# A Framework for Remote Automation, Configuration, and Monitoring of Real-World Experiments

Ismet Aktaş, Oscar Puñal, Florian Schmidt, Tobias Drüner, Klaus Wehrle
Chair of Communication and Distributed Systems,
RWTH Aachen University, Germany
{aktas,punal,schmidt,druener,wehrle}@comsys.rwth-aachen.de

## ABSTRACT

The evaluation of wireless and mobile communication systems in real-world testbeds can be cumbersome and tedious. Experiments require manual intervention to coordinate the execution of involved programs and to collect test results on each involved device. Moreover, the collection of test results from protocols is difficult due to operating system restrictions. In contrast, simulation offers the ability to easily log such information and automate whole experiments. Real-testbeds should offer the same flexibility and convenience to automate whole experiments and to collect test results as in simulation. In this paper, we propose a framework that effectively addresses this challenge. Moreover, with the integration of the cross-layer architecture CRAWLER, we demonstrate that we are able to automate experiments where cross-layer optimizations are involved and while experimenting we centrally monitor and log parameters across protocol layers on different devices.

## Categories and Subject Descriptors

C.2.0 [**Computer Systems Organization**]: Computer-Communication Networks—*Data Communication*

## Keywords

Cross-Layer Design; Monitoring; Testbed; Test Automation

## 1. INTRODUCTION

The design and evaluation of protocols for communication systems in general, and wireless distributed systems in specific, is a challenging task. Typically, either network simulations or, depending on their availability, real testbeds are considered for that purpose. Using the latter has the advantage of implicitly taking hardware, software, and environmental specific issues into account that are typically abstracted in simulations. However, real testbeds usually lack the flexibility of a simulator for accessing performance metrics of interest and relevant system parameters that impact the behavior of the protocol under investigation. Moreover, when multiple devices are involved, running new experiments with different programs and parameterizations and collecting the test results can be a monotonous, time-inefficient task, as it requires manual intervention of the designer [5]. This particularly applies to evaluation of protocols for wireless communication systems, as typically multiple repetitions of the experiment are necessary to neutralize the unpredictable behavior of the channel.

Our goal is to carry over advantages of simulation to real testbeds by (i) providing the flexibility to access and log parameters in devices involved in testing and (ii) enabling the automated execution of experiments. To enable the latter, we propose a client-server architecture that allows for a centralized execution of experiments and collection of results. To achieve the former, we rely on CRAWLER [1], an experimentation platform for monitoring and cross-layer coordination. 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 its 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. This feature not only provides the necessary degree of flexibility to conveniently specify and monitor the desired system information even while experimenting, it also allows us to experiment with cross-layer optimizations. This is in contrast to other *static* architectures [3, 11], where optimizations are designed at compile time and require a system reboot.

In this work, we present a framework that provides the following three key features:

- **Remote test automation** allows a developer to remotely describe whole experimentation setups, that is, start and termination of applications with their respective parameterization and cross-layer optimizations, on a specified set of nodes without physical human interaction.
- **Remote configuration** allows to distribute cross-layer optimizations to remote nodes, to control them (i.e., to add, remove, and modify optimizations), and enables to conveniently access protocol and system information at runtime.
- **Remote monitoring** allows to conveniently specify and log a set of predefined parameters and states of cross-layer optimizations from any node involved in the test and, if desired, to store them centrally. Moreover, live monitoring of parameters and the ability to adjust the level of report granularity are enabled at runtime.


*WiNTECH'14,* September 7, 2014, Maui, Hawaii, USA.

http://dx.doi.org/10.1145/2643230.2643236.

The remainder of this paper is organized as follows: Section 2 introduces CRAWLER and Section 3 presents an overview of our framework. In Section 4, we describe the details of our three key features. Implementation details are provided in Section 4.5. We evaluate our architecture in Section 5 and discuss related work in Section 6. Finally, we conclude the paper in Section 7.

## 2. BACKGROUND

CRAWLER [1] already offers a unified interface which allows to add, remove and modify cross-layer optimizations at runtime. For this, CRAWLER only requires an abstract configuration in its configuration language. Using this language, developers can specify cross-layer optimizations at runtime and at a high level of abstraction without needing to care about implementation details. Applications that want to use these functionalities have to register beforehand at CRAWLER (running as a daemon) using a shared library. This only requires an application to include the shared library's header file `crawler.h` and link against the library. Moreover, CRAWLER's unified interface not only allows applications to provide their own set of cross-layer optimizations but also to exchange information with protocols and system components. In the latter case, the shared library takes care about the interaction between the applications and CRAWLER. To highlight CRAWLER's power, we want to give a small example.

Listing 1 presents a cross-layer optimization to switch TCP's congestion control algorithm at runtime while keeping the internal values such as the congestion window without resetting them. In the first line of the configuration the radio signal strength indicator (RSSI) is accessed. CRAWLER uses so-called stubs which require a fully qualified name, i.e., unique and hierarchical, to access a specific variable in the system. Stubs act as a glue element between the optimizations and the OS. However, in line two of the configuration, the average over 10 values of the RSSI is calculated. In line three this average value is compared to a specific threshold. If the threshold is exceeded as expressed in line three, the congestion control algorithm is switched in line four.

```
1 myRssi:get("wlan0.qual.rssi")
2 rssiAvg:avg(history(myRssi),10)
3 rssiIsLow:less(rssiAvg,60)
4 cwndAlg:if(rssiIsLow,"westwood","vegas")
5 setCwndAlg:set("tcp.cwnd", cwndAlg)
6 load(setCwndAlg)
```

**Listing 1: A cross-layer optimization to switch TCP's congestion control algorithm using Crawler's declarative language.**

Each line in the configuration is a rule which can be nested into complex rule compositions. The keyword `load` in line five instructs CRAWLER to automatically load a specific rule into the system, here rule `setCwndAlg`. In cases of nested rules, the dependencies are automatically resolved and all necessary rules are all loaded into the system. A removal of rules or their replacement work similarly. In particular, by using the keywords `unload` or `replace` a rule is removed or exchanged with another, respectively.

After providing such a configuration, CRAWLER takes care to realize the cross-layer optimization. In particular, based on the configuration, functional units (FUs) are composed

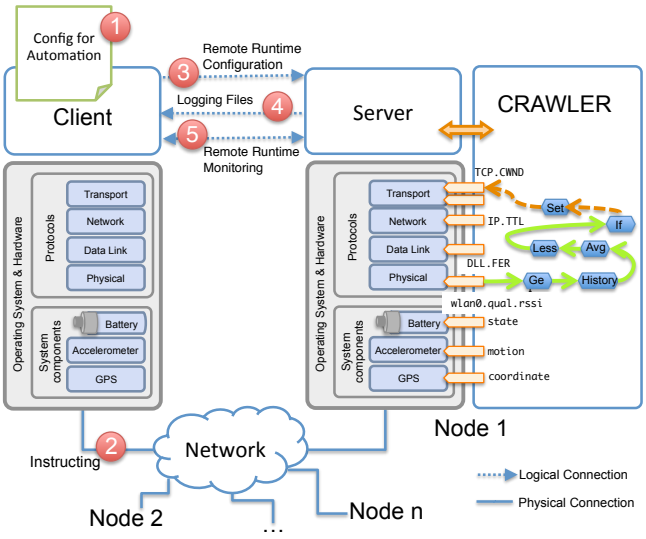


**Figure 1: Conceptual view for remote automation, configuration and monitoring.**

to realize the desired cross-layer optimization (cf. node 1 in Figure 1). FUs are stateful functions that maintain a record of the data and provide results based on that record each time they are called. For example, every instance of `History` keeps its collected values between calls. Moreover, FUs share a unified interface so that they can be flexibly composed with each other. As long as a configuration does not delete FUs but only changes their composition, they will keep their current state and collected information. For example, by exchanging rule 2 (`rssiAvg`) in Listing 1, we can replace the `Avg` FU (cf. Figure 1) with `Min` or `Max` at runtime due to the uniform interface, and still use the collected data from `History`.

## 3. DESIGN OVERVIEW

This section provides an overview of the design and the used components. In order to enable developers to centrally and remotely automate, control and monitor their experiments, we opted for a client-server architecture as shown in Figure 1. The client is the central node that controls several servers running on different devices that are being part of the experiment.

To conduct experiments with our architecture, the developer provides a configuration file (cf. ① in Figure 1). The configuration includes the required parameters for the entire experiment. In particular, it includes information about the devices that are part of the experiment, the programs and cross-layer optimizations that should run on the devices, the schedule of the programs, and information about parameters that should be logged. Subsequently, the client parses the configuration and extracts the necessary tasks. Each task consists of its execution time, target device and instruction (e.g., to run a specific program). At the scheduled execution time, the client sends the given instruction to the listed devices via the network, see ②. Simply put, the client gives instructions about what should be done, when and on which device. The servers receive these instructions and realize them. Only the servers need to run CRAWLER.

We decided to extend Crawler with communication and remote control capabilities because its architecture already supported dynamic loading and unloading of programs and cross-layer optimizations. Crawler so far offered a system-wide interface to provide these abstractions. By enhancing the interface, we enabled Crawler to distribute cross-layer optimizations defined by our central client to other nodes in a network. This capability enables new possibilities to control the servers remotely during experimentation, see ③. The configuration file also contains information about which system parameters to monitor on the servers. At the end of the experiment, the data is automatically aggregated at the client for easy evaluation, see ④. Another advantage of Crawler is its live monitoring feature which conveniently allows to monitor a set of variables in running applications, protocols and system components at runtime. We extended this feature to allow remote access, see ⑤. This enables live monitoring of variables during the experiment, without having to wait for the aggregation of log files at the conclusion of the experiment. This monitoring is supported by a graphical front-end, which also allows controlling and adapting cross-layer optimizations on any server during the experiment.

# 4. ARCHITECTURAL DETAILS

This section provides more details about how we realized the three key features of remote automation, configuration and monitoring and describe how we incorporated them into our interactive front-end.

## 4.1 Remote Automation

In a testbed experiment, all nodes have to be set up with the appropriate software and configuration. For instance, the nodes might be instructed to run different applications and cross-layer optimizations, and produce different data traffic patterns. After the tests, log files have to be analyzed on each node to understand the interplay of the programs. Moreover, due to outside effects in testbeds, especially in wireless networks, experiments have to be repeated many times to lend credibility to the results. Ideally, the adjustment of parameters, the repetition of the test and the collection of log files should be very convenient.

We propose an architecture which allows central configuration and execution of testbed setups. The configuration includes the necessary instructions to automate the execution of desired settings including applications, cross-layer optimizations and further helper programs. An example configuration is shown in Listing 2 which we use later in the evaluation. The configuration is subdivided into sections by keywords enclosed in brackets.

The `[crosslayerremote]` section includes information about the involved remote servers in the test. For example, in line 2 `crosslayer_server_count` defines the number of remote servers used in the test. Line 3 gives the port number which is used by Crawler running on the servers for incoming requests for a remote connection. In the subsequent lines the details about the three servers are configured. To differentiate between several servers, we again use the bracket notation followed by the number of the specific server. For instance, in line 4 we configure server 1 which we assign an alias, here `node01`, followed by the IP-address assigned in line ten. The alias allows assigning mnemonic

```
1 [crosslayerremote]
2 crosslayer_server_count: 3
3 crawler_default_port: 12345
4 [crosslayerremoteserver1]
5 alias: node01
6 ip_address: 192.168.0.1
7 [crosslayerremoteserver2]
8 alias: node02
9 ip_address: 192.168.0.2
10 [crosslayerremoteserver3]
11 alias: node03
12 ip_address: 192.168.0.3
13
14 [schedule]
15 logclear: [([1,2,3], 1, "rm logfile.txt")]
16 daemonclear: [([1,2,3], 1, "rm daemon.txt")]
17 crawler:[([1,2,3],3,"load_crawler cfg80211Layer
18     ethernetLayer")]
19 cfg_n01:[('node01',3,"iwconfig wlan1 txpower 8;")]
20 cfg_n02:[('node02',3,"iwconfig wlan0 txpower 8;")]
21 cfg_n03:[('node03',3,"iwconfig wlan1 txpower 4;")]
22
23 detect01:[('node01',4,"crawlerapp mainJammingApp")]
24 detect02:[('node02',4,"crawlerapp mainJammingApp")]
25 detect03:[('node03',4,"crawlerapp mainJammingApp")]
26 stopdetect:[([1,2,3], 124,"killall mainJammingApp")]
27 on_finish:[([1,2,3],130,"load_crawler cfg80211Layer
28     ethernetLayer")]
29
30 [log]
31 log_crosslayer_remote_server: True
32 log_daemon: True
33 jammingdetLog: [([1,2,3], "logfile.txt")]
```

**Listing 2: Setup of the configuration used for the evaluation of the jamming detection scenario.**

names to servers in the configuration. The two remaining servers are similarly configured (cf. lines 7–12).

The main part of the configuration is the `[schedule]` section which determines when to execute which instruction on which server. For this, we use a three-tuple notation `(<server>, <time>, <command>)`; the first element indicates the server, the second the relative time in seconds since the start of the experiment, and the third the instruction that should be executed. Commands can be Crawler specific functions (such as `load_crawler` and `crawlerapp`) or shell commands. As an example of our tuple notation, in line 19, we use in the first element the alias of the node (`node01`) to specify the server; the second element indicates the time of execution of a specific command (here 3 seconds after starting the experiment); and the third element indicates the command that should be executed (here the manipulation of transmission power `txpower` which is modified using the wireless tools `iwconfig`). The subsequent lines (20–21) instruct the other servers similarly.

To avoid having to execute the same instructions for different servers over and over again, we provide a bracket notation to address all servers at once such as shown in lines 17–18: the three servers are instructed to load Crawler (using the special keyword `load_crawler`) with some of its kernel modules after three seconds. In lines 15–16 we also used this notation to first clean all log files before starting to collect data with Crawler.

However, the main instruction in this configuration is the start of a jamming detection application on each server. This is done in lines 23–25. The `mainJammingApp` loads a previously defined cross-layer optimization (that determines the

jamming detection strategy).[1] Finally, in lines 26–28 the instructions to stop the whole test are given.

To deliver logged parameters to the central place, i.e., the client, the `[log]` section is used. For example, line 33 indicates that all log files from all three servers should be collected.

The configuration presented so far only considered one single test run. To relieve the user from rewriting configuration files for each little parameter change, we support additional custom configuration files. The presented Listing 2 constitutes a main configuration file. The custom configuration describes the relative changes and overwrites the respective values of the main configuration. For example, if we want to have an earlier scheduling time (2 instead of 3) for loading CRAWLER, we only have to add a single modified line in the custom configuration in the respective section (`[schedule]`) such as `crawler: [([1,2,3], 2, "load_crawler cfg80211Layer ethernetLayer")]`. This will then override the command given in line 17 of Listing 2.

## 4.2 Remote Configuration

CRAWLER only allowed a system-wide access to protocols and system components. A remote access from another system was not feasible. Hence, we have enhanced the shared library to allow a remote client to interact with a server, even securely (via OpenSSL). Its use to access remote nodes is similarly simple as for the local use. It only requires to provide few more details about the remote server (e.g., IP address of server and certificate). Furthermore, our framework allows a client to distribute cross-layer optimizations to remote nodes and to control them (i.e., to add, remove, and modify optimizations). This feature together with the ability to remotely access nodes, allow the remote configuration of nodes involved in the experiment with a unprecedented degree of convenience and this even at runtime. To use the shared library, it is only necessary, to include the `crawler.h` file and link against the shared library.

In order to further simplify the process of adding cross-layer optimizations into remote devices, we implemented a helper application which we refer to as `addChainsApp`. The optimization has to be provided as a configuration file using CRAWLER's declarative language. Let we assume a configuration as described in Listing 1. We can now utilize the `addChainsApp` application to remotely configure cross-layer optimizations in a remote system. For example, the following command in the console allows to provide the above described TCP cross-layer optimization contained in the file `myconfig.cfg` to the server reachable via the IP-address 192.168.0.5 and port 12345.

```
1 $>./addChainsApp --host 192.168.0.5 --port 12345
2     --chain myconfig.cfg
```

On calling this command in the console, the `addChainsApp` application first remotely registers at the server as illustrated by step ① in Figure 2. Subsequently, `addChainsApp` sends the configuration containing the cross-layer optimizations to the remote system (cf. step ②). Finally, the server passes the configuration to CRAWLER which automatically realizes the given configuration (cf. step ③).

[1]The details of this specific optimization are presented in [10].

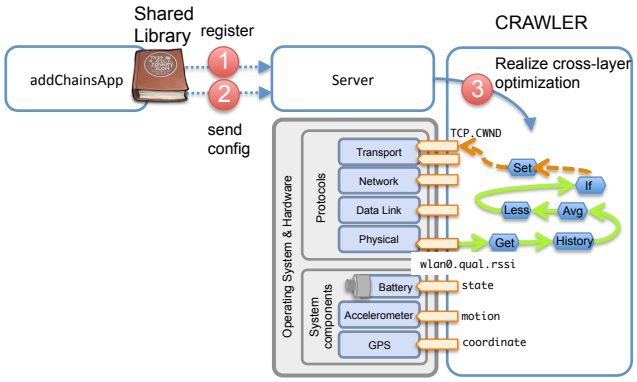


**Figure 2: The `addChainsApp` allows to remotely configure optimizations. After ① registration, the configuration ② instructs the server to add, remove or replace a set of optimizations, which are then ③ realized by the server.**

## 4.3 Remote Monitoring

Access to system information, i.e., to protocols and system components such as sensors, is relatively difficult due to OS limitations. However, the access to system information significantly helps to understand and debug the experiment and the interplay between algorithms and other effects such as the unpredictable wireless channel. Moreover, it will also help to measure and analyze the benefits of the envisioned algorithms. While CRAWLER provides the necessary interfaces to locally access these information, a remote access was initially not enabled. With the improvements to the shared library as presented above, we also lay out the foundation for remote monitoring. The improvements allow us to remotely feed cross-layer optimizations into the system and also to access the desired system information. We implemented the helper application `monitorVariableApp` which makes use of these features. In particular, it generates CRAWLER configurations to conveniently monitor the desired information. For example, by calling the following command in the console it is possible to monitor the RSSI (specified by its fully qualified name `wlan0.qual.rssi.avg`) every 100 milliseconds (update intervals) on the remote server with the IP address 192.168.0.5.

```
1 $>./monitorVariableApp --host 192.168.0.5 --port 12345
2     --variablename "wlan0.qual.rssi.avg" --interval 100
```

On calling this command in the console, `monitorVariableApp` first remotely registers at the server running on the specified remote node as indicated by step ① in Figure 3. In a next step, based on the given name of the variable, a CRAWLER configuration is created that specifies the access to the desired stub (information accessor). This configuration is then sent to the remote system (step ②). Based on the given configuration, CRAWLER realizes the cross-layer optimization to read the specified variable (step ③a). Finally, the variable is continuously monitored and, based on the specified update intervals, provided to the server (step ④a) which delivers the monitored values to the client running the `monitorVariableApp` (step ⑤).

In addition to monitoring system variables, it would be also interesting to monitor intermediate states of the cross-layer optimization, for example, to monitor the average val-

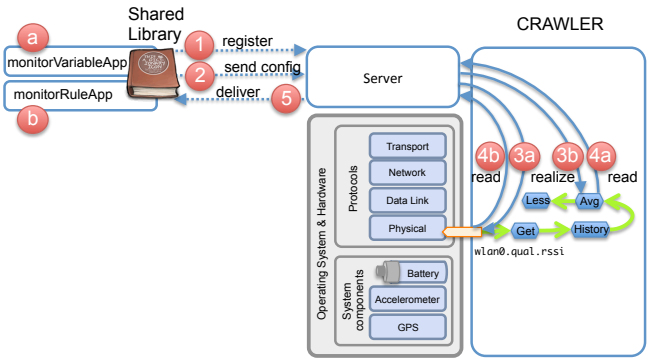


**Figure 3: The `monitorVariableApp` allows to remotely monitor system variables. After ① registration, the configuration ② instructs the server to monitor a variable. Crawler automatically ③ realizes the access, ④ monitors the variable, and ⑤ delivers the values to the client.**

ues of the RSSI in our optimization to modify TCP's congestion control algorithm (which we previously added with `addChainsApp`). As Crawler realizes such cross-layer optimizations by composing functional units (FUs), monitoring FUs already provides the essential monitoring ability to understand and debug cross-layer optimizations. For this purpose, we have implemented the dedicated application `monitorRuleApp` which works similar to `monitorVariableApp` but allows to monitor a specific FUs. For example, by calling the following command in the console, it is possible to monitor the average RSSI value every 500 ms (update interval) on the remote server with the IP address 192.168.0.5.

```
$>./monitorRuleApp --host 192.168.0.5 --port 12345
    --rulename "rssiAvg" --interval 500
```

On calling this command the `monitorRuleApp` first remotely registers at the server running on the specified remote node, cf. step ① in Figure 3. Compared to `monitorVariableApp`, the `monitorRuleApp` accepts only an identifier of a rule instead of a fully qualified name of a stub. Based on the given identifier, a Crawler configuration is created that specifies the access to the desired FU. This configuration is then sent to the remote system (cf. step ②). Upon receiving the configuration, Crawler realizes the access to the specified FU, for instance, the access to the FU that provides the average RSSI values (cf. step ③b). From then on, the specific FU is continuously monitored and based on the specified update intervals provided to the server (cf. step ④b). Finally, these values are further delivered to the `monitorRuleApp` (cf. step ⑤).

To summarize, by using the `addChainsApp`, we enabled to conveniently add cross-layer optimizations into remote systems. While the application `monitorVariableApp` allows to remotely monitor a set of variables in a specified system, the application `monitorRuleApp` allows to remotely monitor internal states of cross-layer optimizations. These helper applications can be conveniently used within any other software, while the applications `monitorVariableApp` and `monitorRuleApp` only requires that the software using them is able to processes and pipe the standard input or rather output. In the following, we demonstrate the benefits of our three helper applications in a graphical and interactive front-end.

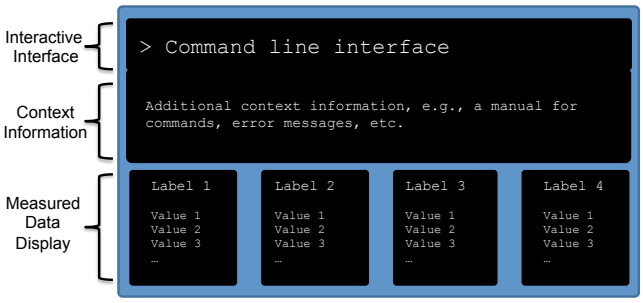


**Figure 4: A sketch of the interface layout**

## 4.4 Graphical and Interactive Front-End

To further support the developer while experimenting, we integrated our helper applications into a graphical front-end that provides the following interactive commands. First, the `addChains` command makes use of the helper application `addChainsApp` and allows to add, remove and modify cross-layer optimizations at runtime. Second, by using the command `monitor` the helper applications `monitorVariableApp` and `monitorRuleApp` are used in the back-end. Third, the `run` command allows to control third-party programs remotely and allows to intervene into a test such as starting or stoping programs, e.g., when misbehavior is observed. Another use could be to start further helper programs such as tcpdump or netem. In effect, this relieves the user from manually opening remote shells to all nodes, giving a unified control interface.

Figure 4 shows a sketch of the interactive front-end. The layout consist mainly of three areas: (i) an area that allows to provide the commands, (ii) an area for given additional information regarding the commands or providing feedback information from our toolchain, and (iii) the data that is being monitored. In the latter case, the area is subdivided into fields reflecting each parameter type such as radio signal strength (RSS). The field is labeled at the top with the node name and the name of the monitored variable, (e.g., `node01 wlan1.cfg80211.signal.avg`) and the most recent values below. Additional fields for monitoring are added or removed at runtime depended on given instructions.

## 4.5 Implementation

The implementation mainly consists of two separate parts: The first part is the extensions to Crawler where we enhanced the application support interface in order to make it accessible over the network. For this it was necessary to add further functionality (remote configuration and monitoring) into the shared library. These enhancements are implemented in C++. Remark, we have also enabled a secure remote configuration using OpenSSL. Second, the remote automation and the interactive remote front-end are implemented in Python. We opted for Python as it allows scripting at a high level, which makes it very suitable as a test automation language. The graphical front-end for the helper applications uses the curses [9] library. Communication between the remote client and the remote servers is realized with PYthon Remote Objects (PYRO). A client connects to remote servers using the PYRO interface and can directly use objects of the corresponding remote server implementation.

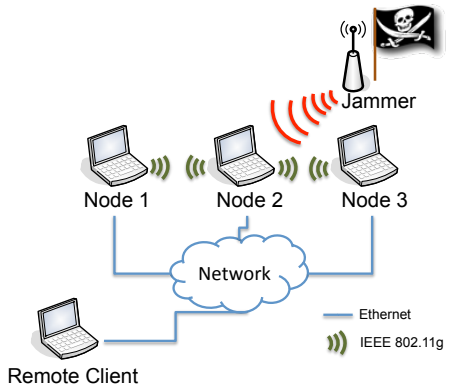


**Figure 5: Scenario used to evaluate our 3 key features. Consists of a three nodes and a remote client. The remote client controls the nodes involved in the experiment and monitors the impact of several metrics under jamming.**

# 5. EVALUATION

In this section we illustrate the benefits and potential of our three contributions by means of a practical experiment. In particular, we use a framework based on CRAWLER that provides 802.11 devices with capabilities for detecting the presence of jamming attacks in a distributed and reliable manner [10]. We enhanced the framework to support remote automation, configuration, and monitoring. In the following, we first describe the experimental setup and later present the main observations, thereby comparing the flexibility of the framework with and without our enhancements.

The experiment was conducted in a small office room in the ComSys Institute at RWTH Aachen University. We used three Linux PCs equipped with 802.11g/n Atheros WLAN cards running the ath9k driver [2]. We let the three PCs build a wireless ad-hoc network and continuously exchange messages. The remote access to the communicating devices was enabled by Ethernet connectivity. Note that the jammer device was not remotely controllable, as it is implemented on the WARP Platform [6], while CRAWLER is specifically designed for x86-based systems. A sketch of our reference scenario is shown in Figure 5.

## 5.1 Evaluating Remote Automation

The preparation of the experiment first requires to open various terminals on each communicating node. In one terminal the kernel modules of CRAWLER are loaded and it later displays kernel logs. A second terminal is used for adjusting the transmission power of the nodes (via iwconfig), starting CRAWLER, and printing log information of the CRAWLER daemon. In a third terminal the jamming detection framework is launched. Once the experiment is finished, the daemon running in second terminal and application running in the third terminal are closed and the respective log files (CRAWLER and kernel) are manually collected from each node. The tasks associated with the preparation and collection of results require several minutes of manual work. Clearly, such an approach does not scale well due to two main reasons. First, the required time increases linearly with the number of devices. Second, multiple experiment runs are generally required for a complete parameter study and to obtain statistical confidence in the results. However, once the short and intuitive configuration was specified as illustrated in Listing 2, the above-mentioned steps with our framework spanned only few seconds.

## 5.2 Evaluating Remote Configuration and Monitoring

In order to validate *remote configuration*, we show the proper working of the commands `addChains` and `run`. For validating the *remote monitoring* feature we need to show that we are able to monitor both (i) system variables and (ii) states of cross-layer optimizations. To validate these features, we selected a set of instructions that were executed while simultaneously running a jamming detection experiment.

```
1 monitor node02 wlan1.cfg80211.signal.avg
2 addChains node02 rssi.cfg
3 monitor node02 maxrssi
4 monitor node02 minrssi
5 run node03 iwconfig wlan1 txpower 20
6 monitor stop node02 wlan1.cfg80211.signal.avg
7 monitor stop node02 minrssi
```

**Listing 3: List of instructions that are conducted successively in the interactive mode.**

With the first line we monitor on `node02` the average signal strength of messages originated at all neighboring nodes. The respective output of our front-end is depicted in Figure 6 which shows reported values (stable between −76 dBm and −77 dBm) for the specified variable. In line two we add the CRAWLER configuration (shown in Listing 4), which includes two cross-layer optimizations. In particular, the instructions in line 2 and 3 compute, over a series of 20 collected values, the maximum and minimum of our monitored RSS variable (as accessed in line 1), respectively. In line 4 and 5 we deliver the corresponding values, that is `maxrssi` and `minrssi`, to the application layer.

```
1 myrssi:get("wlan1.cfg80211.signal.avg")
2 maxrssi:max(maxhistory:history(myrssi,20))
3 minrssi:min(minhistory:history(myrssi,20))
4 appvar1:set("application.mainJammingApp.var1",maxrssi)
5 appvar2:set("application.mainJammingApp.var2",minrssi)
```

**Listing 4: Crawler config that we inserted into the system during experimentation.**

Using the `addChains` command, we inserted these rules into the system. The validation of this step is easily possible with the instructions shown in line three and four of Listing 3 where we used the `monitor` command to monitor both rules. The output of the graphical front-end is depicted in Figure 6 where the middle column shows the `maxrssi` values (area is marked as ③) and the right column the `minrssi` values (area is marked as ④). Note that the newly monitored values are at the top.

To validate that we are able to control third party programs during experimentation we significantly increased the transmission power of the neighboring node `node03` by using the keyword `run` and the program `iwconfig` as shown in line five of Listing 3. We observed that this change improved the received signal strength at `node02`. In particular, `wlan1.cfg80211.signal.avg` significantly increased from −76 dBm to −68 dBm.

If there is no need to continue monitoring a specific variable or optimization, the front-end allows its removal from the display by using the keyword `monitor stop` as shown in line six and seven of Listing 3.

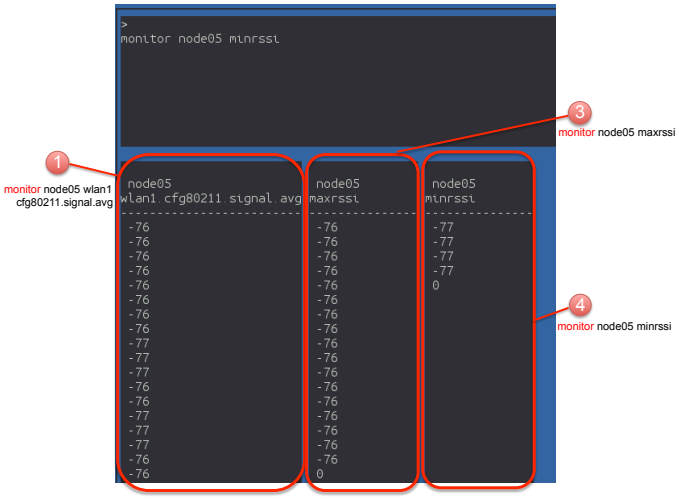


**Figure 6: The interactive mode allows to monitor system variables and cross-layer optimizations by using the keyword `monitor`. Particularly, we monitored on `node02` the system variable `wlan1.cfg80211.signal.avg` and its aggregation, i.e., `maxrssi` and `minrssi`.**

To conclude, in our evaluation we have demonstrated that we are able to automate an experiment with several nodes and programs including cross-layer optimizations. Moreover, during the experiment we showed how to remotely add cross-layer optimizations into the system, run third party programs and monitor different parameters.

## 6. RELATED WORK

The three key features of our framework are remote automation, configuration, and monitoring. In literature there are a few works that offer only one or two of these functionalities.

An example with respect to **remote configuration** is OpenFlow [8] which is a communication protocol that allows remote access to the flow tables of switches and routers. For this, OpenFlow suggests a uniform interface for vendors to allow the configuration of flow tables without revealing internal details of the implementation. The remote devices are controlled via a secure channel using a dedicated controller running on the remote device which allows access only to legitimate peers. However, OpenFlow is primarily focused on high performance flow control and thus limited only to flow table configuration. Hence, it does not allows the configuration of other functionalities besides the network layer, e.g., sensors, protocols, and applications.

A candidate in the field of **remote monitoring** is Ganglia [7] which is a system monitoring tool particularly built for high-performance systems such as clusters and grids. The information distribution of monitored values follows a hierarchical pattern. In particular, servers access the information (called gmond), which is then incorporated into XLM files. These files on the servers can then be accessed via a polling scheme. Intermediate nodes (called gmetad) poll servers or other gmetad and cache these data. Finally, the cached data is accessed by a client for visualization of data. However, the major focus of Ganglia is performance on clusters and its supported set of metrics only consider performance relevant indicators such as CPU and memory utilization. Therefore, neither monitoring of protocols is supported nor cross-layer optimizations.

A similar approach is IETF's standard remote monitoring (RMON) [12] which targets at remote "flow" monitoring in LANs. In its initial version it focused only on monitoring of OSI layers 1 and 2 in Ethernet and Token Ring. Later versions also include the network and application layer. Although RMON has a more general scope in terms of layer support than Ganglia, its concept does not consider other sources of information such as from sensors, transport protocols, other wireless technologies and cross-layer information.

A prominent work regarding **remote automation** is Emulab [4] which targets at combining network simulation, network emulation, and real networks to create a complex testbed. Emulab provides an infrastructure for testbeds including both (real) physical and simulated nodes. It virtualizes nodes and the links between them. It allows to configure and automate test runs, while introducing high cost, restrictions, and complexity to the experimental setup due to additional infrastructure requirements. Emulab targets rather the general case of wireless communication testing. Using Emulab for testing of cross-layer optimizations is not possible. Moreover, the ability of (live) monitoring and logging of variables from system components and protocols to the degree and convenience that we offer is not supported.

To summarize, regarding our three key features, related work only focuses on subsets of the functionality that we provide. For each key feature they target, they focus only on a specific aspect (e.g., performance, layer 1 and 2, etc) which separates their work from ours. Another major distinction is the flexibility and convenience that our approach offers to access information from diverse subcomponents and protocols on remote systems.

## 7. CONCLUSION

We presented a framework that allows to carry over two advantages of simulation to real-world testbeds: (i) centralized, convenient, and flexible access and logging of parameters in devices involved in testing, and (ii) automated execution of experiments.

This is achieved by three complementary key features: First, we propose remote automation which enables the centralized automation and execution of whole test setups with different settings. It only requires to provide a configuration that includes the necessary instructions to automate the experiment. Based on the configuration the necessary commands are parsed and the nodes involved in testing are instructed. Second, the remote configuration feature allows the distribution of cross-layer optimizations which in turn facilitates remote access to devices. Third, the remote monitoring feature allows to log and monitor internal states of cross-layer optimizations as well as variables in protocols and system components. To further support the developer while experimenting, we integrated all three key features into a graphical front-end.

We plan to make the source code of the whole framework available at `http://www.comsys.rwth-aachen.de/research/projects/crawler/`.

## 8. ACKNOWLEDGMENTS

This work was funded by the DFG and the UMIC research cluster of RWTH Aachen University.

## 9. REFERENCES

[1] I. Aktas, F. Schmidt, M. Alizai, T. Drüner, and K. Wehrle. CRAWLER: An Experimentation Platform for System Monitoring and Cross-Layer-Coordination. In *Proc. IEEE WoWMoM*, 2012.

[2] Ath9k - Linux Wireless: Official Website. `http://wireless.kernel.org/en/users/Drivers/ath9k`.

[3] M. Conti, G. Maselli, G. Turi, and S. Giordano. Cross-layering in mobile ad hoc network design. *Computer*, 2004.

[4] Emulab.Net - Emulab - Network Emulation Testbed Home. `http://www.emulab.net/`.

[5] R. S. Gray, D. Kotz, C. Newport, N. Dubrovsky, A. Fiske, J. Liu, C. Masone, S. McGrath, and Y. Yuan. Outdoor experimental comparison of four ad hoc routing algorithms. In *Proc. ACM/IEEE MSWiM*, 2004.

[6] A. Khattab, J. Camp, C. Hunter, P. Murphy, A. Sabharwal, and E. W. Knightly. WARP: A Flexible Platform for Clean-Slate Wireless Medium Access Protocol Design. *ACM SIGMOBILE Mobile Computing and Communications Review*, 2008.

[7] M. L. Massie, B. N. Chun, and D. E. Culler. The Ganglia Distributed Monitoring System: Design, Implementation, and Experience. *Parallel Computing*, 2004.

[8] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. OpenFlow: Enabling Innovation in Campus Networks. *ACM SIGCOMM Computer Communication Review*, 2008.

[9] NCurses (new curses) library. `http://www.gnu.org/software/ncurses/ncurses.html`.

[10] O. Puñal, I. Aktaş, C.-J. Schnelke, G. Abidin, J. Gross, and K. Wehrle. Machine learning-based jamming detection for IEEE 802.11: Design and experimental evaluation. In *Proc. IEEE WoWMoM*, 2014.

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

[12] S. Waldbusser. RFC 4502 - Remote Network Monitoring Management Information Base Version 2, May 2006.