Towards a Flexible and Versatile Cross-Layer-Coordination Architecture

Ismet Aktas, Jens Otten, Florian Schmidt, Klaus Wehrle
Distributed Systems Group
RWTH Aachen University, Germany
Email: {ismet.aktas, jens.otten, florian.schmidt, klaus.wehrle}@rwth-aachen.de

***Abstract*—In wireless and mobile networking, volatile environmental conditions are a permanent challenge, resulting in a demand for cross-layer optimizations. To further increase flexibility, we believe cross-layer architectures should adapt themselves to these changing conditions, just as they adapt the network stack, devices, and applications.**

In this paper, we propose CRAWLER, a novel cross-layer architecture that combines three core properties: signaling between all layers and system components; extensibility and adaptability at runtime; and high usability for cross-layer developers. CRAWLER increases flexibility, as well as expediting and simplifying cross-layer development.

I. Introduction

Traditional network protocol stacks are logically organized in layers. These layers are strictly separated and the cooperation between them is restricted by concise interfaces, which in effect only allow passing packets up and down the stack. In principle, all these layers have been designed to fulfill their functionality without interaction across the layers. History shows that this works well in wired and static environments.

However, today's and upcoming networks consist of wireless links and highly mobile nodes. In order to adapt to the rapidly and frequently changing network conditions under those circumstances, a more sophisticated interaction between protocols than in a traditional layered architecture is desirable. For example, a VoIP application could adapt the codec rate according to the signal quality. The exchange of information across layers, i.e., the cross-layer design paradigm ([4], [9], [10]) is a promising concept to deal with these challenges.

Unfortunately, present architectures have limitations in their flexibility. For example, if we reconsider the VoIP application example above, the codec rate adaptation could not only be based on the signal strength but also on user feedback or further protocol information. Current solutions are not able to dynamically change which of these to use and how to use them, i.e., the adaptation, re-parametrization and addition of cross-layer optimizations during runtime. Moreover, customization of optimizations in existing frameworks is often cumbersome and complicated, if it can be done at all. We envision a system that allows a wider user base, e.g., application developers acting as cross-layer designers (termed in short "designers" from here on), to use a framework to easily develop cross-layer optimizations tailored to their applications' specific needs. Applications can be bundled with an optimization setup that is dynamically added to the framework for as long as they run.

We therefore propose a **cr**oss-layer **a**rchitecture for **w**ire**le**ss netwo**r**ks (CRAWLER) with the following properties:

- Signaling between an arbitrary amount of layers and system components;
- Extensibility of the architecture and adaptability of optimizations at runtime; and
- High usability for cross-layer developers via an abstract description language for optimization rules.

The remainder of this paper is organized as follows. In Section II we shortly discuss related architectures. Section III presents CRAWLER, our proposed cross-layer architecture. Implementation details are given in Section IV. In Section V we present results for a real cross-layer use case utilizing CRAWLER. Finally, we conclude with an outlook in Section VI.

II. Related Work

A plethora of specific cross-layer solutions exists for particular problems, which can mostly be characterized as quick hacks, as compared to full-fledged architectures. Several surveys have been conducted on such solutions [1], [6]. A good overview about design principles and concepts for cross-layer architectures is given in Srivastava et. al. [10]. However, most suggested implementations are only able to signal between two specific layers or are only able to signal in one direction, e.g., from lower layers to upper layers but not vice versa.

In Physical Media Independence (PMI) [3] only device information is propagated layer-by-layer to the upper layers. The opposite direction is not possible. Furthermore, each layer requires a specific adaptation to process signaling information. In [11] ICMP messages are utilized to provide higher layers feedback from lower layers. Nevertheless, higher layers are not able to provide information to lower layers. Again, any-to-any layer signaling is not possible. In [14] an inter-layer signaling pipe (ISP) is suggested. It utilizes packet headers to provide cross-layer feedback from upper layers to lower layers.

There also exist several cross-layer architectures facilitating signaling across all layers, i.e., any-to-any layer signaling. For example, Cross-Layer Signaling Shortcuts (CLASS) [13] enables direct signaling between all layers by message passing. However, any-to-many signaling, i.e., addressing several layers at once, is not possible.

The shared database approach CATS [8] provides a management plane which contains a so-called cross-layer platform.

Protocol information from all layers are available in the platform and are accessible from any layer. However, due to the monolithic architecture, extensibility to new protocols or additional optimizations is difficult, and impossible at runtime.

Mobile Metropolitan Ad-hoc Network (MobileMAN) [2] is also a shared database approach but compared to CATS it introduces an interface to create, read, write and monitor protocol information available in the database. Each layer can store protocol information and make it accessible to other layers. MobileMAN requires to exchange a layer with its redesigned extension to enable the cooperation with the database. In our viewpoint, this is too much intrusion into the network stack resulting in too much dependency and limiting maintainability.

In contrast, ECLAIR [7] solves this issue by introducing so-called Tuning-Layers (TLs). A TL's main task is to provide a platform specific interface from the TL to the protocol stack in order to enable read and write requests to protocol information. Due to the introduction of a generic interface between the architecture and the TL, platform independence is also considered. However, ECLAIR does not support adaptability or extensibility of cross-layer optimizations during runtime.

In comparison, CRAWLER offers any-to-(m)any layer or system component optimizations while allowing extensibility and adaptability during runtime. For this purpose, CRAWLER provides the feature to easily add, remove, reparametrize or adapt cross-layer signaling optimizations. Moreover, by providing an easily usable but powerful configuration language, CRAWLER enables a designer to program and handle cross-layer signaling optimization very flexibly. For example, if a cross-layer designer observes a performance degradation with the already established cross-layer signaling, the designer can reconfigure a new cross-layer signaling or can disable the current one.

III. CRAWLER Design

On an abstract level CRAWLER consists of three components as shown in Figure 1: (1) the logical component (LC) allows designers to express their cross-layer signaling optimizations in a very abstract and intuitive way. For this purpose, we have created a rule-based language customized to cross-layer design purposes. As a result, a designer is able to program cross-layer signaling at a high level by specifying rules (even at runtime). (2) The proper realization of the cross-layer signaling optimizations (given by the LC) is realized by the cross-layer processing component (CPC). Here, the rules are mapped to compositions of small functional units. These compositions can be flexibly changed. (3) Finally, stubs provide (read or write) access to protocol information or sub-system states. In the following, the three components will be discussed.

A. Logical Component

The LC interacts as an abstraction interface between designers and the CPC. Its major goal is to increase the usability of the architecture for designers, allowing them to easily express their desired optimizations without paying too much attention

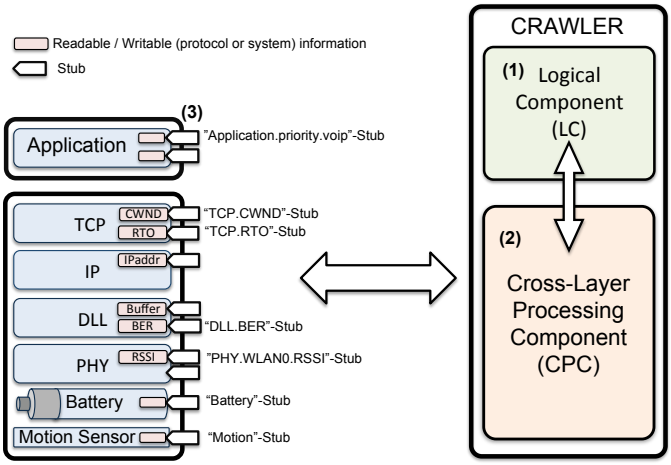

Fig. 1. Conceptual view of CRAWLER's components: (1) the logical component manages cross-layer optimizations via rule-based configuration, (2) the processing component realizes the optimization given by the logical component and (3) stubs provide the access to protocol or system variables.

to implementation details. For that purpose, the LC is divided into four subcomponents. The *configuration* subcomponent enables a designer to express cross-layer optimizations on an abstract level. The *repository* handles configuration setups. The idea of the *application support* subcomponent is to allow applications to share their variables for cross-layer optimizations. Finally, the *interpreter* translates a configuration into a composition of functions which the CPC provides. The four subcomponents will be explained in the following.

1) Configuration: The configuration subcomponent of CRAWLER offers an easily usable but powerful configuration language. The internal realization of the cross-layer signaling is abstracted to a simple declarative configuration. In order to provide such an abstract configuration description, we have created a language based on so-called *rules*. Simply put, rules can be considered as a behavioral description of cross-layer interactions. In Listing 1 we show an example configuration utilizing rules to describe how to access a protocol information, process it and notify an application. Each line of the configuration is a rule. Figure 2 shows a 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 simply specifies which parameter, determined by a unique fully qualified name, should be accessed (how the access itself works will be explained in Section III-C). The second rule, `History` (which saves a certain number, here 4, of collected values), is performed on the evaluation of the first rule. Similarly, rule three evaluates if the average of the evaluation of rule two is below a certain threshold (here 55). In our syntax, rules can be nested within other rules to form rule chains.

So far, we have seen how computations and conditions can be specified using the configuration language. However, sometimes it is desirable to react to events, such as a sudden drop in signal strength. This triggering is denoted by an arrow like in rules 4 and 5. The link quality condition of rule 3 is used

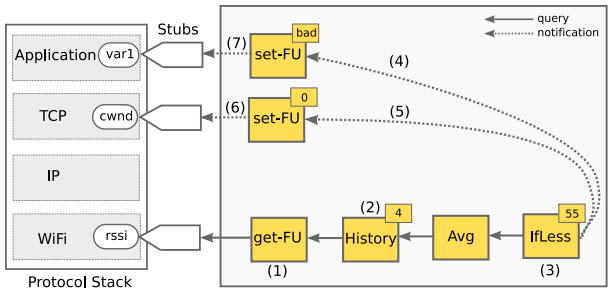

Fig. 2. A simple cross-layer signaling configuration in CRAWLER. We change behavior of the TCP layer and an application based on wireless signal strength.

to inform an application about bad link quality, and to reduce the congestion window of TCP connections to 0, to avoid triggering its congestion avoidance due to data corruption.

The configuration subcomponent provides a flexible adaptation of parameters or whole rules, e.g., a rule can be added to the configuration during runtime. CRAWLER recognizes the change in the configuration and adapts the internal realization of cross-layer optimizations. For example, if we want to change the signal strength threshold at which to react, we only change rule 3, and the framework adapts the rule chain accordingly. This intuitive approach, combined with the abstraction provided by the configuration language, fulfills our design goal of accessibility for designers who are not cross-layer experts. However, as already mentioned, the configuration is only an abstract description of cross-layer interactions that need to be realized. For a better management of the configuration, we have designed the repository subcomponent.

2) Repository: The repository keeps track of all changes that are made to the configuration. It saves configuration setups and therefore facilitates changing between several profiles, or rollback to an earlier setup in case of misconfiguration by the designer. As the name suggests, it behaves similar to a revision control system.

3) Application support: Applications can also utilize cross-layer information for optimizations as described in the VoIP example above. Therefore, the application support subcomponent enables applications to share and retrieve cross-layer information via a shared library. Thus, applications can simply register their variables, e.g., "app.voip.maxdelay". The library takes care of how to access these variables from within CRAWLER. Furthermore, in a later state of the architecture we want to allow applications to bring their own rules (configuration). Thus, each application should be allowed to bring their own optimizations.

4) Interpreter: The LC needs to provide the configuration to the CPC. For this purpose, the configuration is parsed and rules are mapped to specific fine granular instructions termed *commands* which are passed to the CPC. Commands include the references to functions that need to be called by the CPC. The handling of commands and the proper realization of cross-layer interactions will be explained in the next section.

```
1 my_rssi:get("phy.wlan0.rssi")
2 my_history_of_rssi:History(my_rssi, 4)
3 my_rssi_is_Bad:IfLess(Avg(my_history_of_rssi),55)
4 my_rssi_is_Bad-->my_application_var1
5 my_rssi_is_Bad-->my_TCP_Freezer
6 my_appl_var1:set("appl_voip1_var1","bad")
7 my_TCP_Freezer:set("tcp.cwnd","0")
```

Listing 1. A simple cross-layer signaling configuration in CRAWLER. This configuration file defines the setup illustrated in Figure 2.

B. Cross-Layer Processing Component

The cross-layer processing component (CPC) executes the commands received from the LC. In the configuration above, we have seen that rules contain calls to and compositions of *functional units* (FUs) such as `History` or `Avg` (average). As opposed to typical functions, FUs are stateful. For example, every instance of `History` keeps its collected values between calls. Furthermore, if a configuration changes the composition of FUs, but does not create or delete FUs themselves, they will keep their current state and collected information. This serves as one important property towards dynamic reconfigurability and adaptability of cross-layer optimizations at runtime. This is further supported by FUs sharing a uniform interface so that they can be flexibly interconnected with each other. For example, by changing rule 3 in Listing 1, we can exchange the `Avg` FU in Figure 4 with `Min` at runtime due to the uniform interface, and still use the collected data from `History`, because a change in the composition does not reinstantiate all FUs. Finally, the uniform interface facilitates easy extension of the set of available FUs, because at least syntactically, newly designed FUs will be able to interconnect with every other FU previously created.

As previously hinted at, we provide two ways to compose FUs, via queries and via events. The two ways and their applications will be explained below.

1) Query-based Signaling: The query interface allows to explicitly request information. If the query interface of a FU is called, the FU executes and returns the result to the inquiring FU. The result of a FU may depend on the result of further FUs, leading to cascading queries. In case of providing the most up-to-date values, this is the intended behavior. However, to reduce the computational overhead, each FU can cache its already calculated return value and maintain a validity time for it. In case of a new incoming query, the FU can then decide to return the cached value immediately if it still considers it valid.

Figure 3 describes the concept of query-based signaling. (1) A FU `Pre` queries the FU `Func`. `Func` checks the validity for its return value (2). Since the validity time for its return value has expired, (3) `Func` queries `Succ`. Based on the return value of `Succ`, `Func` computes its own new return value and updates its validity time correspondingly. Finally, (4) `Func` sends the result to the predecessor FU `Pre`.

An example composition of FUs utilizing the query interface is shown in Figure 4. `IfLess` queries `Avg` for an average of RSSI values, provided by `History`.

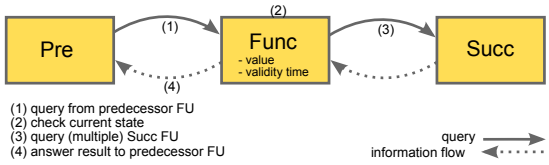

Fig. 3. Query-based signaling

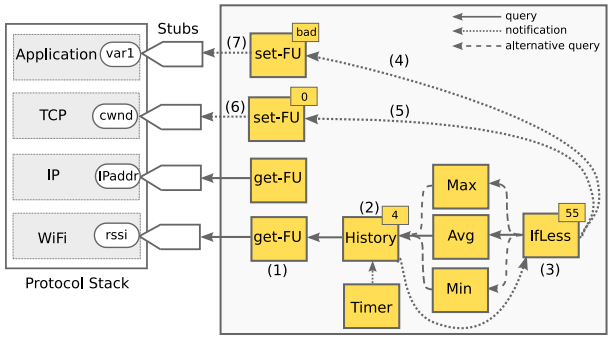

Fig. 4. A slightly extended version of our first simple example.

2) Event-based Signaling: The query-based interface for compositions between FUs results in a polling architecture. To avoid unnecessary polling, we introduce event-based signaling. It is used to notify a FU about value changes measured by another FU. The notified FU can then act based on that triggering. Triggers are defined in the configuration, as in rules 4 and 5 in Listing 1.

Figure 5 describes the concept of event-based signaling. In (1), `Notifier` triggers `Func`. This can be due to an elapsed timer or a measured change in a monitored value. (2) `Func` can now decide to act on that notification, e.g., do a calculation, (3) query its successor FU, or (4) trigger other FUs.

As an example, in Figure 4, the `Timer` FU acts as a trigger to periodically notify the `History` FU, which then takes RSSI samples to save in its log and provide to querying FUs. Furthermore, `IfLess` triggers two stubs (more on stubs in the next section) to set values in the TCP layer and the application.

C. Stubs - Accessing Signaling Information

To access, i.e., read and write, protocol or system information, CRAWLER provides *stubs*. They act as a glue element between the cross-layer optimizations and the data that is accessed. Each piece of data is associated with a stub, so that we again, as in the case of FUs, have opted for a fine granular solution that is easily extensible because stub implementations do not affect each other.

There are several examples of stubs in Figure 4. The CPC encapsulates the data gathered from stubs within a special set of two FUs, `set` and `get`. (In cases where writing values does not make sense, e.g., sensors only providing readable values, stubs with only `get` functionality can be used.) The protocol information or system information is then available within the CPC and all FUs can call `get` or `set` like any other FU. In rule 7 of Listing 1, the `set` FU manipulates protocol

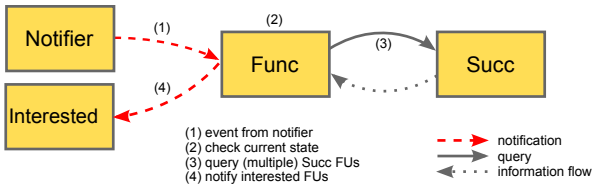

Fig. 5. Event-based signaling

information. Hence, for every protocol information or system information that we want to access, the desired information needs to be identified clearly (compare rule 1 in Listing 1). Thus, to access the desired protocol or system variable, stubs have fully qualified names, i.e., unique and hierarchical names.

Protocol information often changes non-periodically. Because a stub is a FU, it can use the notification interface to notify other FUs about changed protocol information. Not only does this reduce the overhead because polling of protocol information is not necessary, it also increases the responsiveness of rules to changing conditions.

IV. Implementation

We are in the process of implementing CRAWLER on a Linux system. At this point in time, most implementation work on the framework is done, to the point where we can create sets of rules that create simple cross-layer optimizations.

The LC and all its subcomponents are implemented in C++. The LC is running as a daemon in user space and provides a shared library for applications to read and share variables for cross-layer optimizations. For inter-process communication we use Unix Domain Sockets. The CPC is resides in kernel space and is therefore implemented in C. For communication between LC and CPC we use netlink sockets.

The uniform interface between FUs is implemented via a special data type that can contain characters, integers, arrays of any type and struct-like compounds of those types. This facilitates syntactical composability of all FUs. Each FU can then act accordingly to the received type, i.e., slightly different behavior depending on whether it receives a single value or an array of values. For our first evaluation steps, we have so far implemented about 10 FUs and 50 stubs, with the numbers growing with every new testing setup.

V. Evaluation

To give a demonstration of the implementation of our architecture, we created the example that was used in earlier Sections (cf. Figure 2) of this paper in CRAWLER. Our test setup consisted of two PCs. One was running our CRAWLER implementation, and equipped with a WLAN network card. Throughout our test, we ran iperf [12] to constantly create TCP traffic. Furthermore, we used the application Skype, which provides a scriptable API [5], and created a shell script to start and stop a UDP voice stream during the test, which acted as the application endpoint of our cross-layer setup. The other PC was connected to a WLAN access point via Ethernet and served as the other endpoint of our iperf setup.

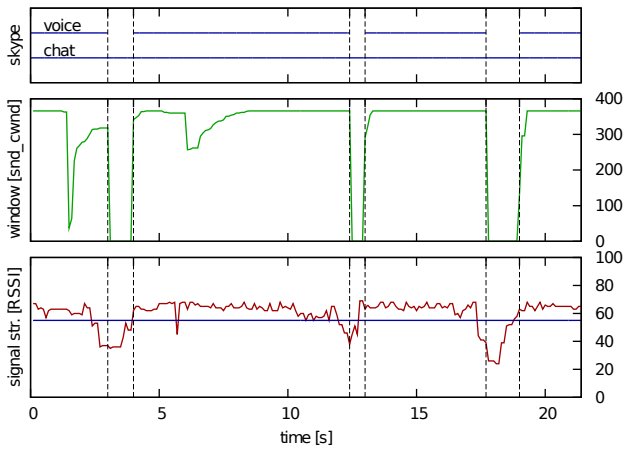

Fig. 6. Results from our testing setup. From bottom up: (1) The measured RSSI and the threshold value of 55. (2) The congestion window, which (on three occasions) is set to 0 when the recent RSSI average is less than the threshold, and re-released when the average rises again. (3) On top, Skype chat is always available, VoIP is deactivated at low RSSI.

We used a stub to read the RSSI value. The `History` FU that read the value was triggered every 100 ms by a `Timer`, and kept the last 4 values. `IfLess` monitored the values, and if the average of these dropped below a certain threshold (in our case 55), it triggered two notifications. The congestion window of ongoing TCP connections was set to 0 (by setting the variable `snd_cwnd` in `tcp_sock` in the Linux kernel), and Skype was notified to stop its VoIP stream, leaving only text chat. If the average rose above 55 again, the congestion window was reset to the previous value, and Skype was allowed to use VoIP again.

In Figure 6 three marked areas indicate the times that the notifications took place, roughly between 3 and 4 seconds, 12 and 13 seconds, and 17 and 18 seconds. The drop of the congestion windows to 0, and its re-release are clearly visible. Furthermore, the uppermost part of the figure visualizes how our signaling stopped and reactivated voice communication at the same time.

VI. Conclusion and Outlook

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, communication sub-systems, respective system components and the coordination of these components during runtime. This has been achieved by the following three components: (1) The logical component provides high usability for cross-layer designers. In particular, we have designed a rule-based configuration language which facilitates an abstract declarative programming of cross-layer optimizations. (2) The cross-layer processing component enables the coordination between an arbitrary amount of layers and system components. It creates an extensible and adaptable framework through reusable and freely composable functional units. (3) stubs allow access to protocol or system information with little infiltration into the protocol stack, which increases the flexibility of the framework to changes in the underlying network stack.

Our Linux implementation of CRAWLER is at a point where simple signaling setups are possible. Most functionality is available. However, several dynamic features are still being developed, such as loading additional FU types during runtime, and recomposition of FUs. These missing features are currently under development.

As a next step, to further simplify the use of our framework for application developers, we plan to design an assistant system that will check rules themselves, and their interaction between each other. This way, we could find dependencies, optimize rules, and prevent variable oscillation, where rules constantly change values and trigger each other. This will be even more important once application can bring their own FUs with them and insert them into CRAWLER during runtime.

We also want to further increase the usability of CRAWLER by providing a visual configuration and monitoring component. The visualization support for monitoring of cross-layer interactions will provide several advantages such as observation of cross-layer interactions over several rules and the ensuing effects.

In conclusion, we intend to provide a tool that will be able to unify most existing isolated cross-layer optimization in one single tool, and expedite the development of novel cross-layer optimizations. We anticipate CRAWLER to be utilizable in diverse research fields such as energy consumption, QoS and routing.

Acknowledgments

Many thanks to Tobias Drüner, Raimondas Sasnauskas, Jó Ágila Bitsch Link, Cem Mengi, and Elias Weingärtner for many fruitful discussions. This work was supported by the UMIC research cluster of RWTH Aachen University.

References

[1] G. Carneiro, J. Ruela, M. Ricardo, and I. Porto. Cross-layer design in 4G wireless terminals. *IEEE Wireless Comm.*, 11(2):7–13, 2004.
[2] M. Conti, G. Maselli, G. Turi, and S. Giordano. Cross-layering in mobile ad hoc network design. *Computer*, 37(2):48–51, 2004.
[3] J. Inouye, J. Binkley, and J. Walpole. Dynamic network reconfiguration support for mobile computers. In *Proc. MobiCom*, pages 13–22. ACM New York, NY, USA, 1997.
[4] X. Lin, N. Shroff, and R. Srikant. A tutorial on cross-layer optimization in wireless networks. *IEEE Journal on Selected Areas in Communications*, 24(8):1452–1463, 2006.
[5] S. Ltd. Skype Public API. *URL: https://developer.skype.com/*, 2008.
[6] V. Raisinghani and S. Iyer. Cross-layer design optimizations in wireless protocol stacks. *Computer Communications*, 27(8):720–724, 2004.
[7] V. Raisinghani and S. Iyer. ECLAIR: An efficient cross layer architecture for wireless protocol stacks. In *Proc. World Wireless Congress*, 2004.
[8] C. Sadler, L. Kant, and W. Chen. Cross-layer self-healing mechanisms in wireless networks. In *Proc. WWC*, volume 254, 2005.
[9] S. Shakkottai, T. Rappaport, and P. Karlsson. Cross-layer design for wireless networks. *IEEE Comm. Magazine*, 41(10):74–80, 2003.
[10] V. Srivastava and M. Motani. Cross-layer design: a survey and the road ahead. *IEEE Communications Magazine*, 43(12):112–119, 2005.
[11] P. Sudame and B. Badrinath. On providing support for protocol adaptation in mobile wireless networks. *MONET*, 6(1):43–55, 2001.
[12] A. Tirumala, F. Qin, J. Dugan, J. Ferguson, and K. Gibbs. Iperf: The TCP/UDP bandwidth measurement tool. *URL: http://dast. nlanr. net/Projects/Iperf*, 2004.
[13] Q. Wang and M. Abu-Rgheff. Cross-layer signalling for next-generation wireless systems. In *IEEE WCNC*, volume 2, pages 1084–89, 2003.
[14] G. Wu, Y. Bai, J. Lai, and A. Ogielski. Interactions between TCP and RLP in Wireless Internet. In *Proc. GLOBECOM*, pages 661–666, 1999.