosh Sabharwal, and Edward W. Knightly. Opportunistic Media Access for Multirate Ad Hoc Networks. In *Proceedings of the 8th Annual International Conference on Mobile Computing and Networking*, MobiCom '02, pages 24–35, New York, NY, USA, September 2002. ACM.

[SMW07] Heiko Schwarz, Detlev Marpe, and Thomas Wiegand. Overview of the Scalable Video Coding Extension of the H.264/AVC Standard. *IEEE Transactions on Circuits and Systems for Video Technology*, 17(9):1103–1120, September 2007.

[SOW13a] Florian Schmidt, David Orlea, and Klaus Wehrle. A Heuristic Header Error Recovery Scheme for RTP. In *Proceedings of the 10th Annual IEEE/IFIP Conference on Wireless On-Demand Network Systems and Services*, WONS '13. IEEE, March 2013.

[SOW13b] Florian Schmidt, David Orlea, and Klaus Wehrle. Support for Error Tolerance in the Real-Time Transport Protocol. Technical Report AIB-2013-19, Rheinisch-Westfälische Technische Hochschule Aachen, Department of Computer Science, December 2013.

[SSCN10] Souvik Sen, Naveen Santhapuri, Romit Roy Choudhury, and Srihari Nelakuditi. AccuRate: Constellation Based Rate Estimation in Wireless Networks. In *Proceedings of the Seventh USENIX conference on Networked systems design and implementation*, NSDI '10, Berkeley, CA, USA, April 2010. USENIX Association.

[SSGA10] Arne Schmitz, Marc Schinnenburg, James Gross, and Ana Aguiar. Channel Modeling. In Klaus Wehrle, Mesut Güneş, and James Gross, editors, *Modeling and Tools for Network Simulation*, chapter 11, pages 191–234. Springer Berlin Heidelberg, 2010.

[SSJ+08] Markus Schnell, Markus Schmidt, Manuel Jander, Tobias Albert, Ralf Geiger, Vesa Ruoppila, Per Ekstrand, and Grill Bernhard. MPEG-4 Enhanced Low Delay AAC – A New Standard for High Quality Communication. In *Proceedings of the 125th Audio Engineering Society Convention*. Audio Engineering Society, October 2008.

[SWLX07] Johan Sjoberg, Magnus Westerlund, Ari Lakaniemi, and Qiaobing Xie. RTP Payload Format and File Storage Format for the Adaptive Multi-Rate (AMR) and Adaptive Multi-Rate Wideband (AMR-WB) Audio Codecs. Request for Comments 4867, Internet Engineering Task Force, April 2007.

[SWT01] Dawn Xiaodong Song, David Wagner, and Xuqing Tian. Timing Analysis of Keystrokes and Timing Attacks on SSH. In *Proceedings of the 10th USENIX Security Symposium*, SSYM '01, Berkeley, CA, USA, August 2001. USENIX Association.

[Tai92] Chen-To Tai. Complementary Reciprocity Theorems in Electromagnetic Theory. *IEEE Transactions on Antennas and Propagation*, 40(6):675–681, June 1992.

[Tor04] Linus Torvalds. LKML archive: How to use floating point in a module? `https://lkml.org/lkml/2004/5/31/5`, May 2004. [Online, accessed 2015-04-14].

[TW11] Andrew S. Tanenbaum and David J. Wetherall. *Computer Networks*. Pearson Education, 5th edition, 2011. International Edition.

[USRP] Ettus Research. The Universal Software Radio Peripheral. `http://www.ettus.com`. [Online, accessed 2015-04-14].

[VBJ09] Mythili Vutukuru, Hari Balakrishnan, and Kyle Jamieson. Cross-Layer Wireless Bit Rate Adaptation. In *Proceedings of the 37th ACM SIGCOMM Conference*, SIGCOMM '09, pages 3–14, New York, NY, USA, August 2009. ACM.

[Vit71] Andrew J. Viterbi. Convolutional Codes and Their Performance in Communication Systems. *IEEE Transactions on Communication Technology*, 19(5):751–772, October 1971.

[VVT12] Jean-Marc Valin, Koen Vos, and Timothy B. Terriberry. Definition of the Opus Audio Codec. Request for Comments 6716, Internet Engineering Task Force, September 2012.

[War07] Henry S. Warren, Jr. The Quest for an Accelerated Population Count. In Andy Oram and Greg Wilson, editors, *Beautiful Code*, chapter 10, pages 147–160. O'Reilly Media, 2007.

[WBBJ67] Paul Watzlawick, Janet Beavin Bavelas, and Don Jackson. Some Tentative Axioms of Communication. In *Pragmatics of Human Communication – A Study of Interactional Patterns, Pathologies and Paradoxes*, pages 48–71. W. W. Norton, 1967.

[WCB+13] Dan Wing, Stuart Cheshire, Mohamed Boucadair, Reinaldo Penno, and Paul Selkirk. Port Control Protocol (PCP). Request for Comments 6887, Internet Engineering Task Force, April 2013.

[WCN+14] Tao Wang, Xiang Cai, Rishab Nithyanand, Rob Johnson, and Ian Goldberg. Effective Attacks and Provable Defenses for Website Fingerprinting. In *Proceedings of the 23rd USENIX Security Symposium*, USENIX Security '14, pages 143–157. USENIX Association, August 2014.

[WEKJ11] Ye-Kui Wang, Roni Even, Tom Kristensen, and Randell Jesup. RTP Payload Format for H.264 Video. Request for Comments 6184, Internet Engineering Task Force, May 2011.

[Wel84] Terry A. Welch. A Technique for High-Performance Data Compression. *IEEE Computer*, 17(6):8–19, June 1984.

[WKHW02] Andreas Willig, Martin Kubisch, Christian Hoene, and Adam Wolisz. Measurements of a Wireless Link in an Industrial Environment Using an IEEE 802.11-Compliant Physical Layer. *IEEE Transactions on Industrial Electronics*, 49(6):1265–1282, December 2002.

[WKSK07] Grace R. Woo, Pouya Kheradpour, Dawei Shen, and Dina Katabi. Beyond the Bits: Cooperative Packet Recovery Using Physical Layer Information. In *Proceedings of the 13th Annual International Conference on Mobile Computing and Networking*, MobiCom '07, pages 147–158, New York, NY, USA, September 2007. ACM.

[WPY07] Cheng-Xiang Wang, Matthias Pätzold, and Qi Yao. Stochastic Modeling and Simulation of Frequency-Correlated Wideband Fading Channels. *IEEE Transactions on Vehicular Technology*, 56(3):1050–1063, May 2007.

[WSBL03] Thomas Wiegand, Gary J. Sullivan, Gisle Bjøntegaard, and Ajay Luthra. Overview of the H.264/AVC Video Coding Standard. *IEEE Transactions on Circuits and Systems for Video Technology*, 13(7):560–576, July 2003.

[WYLB06] Starsky H. Y. Wong, Hao Yang, Songwu Lu, and Vaduvur Bharghavan. Robust Rate Adaptation for 802.11 Wireless Networks. In *Proceedings of the 12th Annual International Conference on Mobile Computing and Networking*, MobiCom '06, pages 146–157, New York, NY, USA, September 2006. ACM.

[YV85] Joseph H. Yuen and Q. D. Vo. In Search of a 2-dB Coding Gain. TDA Progress Report 42-83, NASA Jet Propulsion Laboratory, Communications Systems Research Section, July–September 1985.

[ZL77] Jacob Ziv and Abraham Lempel. A Universal Algorithm for Sequential Data Compression. *IEEE Transactions on Information Theory*, 23(3):337–343, May 1977.

[ZvM04] Xiaoming Zhou and Piet van Mieghem. Reordering of IP Packets in Internet. In Chadi Barakat and Ian Pratt, editors, *Passive and Active Network Measurement*, volume 3015 of *Lecture Notes in Computer Science*, pages 237–246. Springer Berlin Heidelberg, 2004.