CRAWLER: An Experimentation Platform for System Monitoring and Cross-Layer-Coordination

Ismet Aktas, Florian Schmidt, Muhammad Hamad Alizai, Tobias Drüner, Klaus Wehrle
Communications and Distributed Systems
RWTH Aachen University, Germany
Email: {aktas,schmidt,alizai,druener,wehrle}@comsys.rwth-aachen.de

***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 monitoring, experimenting and specifying 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 for accessing application, protocol and system information. The generic design of this interface further enables a convenient and declarative way to specify and experiment how a set of cross-layer optimizations should be composed and adapted at runtime. Our evaluation demonstrates the usability of CRAWLER by experimenting, monitoring and improving TCP's congestion control algorithm.

I. Introduction

Developing real-world protocols and applications for wireless and mobile systems is difficult. The volatile nature of the wireless medium and mobility complicate their 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.

Network analysis tools, such as wireshark [1], only allow the inspection of traffic at few 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, recent research [2] has shown that availability of cross-layer information would allow an application to be more adaptive. Not only applications, but also protocols show significant adaptability advantages when following the cross-layer paradigm. For example, in mobile and wireless systems, even a single cross-layer optimization at the MAC layer can achieve throughput speedups of up to 20 times and latency reduction of up to 10 times over regular TCP [3]. Despite this tremendous potential to enhance system performance and boasting 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].

Although several static cross-layer architectures have been proposed, we lack a generic and flexible architecture that enables developers to specify and experiment with cross-layer optimizations. A static cross-layer architecture [5]–[8] facilitates easy manipulation of protocol-stack parameters and combines several specific cross-layer optimizations. In current architectures of this type, cross-layer optimizations are composed offline (i.e., at compile time) and are deeply embedded within the OS. This approach has two key limitations that motivate the ideas presented in this paper.

First, the process of adding or removing an optimization is cumbersome: optimizations need to be patched with the architecture, and because the architecture is deeply embedded with the OS, recompiling the kernel and rebooting the system are typical consequences. Hence, the developer has to deal with too many system internals before 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. Hence, an application or environment specific optimization 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-in to 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 [9] which underlines the need for dynamic adaptation of the system behavior (i.e., protocols, system components, and applications) based on the application requirements and the network conditions.

In this paper we present CRAWLER, a new experimentation architecture for system monitoring and cross-layer-coordination that facilitates the evaluation of applications and wireless network protocols. The target audience of CRAWLER are application and system developers of wireless networks who want to experiment and evaluate their software. Specifically, CRAWLER provides the following key features that illustrate its departure from the existing work and mark the contributions of this paper.

- It simplifies the process of monitoring and experimentation by providing a unified interface for accessing application, protocol and system information which is independent of the OS internals.
- The generic design of the unified interface further simplifies the process of specifying cross-layer optimizations by providing a declarative way to specify how a set of optimizations should be composed and adapted at runtime.
- It offers (i) a very high degree of flexibility, to experiment with different compositions of cross-layer optimizations, and (ii) extensibility, to include heterogeneous protocols and system components 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.

The remainder of this paper is organized as follows. Section II presents a system overview, highlights our design goals, and comprehends the scope of our architecture. Based on our design goals, Section III describes our architecture from a conceptual point of view. We evaluate CRAWLER in Section V. Finally, we discuss related work in Section VI before concluding the paper in Section VII.

II. Design Overview

CRAWLER consists of two main components as shown in Figure 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 to applications for (i) providing their own optimizations on demand, and (ii) exchanging information with the protocol stack and system components.

The cross-layer optimizations given by the LC are realized by the *cross-layer processing component* (CPC). Here, the rules are mapped to compositions of small *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 the optimization rules can be done at runtime

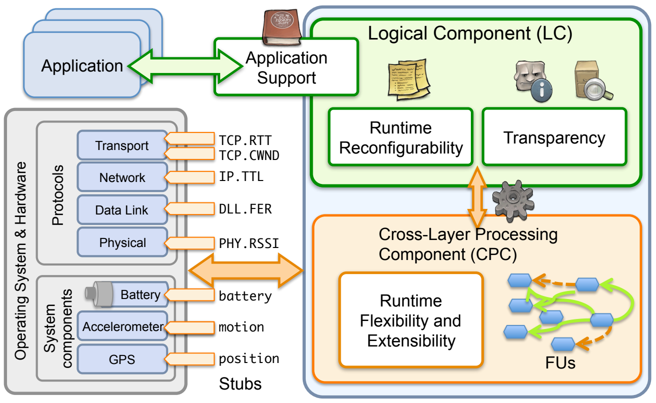


Figure 1. Conceptual view of CRAWLER's components. The logical component (LC) abstracts from the implementation of cross-layer optimizations via an easily usable but powerful rule-based configuration language. The cross-layer processing component (CPC) realizes the optimizations given by the LC which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

using the LC. These changes are reported to the CPC, which adapts the FU compositions accordingly.

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

A. Design Goals

Our design is centered around the following goals:

Transparency: 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 when developing new protocols and system components: Cross-layer optimizations should be transparent to the system developers. Nevertheless, as experts of their system it is beneficial if they already provide access to the protocol and system variables to support cross-layer interactions.

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.

Runtime Reconfigurability: The architecture should offer the ability to (i) detect the underlying environmental changes, and (ii) respond to the evolving application monitoring and optimization demands, by automatically loading the adequate set of optimizations at runtime. For example, energy saving optimizations may not be necessary if the device is plugged-in to a power supply.

B. *Cross-Layering in* CRAWLER*: Discussion*

CRAWLER runs on end-hosts and coordinates local information such as from the protocol stack and system components. CRAWLER itself does not provide information exchange among nodes in a network, such as in [10], 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.

Currently, CRAWLER neither provides protection against mis-configurations of optimizations nor resolves conflicts among concurrent optimizations. How to automatically detect such mis-configuration and conflicts [11] is a research challenge in itself, and a detailed exploration of the design space is beyond the scope of this paper. However, we plan to extend CRAWLER with features that allow easier detection of such problems. Nevertheless, CRAWLER is a playground to observe the effects of applications and protocols, and their different set of optimizations. Hence such mis-configurations and conflicts can already be easily detected manually. Similarly, it offers a very high degree of flexibility and thus simplifies the process of experimenting with finding the right set of optimizations for a certain use case.

III. 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 II-A.

A. *Goal 1: Achieving Transparency*

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 Figure 2. The *configuration* subcomponent allows a developer to express cross-layer optimizations on an abstract level and thus it 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 their current state 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 *transparency*. We postpone discussion on the application support subcomponent to Section III-B because it meets another design goal, namely design goal 2.

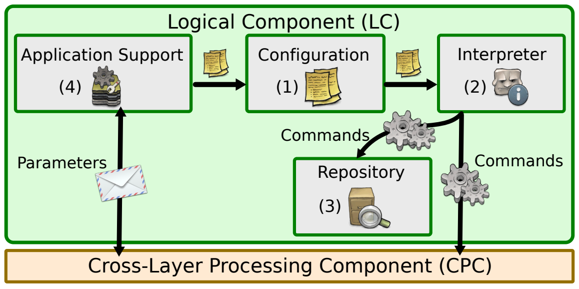


Figure 2. The LC comprises four subcomponents. (1) The configuration is an abstract description of a cross-layer optimization. (2) The interpreter parses the configuration. (3) The repository saves snapshots of configuration setups, allowing easy access to the current and past setups. (4) The application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

1) Configuration: The first step in CRAWLER's construction 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 a declarative configuration. Each *rule* is a behavioral description of a cross-layer interaction within an optimization. Rules can be nested within other rules to form rule chains. 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. Figure 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 III-C2 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, in this example `55`.

```
1 my_rssi:get("phy.wlan0.rssi")
2 my_history_of_rssi:History(my_rssi, 4)
3 my_rssi_is_bad:Less(Avg(my_history_of_rssi), 55)
4 my_rssi_is_bad->my_appl_var1
5 my_rssi_is_bad->my_TCP_Freezer
6 my_appl_var1:set("application.appl.voip_var1", "bad")
7 my_TCP_Freezer:set("transport.tcp.cwnd", "0")
8 my_timer:Timer(200)
9 my_Timer->my_rssi_is_bad
```

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

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 triggering 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 III-D.

2) Interpreter: In the next step, this high level configuration of cross-layer interactions needs to be transformed into the actual optimization. To this end, the interpreter 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 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 mapped into the commands `createFU(History)`, `addParameter(my_rssi)` and `addParameter(4)` which results in the wiring of corresponding FUs as shown in Figure 3. The handling of commands and the realization of cross-layer interactions are explained later in Section III-C.

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 and filter the rules again. In a running system, this allows more efficient switching between preprocessed sets of optimizations. Second, while designing and testing new cross-layer optimizations, the repository allows the developers to roll back to a previous optimization for debugging purposes.

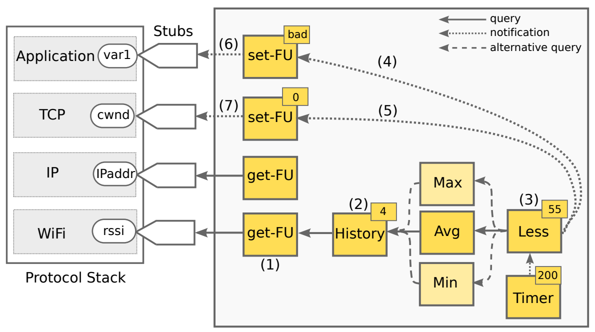


Figure 3. A simple cross-layer configuration in CRAWLER. We change the behavior of the TCP layer and an application based on signal strength.

Summarizing, the declarative approach of specifying cross-layer interactions and the ability to troubleshoot cross-layer optimization at the very early stages of development enhances the usability and maintainability of CRAWLER. Similarly, because it allows to specify cross-layer optimization at a high level of abstraction, CRAWLER does not impose any specific requirements on protocol and system developers. Hence, the collaboration of these three subcomponents of the LC fulfills our design goal of transparency.

B. 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 would offer 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 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. For monitoring purposes, we have implemented a monitoring application (works using the shared library) that relieves the developer from any configuration efforts. In order to conveniently monitor a set of already available variables within the OS, the monitoring application can be simply called from the console with a variable such as the sending congestion window (`snd_cwnd`) of TCP as an argument. This simple call starts the application and constantly monitors and logs the desired variable.

C. Goal 3: Flexibility & 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.

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), 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 Figure 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.

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 a 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. In case of a new incoming query, the FU can then decide to return the cached value or recompute a new one.

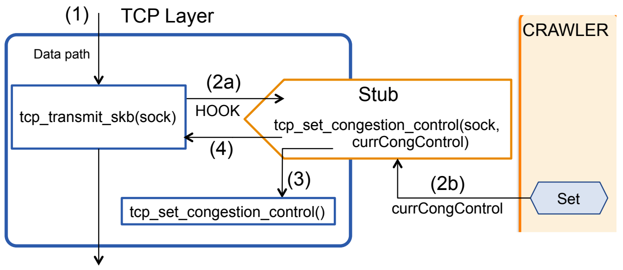


Figure 4. Stub for changing TCP's congestion control algorithm.

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.

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: Each protocol and system variable has its 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.

CRAWLER's runtime, the CPC, associates set and get FUs with each stub included in the architecture, as shown in Figure 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.

Figure 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`. (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 section V-A to demonstrate a use case.

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.

D. Goal 4: Runtime Reconfigurability

Runtime reconfigurability 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. Dependencies are automatically resolved 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 III-A2.

unload(rule_name): The rule `rule_name` is unloaded at runtime. 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 Listing 1.

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 allow reconfiguration of chains at runtime. CRAWLER also provides mechanisms to automatically execute the commands associated with these keywords based on the 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, 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 line 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.

```
1  [manual]
2  rssiavg:avg(history(get("wlan0.qual.rssi"),10))
3  less1:less(rssiavg,60)
4  packetLossRate:get("app.switchCwnd.packetLossRate")
5  less2:less(4,packetLossRate)
6  changeCwnd:and(less1,less2)
7  cwndAlg:if(changeCwnd,"westwood","vegas")
8  initPort:set("tcp.activate.outgoingPacketsPort",5001)
9  setCwndAlg:set("tcp.cong_control_5001", cwndAlg)"
10
11 [contextEnter]
12 loadOpt:get("app.switchCwnd.loadOpt")
13 loadOpt->load(setCwndAlg)
14
15 [contextExit]
16 removeOpt:get("app.switchCwnd.removeOpt")
17 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.

IV. Implementation

We implemented CRAWLER[1] 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 [12]. 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.

V. Evaluation

Our evaluation of CRAWLER focuses on two aspects: First, we give an example of how to use CRAWLER for monitoring and cross-layer adaptation purposes by focusing on a well-known cross-layer example, that is, TCP congestion control in the wireless world. Second, we evaluate the overhead by using benchmarks for time critical parts of the architecture.

[1]This paper 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/

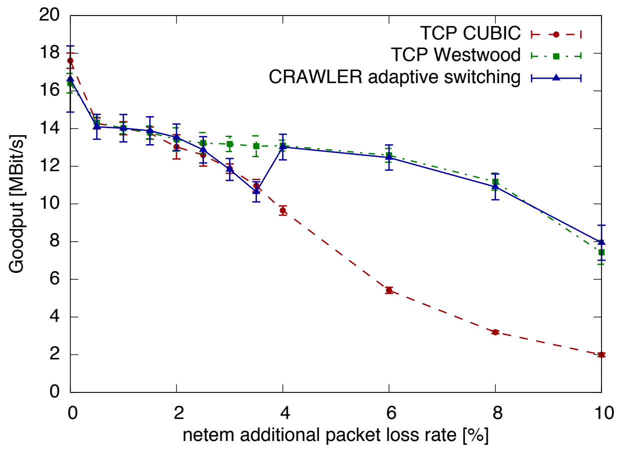


Figure 5. Adapting TCP's congestion control: The switch from CUBIC to Westwood is performed at a packet loss rate of 4%. The error bars represent the 95% confidence interval of ten repeated experimental runs.

A. *Use case: Monitoring and Adaptation of TCP Congestion Control Algorithms*

To give an insight into how modeling and 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 [13] and Westwood [14], 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 802.11g WLAN card. We use Iperf [15] to create TCP traffic, and netem [16] 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 802.11g 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 Figure 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 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.

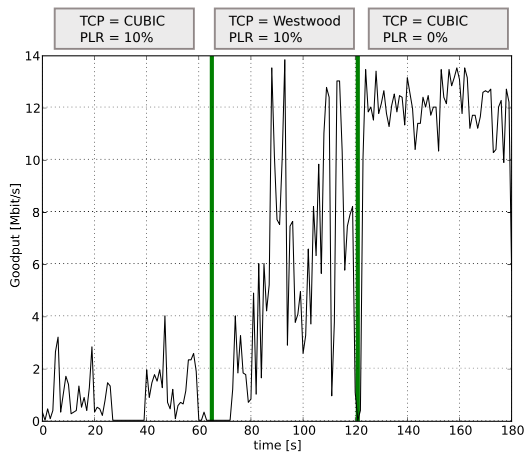


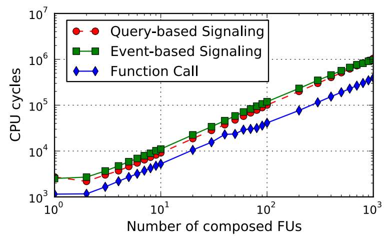
Figure 6. Goodput of a TCP transmission over time under varying environmental conditions and congestion control algorithms. The optimization is loaded after 60 seconds which triggers the switch from CUBIC to Westwood. The switch back to CUBIC is triggered when the packet loss rate (PLR) falls below the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

The effect of this optimization is shown in Figure 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.

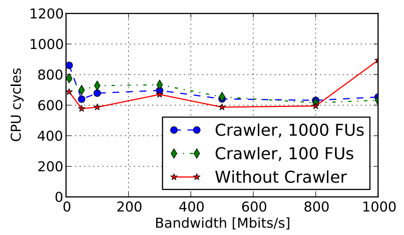
Figure 6 shows our results for a longer experimental run, and also highlights the possibility to load rules at runtime. For the first 60 seconds, we did not load the optimization into the CPC, as depicted by the low TCP goodput achieved during this time. The optimization is loaded at 60 seconds which triggers the switch from CUBIC to Westwood and subsequently improves the goodput. Similarly, at 120 seconds, 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.

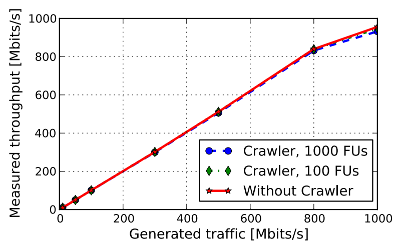
This example demonstrates the correctness of CRAWLER's



(a) Signaling overhead for query-based signaling and event-based signaling



(b) Packet processing duration within the protocol stack



(c) Measured throughput for different FU composition sizes

Figure 7. 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.

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

B. *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, (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 signalling (cf. Section III-C1) 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.

Figure 7(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 [15]. Figure 7(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. Figure 7(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.

VI. Related Work

A plethora of **specific cross-layer solutions** [4], [17], [18] 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. 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 [5] 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 [6] 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 [7] 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 [8] is the most advanced cross-layering architecture that provides a generic interface for accessing protocol stack. Its generic interface facilitates platform independence but it does not support dynamic adaptability of cross layer optimizations at runtime.

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, and (iii) enable runtime adaptability of cross layer optimizations depending upon the underlying network conditions. To the best of our knowledge, these key features are not supported by the existing cross-layer architectures.

VII. Conclusions and Future Work

In this paper, 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. 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 plan to extend CRAWLER with the ability to automatically detect mis-configurations and conflicting optimizations. This is important since each application can specify its own optimizations and load them at runtime. We 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.

VIII. Acknowledgments

We would like to thank Raimondas Sasnauskas, Dominik Dennisen, Hendrik vom Lehn, Martin Henze, Hanno Wirtz, Tobias Heer, Jó Bitsch and Cem Mengi for many fruitful discussions and valuable feedback. This research was funded in part by the DFG Cluster of Excellence on Ultra High-Speed Mobile Information and Communication (UMIC).

References

[1] "Wireshark network protocol analyzer," http://www.wireshark.org/, accessed Nov. 5, 2011.

[2] S. Khan, Y. Peng, E. Steinbach, M. Sgroi, and W. Kellerer, "Application-driven cross-layer optimization for video streaming over wireless networks," *IEEE Communications Magazine*, vol. 44, no. 1, pp. 122–130, 2006.

[3] H. Balakrishnan, S. Seshan, and R. H. Katz, "Improving reliable transport and handoff performance in cellular wireless networks," *Wireless Networks*, vol. 1, pp. 469–481, 1995.

[4] V. Srivastava and M. Motani, "Cross-layer design: a survey and the road ahead," *IEEE Communications Magazine*, vol. 43, no. 12, pp. 112–119, 2005.

[5] Q. Wang and M. Abu-Rgheff, "Cross-layer signalling for next-generation wireless systems," in *IEEE WCNC*, vol. 2, 2003, pp. 1084–89.

[6] C. Sadler, L. Kant, and W. Chen, "Cross-layer self-healing mechanisms in wireless networks," in *Proc. WWC*, vol. 254, 2005.

[7] M. Conti, G. Maselli, G. Turi, and S. Giordano, "Cross-layering in mobile ad hoc network design," *Computer*, vol. 37, no. 2, pp. 48–51, 2004.

[8] V. Raisinghani and S. Iyer, "ECLAIR: An efficient cross layer architecture for wireless protocol stacks," in *Proc. World Wireless Congress*, 2004.

[9] I. Aktas, J. Otten, F. Schmidt, and K. Wehrle, "Towards a flexible and versatile cross-layer-coordination architecture," in *Proc. INFOCOM'10 Work-in-Progress Session*, 2010.

[10] R. Winter, J. Schiller, N. Nikaein, and C. Bonnet, "Crosstalk: Cross-layer decision support based on global knowledge," *Communications Magazine, IEEE*, vol. 44, no. 1, 2006.

[11] V. Kawadia, P. Kumar, B. Technol, and M. Cambridge, "A cautionary perspective on cross-layer design," *IEEE Wireless Communications*, vol. 12, no. 1, pp. 3–11, 2005.

[12] P. Neira-Ayuso, R. Gasca, and L. Lefevre, "Communicating between the kernel and user-space in Linux using Netlink sockets," *Software: Practice and Experience*, 2010.

[13] S. Ha, I. Rhee, and L. Xu, "CUBIC: a new TCP-friendly high-speed TCP variant," *SIGOPS Oper. Syst. Rev.*, vol. 42, no. 5, pp. 64–74, 2008.

[14] S. Mascolo, C. Casetti, M. Gerla, M. Y. Sanadidi, and R. Wang, "TCP westwood: Bandwidth estimation for enhanced transport over wireless links," in *Proc. MobiCom'01*. New York, NY, USA: ACM, 2001, pp. 287–297.

[15] A. Tirumala, F. Qin, J. Dugan, J. Ferguson, and K. Gibbs, "Iperf: The TCP/UDP bandwidth measurement tool," http://iperf.sourceforge.net/, 2004.

[16] S. Hemminger, "Netem-emulating real networks in the lab," in *Proc. Linux Conference Australia*, 2005.

[17] G. Carneiro, J. Ruela, M. Ricardo, and I. Porto, "Cross-layer design in 4G wireless terminals," *IEEE Wireless Comm.*, vol. 11, no. 2, pp. 7–13, 2004.

[18] V. Raisinghani and S. Iyer, "Cross-layer design optimizations in wireless protocol stacks," *Computer Communications*, vol. 27, no. 8, pp. 720–724, 2004.