Ad Hoc Networks 13 (2014) 444–461

Contents lists available at ScienceDirect

Ad Hoc Networks

journal homepage: www.elsevier.com/locate/adhoc

Harnessing cross-layer-design

Ismet Aktas [*], Muhammad Hamad Alizai, Florian Schmidt, Hanno Wirtz, Klaus Wehrle

Communication and Distributed Systems (ComSys), RWTH Aachen University, Germany

ARTICLE INFO

Article history:
Received 11 April 2013
Received in revised form 7 August 2013
Accepted 20 September 2013
Available online 29 September 2013

Keywords:
Cross-layer design
Cross-layer conflict detection
Software architecture
Runtime adaptive systems
Cross-layer interactions

ABSTRACT

Applications and protocols for wireless and mobile systems have to deal with volatile environmental conditions such as interference, packet loss, and mobility. Utilizing cross-layer information from other protocols and system components such as sensors can improve their performance and responsiveness. However, application and protocol developers lack a convenient way of specifying, monitoring, and experimenting with optimizations to evaluate their cross-layer ideas.

We present CRAWLER, a novel experimentation architecture for system monitoring and cross-layer-coordination that facilitates evaluation of applications and wireless protocols. It alleviates the problem of complicated access to relevant system information by providing a unified interface to application, protocol and system information. The versatile design of this interface further enables a convenient and declarative way to specify and experiment with compositions of cross-layer optimizations and their adaptions at runtime. CRAWLER also provides the necessary support to detect cross-layer conflicts, and hence prevents performance degradation when multiple optimizations are enabled across the protocol stack. We demonstrate the usability of CRAWLER for system monitoring and cross-layer optimizations with three use cases from different areas of wireless networking.

1. Introduction

Developing real-world protocols and applications for wireless and mobile systems is difficult. The volatile nature of the wireless medium as well as network and channel dynamics induced by mobility complicate their comprehensive development. This is further aggravated by the isolated nature of today's applications, protocols, and the operating system. Although the isolation of applications from each other, protocols, and the operating system attains reasonable software engineering advantages, it disregards (i) access to relevant system information, such as protocol states, for monitoring and experimentation and (ii) coordination among different components to optimize the performance in the face of changing conditions or mobility effects.

In order to achieve in vivo access to such relevant system information, network analysis tools, such as wireshark [1], allow the inspection of traffic specific points in the protocol stack. However, such tools lack the ability to monitor protocol states, variables, and system components, e.g., battery, motion indicators, and CPU utilization. This is mainly because the protocol stack and system component drivers are deeply integrated into the operating system which strongly limits external access to their internal states. Therefore, application and system developers are unable to access vital system information for monitoring, experimentation and performance optimization.

However, breaking this isolated layer and application paradigm, recent research [2] has shown cross-layer information, i.e., information provided also over non-adjacent layers, to allow both diverse applications and protocols to be significantly more adaptive. For example, in mobile and wireless systems, even a single cross-layer optimization at the MAC layer can achieve TCP throughput

* Corresponding author. Tel.: +49 241 80 21418.
E-mail address: Aktas@comsys.rwth-aachen.de (I. Aktas).

1570-8705/$ - see front matter © 2013 Elsevier B.V. All rights reserved.
http://dx.doi.org/10.1016/j.adhoc.2013.09.003

speedups of up to 20 times and latency reduction of up to 10 times over unmodified systems [3]. However, despite its proven potential to enhance system performance and a fair share of research investment in recent years, the cross-layer paradigm has not been able to leverage its utility beyond few promising yet concentrated research efforts [4–8].

Although several static cross-layer architectures have been proposed, networking researchers and application developers lack a generic and flexible architecture that enables specification and experimentation with cross-layer optimizations. Specifically, existing *static* cross-layer architectures [5,9–11] facilitate manipulation of protocol-stack parameters and combine several dedicated cross-layer optimizations. However, in current architectures of this type, cross-layer optimizations are composed offline (i.e., at compile time) and are deeply embedded within the operating system (OS). This approach has three key limitations that motivate the ideas presented in this article.

First, the process of adding or removing an optimization is impractical: optimizations are hard-wired with the architecture, and because the architecture is deeply embedded into the OS, recompiling the kernel and rebooting the system are typical consequences when changing optimizations. Furthermore, the developer has to deal with too many system internals such as OS programming language, application programming interfaces (APIs) and primitives before actually experimenting with cross-layer optimizations.

Second, because of this static nature of the existing architectures, an optimization will change the system behavior even if it is not needed or intended to take effect. Precisely, an optimization that is specific to an application or environment is not required when that application is not running or the underlying conditions have changed. For example, energy saving optimizations may not be necessary if the device is plugged into a power supply. Therefore, this optimization and its interaction with the network stack is superfluous and may even adversely affect other active applications. We strongly believe that this is against the original spirits of the cross-layer paradigm [12] which emphasize the need for dynamic adaptation of the system behavior (i.e., protocols, system components, and applications) based on the current application requirements and the network conditions.

Third, compile-time installation of optimizations significantly complicates the detection of cross-layer conflicts, i.e., possible performance degradations [13] caused by multiple, contradicting optimizations. Detecting such conflicts thereby remains one of the major unresolved challenges in the cross-layer development domain [4,8,13].

In this article we present CRAWLER, a novel experimentation architecture for system monitoring and cross-layer-coordination that facilitates the evaluation of applications and wireless network protocols. CRAWLER thereby benefits developers of wireless and mobile applications, protocols, and systems and supports them in experimenting with and evaluating their cross-layering ideas. Specifically, CRAWLER provides the following key features that illustrate its departure from the existing work and mark the contributions of this article.

- CRAWLER simplifies the process of monitoring and experimentation by providing a unified interface for accessing application, protocol, and system information, independent from the OS internals.
- The generic, versatile design of this interface further facilitates specifying cross-layer optimizations by providing a declarative way of composing a set of optimizations and their adaption and adaptability at runtime.
- It offers (i) a very high degree of flexibility, to fluently experiment with changing compositions of cross-layer optimizations and (ii) extensibility, to include and remove heterogeneous protocol and system components in order to find the right set of optimizations for a certain use-case. Hence, CRAWLER is well suited as a rapid prototyping tool for application and system developers.
- It enables cross-layer conflict detection support to provide feedback to the developers regarding conflicting interdependencies when experimenting with multiple, concurrent cross-layer optimizations.

The remainder of this article is organized as follows. Section 2 presents a system overview, highlights our design goals, and comprehends the scope of our architecture. Based on our design goals, Section 3 describes our architecture from a conceptual point of view. The practical value of CRAWLER is demonstrated in Section 4 where three different use cases from divers networking fields are presented. In Section 5 we show how CRAWLER supports a developer to detect conflicting interdependencies between multiple cross-layer optimizations. The implementation details and the architectural overhead of CRAWLER are presented in Section 6. Finally, we discuss related work in Section 7 before concluding the article in Section 8.

2. Design overview

CRAWLER consists of two main components as shown in Fig. 1: the *logical component* (LC) allows cross-layer developers to express their monitoring and optimization needs in an abstract and declarative way. For this purpose, we have created a rule-based language customized to cross-layer design purposes. Using this language, developers can specify cross-layer signaling at a high level without needing to care about implementation details. Additionally, the LC offers a uniform interface that allows applications (i) to provide their own optimizations on demand and (ii) exchange information with the protocol stack, system components and other applications.

The cross-layer optimizations as specified in the LC are realized by the *cross-layer processing component* (CPC). Here, rules are mapped to compositions of self-written *functional units* (FUs). Finally, *stubs* provide read/write access to protocol information and sub-system states via a generic interface that abstracts from a specific implementation. Thus, additions and changes in optimization rules can be done at runtime using the LC. These changes are reported to the CPC, which adapts the FU compositions accordingly.

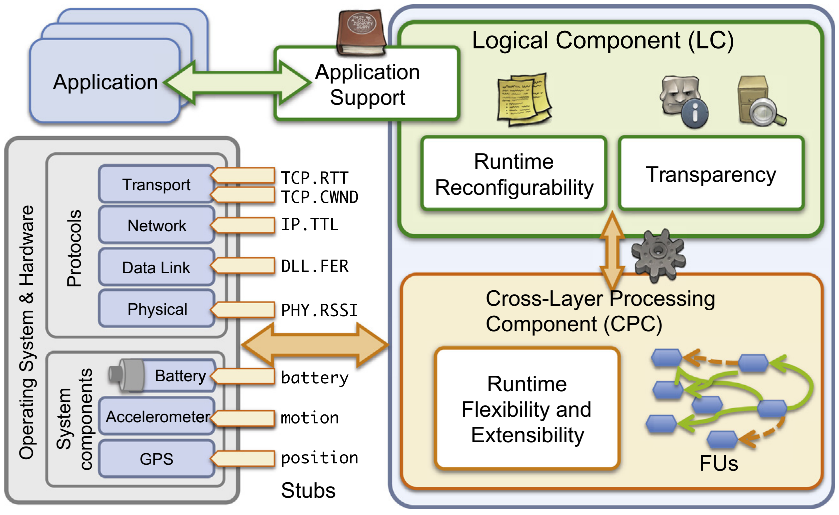

Fig. 1. Conceptual view of CRAWLER's components. The logical component (LC) abstracts from the implementation of cross-layer optimizations via an easily usable but powerful rule-based configuration language. The cross-layer processing component (CPC) realizes the optimizations given by the LC which can be readjusted flexibly at runtime.

Before going into further details of the architecture, we present our design goals and briefly highlight the scope of our approach.

2.1. Design goals

Our design is centered around the following goals:

Manageability: Cross-layer interactions should not impair the key software engineering properties, such as modularity, maintainability, and usability, of the layered protocol stack despite introducing dependencies across non-adjacent layers. Similarly, the cross-layer architecture should not impose additional requirements such as protocol dependencies when developing new protocols and system components: cross-layer optimizations should be easily maintainable and usable for application and system developers without having too much knowledge about system details and architectural requirements.

Application support: Unlike existing approaches, the architecture should provide a unified interface for application developers to (i) specify and add their own monitoring and optimization needs into the system and (ii) bundle these optimizations with their applications, without needing to deal with OS level details. Moreover, it should simplify the process of accessing protocol and system information, typically placed in the OS, which today is limited to only a few interfaces and thus requires manual inspection and adaptation of the very large OS code base.

*Runtime flexibility and extensibility:*The architecture should offer flexibility that is essential for adjusting and experimenting with different sets of optimizations, and further, the extensibility for involving all possible protocols and system components. In other words, for designing an optimization, the exchange of information between any number of layers and system components and the composition of any number of specific cross-layer optimizations should be possible at runtime. To achieve this, the design of an architecture has to offer sufficient versatility to cope with the diversity and permanent evolution of protocols and application requirements.

Context adaptability: The architecture should offer the ability to (i) detect the underlying environmental changes and (ii) respond to the changing application monitoring and optimization demands (e.g., when starting/terminating applications), by automatically loading the adequate set of optimizations at runtime. For example, energy saving optimizations may not be necessary if the device is plugged-into a power supply. This necessitates detecting the right condition (i.e., plugged to power) and loading the right set of optimizations (e.g., better performing but energy-consuming optimizations).

2.2. Cross-layering in CRAWLER: design scope

CRAWLER runs on end hosts and coordinates local information such as from the protocol stack and system components. CRAWLER itself does not establish information exchange among nodes in a network, such as in [14], because we believe that a monitoring and cross-layer experimentation architecture should not be responsible for establishing such information exchange mechanisms. Rather, this is the domain of a communication protocol. Nonetheless, a combination of such a protocol with CRAWLER could be used to share cross-layering information between nodes in a network. For example, in Section 4.3 we show a use case of CRAWLER where we shared a monitored parameter among neighboring nodes in an ad hoc network in order to improve the detection of a jammer.

3. Architectural details

We present a goal-driven description of CRAWLER by highlighting, with the help of simple examples, how our design achieves the four goals we laid out in Section 2.1.

3.1. Goal 1: achieving managability

The LC is the interface between developers and the CPC. Its major goal is to increase the usability and maintainability of cross-layer optimizations for developers, allowing them to easily express their desired optimizations without paying too much attention to implementation details. For this purpose, the LC is divided into four subcomponents as shown in Fig. 2. The *configuration* subcomponent allows a developer to express cross-layer optimizations on an abstract level. It thus hides their implementation details for a particular operating system. The *interpreter* subcomponent is responsible for parsing and mapping this abstract description to so-called *commands*. These commands instruct the CPC on how to realize the given cross-layer description. In addition, these commands are stored in a *repository* subcomponent that maintains a view of the current realized cross-layer optimizations in the CPC. The *application support* subcomponent allows applications to share their variables for cross-layer optimizations. Additionally, it allows applications to add their own monitoring and optimization needs. In the following we discuss the first three subcomponents which are intended to meet our design goal of *manageability*. We postpone discussion on the application support subcomponent to Section 3.2 to dedicatedly describe how this subcomponent realizes our second design goal of application support.

3.1.1. Configuration

The first step in CRAWLER's functionality is to allow the developers to specify their cross-layer optimizations. CRAWLER provides an easy to use but powerful rule-based language for specifying optimizations in an abstract and declarative configuration. Each *rule* is a behavioral description of a part of a cross-layer interaction such as accessing protocol information and aggregation. Rules can be nested within other rules to form rule chains, i.e., to develop cross-layer optimizations. In Listing 1, we present an example configuration with rules that specify how to access and process protocol-stack information and when to notify it to the application. Each line in the configuration is a rule. Fig. 3 shows a (slightly extended) graphical representation of this configuration. The figure is marked with numbers which correspond to the line numbers, i.e., rules, in the configuration.

The first rule `my_rssi` simply specifies which parameter, determined by a unique fully qualified name, should be accessed (see Section 3.3.2 for further details regarding the access mechanism). The second rule `my_history_of_rssi` collects the `History` of RSSI (received signal strength indication) values, i.e., the last `4 RSSI` values of the `wlan0` interface in this case. Similarly, the third rule `my_rssi_is_bad` determines if the average of these RSSI values is below a certain threshold.

So far, we have seen how computations and conditions can be specified using rules. However, sometimes it is desirable to react to events, such as a sudden drop in signal strength. This notification is denoted by an arrow such as in rules 4 and 5. The link quality condition of rule 3 is used to inform an application about the bad link quality (rule 6) and to reduce the sending congestion window of TCP connections to 0 (rule 7), i.e., to avoid triggering its congestion avoidance due to data corruption.

CRAWLER also allows the developers to modify or add new rules during runtime. It recognizes these changes in the configuration and adapts the internal composition of cross-layer optimizations accordingly. For example, if we want to change the signal strength threshold, we only need to modify rule 3. We defer further discussion on dynamic reconfiguration to Section 3.4.

Overall, the choice for a declarative and abstract language provides accessibility for developers who do not need to be cross-layer experts. Our language is customized to cross-layer needs in a way that all necessary functionalities for any kind of cross-layer interaction can be implemented. However, as already mentioned, the configuration is only an abstract description of cross-layer interactions that need to be realized. But beforehand, we will explain next how the necessary information to realize the desired optimization is extracted from the configuration.

3.1.2. Interpreter

In the next step, such high level configurations of cross-layer interactions need to be transformed into the actual, resulting optimization. To this end, the interpreter

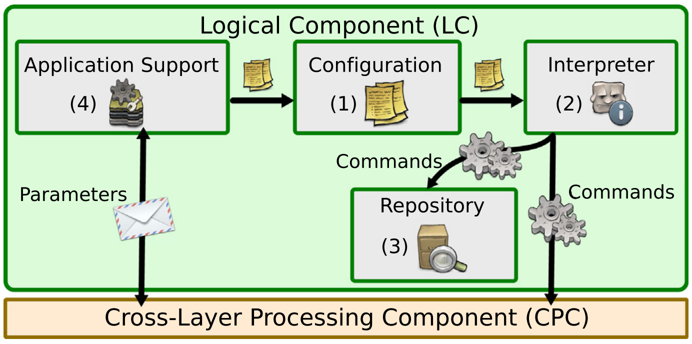

Fig. 2. The LC comprises four subcomponents. (1) The configuration is an abstract description of a cross-layer optimization. (2) The interpreter parses the configuration. (3) The repository saves snapshots of configuration setups, allowing easy access to the current and past setups. (4) The application support component provides an interface to applications for communication with CRAWLER in order to provide own optimizations and access to parameters.

```
my_rssi:get("phy.wlan0.rssi")
my_history_of_rssi:History(my_rssi, 4)
my_rssi_is_bad:Less(Avg(my_history_of_rssi), 55)
my_rssi_is_bad->my_appl_var1
my_rssi_is_bad->my_TCP_Freezer
my_appl_var1:set("application.app1.voip_var1", "bad")
my_TCP_Freezer:set("transport.tcp.cwnd", "0")
my_timer:Timer(200)
my_Timer->my_rssi_is_bad
```

Listing 1. A simple cross-layer signaling configuration in CRAWLER. This configuration file defines the setup illustrated in Fig. 3.

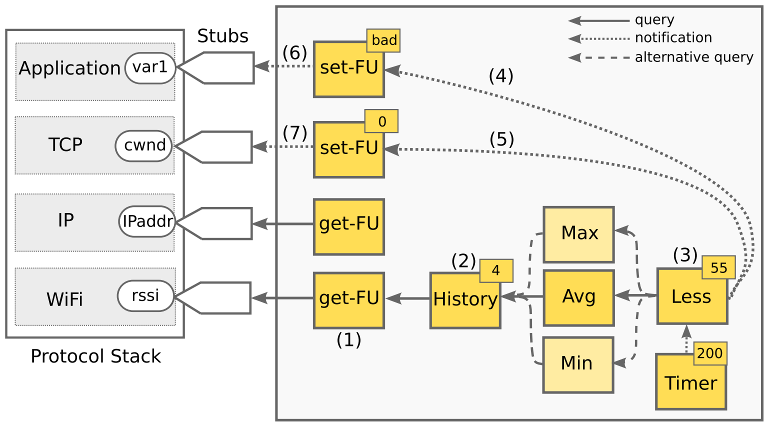

Fig. 3. A simple cross-layer configuration in CRAWLER. We change the behavior of the TCP layer and an application based on signal strength.

subcomponent of the LC parses the configuration and maps rules to fine-grained instructions called *commands*. These commands hold instructions for the CPC on how to wire and parameterize different *functional units* FUs to compose a certain optimization. FUs are special stateful functions that keep their private variables between calls, and that have a uniform interface to simplify the wiring of FUs. For example, rule 2 in Listing 1 is build from the commands `createFU (History)`, `addParameter (my_rssi)` and `addParameter (4)` by wiring the corresponding FUs as shown in Fig. 3. The handling of commands and the realization of cross-layer interactions are explained later in Section 3.3.

3.1.3. Repository

The repository keeps track of all the changes in a configuration. As the name suggests, it behaves similar to a revision control system: each time the configuration changes, the commands (as created by the interpreter) are automatically committed as a new revision. As a result, several revisions of a configuration can be stored in a preprocessed state. The benefit of this is twofold: first, this assists CRAWLER in switching between different optimizations without needing to parse the rules again. In a running system, this allows more efficient switching between preprocessed sets of optimizations, e.g., if a certain context is available. Second, while designing and testing new cross-layer optimizations, the repository allows the developers to roll back to a previous, well tested and running optimization for debugging purposes.

Summarizing, the declarative approach of specifying cross-layer interactions enhances the usability and maintainability of CRAWLER. Non of the existing architectures simplify the specification of cross-layer optimizations to a degree where even developers who are not experts can describe cross-layer optimizations. Thus, because CRAWLER allows to specify cross-layer optimization at a high level of abstraction, it does not impose any system specific requirements on protocol and system developers. Hence, the collaboration of these three subcomponents of the LC fulfills our design goal of manageability.

3.2. Goal 2: application support

In the previous section, we discussed how a cross-layer developer specifies rules to describe cross-layer optimizations. However, to provide rich application support, we also need an interface between applications and CRAWLER. Such an interface allows developers to enable applications and the OS to work together to make informed joint adaptation decisions. For example, in a handheld device, this could allow the OS to opt for a low-power mobile connection for background always-on services and switch over to a high-speed WiFi connection if the application requires a high-volume streaming connection. Similarly, an application could request a certain minimum and maximum required bandwidth and the OS could inform it about the bandwidth to be expected. The application can then choose a suitable transmission quality.

CRAWLER provides a rich interface for developers: It enables the applications to specify their needs (i) by accessing system information and sharing their own information and (ii) by providing own optimizations without needing to deal with implementation details of the OS or CRAWLER.

Listing 1 presents an example of information exchange between an application and CRAWLER in rules 4 and 6. A VoIP application creates a (user space) variable and provides an accessor `my_appl_var1` to it. This variable is set to a certain value when the RSSI falls below a certain threshold (rule 4). The application can then react to this change accordingly. To make use of such a configuration, CRAWLER applications can register variables that facilitate signaling of states via a system-wide shared library. This only requires an application to include the library's header file `crawler.h`, provide callback functions to read or write to the application variables, and link against the library. The interaction between CRAWLER and applications is performed by the shared library itself.

3.3. Goal 3: flexibility and extensibility at runtime

The flexibility of CRAWLER is associated with how a cross-layer optimization is composed and modified. CRAWLER provides a flexible wiring mechanism between FUs, the basic building blocks of an optimization, to enable the developers in experimenting with different compositions of an optimization. Similarly, extensibility deals with the underlying mechanism employed to access protocol-stack and system-component information. CRAWLER provides *stubs* as an extensible interface between cross-layer optimizations and the OS.

3.3.1. FU wiring

FUs possess two properties which form the basis for dynamic reconfigurability and adaptability of cross-layer optimizations.

First, FUs are stateful functions that maintain record of the data and provide results based on that record each time they are called. In contrast to stateless functions, whose output only depend on the input and the global state of the system, each FU keeps its private state (variables) as long as CRAWLER runs, much like an object in an object-oriented language. The output of an FU therefore depends on input, global system state, and private state of the FU. For example, every instance of `History` keeps its collected values between calls. As long as a configuration does not delete FUs but only changes their wiring, they will keep their current state and collected information.

Second, FUs share a unified interface so that they can be flexibly wired with each other. For example, by changing rule 3 in Listing 1, we can exchange the `Avg` FU in Fig. 3 with `Min` or `Max` at runtime due to the uniform interface, and still use the collected data from `History`. This is because a change in the wiring does not re-instantiate all FUs. This uniform interface also facilitates easy extension of FUs as newly designed FUs can easily be wired with the existing ones. CRAWLER supports two mechanisms to wire FUs, queries and events. Both types together cover the full range of information retrieval and aggregation to design any kind of cross-layer signaling.

Query-based Signaling: The query interface allows to explicitly request information. If the query interface of an FU is called, it returns the result to the inquiring FU. The query result of an FU may depend on the result of further FUs, leading to cascading queries. However, to reduce the computational overhead, each FU can cache its previously returned value and set a validity time for it. For example, on a query the `History`-FU returns immediately its collected and stored values instead of recollecting them. In case of a new incoming query, the FU can then decide to return the cached value or recompute a new one.

Event-based Signaling: The query-based interface for compositions between FUs results in a polling architecture. To avoid unnecessary polling, CRAWLER also supports an event-driven signaling that notifies interested FUs about the occurrence of an event, for example a significant change in a certain value measured by another FU.

Finally, to enhance the extensibility of the architecture, CRAWLER also maintains a *toolbox* that stores FUs. It helps in reusing generic FUs, such as `Timer` and `History`, or compose more complex FUs, such as a handoff estimation, by combining several small FUs.

Flexibility of CRAWLER is achieved with both signaling schemes to compose FUs, the building blocks of optimizations, to allow developers to experiment and modify optimization at runtime. In contrast, extensibility requires mechanisms to adapt to evolving nature of the OS, i.e., new or enhanced protocols and system-components. Therefore, this mechanisms should avoid dependencies between the cross-layer architecture and the OS. CRAWLER provides *stubs* as an extensible interface between cross-layer optimizations and the OS.

3.3.2. Stubs – accessing signaling information

Stubs provide read and write access to protocol and system information. They act as a glue element between the cross-layer optimizations and the OS. Stubs offer a common interface and a very find-grained access to system information: protocol and system variables have their own `get` and `set` stubs. Thus, to access the desired protocol or system variable, stubs need fully qualified, i.e., unique and hierarchical, names (cf. rule 1 in Listing 1). In cases where writing values is not possible, e.g., sensors that provide read-only variables, stubs with only `get` functionality can be used.

In CRAWLER's runtime the CPC automatically associates set and get FUs with each stub included in the architecture, as shown in Fig. 3. Protocol information often changes non-periodically and unpredictably as network conditions change. Because a stub is accessed by CRAWLER via FUs, these FUs can use the event-based signaling to notify other interested FUs about any change in protocol information. This increases the responsiveness of rules to changing conditions.

Fig. 4 shows an example of a stub that changes TCP's congestion control algorithm. The basic four steps to change TCP's congestion control algorithm are as follows:

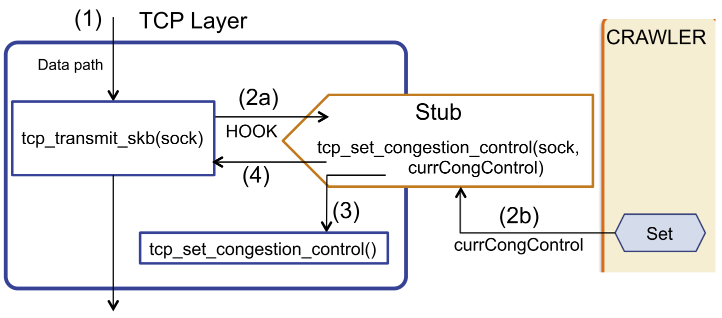

Fig. 4. Stub for changing TCP's congestion control algorithm works as follows: (1) While processing a packet in TCP, the function `tcp_transmit_skb` is called. (2a) Here we hook into the processing and redirect the processing to the stub. (2b) In parallel (asynchronous), `CRAWLER` sets the adequate congestion control algorithm via a `Set`-FU. (3) Based on the given value, the congestion control algorithm is selected. (4) Finally, processing continuous.

(1) after receiving and processing a packet `tcp_transmit_skb` is called right before delivering the packet to IP. (2a) Here, we inject a hook that redirects the Fig. 4 shows an example of a stub that changes TCP's congestion control algorithm. The basic four steps to change TCP's congestion control algorithm are as follows: (1) after receiving and processing a packet `tcp_transmit_skb` is called right before delivering the packet to IP. (2a) Here, we inject a hook that redirects the processing to the stub. (2b) The stub receives the current congestion control algorithm `currCongControl` from CRAWLER via a `Set`-FU in parallel (asynchronous). (3) If a change in the congestion control algorithm is requested, TCP's `tcp_set_congestion_control` algorithm is called for a certain socket. (4) Afterwards, the packet processing continues as normal. This stub is later used in the evaluation (see Section 4.1) to demonstrate a use case of CRAWLER.

Overall, stubs allow CRAWLER to monitor and coordinate a diverse set of protocols, system components and applications. Moreover, with a unified wiring interface between FUs, their different types of interconnection, and the ability to reuse and wire further FUs, provide a very high degree of extensibility and flexibility at runtime.

3.4. Goal 4: context adaptability

Context adaptability is one of the key features of CRAWLER. Application support is not possible with a static set of rules that cannot adapt to application demands. Specifically, application specific rules might not be known at system start time; they have to be loaded when the application starts and removed when it terminates.

In order to dynamically add, modify, and remove rules at runtime, CRAWLER provides the following three keywords that can be used in the configuration:

load (rule_name): The rule `rule_name` is loaded at runtime, e.g., if a developer wants to monitor the effects of an enabled optimization. Dependencies are automatically satisfied if `rule_name` references another rule which is not loaded in the CPC. For example, for the configuration defined in Listing 1, `load (my_rssi_is_bad)` will automatically load `my_history_of_rssi`. The new rules are composed into FU compositions as discussed in Section 3.1.2.

unload (rule_name): The rule `rule_name` is unloaded at runtime, e.g., if a developer wants again compare the system without the certain optimization. The internal handling of unloading a rule is more complex than loading it since unloading can result in unreferenced FUs. To address this problem, CRAWLER associates a reference counter with each FU. As an example, `unload (my_rssi_is_bad)` will also unload the rule `my_history_of_rssi` unless it is used by another rule that is not listed in Listing1.

replace (rule_old, rule_new): The rule `rule_old` is replaced with `rule_new` at runtime. Note, to achieve this some connections of the exchanged FU have to be rewired.

These keywords trigger the functionality to add, modify, and remove rules at runtime. CRAWLER also provides mechanisms to automatically execute the commands associated with these keywords based on context changes such as environmental conditions. For example, Listing 2 shows how application specified rules are automatically loaded or unloaded at runtime based on different conditions. The `[manual]` section contains rules that are parsed by the Interpreter but are not directly applied in the CPC. `[contextEnter]` specifies which rules from the `[manual]` section should be loaded when a certain condition (also specified in the form of a rule) is met. Therefore, lines 12 and 13 specify that the rule `setCwndAlg` will be loaded when the application sets its variable `loadOpt` to true. Note that this configuration will be later used in the evaluation section to demonstrate the change of the congestion control algorithm of TCP. `contextExit` is the opposite of `contextEnter` to unload rules when a certain condition is met. For example, in lines 16 and 17 based on the application's variable `removeOpt` all rules are unloaded.

Summarizing, by supplying keywords to load, unload, and replace rules, CRAWLER achieves reconfigurability at runtime. It also provides necessary support to automatically execute these rules depending upon the conditions defined by the developers.

4. Versatility of CRAWLER's application area

This section focuses on how CRAWLER can be utilized for monitoring and cross-layer adaptation purposes in diverse networking areas. For this we show three completely different use cases from different research fields. First we

```
[manual]
rssiavg:avg(history(get("wlan0.qual.rssi"),10))
less1:less(rssiavg,60)
packetLossRate:get("app.switchCwnd.packetLossRate")
less2:less(4,packetLossRate)
changeCwnd:and(less1,less2)
cwndAlg:if(changeCwnd,"westwood","vegas")
initPort:set("tcp.activate.outgoingPacketsPort",5001)
setCwndAlg:set("tcp.cong_control_5001", cwndAlg)"

[contextEnter]
loadOpt:get("app.switchCwnd.loadOpt")
loadOpt->load(setCwndAlg)

[contextExit]
removeOpt:get("app.switchCwnd.removeOpt")
removeOpt->unload(ALL)
```

Listing 2. Configuration of an application-specified optimization: TCP's congestion control algorithm is changed based on packet loss rate (PLR) and RSSI values. If the PLR is high and the RSSI is low, TCP's congestion control algorithm is set from TCP CUBIC to TCP Westwood. If either of the conditions is not satisfied, the congestion control algorithm is set back to TCP CUBIC.

start with the classical and well-known cross-layer example, that is, TCP congestion control. Afterwards, we show how we used CRAWLER to suggest a VoIP codec switching scheme to improve perceived user quality. Finally, we demonstrate CRAWLER's monitoring and cross-layer adaptation features for jamming detection and reaction.

4.1. Use case: monitoring and adaptation of TCP congestion control algorithms

To give an insight into how modeling a cross-layer adaptation with crawler is set-up, we now present an exemplary optimization that controls TCP's congestion control mechanism. The goal of this optimization is to dynamically switch between different congestion control algorithms, such as CUBIC [15] and Westwood [16], depending upon the underlying network conditions. CUBIC is the standard congestion control algorithm in the Linux kernel since 2.6.19 due to its superior performance and fairness properties under different network conditions. Westwood is specifically developed for wireless communications (such as in WLAN), and provides better throughput in challenging network conditions with high loss rates.

Our **test setup** consists of two PCs. One PC runs our CRAWLER implementation and is equipped with an 802.11 g WLAN card. We use Iperf [17] to create TCP traffic, and netem [18] to create different packet loss rates (PLRs) and to produce repeatable results in order to stress test our architecture. The other PC connects to an 802.11 g WLAN access point and serves as the destination for Iperf traffic.

As a first step, we model different loss conditions and measure the TCP goodput via Iperf. The results of this monitoring step can be seen in the first two curves in Fig. 5. It can be seen that Westwood outperforms CUBIC in high packet loss scenarios.

As a second step, we therefore specified a CRAWLER optimization that switched between different congestion control algorithms at runtime without re-initializing the TCP connection: If the packet loss rate exceeds 4% (a significant

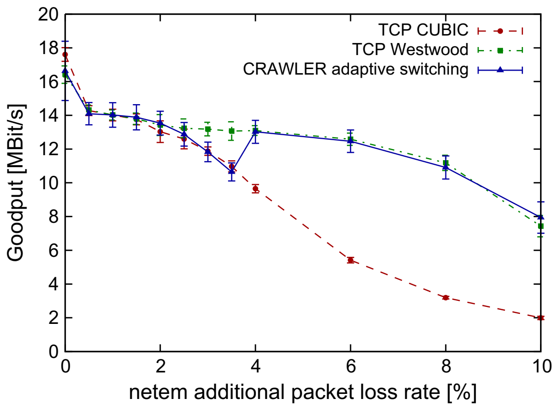

Fig. 5. Adapting TCP's congestion control: The switch from CUBIC to Westwood is performed at a packet loss rate of 4%. The error bars represent the 95% confidence interval of 10 repeated experimental runs.

amount for TCP) and the RSSI value falls below 60, TCP switches from CUBIC to Westwood congestion control. A switch back to CUBIC is initiated when the network conditions become stable again. The complete configuration script (in the form of an application-provided optimization) is presented in Listing 2.

The effect of this optimization is shown in Fig. 5. The variation in the results (specified by the 95% confidence intervals) can be attributed to different environmental conditions observed during the course of 10 repeated experimental runs in an indoor environment with several co-exiting WLANs deployments in the same frequency range. Note that a switch at a lower PLR of 3% could also improve performance of the optimization. However, as our main goal is show an exemplary optimization, the switch at 4% highlights its effects.

Fig. 6 shows our results for a longer experimental run, and also highlights the possibility to load rules at runtime. For the first 60 s, we did not load the optimization into the CPC, as depicted by the low TCP goodput achieved during

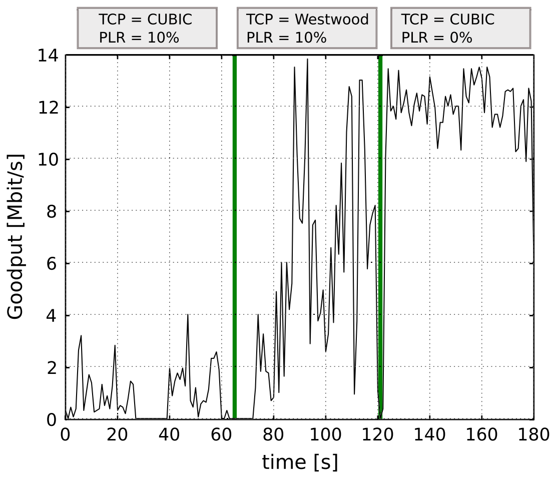

Fig. 6. Goodput of a TCP transmission over time under varying environmental conditions and congestion control algorithms. The optimization is loaded after 60 s which triggers the switch from CUBIC to Westwood. The switch back to CUBIC is triggered when the packet loss rate (PLR) falls below the application-specified threshold of 4%.

this time. The optimization is loaded at 60 s which triggers the switch from CUBIC to Westwood and subsequently improves the goodput. Similarly, at 120 s, when the PLR falls below its 4% threshold, TCP switches back to CUBIC and thus achieves a consistently higher goodput.

As a final step, we investigate if our on-the-fly algorithm change produces undesirable side effects. For example, the behavior of TCP's cwnd (congestion window) across different congestion control algorithms could lead to unexpected behavior. To monitor the behavior during the algorithm switch, we monitored the cwnd variable via CRAWLER's monitoring application by simply executing `monitorapp 'transport.tcp.cwnd'` in the console. In contrast, a manual setup would require changes to the kernel to introduce hooks and to create an interface to access the collected data. CRAWLER relieves the developer from these steps and expedites the testing and monitoring of variables and setups.

This example demonstrates the correctness of CRAWLER's implementation. It also shows that CRAWLER provides simple and rapid access to system variables and parameters. A 15-line configuration can be used to adapt TCP's congestion control without needing to re-initialize the end-to-end connection. Similarly, the congestion window can easily be monitored by executing the relevant monitoring application of CRAWLER.

4.2. Use case: VoIP codec switching

Today's VoIP applications are bound to use only one negotiated audio codec during the whole duration of a call. Although some codecs are adaptable to a certain degree, wireless networks demand even more adaptability to cope with the underlying network conditions. In this section we show that how CRAWLER improves the user-perceived quality of VoIP by automatically switching the codec during a phone call. This automatic switching is based on the observed network parameters – such as packet loss, jitter and bandwidth – that strongly impact the user-perceived quality of a VoIP call.

In order to objectively evaluate perceived speech quality, we rely on the PESQ tool [19,20] that rates perceived quality with a MOS-LQO (mean opinion score – listening quality objective) score ranging from 1 (bad quality) to 5 (excellent quality). Our *test setup* consist of two notebooks running Linux Ubuntu 10.04 and a router in between. The two notebooks are connected to the router via a 100 Mb/s Ethernet connection. A wired connection has been chosen to avoid uncontrollable wireless interference that can impact PESQ-MOS results. The whole test has been automated to achieve repeatability and to conduct manifold tests for credible results. We use the open source VoIP client Linphone [21] as it provides a wide range of VoIP codecs. We used netem [5] to insert jitter or packet loss into the connection, and employed traffic shaping via the token bucket filter [9] to reduce the available bandwidth. For each combination of codec and a certain packet loss/jitter/bandwidth, we repeated the experiment 100 times. One notebook, the sender, initiates the call and transmits the ITU-T test file via Linphone. The other notebook, the receiver, answered the call and recorded the audio output.

Figs. 7(a and b) and 8 show our results for the codecs GSM, Speex, PCMU (PCM with μ-law encoding), and PCMA (PCM with A-law encoding) under different network conditions manipulated with the help of netem [22]. The performance of different codecs under varying packet loss and jitter show a similar trend. In contrast, Fig. 8 shows that the variation in bandwidth results in a diverse behavior for different codecs. For example, from 20 kb/s to 30 kb/s GSM performs slightly better than the Speex. Similarly, from 40 kb/s to 90 kb/s Speex outperforms all other codecs. Finally, from 90 kb/s onwards, PCMU and PCMA achieve the best results. We can therefore conclude that bandwidth is the most suitable network parameter to characterize the performance of the available set of codecs. We can also derive a simple codec switching scheme: *A VoIP call should use Speex for a bidirectional bandwidth between 0 kb/s and 80 kb/s and PCM*[1] *otherwise.*

To implement such a switching scheme using CRAWLER, we used Wbest [23] to measure bandwidth during a VoIP session since Linphone does not offer such measurements. CRAWLER gets the IP addresses of a call from Linphone and provides it to Wbest's bandwidth measurements. Similarly, it gets bandwidth measurements from Wbest, calculating using a sliding window based average, and provides it to Linphone to initiate an appropriate codec switching. The overall cross layer optimization for codec switching works as follows: At the beginning of the call, CRAWLER measures the bandwidth and initiates a call by sending a SIP-invite message with the appropriate codec. During a call CRAWLER monitors network conditions. If a high bandwidth-consuming high quality codec is used, CRAWLER checks if the

[1] *PCMA or the PCMU codec curves are very similar in all cases, therefore from now on we use only the term PCM to indicate both. Similarly, we are ignoring the marginal performance difference between other codes in for different packet loss, jitter and bandwidth to maintain (i) the simplicity of our scheme and (ii) focus on the viability of* CRAWLER *in codec switching.*

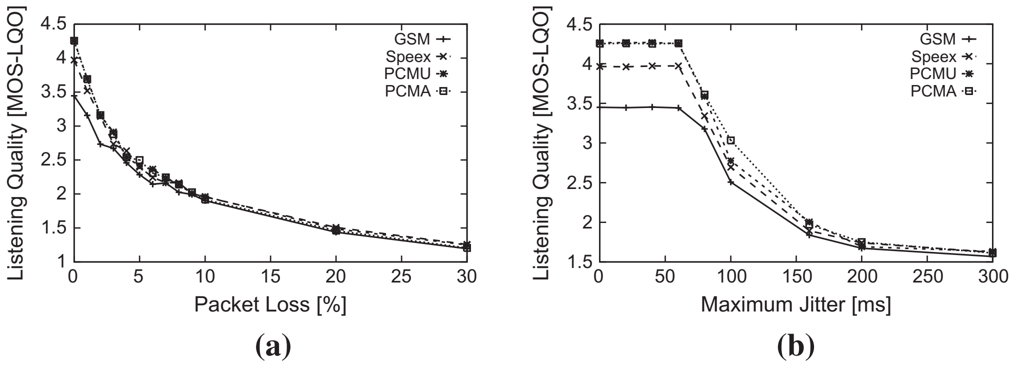

Fig. 7. All codecs are similarly impacted, no benefit achievable by switching codecs. (a) Influence of packet loss on perceived speech quality for several voice codecs. (b) Influence of delay jitter on perceived speech quality for several voice codecs.

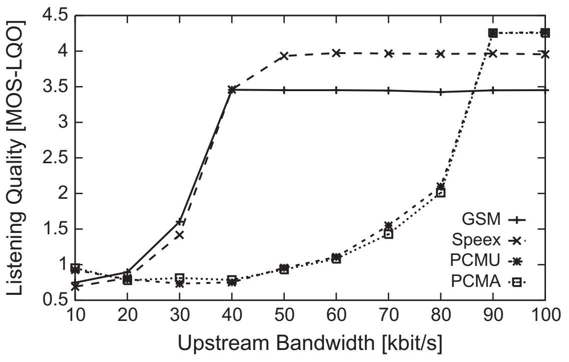

Fig. 8. Influence of available bandwidth on perceived speech quality for several voice codecs. Codec performance clearly depends on bandwidth, so codec switching depending on available bandwidth is sensible.

current packet loss raises above a certain threshold to decide whether to use a low bandwidth-consuming low quality codec, and vice versa. We compare our cross layer optimization of codec switching scheme with a static use of either Speex or PCM.

Fig. 9(a) shows the results for decreasing bandwidth from 200 kb/s to 65 kb/s after 24 s. We use the same ITU-T test file for all experiments. In Fig. 9(a) the speech quality of the Speex codec stay constant throughout the test, because the bandwidth limitation to 65 kb/s is still above Speex's requirements. On the other hand, PCM shows a strong degradation of quality after the bandwidth reduction. Pleas note that CRAWLER correctly chooses PCM as the initial codec. The overall effect of our codec switching scheme can clearly be seen. The temporary degradation between 24 and 32 s can be attributed to two factors. (1) The bandwidth decrease is detected due to frame losses, which reduces the listening quality in the time before the switch takes place. (2) Linphone's current implementation reacts to a re-INVITE codec switch with a small playback gap of about 200 ms, which also decreases the perceived quality. Similarly, Fig. 9(b) shows the results for increase in bandwidth from 65 kb/s to 200 kb/s after 24 s. The overall effect of our codec switching scheme can clearly be seen.

Concluding, our experiments show that the codec switching optimization selects the specified codec properly at the beginning and during the VoIP call. It improves listening quality compared to a static codec choice, except for the short time required to perform the switching

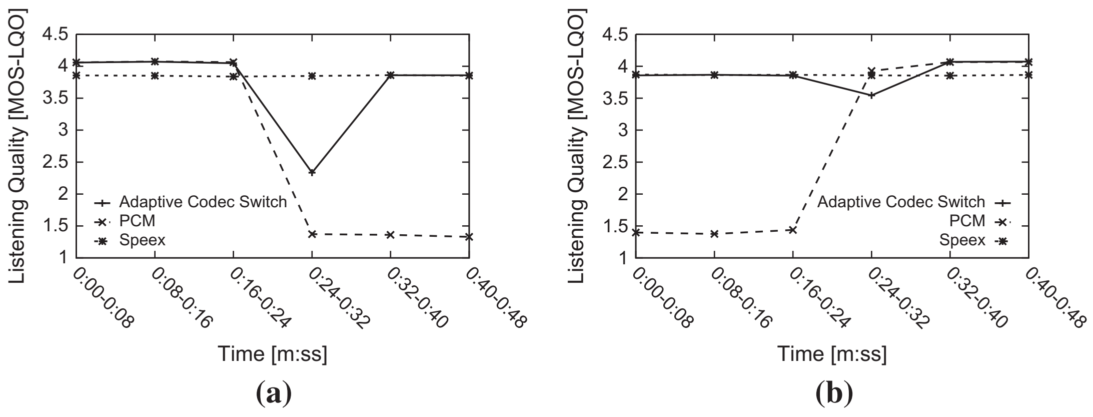

Fig. 9. MOS-LQO comparison for our codec switching scheme and pure PCM and Speex codec in case of changing bandwidth conditions. (a) MOS-LQO values for decreasing bandwidth. At 24 s, bandwidth was reduced from 200 kb/s to 65 k/s, and our scheme switched from PCM to Speex. (b) MOS-LQO values for increasing bandwidth. At 24 s, bandwidth was increased from 65 kb/s to 200 kb/s, and our scheme switched from Speex to PCM.

```
MaxPDR = max{PDR(N): N in Neighbors};
IF (MaxPDR<THRESHOLD1) THEN
  SS = SampleSignalStrength();
  IF (SS>THRESHOLD2)  THEN
    Jammer detected;
```

Listing 3. Jamming detection rule proposed by Xu et al. [24] using signal strength consistency checks.

operation. However, in a real setup with long conversations and only occasional codec switches, we expect the overall voice quality improvement to strongly outweigh these short degradations.

4.3. Use case: coping with jamming-attacks

The shared nature of the wireless medium allows a special kind of security attacks, the so-called jamming attacks which target at completely shutting down the communication. Different strategies of jamming attacks have been suggested [24–26]. These strategies can roughly be classified into basic and intelligent jammers. While intelligent jammers try to exploit weakness in the medium access protocol, basic jammers are protocol unaware. However, to cope with the problem of jamming effectively, an approach based on cross-layer design is required [26] since a jamming attack could target multiple layers simultaneously. Thus, several parameters from several layers have to be correlated.

The majority of exiting strategies [24,27,28] against jamming attacks are customized for a specific jammer in a certain scenario. Consequently, it is difficult to use those approaches in combination eventhough this could help in obtaining a unified system that can fight jamming in an adequate manner [26], i.e., where the strategy is selected based on the scenario and jammer. CRAWLER provides the necessary platform support to combine such strategies for three main reasons: (i) monitoring of parameters since it already provides access to a fair share of parameters, (ii) correlation of parameters as CRAWLER offers a simple and abstract configuration language, and (iii) the ability to add and remove jamming detection and reaction strategies at runtime to adapt to a certain scenario and jammer.

For example, we implemented the strategy suggested by Xu et al. [24] as it is the most widely used approach in the field of jamming detection for sensor networks. With CRAWLER we tried to rebuild this detection strategy for 802.11. The strategy of Xu et al. follows a cooperative approach using the packet delivery ratio (PDR) and the radio signal strength (RSS). The PDR is exchanged continuously with all its one-hop neighboring sensor nodes. This base amount of traffic also contributes to the RSS measurements. The detection strategy for this approach is shown in Listing 3. In a first step, from all neighboring nodes via PDR exchange messages the maximum PDR is taken. Subsequently, if the maximum PDR falls below a predefined threshold, a signal strength measurement is initiated. If this measurements also falls below another predefined threshold, the jammer is considered being detected.

The second RSS consistency check helps to differentiate if the PDR is low due to another influences such as mobility. However RSS measurements (or radio signal strength indicator (RSSI)[2]) in 802.11 are coupled with packet reception. That is, RSSI is calculated over the preamble and if a packet is not detected at all due to jamming, there are no RSSI measurements. Therefore, we adapted Xu's approach to use noise instead of RSS. This is because noise is strongly affected in the presence of a jammer. We have also implemented an ad hoc PDR exchange messaging scheme where all nodes exchange their currently measured PDR via broadcast with all their neighboring nodes.

We evaluated our detection strategy on a military testing ground in Greding, Bavaria, Germany. This testing environment was a spacious rural area permitting wireless communication without major disturbances. The testing scenario was as following. A military vehicle was instructed to escort a non governmental organization (NGO) vehicle in order to protect the NGO vehicle from enemies attached with jammers. Both vehicles were equipped with x86 computers running Vyatta-Linux, CRAWLER and an application for jamming detection. The scenario is shown in Fig. 10.

At the beginning both vehicles were outside of the jamming affected area. The NGO and military vehicles were 20 m apart from each other and moved along a road at a constant speed of about 20 km/h in direction to the jammer. The constant jammer used a directional antenna and an additional amplifier of 1 W enabling the jammer to disrupt a wireless communication entirely, up to a distance of at about 460 m. The jammer was hidden next to the road. Throughout the experiment, the vehicles exchanged ping messages and PDR exchange messages secured by ssh connections. Beside this, no further traffic was generated, neither by possible surrounding wireless nodes nor other radio access technologies. After both vehicles reached the jamming-affected area, our detection strategy was able to detect the presence of the jammer as shown in Fig. 11.

The curve shape of the PDR, Noise and RSS is very stable outside of the range of the jammer. Communication was not effected, i.e., all packets sent between the NGO and military vehicle could be received. With decreasing distance to the constant jammer, both vehicles entered the jamming-affected area. This resulted in an increase of the noise level, −62 dbm at maximum, and in a reduced RSS of −82 dbm at minimum. Since still some packets were received at this stage, we could measure RSSI values. The collisions between packets, sent from the embedded computers inside the vehicle, and the jamming signal caused the maximum PDR to drop until a PDR of 0% has

[2] Note that RSS is not provided by consumer network cards and the RSSI measurements are calculated differently in each brand.

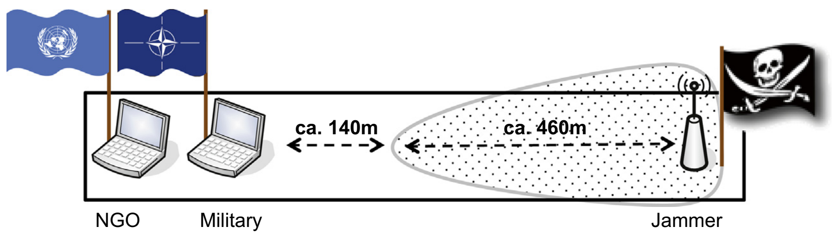

Fig. 10. Mobile test scenario conducted in Greding, Bavaria, Germany. The military vehicle and the NGO were driving along a road, starting around 600 m away from the constant jammer. After approximately 140 m, the vehicles reached area affected by the jammer and further approached the jammer until they reached the it after another approximately 460 m.

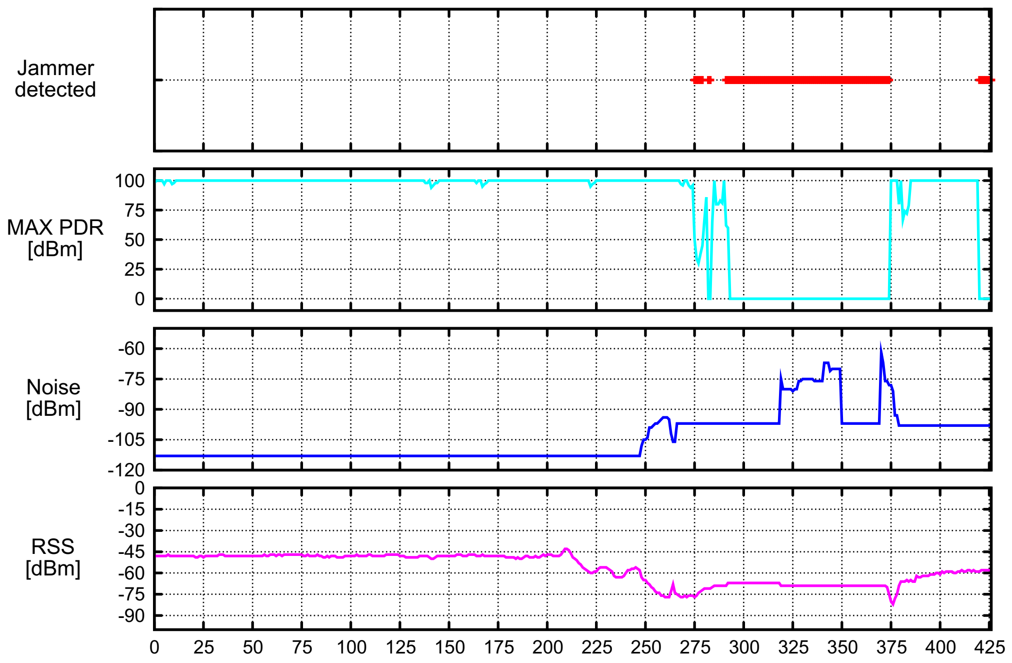

Fig. 11. Results of the first experiment of the use case scenario. Both vehicles were 20 m apart and drove with nearly constant speed of 20 km h. The constant jammer was emitting a jamming signal of −13 dbm amplified by 1 W.

been reached. As a result, the jammer was detected and further countermeasures could be initiated.

In our next experiment, the vehicles stayed in jamming range for a longer period of time. Here we gained some new interesting insights as shown in Fig. 12. After entering the jamming range, indicated by the red bar, the noise starts fluctuating although the vehicles do not move. This exhibits a periodic behavior which has also been observed in [29] using a constant jammer and an Atheros wireless network card. Puñal et al. assume that this is caused by ANI which probes different noise immunity configurations (in discrete steps) due to false signal detections. Despite ANI, the embedded computers were not able to decode received packets correctly or rather did not detect packets anymore indicated by the maximum PDR at 0% and the missing RSSI reports. After a while, the vehicles start moving again, increasing the distance to the jammer. As a result, all monitored values start recovering again.

So far, we are able to detect a jammer, but with CRAWLER it is also similarly easy to build reaction strategies. After detecting a jammer for this particular scenario (that we run in the scope of a project), we informed a headquarter via a satellite link about the presence of a jammer. Within this project it was necessary to create SOAP-messages that were labeled at the IP-layer. These specifically labeled packets are treated differently by the routing protocol. To achieve this, we filtered packets belonging to our jamming application at the IP layer using CRAWLER, i.e., by modifying the TOS field of IPv6 (IPv4 is also supported) packets and fed them back to the network stack. These packets arrived successfully at the headquarter.

To summarize, we were able to successfully detect a jammer and react to it using CRAWLER to monitor, correlate and manipulate protocols behavior at different layers.

5. Conflict detection support

The overall goal of a cross-layer optimization is to improve a performance metric such as energy, throughput, delay or user perceived quality of service. While we

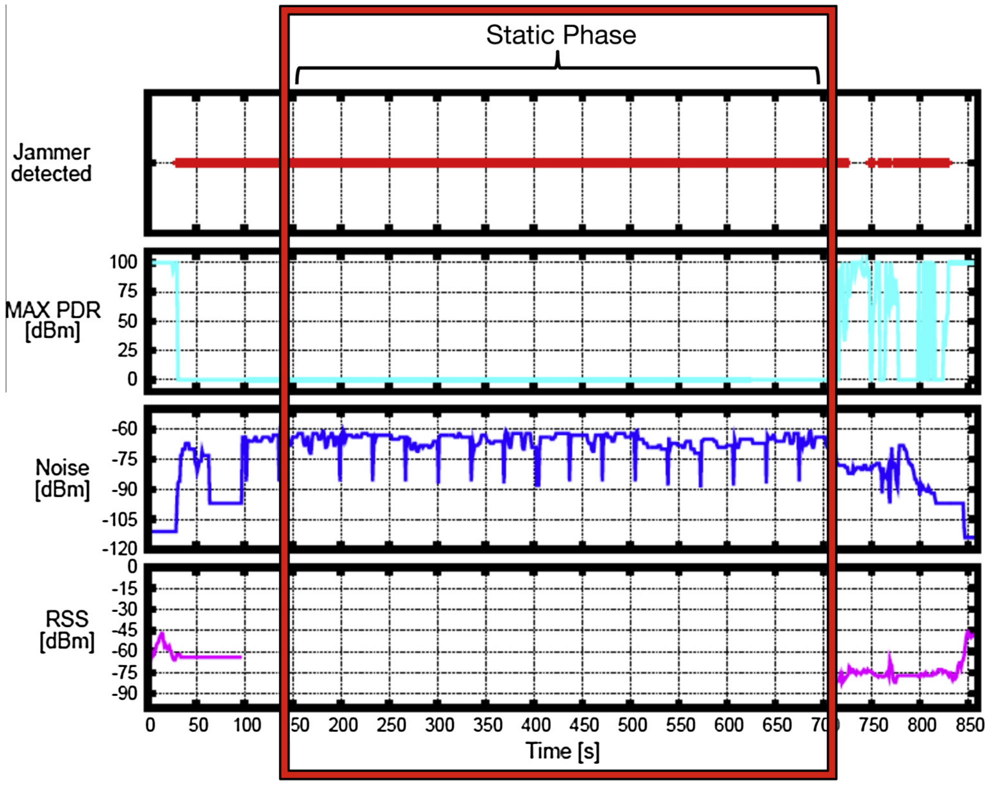

Fig. 12. Results of the fourth experiment of the use case scenario. Both vehicles were 30 m apart and drove with nearly constant speed of 30 km h. The constant jammer was emitting a jamming signal of −19 dbm amplified by 1 W.

observed that running a single optimization leads to respective performance improvements (cf. Section 4), multiple cross-layer optimizations in parallel could lead to unintended contradicting effects resulting in severe performance degradation. In the cross layering domain this problem is referred to as *cross-layer conflicts*. Although cross-layer conflicts are a well known problem [4,8,13], the existing cross-layer architectures fail to assist the developers in detecting such conflicts and in finding the right set of optimizations. In this section we present an architectural extension of CRAWLER that classifies and detects cross-layer conflicts. This architectural extension provides necessary feedback to the developers regarding conflicting cross-layer optimizations that may influence each other. Hence, it helps the developers in resolving such conflicts early in the experimentation phase. In the following we first classify different types of cross-layer conflicts before discussing how CRAWLER deals with them.

5.1. Classification of cross-layer conflicts

We classify cross-layer conflicts into (1) direct conflicts and (2) indirect conflicts based on how difficult it is to detect a specific conflict.

Direct conflicts occur when multiple cross-layer optimizations try to manipulate the same variable in a certain protocol as shown in Fig. 13(a). Here multiple optimizations try to manipulate a single parameter at a certain layer via a set-FU. Hence, it is possible that two conflicting optimizations have contradicting effects on the variable leading to the oscillation of a parameter, and accordingly, an overall performance degradation. For example, an energy

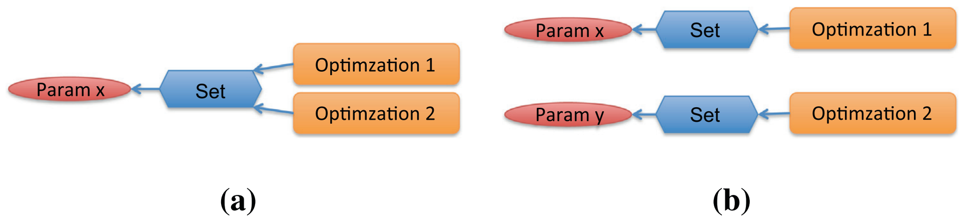

Fig. 13. Conflict classification: direct and indirect conflicts. (a) Direct conflicts: Multiple optimizations manipulating the same parameter in a certain protocol. (b) Indirect conflicts: Optimizations manipulate different parameters but nevertheless affect each other.

related optimization is interested in saving transmission energy by decreasing the transmission power. Conversely, a connectivity oriented optimization is interested in keeping long range connectivity by increasing the transmission power. Hence, both these optimizations, when run in parallel, could lead to the oscillation of transmission power and an overall misbehavior of the system.

Indirect conflicts are caused by multiple optimizations that influence each other even though they do not manipulate the same parameter. Hence, such conflicts are more difficult to detect when compared with direct conflicts. Fig. 13(b) shows variables of different protocols being manipulated by different cross-layer optimizations. For example, two different optimizations try to improve the ARQ error control at TCP and MAC layers, respectively. Two error control optimizations enabled at the same time may lead to throughput degradations if not coordinated properly. This is because each of the ARQ error control causes additional overhead which decreases throughput.

5.2. Detecting direct conflicts

Since in direct conflicts several optimizations compete for the same parameter, detecting such conflicts is rather straightforward. The underlying idea is to determine the number of parallel optimizations that are manipulating the same variable. For this purpose, CRAWLER automatically counts the number of FUs accessing a *set-FU* for a certain variable. The manipulation of a certain variable by FUs acquired by different optimizations is a strong indication of the potential root cause for a conflict.

Moreover, we have implemented three FUs to further analyze the impact of variable manipulations by different optimizations. Hence, besides describing rules for cross-layer optimizations, we also use CRAWLER's description language to detect and analyze conflicts. This process is similar to writing programming code and adding debug-information such as assertions. In the following, we describe these FUs that enhance CRAWLER's monitoring and introspection capabilities.

Frequency FU (FRQ): This FU counts the number of accesses to a variable over a certain period of time. The higher the frequency of access to a variable, the greater is the probability of a conflict. This is because frequent access to a certain variable is a strong indication that two optimizations are in conflict with each other with regard to the suitable value for that variable. Hence, this conflict automatically increases the number of accesses to that variable. Fig. 14 shows how two different optimizations try to access and manipulate a variable. The frequent changes may occur due to the reason that the optimizations work contradictively. The frequency FU can detect these

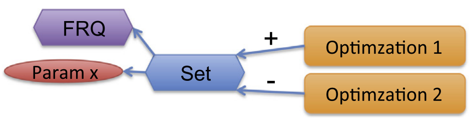

Fig. 14. Multiple optimizations try to change the variable to frequent. Our frequency checking FU FRQ is able to detect and report this to the application.

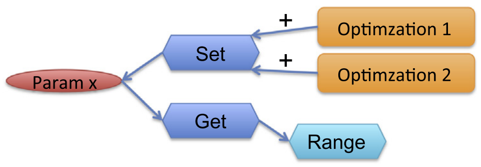

Fig. 15. Multiple optimizations increase the variable *x*. Our range FU checks if the value is unintentionally changed twice resulting in out of bound increase.

frequent changes and provide necessary feedback to the conflict monitoring application.

Range FU: This FU checks if the variable is assigned values from a certain predefined range. If, for example, two optimizations increase the value of a variable simultaneously, it may result in out of bound increase resulting in unintended misbehavior. Fig. 15 shows two optimizations try to increase a variable which is also monitored and checked by the Range FU if the changes exceed the predefined thresholds.

Oscillation FU: This FU observes if the values of a certain variable are fluctuating considerably. It has two functions: First, the FU observes whether two subsequent assignments of a variable deviate beyond a certain predefined margin, reporting a possible misbehavior. Second, it also provides the ability to measure the scale of these deviation. For example, if the sampling frequency of the FU is set to high, then single peaks within a short timeframe are detected as misbehavior. In contrast, if the sampling frequency is set to a low value, then short peaks are not considered but alterations over a longer period can be detected as misbehavior.

Overall, these three FUs assist a developer in detecting possible conflicts when running multiple optimizations.

5.3. Detecting indirect conflicts

Indirect conflicts can introduce complex interdependencies among variables of different protocol layers resulting in performance degradation in terms of metrics such as energy, throughput, jitter or delay. To detect indirect conflicts CRAWLER monitors these metrics based on the recently observed network conditions traffic conditions. For example, we use Wbest [23] with CRAWLER to probe for current network conditions such as bandwidth and delay.

Besides observing current network conditions, CRAWLER also needs a first hand knowledge about the application requirements to determine whether multiple concurrent optimizations are in conflict with each other. For example, the throughput decrease during a VoIP call initialization might be irrelevant for a VoIP application but delay is not. Therefore, the performance requirements of an application is an essential information to decide if the performance degradation has occurred due to an optimization conflict. Hence, to find out the basic conditions given by applications, we need to classify applications. For this purpose, we classify different types of networking applications to establish their basic set of requirements. Our classification is based on the comprehensive QoS based classification in [30].

In our current implementation, the QoS class of an application is statically determined when the application is registered with CRAWLER. It keeps track of all the registered applications and their classes, and accordingly instructs Wbest to probe for the corresponding network requirements. If many applications are loaded simultaneously, the corresponding rules for monitoring respective metrics are established automatically by CRAWLER. If performance deteriorates with respect to any of these metrics, CRAWLER notifies all the registered applications regarding the possible occurrence of a conflict among concurrent optimizations.

To conclude, we first classified the cross-layer conflict problem into two classes, that is, direct and indirect conflicts. Afterwards, we showed how we extended CRAWLER with debugging capabilities to counter these two classes of conflicts. To automatically detect direct conflicts, CRAWLER simply counts the number of accessors to a certain variable. For manual debugging, we added further FUs to provide sophisticated debugging support. For indirect conflicts, we extended CRAWLER with Wbest to monitor network traffic conditions. Based on the observed network conditions and application demands, CRAWLER tries to detect indirect conflicts due to multiple cross-layer optimizations.

6. Implementation and architectural overhead

In this section we discuss the implementation details of the architecture and evaluate the architectural overhead when running CRAWLER.

6.1. Implementation

We implemented CRAWLER[3] for Linux (kernel 2.6.32). The LC and all its subcomponents are implemented in C++. It runs as a daemon in user space. The CPC resides in kernel space and is implemented in C. This reduces the number of expensive context switches between kernel and user space during runtime. The communication between LC and CPC takes place via flexible interfaces provided by generic netlink sockets [31]. For using CRAWLER, applications can link against a shared library that contains all the functionality to interface with the LC.

The wiring between FUs is implemented using a special data type that can contain characters, integers, boolean values, arrays, and a struct-like compounds of these types. So far, we have implemented about 20 FUs and 160 stubs, with the numbers growing with every new testing setup.

6.2. Architecture overhead

We now measure the runtime overhead of our architecture. CRAWLER's runtime, the CPC, provides two main functionalities: (i) registering and wiring FUs and stubs and (ii) signaling between FUs and stubs to access protocol and component information. The registration of FUs and stubs is not time-critical since this only happens when a new optimization is loaded into the system. During the registrations, each newly created FU and stub is checked to prevent duplicates. For each of them, this has a runtime of $O(n + m)$ where n and m are the number of already existing FUs and stubs, respectively.

Query-based and event-based signaling (cf. Section 3.3.1) play a vital role in determining the processing overhead of CRAWLER. To measure this, we use a simple benchmark of several wired `Forwarder` FUs. These do not contain any complex logic: they simply relay the query to the next FU. The idea here is to keep the complexity of the FUs as low as possible to measure the signaling overhead between FUs.

Fig. 16(a) shows the results for both the signaling mechanisms of CRAWLER when compared with a standard Linux function call (note the logarithmic scale on both axes). We created chains of `Forwarder` FUs of different lengths, from one to one thousand chained FUs. Afterwards, we measured the CPU cycles required to traverse all `Forwarder` FUs, repeating each benchmark 100 times. The results show that query-based and event-based signaling mechanisms introduce an overhead of a factor 2.1 and 2.8 when compared with native Linux function call, respectively. However, we can clearly see that the overhead increases linearly with the length of the chains.

However, this processing overhead does not increase the processing time of network packets. To show this, we connect two notebooks via a Gigabit Ethernet. The sender notebook runs our CRAWLER implementation with an optimization that changes each outgoing packet by manipulating the TTL field of the IP header. The optimization consists of two rules: Rule 1 creates a chain of `Forwarder` FUs of different lengths. At the end of this FU chain, we added a simple FU that incremented an integer value. Rule 2 registers a netfilter hook in the IP output path that sets the TTL to that value. We then create different amounts of UDP traffic via Iperf [17]. Fig. 16(b) shows the length of rule chains does not contribute noticeably to the per-packet processing time. This highlights the fact the runtime overhead of CRAWLER is asynchronous to packet processing. Fig. 16(c) depicts the throughput measurements for the same experiments.

Overall, these results conclude that, while CRAWLER introduces processing overhead, this overhead does not deteriorate network performance in terms of throughput and packet processing time.

7. Related work

A plethora of *specific cross-layer solutions* [4,6,32] have been proposed that optimize a specific behavior of the system rather than creating full-fledged architectures. The majority of these solutions either enables cross-layer signaling between two specific layers or between many layers but in only one direction, e.g., from lower layers to upper layers but not vice versa. For instance, PMI [33] only propagates device information layer-by-layer to the upper ayers. Similarly, ICMP messages have also been utilized to provide feedback from lower layers to upper layers

[3] This article focuses on the main features of the CRAWLER architecture that support our design goals. The source code and documentation of the whole architecture can be accessed via `http://www.comsys.rwth-aachen.de/research/projects/crawler/`

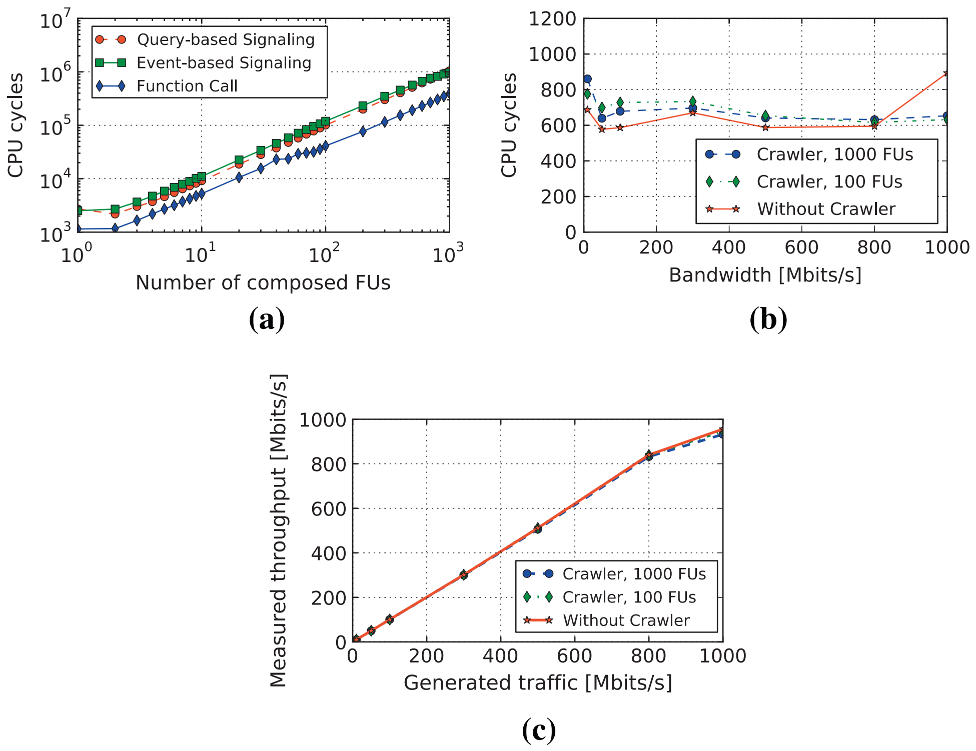

Fig. 16. Performance measurements of CRAWLER. (a) The signaling overhead has a linear increase of CPU cycles with increasing amount of wired FUs. (b) As CRAWLER's rules run asynchronously, packet processing time is independent of the amount of wired FUs. (c) Likewise, throughput is not influenced.

Table 1
Comparison of cross-layer architectures.

Architecture	CLASS [9]	CATS [10]	MobileMAN [5]	ECLAIR [11]	CRAWLER
Multiple optimizations	✔	✔	✔	✔	✔
Any-to-(m) any layers coordination	✗	✔	✔	✔	✔
Extensibility or flexibility	✗	✗	✔	✔	✔
Protocol stack abstraction	✗	✗	✗	✔	✔
Manageability	✗	✗	✗	✗	✔
Appl. support	✗	✗	✗	✗	✔
Runtime extensibility and flexibility	✗	✗	✗	✗	✔
Context adaptation	✗	✗	✗	✗	✔
Context detection	✗	✗	✗	✗	✔

[34]. The inter-layer signaling pipe (ISP) [35] utilizes packet headers to provide cross-layer feedback from upper layers to lower layers. In contrast, CRAWLER is an architecture that facilitates realization of all these specific solutions, potentially in parallel.

In recent years, a number of *cross-layer architectures* have been proposed that facilitate signaling across all layers in both directions, i.e., any-to-any layer signaling. For example, CLASS [9] enables direct signaling between all layers by message passing. However, any-to-(m) any layer signaling, i.e., addressing several layers at once, is not possible with CLASS. CATS [10] provides a management plane that supports such any-to-(m) any layer signaling. However, CATS has a monolithic architecture that does not specify any generic interface for signaling among different layers and hence is unable to cope with permanent evaluation of protocols and system components. MobileMAN [5] provides a database where each layer can store protocol information and make it accessible to other layers in a unified fashion. Thus, MobileMAN requires extensive modifications in the protocol-stack to enable such database interactions. This limits extensibility and maintainability of this architecture. ECLAIR [11] is the most advanced cross-layering architecture that provides a generic interface for accessing protocol stack. Its generic interface facilitates platform independence but it is does not support dynamic adaptability of cross layer optimizations at runtime.

However, the need for application support has been emphasized in recent years to allow applications to acquire system or protocol information. Unfortunately, this emphasis has been limited to mere architectural concepts [36,37], or to specific cross-layer optimizations [2,38]. Similarly, the problem of cross-layer conflicts has been introduced in few papers [4,8,13] but a feedback for developers has not been suggested. Finally, to the best of our knowledge, the need for manageability, context adaptation and runtime flexibility and extensibility has not been addressed by any of the research papers.

Our main departure from the existing work is that our architecture (i) allows the developers to specify cross layers optimizations at a very high level of abstraction, (ii) provides rich application support by enabling applications to interact with CRAWLER and specify their own optimizations, (iii) enable runtime adaptability of cross layer optimizations depending upon the underlying network conditions, and (iv) provide the necessary support for developers to detect cross-layer conflicts. To the best of our knowledge, these key features are not supported by the existing cross-layer architectures as depicted in Table 1.

8. Conclusions and future work

In this article, we have presented CRAWLER, a **cr**oss-layer **a**rchitecture for **w**ire**le**ss netwo**r**ks that enables flexible and versatile adaptation of protocols, system components, and applications. One key novelty is that CRAWLER can react to unpredictable changes in a device's environment by adapting all its optimization at runtime. The rule-based language enhances the usability and maintainability of CRAWLER by allowing to express cross-layer optimizations in an OS-independent fashion. Runtime reconfigurability is achieved via the flexible wiring between different functional units within an optimization. Our evaluation demonstrates the utility and correctness of CRAWLER's implementation with help of simple use cases. It also shows that CRAWLER does not deteriorate the network performance parameters such as throughput and packet processing time.

Developing novel cross-layer optimizations is our primary focus as a future work. We also want to improve the usability of CRAWLER even further by providing a visual configuration and monitoring component. The visualization support for monitoring cross-layer interactions will provide several advantages such as observing complex cross-layer interactions and the ensuing effects.

References

[1] Wireshark network protocol analyzer, <http://www.wireshark.org/>, 2011 (accessed 05.11.11).
[2] S. Khan, Y. Peng, E. Steinbach, M. Sgroi, W. Kellerer, Application-driven cross-layer optimization for video streaming over wireless networks, IEEE Communications Magazine 44 (1) (2006) 122–130.
[3] H. Balakrishnan, S. Seshan, R.H. Katz, Improving reliable transport and handoff performance in cellular wireless networks, Wireless Networks 1 (1995) 469–481.
[4] V. Srivastava, M. Motani, Cross-layer design: a survey and the road ahead, IEEE Communications Magazine 43 (12) (2005) 112–119.
[5] M. Conti, G. Maselli, G. Turi, S. Giordano, Cross-layering in mobile ad hoc network design, Computer 37 (2) (2004) 48–51.
[6] V. Raisinghani, S. Iyer, Cross-layer design optimizations in wireless protocol stacks, Computer Communications 27 (8) (2004) 720–724.
[7] S. Shakkottai, T. Rappaport, P. Karlsson, Cross-layer design for wireless networks, IEEE Communications Magazine 41 (10) (2003) 74–80.
[8] V. Kawadia, P. Kumar, B. Technol, M. Cambridge, A cautionary perspective on cross-layer design, IEEE Wireless Communications 12 (1) (2005) 3–11.
[9] Q. Wang, M. Abu-Rgheff, Cross-layer signalling for next-generation wireless systems, in: IEEE WCNC, vol. 2, 2003, pp. 1084–1089.
[10] C. Sadler, L. Kant, W. Chen, Cross-layer self-healing mechanisms in wireless networks, in: Proc. WWC, vol. 254, 2005.
[11] V. Raisinghani, S. Iyer, ECLAIR: an efficient cross layer architecture for wireless protocol stacks, in: Proc. World Wireless Congress, 2004.
[12] I. Aktas, J. Otten, F. Schmidt, K. Wehrle, Towards a flexible and versatile cross-layer-coordination architecture, in: Proc. INFOCOM'10 Work-in-Progress Session, 2010.
[13] A. Willig, Recent and emerging topics in wireless industrial communications: A selection, IEEE Transactions on Industrial Informatics 4 (2) (2008) 102–124.
[14] R. Winter, J. Schiller, N. Nikaein, C. Bonnet, Crosstalk: cross-layer decision support based on global knowledge, Communications Magazine, IEEE 44 (1) (2006) 93–99.
[15] S. Ha, I. Rhee, L. Xu, CUBIC: a new TCP-friendly high-speed TCP variant, SIGOPS Operating Systems Review 42 (5) (2008) 64–74. doi: http://dx.doi.org/10.1145/1400097.1400105.
[16] S. Mascolo, C. Casetti, M. Gerla, M.Y. Sanadidi, R. Wang, TCP westwood: bandwidth estimation for enhanced transport over wireless links, in: Proc. MobiCom'01, ACM, New York, NY, USA, 2001, pp. 287–297. doi: http://dx.doi.org/10.1145/381677.381704.
[17] A. Tirumala, F. Qin, J. Dugan, J. Ferguson, K. Gibbs, Iperf: The TCP/UDP bandwidth measurement tool, 2004.
[18] S. Hemminger, Netem-emulating real networks in the lab, in: Proc. Linux Conference Australia, 2005.
[19] Pesq, <http://www.pesq.org/>.
[20] ITU-T 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 (2001).
[21] Linphone, <http://www.linphone.org>, 2010 (accessed 11.04.10).
[22] netem, <http://www.linuxfoundation.org/collaborate/workgroups/networking/netem>.
[23] Wbest: a bandwidth estimation tool for ieee 802.11 wireless networks, <http://web.cs.wpi.edu/claypool/papers/wbest/>.
[24] W. Xu, W. Trappe, Y. Zhang, T. Wood, The feasibility of launching and detecting jamming attacks in wireless networks, in: Proceedings of the 6th ACM International Symposium on Mobile Ad hoc Networking and Computing, MobiHoc '05, ACM, New York, NY, USA, 2005, pp. 46–57. doi: http://dx.doi.org/10.1145/1062689.1062697, <http://doi.acm.org/10.1145/1062689.1062697>.
[25] G. Thamilarasu, S. Mishra, R. Sridhar, Improving reliability of jamming attack detection in ad hoc networks, International Journal of Communication Networks and Information Security (IJCNIS) 3 (1), 2011.
[26] K. Pelechrinis, M. Iliofotou, S. Krishnamurthy, Denial of service attacks in wireless networks: The case of jammers, IEEE Communications Surveys Tutorials 13 (2) (2011) 245–257. doi: http://dx.doi.org/10.1109/SURV.2011.041110.00022.
[27] M. Li, I. Koutsopoulos, R. Poovendran, Optimal jamming attacks and network defense policies in wireless sensor networks, in: INFOCOM 2007. 26th IEEE International Conference on Computer Communications, IEEE, 2007, pp. 1307–1315. doi: http://dx.doi.org/10.1109/INFCOM.2007.155.
[28] M. Strasser, B. Danev, S. Čapkun, Detection of reactive jamming in sensor networks, ACM Transactions on Sensor Networks 7 (2) (2010) 16:1–16:29. doi: http://dx.doi.org/10.1145/1824766.1824772, <http://doi.acm.org/10.1145/1824766.1824772>.
[29] O. Puñal, A. Aguiar, J. Gross, In VANETs we trust?: characterizing RF jamming in vehicular networks, in: Proceedings of the Ninth ACM International Workshop on Vehicular Inter-networking, Systems, and Applications, VANET '12, ACM, New York, NY, USA, 2012, pp. 83–92. doi: http://dx.doi.org/10.1145/2307888.2307903, <http://doi.acm.org/10.1145/2307888.2307903>.
[30] A. Arunachalam, J. Reed, Quality of service (QOS) classes for BWA, A contribution to the IEEE 802.
[31] P. Neira-Ayuso, R. Gasca, L. Lefevre, Communicating between the kernel and user-space in Linux using Netlink sockets, Software: Practice and Experience.
[32] G. Carneiro, J. Ruela, M. Ricardo, I. Porto, Cross-layer design in 4G wireless terminals, IEEE Wireless Communication 11 (2) (2004) 7–13.
[33] J. Inouye, J. Binkley, J. Walpole, Dynamic network reconfiguration support for mobile computers, in: Proc. MobiCom, ACM, New York, NY, USA, 1997, pp. 13–22.
[34] P. Sudame, B. Badrinath, On providing support for protocol adaptation in mobile wireless networks, MONET 6 (1) (2001) 43–55.
[35] G. Wu, Y. Bai, J. Lai, A. Ogielski, Interactions between TCP and RLP in Wireless Internet, in: Proc. GLOBECOM, 1999, pp. 661–666.

[36] C. Efstratiou, K. Cheverst, N. Davies, A. Friday, An architecture for the effective support of adaptive context-aware applications, in: Mobile Data Management, Springer, 2001, pp. 15–26.
[37] M. Sama, D. Rosenblum, Z. Wang, S. Elbaum, Multi-layer faults in the architectures of mobile, context-aware adaptive applications, in: Journal of Systems and Software 83 (6) (2010) 906–914.
[38] O. Nemethova, W. Karner, A. Al-Moghrabi, M. Rupp, Cross-layer error detection for H.264 video over UMTS, in: Proc. WPMC'05, 2005.

Ismet Aktas is a Research Assistant at the Chair of Communication and Distributed Systems, RWTH Aachen University. He received his Diploma Degree from RWTH Aachen University in 2007. His studies were funded by the excellence initiative "Ultra High Speed Information and Communication – UMIC". He started his PhD in the June of 2007. His research interests include cross-layer design, architecture design and jamming detection.

Muhammad Hamad Alizai is a Research Assistant at the Chair of Communication and Distributed Systems, RWTH Aachen University. He received his Masters Degree from RWTH Aachen in 2007. His studies were funded by "DAAD – Siemens scholarship Asia 21st century". He started his PhD in the March of 2008 under the PhD scholarship program of DAAD. His research interests include wireless sensor networks mainly focusing on modeling & simulation, protocol design, and link level optimizations.

Florian Schmidt is a Research Assistant at the Chair of Communication and Distributed Systems, RWTH Aachen University. He received his Diploma Degree from RWTH-Aachen University in 2008. His studies were funded by the excellence initiative "Ultra High Speed Information and Communication – UMIC". He started his PhD in the September of 2008. His research interests include heuristic header recovery, network emulation, iterative source–channel decoding and rate adaptation.

Hanno Wirtz is a Research Assistant at the Chair of Communication and Distributed Systems, RWTH Aachen University. He received his Diploma Degree from RWTH–Aachen University in 2008. His studies were funded by the excellence initiative "Ultra High Speed Information and Communication – UMIC". He started his PhD in the October of 2008. His research lies in (mobile) wireless networks, peer-to-peer-systems and routing mechanisms.

Klaus Wehrle is professor of Computer Science and head of the Chair of Communication and Distributed Systems (Informatik 4) at RWTH Aachen University, Germany. He received his Diploma (1999) and PhD (2002) degrees from University of Karlsruhe (now KIT), both with honors. He joined the International Computer Science Institute at University of California at Berkeley from 2002 to 2003. In 2004 he was awarded a DFG Emmy Noether grant and established a junior research group on Protocol Engineering and Distributed Systems at University of Tübingen. In 2006, he joined RWTH Aachen University as associate professor, later as full professor. His research activities are focused on (but not limited to) engineering of networking protocols, (formal) methods for protocol engineering, sensor networks, network simulation, peer-to-peer-networking as well as all operating system issues of networking. He is a member of IEEE, ACM, Sigcomm, GI (German Informatics Society), VDE, and GI/ITG-Fachgruppe KuVS.