Architecting Protocol Stack Optimizations on Mobile Devices

*Vijay T. Raisinghani**
Tata Infotech Ltd.(ATG) and
School of Information Technology
IIT Bombay
rvijay@it.iitb.ac.in

Sridhar Iyer
School of Information Technology
IIT Bombay
sri@it.iitb.ac.in

Abstract— **Applications using traditional protocol stacks (e.g TCP/IP) from wired networks do not function ef£ciently in mobile wireless scenarios. This is primarily due to the layered *architecture* and *implementation* of protocol stacks.**

Cross layer feedback is one of the mechanisms to improve the performance of a layered stack, in mobile wireless environments. For example, transport layer retransmissions could be reduced by making it aware of network disconnections or handoff events. However, since the protocol stack is an integral part of the operating system, any such cross layer modi£cation to the stack should not impact its *ef£ciency*, *correctness* and *maintainability*. An appropriate architecture would help ensure that cross layer modi£cations con£rm to this requirement.

In this paper, we present an architecture *ECLAIR* for cross layer feedback. As compared to other approaches, ECLAIR requires minimal or no modi£cation to the stack.

To evaluate the ef£ciency of cross layer architectures, we identify *time/space complexity*, *design complexity*, *user-kernel crossing* and *data path delay* as performance metrics. We validate and evaluate ECLAIR through a prototype implementation and experiments. Our results and analysis show that ECLAIR is an ef£cient cross layer architecture. To enhance the ef£ciency of ECLAIR, we propose a sub-architecture to reduce the runtime overheads.

We also present a design guide for cross layer optimizations using ECLAIR.

I. INTRODUCTION

To ensure interoperability with the existing Internet, standard protocol stacks (e.g. Transmission Control Protocol(TCP)/Internet Protocol(IP) [14]) are being deployed even in mobile wireless setups i.e. on mobile devices and intermediate nodes in the wireless network. However, these standard protocol stacks function inef£ciently in mobile wireless environments. This is primarily due to the layered *architecture* [6] and *implementation* of protocol stacks. For example, TCP reduces its throughput in wireless environments since it misinterprets wireless packet losses as congestion losses [1], [19].

It is evident from above that adapting TCP behavior, based on wireless channel information from lower layers, would improve its throughput. In a similar manner the performance of other layers in the protocol stack can be improved by enabling such cross layer optimizations [11]. The cross layer feedback to a layer could be from layers above or below it. Cross layer

optimizations may implemented at the intermediate nodes [1], [17] or mobile devices. We focus on cross layer feedback in the mobile device since we believe that it would be easier to implement changes on end-devices than in the network.

As new wireless networks are deployed, various cross layer optimizations would be required to enhance the performance of existing protocol stacks. However, if these optimizations are implemented in an *ad hoc* manner it could lead to (1) decreased throughput of the stack, (2) dif£culties in ensuring protocol correctness and (3) substantial effort in maintenance of the cross layer optimizations. By *ad hoc* we mean introduction of additional code in a layer's code, to implement a cross layer optimization. For example, to allow TCP to get hand-off information from the link layer, additional code will be introduced in TCP and link layer. This would reduce the protocol layer throughput since it will now have to execute the cross layer code also. Further, it would be dif£cult to ensure protocol correctness as additional cross layer code is introduced. An appropriate architecture would help ensure that cross layer modi£cations do not impact the *ef£ciency*, *correctness* and *maintainability* of the protocol stack.

In this paper we present the internal details of our architecture ECLAIR (section II). ECLAIR [12] exploits the fact that protocol behavior is determined by the values stored in the protocol data-structures. In ECLAIR, a *Tuning Layer* (TL), one for each layer, provides an interface to manipulate and monitor these protocol data-structures. The TLs are used by the *Protocol Optimizers* (POs) which contain the cross layer feedback algorithms. The POs constitute the *Optimizing SubSystem* (OSS). The POs take input from various layers and determine appropriate adaptations. A PO uses one or more TLs for collecting information and effecting adaptation. The key bene£ts of ECLAIR are as follows: enables *rapid prototyping* of new cross layer algorithms, ensures *minimum intrusion* into the existing stack, facilitates easy *portability* to different systems, and imposes *minimal overhead* on the existing stack [12].

We validate ECLAIR by implementing RWC [13] and running experiments over wireless LAN (section III). We compare this ECLAIR implementation with a non-ECLAIR RWC implementation [10] (section IV). Through this comparison we show that ECLAIR is an ef£cient cross layer architecture. For comparing the implementations we select the

---

following metrics: *time/space complexity*[1], *design complexity*[2], *user-kernel crossing*[3] and *data path delay*[4].

To minimize the overhead of an ECLAIR cross layer implementation we propose a sub-architecture (section V). The idea is to create a separate set – *core* – of selected *high utility* cross layer information.

To ensure *correct* and *efficient* cross layer feedback, the *type* of cross layer feedback should be known. We present a design guide for cross layer design using ECLAIR in section VI. We broadly classify cross layer feedback into *synchronous* or *asynchronous* i.e. the adaptive action at a layer based on cross layer feedback from another layer may be *synchronous* or *asynchronous*. In synchronous cross layer feedback, whenever a layer receives some cross layer information, it proceeds with its *regular* execution only after executing the cross layer adaptation required. For example, assume there is *network disconnection* event sent to TCP from the link layer. In the synchronous case, TCP's regular execution is stopped, appropriate adaptation is carried out in TCP and then regular TCP execution proceeds. In the asynchronous case, the control data structures of TCP would be updated while TCP's execution is in progress. We discuss the application of ECLAIR to *asynchronous* and *synchronous* cross layer feedback. We also provide guidelines for ECLAIR implementations of single and multiple cross layer optimizations.

In section VII we present related work. We conclude the paper and discuss future work in section VIII.

II. ECLAIR DETAILS

A. ECLAIR Overview

ECLAIR is split into two subsystems – *Tuning Layers* and *Optimizing SubSystem* (figure 1). ECLAIR enables *rapid prototyping* of new cross layer feedback algorithms since its components are outside the existing protocol stack.

Tuning Layers (TLs): The purpose of a tuning layer is to provide an interface to the protocol data-structures. For example, TCP tuning layer (TCPTL) is provided for TCP. Since the functionality for manipulating protocol data-structures is built in to the TLs, minimal modification is required to the existing protocol stack. This facilitates incorporation of new cross layer feedback algorithms with *minimum intrusion*. Further, for the purpose of *portability* each TL is subdivided in to a *generic* and an *implementation specific sublayer*. This subdivision is essential since the implementation of protocols is different across systems, even though they confirm to the protocol standards. For example, the TCP data-structure names are different in NetBSD and Linux. For ease of reference we group the tuning layers according to their function. For example, *Transport Tuning Layer* refers to the collection of transport protocol tuning layers such as *TCPTL* for TCP, *UDPTL* for UDP, etc.

---

[1]Time/space complexity is a measure of the computing resources required by the system.
[2]Design complexity is a measure of the design effort.
[3]User-kernel crossing is a measure of the time delay due to function calls from user space to kernel space.
[4]Data path delay is the delay introduced in stack's existing call flow.

Optimizing SubSystem (OSS): The optimizing subsystem contains the algorithms and data-structures for cross layer optimizations. The OSS contains many *Protocol Optimizers* (POs). A PO contains the algorithm for a particular cross layer optimization. The OSS interaction with the TLs is shown in figure 1. The solid line with a solid arrow head indicates an *optimizing action* by a PO i.e. a PO invokes a function in the TL for modifying protocol behavior. The dashed line with hollow arrow head indicates that a PO registers for events with a TL and that TL notifies the registered PO whenever an event occurs. For example, a *TCP-handoff* PO would register with the MAC-TL for hand-off events such as *handoff-start*, *handoff-end*, etc. This PO would contain the algorithm to appropriately manipulate TCP behavior whenever hand-off events occur.

The OSS executes concurrently with the existing protocol stack and does not increase the stack processing overhead.
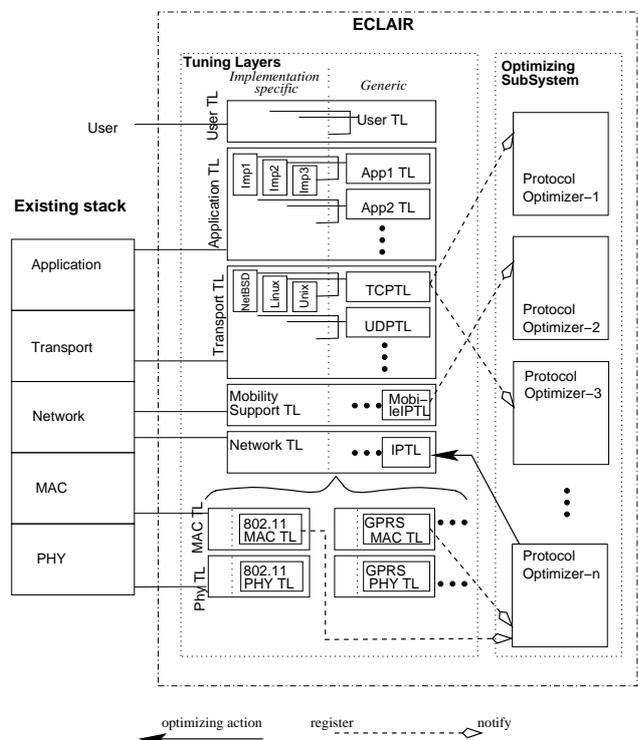

Fig. 1. ECLAIR: Cross Layer Feedback Architecture

Besides the benefits indicated earlier, other salient features of ECLAIR are as follows:

(a) *Cross layer feedback switch*: Since the cross layer system is separate, it can be easily/dynamically enabled or disabled. Also, individual POs may be enabled or disabled.
(b) *Enables seamless mobility*: A seamless mobility PO could monitor multiple wireless interfaces and dynamically switch to the interface that has a better signal strength.
(c) *User Tuning Layer*: Besides the layer specific TLs, ECLAIR also has a *User Tuning Layer* (UTL). UTL allows a device user or an external entity (e.g.: a distributed algorithm or a base station) to tune the device behavior.

B. ECLAIR Details

TL interface to protocol stack:

A tuning layer reads from / writes to the data-structures of a protocol. A protocol implementation typically has data-structures for *control* and *data*. A protocol's behavior is determined by its control data-structures. For example, in Linux, TCP control information is stored in a data-structure `struct tcp_opt` embedded within the socket data-structure `struct sock` [4]. Some of the fields in `tcp_opt` are *retransmission time out* `rto`, *smoothed round trip time* `srtt`, *maximal window to advertise* `window_clamp` and *slow start threshold* `snd_ssthresh`. These fields are read and written to at various points in the TCP code. The values in these and other fields determines TCP behavior. For example, TCP retransmits packets when the `rto` timer expires. The value in `tcp_opt.rto` is used for setting TCP's retransmission time out. Tuning layers are aware of such implementation details of protocol data-structures. TLs manipulate the values in the protocol's control data-structures for modifying protocol behavior. TLs also monitor the protocol data-structures for monitoring events within a layer.

TL and OSS interface:

Tuning layers export an *application programming interface* (API) to the protocol optimizers (POs) in the Optimizing SubSystem. A PO uses a TL's *register* API to register with the TL, for information about *events* at a layer. Multiple POs can register for the same event with a TL. A PO decides the *optimizing action* to be taken based on the *event* information it receives from the TLs with which it has registered. The PO also uses TL APIs for querying the current *state* of the protocol layer which is to be modified (e.g.: TCP's state could be *congestion avoidance* or *slow start* phase). The PO modifies the target protocol's behavior by invoking the appropriate API of the protocol's TL with the necessary parameters. The TL ensures that the correct field in the protocol's data-structure is updated or the appropriate function in the operating system is invoked such that the target protocol's behavior is modified.

The POs invoke the *generic* sublayer within a TL. The generic sublayer in turn invokes the *implementation specific sublayer* APIs for system specific actions. For example, for tuning TCP receiver window, the generic tuning sublayer API is `set_recv_win()`. The corresponding implementation specific API for Linux is `linux_set_recv_win()`. Some additional APIs are presented in [12].

In this section we presented the internal details of ECLAIR. In the next section we validate ECLAIR through a prototype implementation.

III. ECLAIR VALIDATION

To validate ECLAIR we choose Receiver Window Control (RWC) [13] as the candidate cross layer optimization.

A. Receiver Window Control

Users can provide useful feedback to improve the performance of the stack or the *user experience*[13]. For example, a user may want a file download to complete faster than another simultaneous download on the device.

One method of controlling an application's bandwidth share, on a *receiving* device, is through manipulation of the *receiver window* of its TCP connection i.e. *Receiver Window Control* [10], [13]. TCP uses congestion and flow control mechanisms to avoid swamping the network or the receiver [9], [14]. The receiver reflects its receive buffer status by the *advertised window* field in the acknowledgments to the sender. If the advertised window decreases, the sender also reduces its send rate. This TCP behavior can be exploited to reduce the throughput of some applications and consequently increase throughput of rest of the applications, on the receiver.

B. Receiver Window Control Design

The design of *Receiver Window Control* (RWC) using ECLAIR is shown in figure 2. The *RWC PO* contains the algorithm. This PO calculates an application's receiver window based on the priority assigned to it by the user [13].

The explanation of the sequence shown in figure 2 is as follows: **(1)** TCPTL gets TCP's control data-structure memory location information, at system start. **(2a)** PO registers for *user* events. **(2b)** User changes priorities for running applications. **(3)** Application identification and respective priority information is passed to the RWC PO. **(4a),(4b)** RWC PO collects current receiver window/buffer information via TCPTL. It uses this information to recalculate the new receiver window values for the various applications. We assume that the applications can be identified by their sockets. **(5a),(5b)** RWC PO sets the receiver window values for each application via TCPTL.
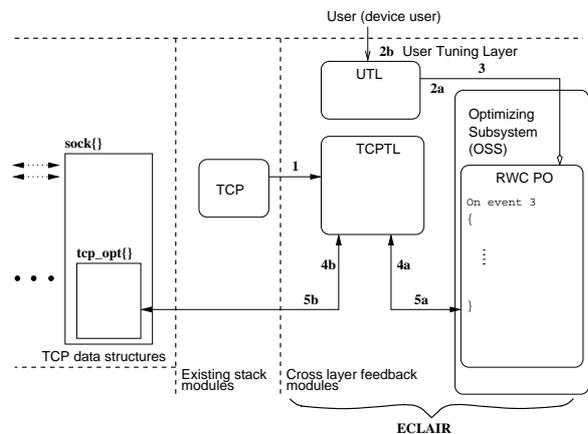

Fig. 2. ECLAIR architecture: Receiver Window Control

C. Receiver Window Control Implementation (Linux)

First we present some internal details of the Linux TCP/IP kernel which are relevant for Receiver Window Control implementation. The relevant TCP data-structures are in the header file `sock.h`. `tcp_opt` is TCP's control data-structure. `sock` is the *socket* data-structure. `window_clamp` and `rcv_ssthresh` are used for controlling the advertised window in TCP. For browsing the Linux kernel code, we used source code browsing tools such as cscope [22], cbrowser [20] and the Linux Cross Reference website [21].
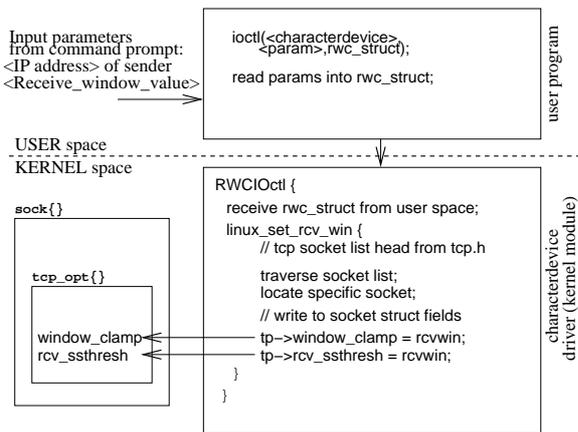
Fig. 3. Call flow: RWC using ECLAIR

Figure 3 shows the call flow of the RWC prototype implementation using ECLAIR. Our current implementation has largely TL functionality only. In this prototype the RWC *calculation* is done by the user. The user runs a program in user space with the parameters: IP address of the sender (to identify the application) and the receiver window value. These parameters are passed to the TL, in `rwc_struct` via `ioctl`, to change the control parameters (receiver window) in the socket. The *IP address* parameter is used to identify the application's TCP socket within which the receiver window value is to be changed. We implemented the receiver window control TL as a Linux kernel loadable module [3]. No modification was required to the TCP code in the kernel.

Using the above implementation we conducted the experiments over WLAN.
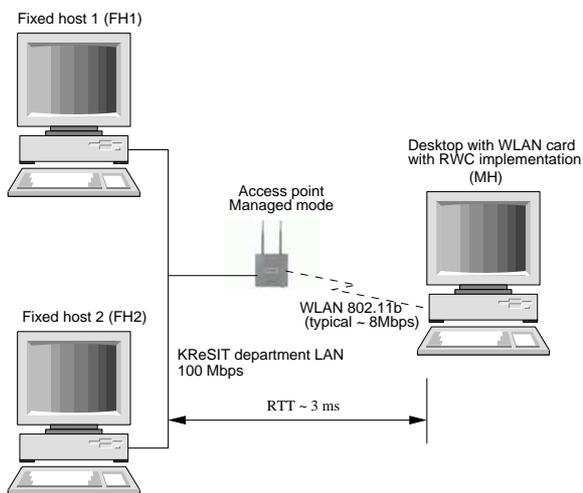
D. Receiver Window Control Experiments

Fig. 4. RWC: Experiment setup

Figure 4 shows the experimental setup. For our experiments we used a Air Premier D-Link Enterprise 2.4GHz Wireless Access Point – DWL-1000AP+ (managed mode), and D-link 520+ Wireless PCI Adapter (2.4 GHz) on the desktop (Redhat Linux Kernel 2.4.x, Pentium4, 1.9 GHz, 256 MB

RAM) which had the RWC implementation. To match the environment characteristics (noise on the wireless channel), we set the sensitivity threshold of the WLAN card to 60 using the script available with the driver package [23]. We used the default MTU of 1500.

We ran two simultaneous *http* file download sessions. FH1 and FH2 were the senders and MH was the receiver (figure 4). The downloaded files were 2MB each. We invoked RWC, on MH, from the command prompt during the http transfers. For the experimental setup the approximate bandwidth-delay product was 3KB. (8000/8 KBps * 3/1000 sec, see figure 4). Thus to throttle a sender the receiver window was kept below 3KB. Since in a WLAN network packet losses reduce the throughput, we started with a receiver window of 2KB to ensure that the sender throughput is bound by the receiver window. The results of our experiments are shown in figure 5.

RWC experiment observations:
Scenario 1 - No RWC (figure 5(a)): The default bandwidth available to the flows is shared unequally. The flow that starts first (Flow-1) gets most of the bandwidth.
Scenario 2 - RWC: We reduced the receive window of Flow-1. There was no need to increase the receive window of the other flow (Flow-2), since the default (system) receive window is 64KB which is much larger than the bandwidth-delay product of 3KB.

- *Receive Window 2KB (figure 5(b)):* As expected, Flow-1 is throttled due to the reduced receive window.
- *Receive Window 1KB/0.5KB (figures 5(c) - 5(d)):* Due to further reduction in the receive window, the bandwidth available to Flow-1 is lesser than in the above case.

The above results validate our user feedback (RWC) implementation.

In this section we validated ECLAIR through a prototype implementation of Receiver Window Control. In the next section we use the above implementation to evaluate the performance of ECLAIR.

IV. ECLAIR PERFORMANCE EVALUATION

We first select the metrics for evaluating cross layer feedback architectures. We use [10] as the non-ECLAIR Receiver Window Control implementation for comparison with the ECLAIR implementation.

A. Evaluation Metrics

Cross layer feedback is essentially a modification to the protocol stack. The intent is to enhance performance of the stack by reading information from a layer, interpreting that information and effecting a change at another layer. However, cross layer feedback entails the overhead of running additional instructions or programs, in the kernel or user space. In light of this, we believe that the primary performance measure for cross layer feedback should be the *time* and *space* overhead. Further, it is essential that any cross layer feedback idea is easy to implement and maintain. Thus, *design complexity* is the second important criteria. Lastly, even if the time/space overhead or design complexity of a cross layer feedback
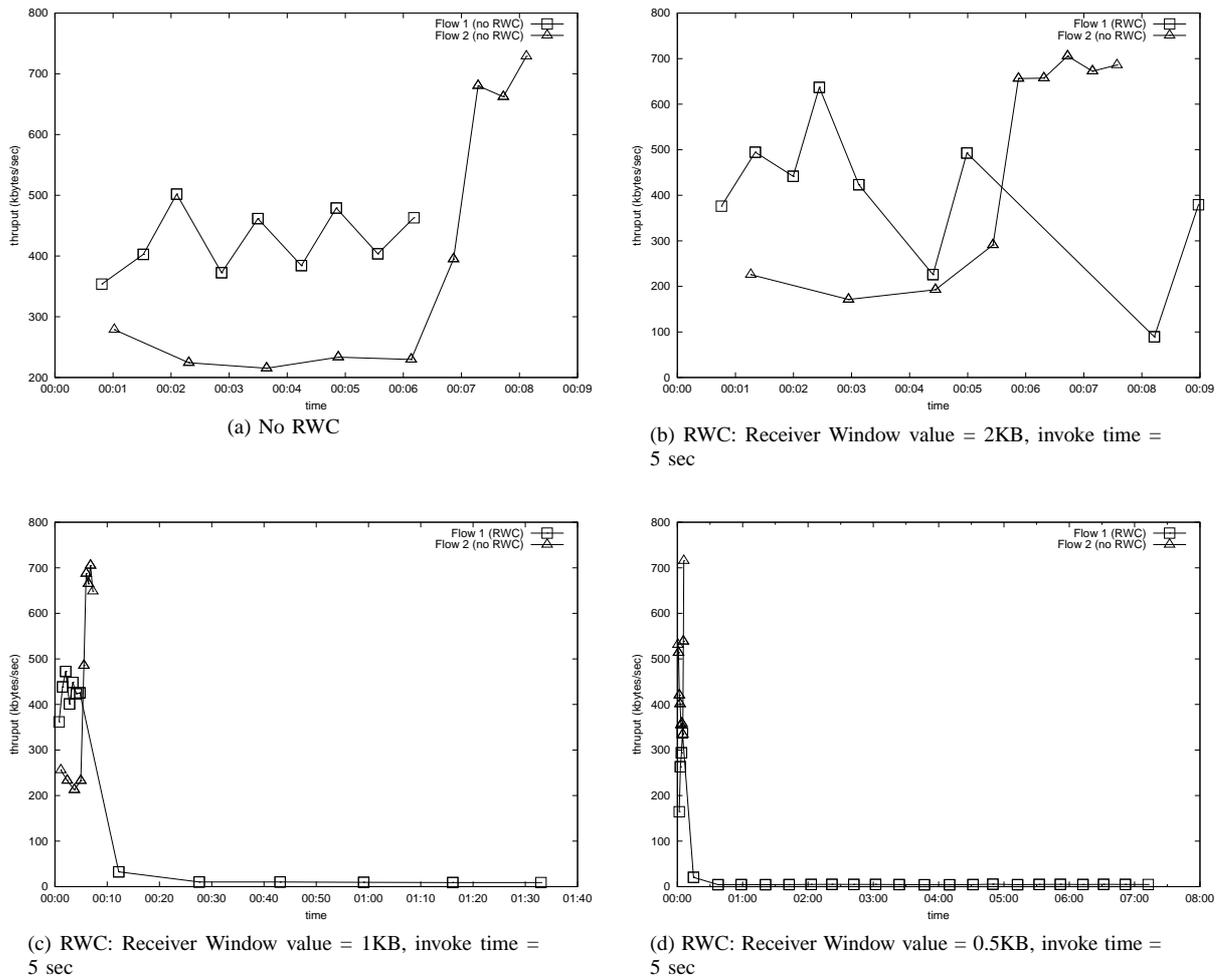
(a) No RWC

(b) RWC: Receiver Window value = 2KB, invoke time = 5 sec

(c) RWC: Receiver Window value = 1KB, invoke time = 5 sec

(d) RWC: Receiver Window value = 0.5KB, invoke time = 5 sec

Fig. 5.   Receiver Window Control experiments over WLAN (802.11)

implementation is low, *user-kernel* crossing could signi£cantly add to the run-time overhead of cross layer feedback. Hence, we also consider user-kernel crossing as an important performance metric. Besides the above metrics which highlight the *overhead* imposed by a cross layer feedback mechanism, another important metric is *data path delay*. Data path is the sequence of functions within the protocol stack which are concerned only with sending and/or receiving packets. *Data path delay* means the time delay on the data path. Data path delay may form a part of the time/space overhead or user-kernel crossing overhead of a cross layer feedback mechanism. However, it is important enough to be considered as an independent performance metric since it directly impacts the throughput of the stack.

We believe that these four performance metrics i.e. time/space overhead, design complexity, user-kernel crossing overhead and data path delay are suf£cient to assess the overheads of any implementation of cross layer feedback.

We use the above metrics to compare the ECLAIR implementation of RWC (section III) with the non-ECLAIR [10] implementation. For this paper, we assume that the time-space and design complexity of both the implementations is similar.

Hence we focus only on the user-kernel crossing impact and data path delay. For comparison we analyze the design of both the implementations of receiver window control. We use the standard software engineering design techniques like *structure chart* and *sequence diagram*[7] for our analysis.

B. User Kernel Crossing Comparison

Figure 6 shows the structure chart for the non-ECLAIR implementation of RWC. No speci£c architecture has been explicitly stated in [10]. It is essentially a *user space* implementation. We call this *non-ECLAIR* for convenience.

Applications *register* with the RWC module. This RWC *module* is invoked on every `read()` of registered applications. To modify the receiver window values, the operating system calls `getsockopt()` and `setsockopt()` are invoked from within `read()`. This is implemented by modifying the standard C library – libc. From £gure 6 it can be deduced that for this non-ECLAIR implementation the user-kernel crossing is:

$$O(m \times n)$$

where, n is the number of applications and m is the number of *reads* per application.
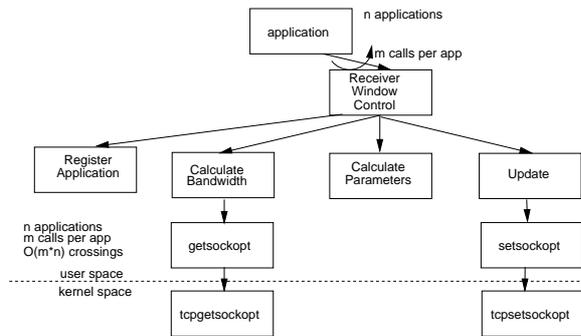
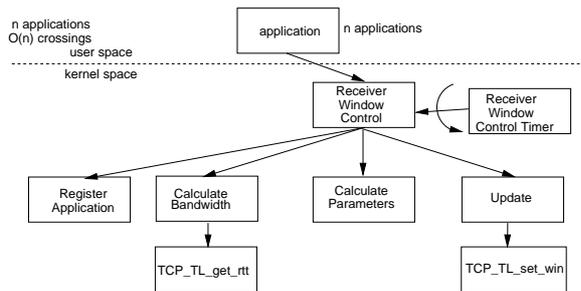Fig. 6.   Receiver Window Control: Structure chart – non-ECLAIR

Fig. 7.   Receiver Window Control: Structure chart – ECLAIR

In the ECLAIR implementation (£gure 7), the application *registers* for RWC and passes the required parameters to the RWC module, using `ioctl()`. This is the only user-kernel crossing. From £gure 7 we can see that the user-kernel crossing is:

$$O(n)$$

where, n is the number of applications. RWC is invoked at regular intervals by a timer within the kernel. Since this invocation is from within the kernel, the overhead on the system is quite less. The tuning layer has access to the TCP data structures and updates the receiver window values directly.

C. Data Path Delay Comparison

To understand the concept of data path delay, we £rst present the sequence diagram of send and receive data paths of an *unmodi£ed* protocol stack in £gure 8.

Figure 9(a) shows the modi£ed data path for the non-ECLAIR [10] implementation. The `data-a` return shows the actual return path if `read()` was not modi£ed. It can be seen from the £gure that the data return from the modi£ed `read()` is delayed till the RWC algorithm completes its run. This is the data path delay. Considering all the *registered* applications together, excluding other overheads, the overall data path delay is $O(m \times n)$ (see £gure 6).

Figure 9(b) shows the data path for the ECLAIR implementation of receiver window control. The ECLAIR implementation is *asynchronous* and hence does not impact the data path of the application.
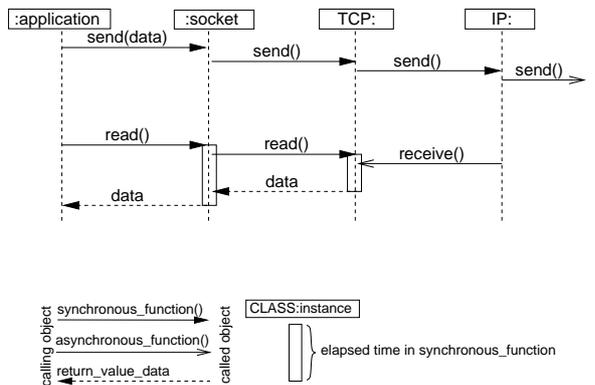

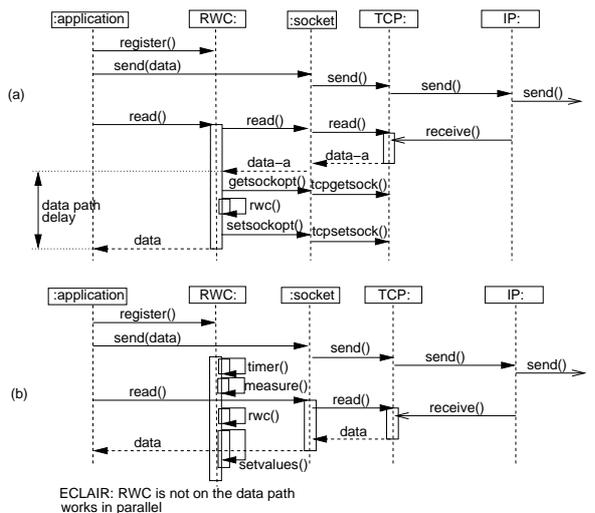Fig. 8.   Sequence diagram of data send and receive paths for an unmodi£ed protocol stack

Fig. 9.   Data send and receive with RWC: (a) non-ECLAIR (b) ECLAIR

D. ECLAIR Evaluation Summary

ECLAIR pro£ling: The ECLAIR implementation uses `ioctl()` for communicating user inputs to the stack. Table I shows the time measurements for ECLAIR.

ECLAIR function	User kernel crossing
`ioctl()`	82

TABLE I
ECLAIR PROFILING (TIME IN μSECS)

non-ECLAIR pro£ling: Since the non-ECLAIR implementation uses `getsockopt()` and `setsockopt()`, we pro£led the time taken by `getsockopt()` and `setsockopt()`. Table II shows the time measurements for these *system calls*.

Table III shows the comparative performance summary of ECLAIR and non-ECLAIR implementations of receiver window control.

As seen earlier, that the total user kernel crossing impact for ECLAIR implementation is $O(n)$ while that for non-ECLAIR its $O(m \times n)$. Hence for ECLAIR, since the `ioctl()` delay is 82 μsec (table I), the overall impact due to user-kernel

System call	User kernel crossing
`getsockopt()`	21
`setsockopt()`	20

TABLE II

NON-ECLAIR PROFILING (TIME IN μSECS)

crossing is $82 \times O(n)$. For the non-ECLAIR case, the overall impact is $(20 + 21) \times O(m \times n)$. There is no data path delay for ECLAIR while there is a data path delay of $O(m \times n)$ for the non-ECLAIR implementation. This is summarized in table III.

Performance metric	ECLAIR	non-ECLAIR
User-kernel crossing (μsec)	$82 \times O(n)$	$(20+21) \times O(n \times m)$
Data path delay (μsec)	-	$(20+21) \times O(n \times m)$

TABLE III

ECLAIR AND NON-ECLAIR QUANTITATIVE COMPARISON

In this section we selected metrics for evaluating any cross layer feedback mechanism. We compared ECLAIR and non-ECLAIR implementations of receiver window control using user-kernel crossing overhead and data path delay metrics. The evaluation results highlight the efficiency of ECLAIR.

In the next section we present the sub-architecture to minimize ECLAIR overheads.

V. ECLAIR SUB-ARCHITECTURE

To maximize the benefit from cross layer feedback a well-defined methodology is required for (a) identifying critical cross layer *data items* and (b) minimizing overhead for cross layer feedback. A cross layer *data item* is information that is available at a layer which can be used for cross layer feedback to other layers. For e.g. bit-error-rate information at the physical layer is a data item.

In this section we present: (a) a method for quantifying the contribution of a data item and using this to identify the critical data items and (b) a sub-architecture for cross layer feedback which complements ECLAIR.

A. Identifying critical data items

Each data item used in cross layer feedback could provide a certain *utility* whenever it is used by a layer. This *utility* could be reduction of CPU cycles or power consumption and reduction of memory. On the other hand, the *cost* of a data item is the CPU cycles or power and memory required to enable cross layer feedback for that data item. The first step in cross layer optimization is identifying those elements which have high *utility*. These data items are the *critical* data items for cross layer optimization.

To quantify the utility of a data item the exact saving achieved by using the data item needs to be found. The saving could be determined either by precise models, simulations or actual measurements. At the design stage, we feel that the estimated frequency of use of the data item, for cross layer feedback, can serve as an indicator of its utility.

Let, d_i be a data item at a layer j. i is an index of the set of data items available for cross layer feedback throughout the stack. The total number of times, ω_i, the data item is accessed by various layers, other than j, is an indicator of the *utility* of the data item. Critical data items are the ones for which ω_i is high. The designer may choose to define a threshold or cutoff value for ω_i.

After the critical data items have been identified the next step is defining the sub-architecture for cross layer feedback, within ECLAIR.

B. Sub-architecture for cross layer feedback

The highlight of our sub-architecture is the creation of a special subset of data items from the critical data items. We call this subset the *core*. A data item is placed in the *core* if the cost of cross layer feedback for the data item is lower when it is placed in the core. Figure 10 illustrates the concept of core.
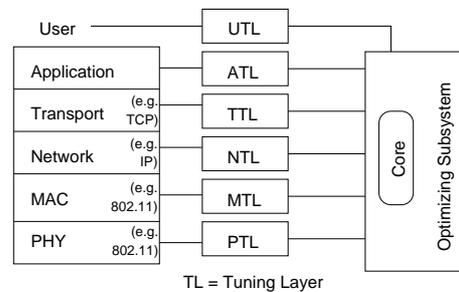

Fig. 10. ECLAIR with core

Core:

Let the *critical* set of data items available for cross layer feedback be $\mathcal{D} = \{d_i : \omega_i > \upsilon\}$, where υ is a threshold on utility for identifying the critical data items. With reference to core, *writing* a data item into the core means copying the value of the data item into core. Similarly, *reading* a data item from core means reading the value of the data item from core by other layers.

Costs related to a data item d_i: Let,

- the cost of writing the data item into the core:

$$\phi'_i = c_w \times \omega'_i \quad (1)$$

where,
c_w = cost of a single write of the value of the data item into the core. Assumed to be constant for all data items.
ω'_i = estimated frequency of writing the value of the data item into the core

- the cost of reading a data item from the core:

$$\phi_i = c_r \times \omega_i \quad (2)$$

where,
c_r = cost of a single read of the data item if it is in core. Assumed constant for all data items.

ω_i = sum of estimated frequency of access of the data item by all layers, other than the layer generating the data item value.

- $\bar{c}_r$ = cost of a single read of the data item if it is not in core. Assumed constant for all data items.

The values of the various costs ($c_r, \bar{c}_r, c_w$) would depend on the system characteristics.

Def. 5.1 (Core Interaction Cost (CIC)): **CIC** (Υ_i) of a data item d_i is defined as sum of the cost of writing and reading the data item from the core. From equations (1) and (2)

$$\Upsilon_i = \phi_i' + \phi_i \qquad (3)$$

The total *cost of core* Ψ of core C is the sum CIC of all elements in C

$$\Psi = \sum \Upsilon_i \text{ for all } d_i \in C \qquad (4)$$

Similarly, the total *utility of core* Θ is the sum of the utilities of all the data items in C

$$\Theta = \sum \omega_i \text{ for all } d_i \in C \qquad (5)$$

Next, it is to be decided whether an item is suitable for the core or not. For this, the power saving obtained by putting an item in core needs to be evaluated.

Def. 5.2 (Core Potential Score (CPS)): **CPS** (κ_i) of a data item d_i is defined as the reduction in power obtained by placing the item in core.

$$\kappa_i = (\bar{c}_r \times \omega_i) - \Upsilon_i \qquad (6)$$

An item d_i is suitable for the core C only if $\kappa_i > 0$.

Rearranging the terms of equation (6) and from equations (1), (2), (3), we get

$\kappa_i > 0$, if and only if

$$1 - \frac{c_r}{\bar{c}_r} - \frac{c_w}{\bar{c}_r} \times \frac{\omega_i'}{\omega_i} > 0 \qquad (7)$$

Since all the terms in equation (7) are positive, it can be easily seen that:

(a) If $c_r \geq \bar{c}_r$ then $\kappa_i < 0$ i.e. the data item is not suitable for core
(b) If $c_r \ll \bar{c}_r$, $c_w \ll \bar{c}_r$ and $\omega_i' \ll \omega_i$ then $\kappa_i \gg 0$ i.e. the data item is most suitable for core
(c) If $c_r \ll \bar{c}_r$, $c_w \ll \bar{c}_r$ and $\omega_i' \approx \omega_i$ then $\kappa_i > 0$
(d) If $c_r \ll \bar{c}_r$, $c_w \approx \bar{c}_r$ and $\omega_i' \ll \omega_i$ then $\kappa_i > 0$
In the above two cases also the data item is suitable for the core
(e) If $c_r \ll \bar{c}_r$, $c_w \approx \bar{c}_r$ and $\omega_i' \approx \omega_i$ then $\kappa_i \approx 0$ i.e. the data item is not suitable for core.

Algorithm for selecting the elements for core:
Initially C is empty. Let τ be some *threshold* (design criteria) for the core defined by the designer. The ordering of the data items ensures that first the *high utility* data items are picked for the core. The algorithm is presented in figure 11.

```
1.
Sort the elements in D based on their CPS' i.e. the element(s) with the
maximum saving is first in the set. Let, D' be the sorted set of data items.
2.
{Check each element}
for all d_i ∈ D' do
    {Check net utility if item in core}
    if Θ − Ψ < τ then
        C = C ∪ {d_i}
    else
        break
    end if
end for
```

Fig. 11. Core algorithm

C. Usage Scenario

We show the use of the sub-architecture through an example. We assume certain cross layer feedback items.

Our example assumes the following $\bar{c}_r = 1, c_r = 0.5, c_w = 0.5$.

Cross layer data items:
For the sake of simplicity, we consider only four data items:

- d_1 = Retransmission information at link layer
- d_2 = Losses acceptable to an application (application layer)
- d_3 = User defined application priority
- d_4 = Wireless channel bit-error rate

Next, we assume some frequency of write and access for the data items.

- d_1: Write frequency $\omega_1' = 50$ per second. The layers that could use this information are (1) TCP, for adapting its retransmission timeout value and (2) application layer to get an estimate of the channel condition and adapt its sending rate. We assume TCP uses this 10 times per sec, while the application uses this information once per second. Thus, $\omega_1 = 10 + 1 = 11$.
- d_2: Write frequency: $\omega_2' = 1/600$ per second (i.e. application may change its requirements once in ten minutes). The layers that may read this information are link layer and IP layer. Link layer could use this information to adapt its error control mechanisms according to application requirements and channel conditions. IP layer would read this information to determine the interface on which to send the packets. We assume that link layer reads this information 50 times per sec and IP layer reads this information 10 times per second. Thus $\omega_2 = 50 + 10 = 60$.
- d_3: Write frequency: $\omega_3' = 1/600$ per second (i.e. user may update application priority once in ten minutes). This information may be used by RWC (see section III) to manipulate the receiver window for current applications. We assume that RWC reads this once every ten minutes. Thus $\omega_3 = 1/600$.
- d_4: Write frequency: $\omega_4' = 10$ per second (bit-error information from physical layer). MAC, IP, TCP, and application layers may read this information for adaptation. We assume each reads this information 10 times per second. Thus $\omega_4 = 40$.

Based on ω_i the critical data items can be determined. If the cut-off for ω_i was 10, then d_1, d_2 and d_4 would be the

critical data items.

Core:
Using this information and the equations (1),(2),(6) we get:
$\kappa_1 = 11 \times 1 - (50 \times 0.5 + 11 \times 0.5) = -19.5$
$\kappa_2 = 60 \times 1 - (\frac{1}{600} \times 0.5 + 60 \times 0.5) \approx 30$
$\kappa_4 = 40 \times 1 - (10 \times 0.5 + 40 \times 0.5) = 15$

From the values of κ_i we can see that d_1 is not suitable for the core since $\kappa_1 < 0$.

Using the *core algorithm* in £gure 11, if $\tau = 35$ then d_2 will be in the core.

In this section, we presented a sub-architecture for ECLAIR which reduces the cross layer feedback overheads. In the next section we present a guideline for applying ECLAIR to cross layer feedback implementations.

VI. Cross Layer Feedback Design Guide

We believe that the primary criteria for selecting a cross layer feedback architecture is the *type* of cross layer feedback.

A. Asynchronous v/s Synchronous Cross Layer Feedback

The adaptive action at a layer based on cross layer feedback from another layer may be *synchronous* or *asynchronous*. In synchronous cross layer feedback, whenever a layer receives some cross layer information, it proceeds with its *regular* execution only after executing the cross layer adaptation required. For example, assume there is *network disconnection* event sent to TCP from the link layer. In the synchronous case, TCP's regular execution is stopped, appropriate adaptation is carried out in TCP and then regular TCP execution proceeds. In the asynchronous case, the control data structures of TCP would be updated while TCP's execution is in progress.

To ensure *correct* and *ef£cient* cross layer feedback appropriate architecture is required suited to the type of cross layer feedback.

As an example, we consider receiver window control explained earlier (section III). In this case, the primary requirement is to apportion application bandwidth. It may not be essential to tune application bandwidth *synchronously* with each `read()` of an application, as proposed in [10]. Thus the architecture proposed in [10] reduces the application throughput. However, as shown in section IV, if an asynchronous architecture e.g. ECLAIR is used, the data path delay and hence application throughput is not reduced.

Further, the cross layer feedback behavior would be incorrect, if an architecture suitable for asynchronous feedback is used for synchronous feedback. For example, cross layer feedback adaptation which is to be triggered by information contained in each packet would fail if an asynchronous architecture like ECLAIR is used.

Subsequent to the architecture choice based on the type of cross layer feedback, it is essential to minimize the overheads of the cross layer feedback implementation. In the following sections we discuss the design guide for ECLAIR implementations for single and multiple PO (protocol optimization) cases.

B. Single Cross Layer Optimization

Separating the Protocol Optimizers and Tuning Layers into a separate cross layer system, outside the stack, introduces the overhead of additional function calls. Hence, in case only a single cross layer optimization is planned and the cross layer system is not to be ported / deployed on multiple operating systems then it is better to incorporate the protocol optimizer (PO) and tuning layers (TLs) within the existing stack itself. This would reduce the overhead of multiple function calls between PO and TL and hence would increase the ef£ciency of the implementation. However, if additional cross layer feedback optimizations are to be introduced later, PO and TL should be implemented as separate modules. This is to avoid the *maintainability* and *portability* issues later.

C. Multiple Cross Layer Optimizations

In case of multiple cross layer optimizations, POs and TLs should be implemented as indicated in the ECLAIR architecture.

If multiple cross layer optimizations or POs directly access the layers, then the dependency of the POs is high on the layer's code. Any change to the layer code will lead to a change in all the POs interacting with that layer. Reducing such *coupling* is useful for ease of maintenance and evolution of the cross layer system. Introduction of a tuning layer, leads to reduction in the coupling between the layer code and POs. Further, *core* should be introduced for reducing the cross layer overheads.

In summary, ECLAIR should be used if the cross layer type is asynchronous. Further, POs and TLs should be implemented, as proposed in ECLAIR, if multiple cross layer optimizations are to be implemented or if the cross layer system is to be ported to multiple operating systems. Core should be introduced to reduce the cross layer overheads.

VII. Related Work

For cross layer feedback on the device [15] propose that the network layer monitors lower layers for events and generates Internet Control Message Protocol (ICMP) messages within the protocol stack. A special handler traps these messages and adapts protocols based on adaptations de£ned by the application developer. This is essentially an architecture that passes lower layer information to higher layers. There is no mechanism to pass information from higher to lower layers e.g. TCP cannot pass information to the MAC layer. Another disadvantage is that all *events* are wrapped in ICMP packets, which increases the cross layer feedback overhead.

In [18] cross layer information is exchanged through packet headers. Hence, this is suitable for selected types of synchronous cross layer feedback e.g. adaptation at lower layers based on information in each packet from higher layers. However this requires that lower layers be able to read higher layer headers. Further substantial modi£cation is required to the layer code where adaptation is required. For information from lower to higher layers, the lower layers would need to change the packet header, which could lead to packet errors.

The framework in [2] proposes a *cross layer manager* that corrects the behavior of a protocol based on events it receives from other protocols in the stack. This too seems to be suitable for *asynchronous* cross layer feedback. However, internal details, implementation and performance issues have not been discussed.

Cross layer signaling shortcuts (CLASS) are proposed in [16]. A representative survey is also presented. However, the CLASS mechanism would have drawbacks similar to that of ad hoc implementations.

The aforementioned cross layer mechanisms would require substantial modifications to the stack as compared to ECLAIR. Further, they do not address in detail the issue of cross layer feedback overheads and evaluation and metrics.

Additional research useful for cross layer feedback design is as follows: Useful caveats and principles related to cross layer feedback design are presented in [8]. Therein the issue of cross layer feedback conflict is also highlighted. Power aware protocols in ad hoc networks are discussed in [5]. References therein provide insight into the various power aware protocol proposals and design issues. A survey of cross layer feedback optimizations is presented in [11].

VIII. CONCLUSION

The performance of layered protocol stacks can be improved by cross layer feedback. In this paper we highlighted the problems associated with *ad hoc* cross layer implementations. Ad hoc cross layer implementations can affect the efficiency, correctness and maintainability of the protocol stack. This highlights the need for a appropriate cross layer feedback architecture.

We presented the internal details of our cross layer architecture ECLAIR. ECLAIR provides the benefits of rapid prototyping, minimum intrusion, portability and minimal overheads. ECLAIR also enables (a) dynamic control (switch on/off) of cross layer algorithms, (b) seamless mobility and (c) user feedback.

We validated ECLAIR through a prototype implementation of Receiver Window Control (RWC), on a Linux desktop and experiments over WLAN. To evaluate cross layer feedback architectures we selected the metrics of *time/space complexity*, *design complexity*, *user-kernel crossing* and *data path delay*. We used these metrics to compare ECLAIR with another implementation of RWC. Our results and analysis show that ECLAIR is an efficient architecture, since it minimizes the *user-kernel crossing* and *data path delay*. We also proposed a sub-architecture *core* which is a set of cross layer items which provide high performance gains. Core helps reduce the overheads of cross layer feedback in ECLAIR.

ECLAIR is suited for *asynchronous* cross layer optimizations. For single protocol optimizations ECLAIR overheads may be too high. However, ECLAIR is beneficial when there are multiple protocol optimizations to be implemented. With suitable modifications it may be possible to use ECLAIR for synchronous cross layer feedback also, however, this would require substantial modifications to the stack could lead to increased delay on the data path.

ECLAIR also allows a distributed implementation of cross layer feedback. For example, in case of a laptop connected to the network through a wireless modem, the tuning layers of the link and physical layers could reside on the wireless modem and the rest of ECLAIR TLs and OSS could reside on the laptop.

REFERENCES

[1] H. Balakrishnan, V. N. Padmanabhan, S. Seshan, and R. H. Katz. A Comparison of Mechanisms for Improving TCP Performance over Wireless Links. *IEEE/ACM Transactions on Networking*, 5(6):756–769, December 1997.
[2] G. Carneiro, J. Ruela, and M. Ricardo. Cross Layer Design in 4G Wireless Terminals. *IEEE Wireless Communications*, 11(2):7–13, April 2004.
[3] Jonathan Corbet, Alessandro Rubini, and Greg Kroah Hartman. Linux Device Drivers. O'Reilly, February 2005. Third edition.
[4] J. Crowcroft and I. Phillips. TCP/IP & Linux Protocol Implementation: Systems Code for the Linux Internet. John Wiley & Sons, October 2001. 1st edition.
[5] A.J. Goldsmith and S.B. Wicker. Design Challenges for Energy-constrained Ad hoc Wireless Networks. *IEEE Wireless Communications*, 9(4):8–27, August 2002.
[6] ITU. Information technology - OSI - Basic Reference Model, July 1994. X.200.
[7] P. Jalote. *An Integrated Approach to Software Engineering*. Springer Verlag New York, 1997. Second Edition.
[8] V. Kawadia and P. R. Kumar. A Cautionary Perspective on Cross Layer Design. *IEEE Wireless Communications*, 12(1):3–11, February 2005.
[9] M. Allman and V. Paxson and W. Stevens. RFC2581: TCP Congestion Control, April 1999.
[10] P. Mehra, A. Zakhor, and C. Vleeschouwer. Receiver-Driven Bandwidth Sharing for TCP. In *IEEE INFOCOM*, SF, USA, April 2003.
[11] V. T. Raisinghani and S. Iyer. Cross-layer Design Optimizations in Wireless Protocol Stacks. *Computer Communications (Elsevier)*, 27(8):720–724, May 2004.
[12] V. T. Raisinghani and S. Iyer. ECLAIR: An Efficient Cross Layer Architecture for Wireless Protocol Stacks. In *World Wireless Congress*, SF, USA, May 2004.
[13] V. T. Raisinghani, A. K. Singh, and S. Iyer. Improving TCP Performance over Mobile Wireless Environments using Cross Layer Feedback. In *IEEE ICPWC*, New Delhi, India, December 2002.
[14] W. Richard Stevens. *TCP/IP Illustrated, Volume I, The Protocols*. AWL, 1994.
[15] P. Sudame and B. R. Badrinath. On Providing Support for Protocol Adaptation in Mobile Networks. *Mobile Networks and Applications*, 6(1):43–55, 2001.
[16] Qi Wang and M.A. Abu-Rgheff. Cross-layer Signalling for Next-Generation Wireless Systems. In *Wireless Communications and Networking (WCNC)*, volume 2, pages 1084–1089. IEEE, March 2003.
[17] C. Wijting and Ramjee Prasad. A Generic Framework for Cross-Layer Optimisation in Wireless Personal Area Networks. *Wireless Personal Communications*, 29(1–2):135–49, April 2004.
[18] Gang Wu, Yong Bai, Jie Lai, and A. Ogielski. Interactions between TCP and RLP in Wireless Internet. In *IEEE GLOBECOM*, volume 1B, pages 661–666, Rio de Janeireo, Brazil, December 1999. IEEE.
[19] G. Xylomenos and G. C. Polyzos. Internet Protocol Performance over Networks with Wireless Links. *IEEE Network*, 13(4):55 – 63, July/August 1999.
[20] Cbrowser. http://cscope.sourceforge.net/.
[21] Cross referencing linux. http://lxr.linux.no/source/.
[22] Cscope. http://cscope.sourceforge.net/.
[23] The ACX100/ACX111 Wireless Network Driver Project. http://www.linuxforum.com/linux-kernel-programming/index.html, 2004. version 0.20pre8.