XIAN Automated Management and Nano-Protocol to Design Cross-Layer Metrics for Ad hoc Networking

Hervé Aïache, Vania Conan, Laure Lebrun, Jérémie Leguay,
Stéphane Rousseau, Damien Thoumin

Thales Communications[**]
160, boulevard de Valmy, BP82
92704 Colombes Cedex, France
{firstname.name}@fr.thalesgroup.com

Abstract. In the highly dynamic and unpredictable environment of MANETs, cross-layer design is receiving growing interest but lacks experimental validation tools. This paper presents *XIAN* (Cross-layer Interface for wireless Ad hoc Networks), a generic framework for experimenting cross-layer designs in Linux testbeds with 802.11 wireless cards using the *MadWifi* driver. *XIAN* can be used as a service by other layers or system components to access MAC/PHY configuration and performance information. It provides experimenters with an open framework to create automatically complex metrics from both local and neighbour node measurements. The defined and implemented software architecture introduces the *XIAN Nano-protocol* and its automated management. We exemplify their benefits through the implementation of the well-known ETX (*Expected Transmission count*) metric and we provide results from real experimentations.

1 Introduction

In Mobile Ad hoc Networks (MANETs [1]), nodes can be mobile and share one or more wireless channels without centralized control. In such dynamic and unpredictable distributed environments, traditional networking principles, such as the layer isolation of the OSI (Open Systems Interconnection) model, are challenged. Recent research studies have explored more flexible approaches to networking, called *Cross-Layer* approaches, that revisit the classical IP stack design. The central idea of cross layering consists in allowing a more flexible exchange of status or control information between the different components of the communication system. With a better knowledge of available resources from different layers of the ad hoc network stack, the system is expected to be more reactive to the wireless environment and responsive to quality requirements of applicative-oriented elements.

When compared to the usual OSI reference model, existing cross-layer solutions span a wide spectrum of options: some advocate global exchange of information between components (e.g. Conti et al. [2]), others prefer to limit them to adjacent layers (e.g. Kawadia et al. [3]), depending on how they impact or differ from this reference model. In any case,

[**] This work has been partially supported by the European IST FP6 CHORIST project and the French RNRT Airnet project.

cross-layering calls for software architectures and implementations that support a more flexible sharing of information and status reports between the processes and functional modules of the communication system. Experimenting with cross layer design for MANETs remains difficult, most ad hoc testbeds making use of 802.11 cards which lack appropriate API support.

This paper builds upon our earlier work [4] to define XIAN (Cross-layer Interface for wireless Ad hoc Networks) which consists in a framework that facilitates cross-layer integrations and experimentations by easing the access to information from MAC/PHY layers. The ultimate goal of XIAN is to encourage and facilitate cross-layer studies and experimentations over MANET testbeds. It has been implemented and is available in open source [7] for Linux over the *MadWifi* 802.11 driver [5]. This paper presents a major extension to the earlier basic framework by supporting the implementation of complex cross-layer metrics that can combine both local information and measurements obtained from the node's neighbours by the *XIAN Nano-protocol*. We exemplify its use with the implementation of the ETX (Expected Transmission Count) metric proposed by De Couto et al. [6] and we provide experimental demonstration of potential benefits.

The remainder of the paper is structured as follows. Section 2 presents the XIAN approach and software architecture. Section 3 describes the XIAN programming interfaces and examples of accessible metrics. Section 4 and Section 5 present the use and experimental benefits of the proposed XIAN extension for the implementation of ETX. Section 6 concludes the paper, discussing directions for future work.

2 XIAN: Cross-layer Interface for wireless Ad hoc Networks

This section introduces the overall XIAN framework. We first present the Linux based software architecture. Then we show how to integrate new complex metrics to extend and customize the framework. Finally we describe the *XIAN Nano-protocol*, integrated at the MAC layer, that automatically handles metric exchanges between neighbour nodes.

2.1 XIAN basic components for cross-layer exchanges

MAC/PHY state information, such as number of transmission retries or number of transmitted frames with CTS enabled, is available at driver level and a large part of this information may be of interest to higher layers. To support flexible cross-layer access to this information XIAN implements a software architecture composed of three main components:

- The ***Kernel Space XIAN Interface*** (KSI) is dedicated to kernel space components (e.g. TCP or UDP implementations) and implemented as a Linux kernel module. It interacts directly with the *MadWifi* driver to retrieve its internal states or statistics or with the XIAN components involved in the definition of complex metrics (see section 2.2).
- The ***User Space XIAN Interface*** (USI) mimics the KSI but at user space level. This API is implemented as an ordinary C library in order to facilitate its integration with user space programs (e.g. routing daemons or applications).
- The ***XIAN Information Transport Module*** (ITM) allows to pass information and statistics from the kernel space to the user space, by connecting the two previous

XIAN APIs. This module is implemented in this version of XIAN as a special character device.

In addition, a complementary component, called the ***XIAN User Space Extended Interface*** (or USEI), provides simple means for experimenters to perform additional processing of raw measurements, such as averaging, metric combination or notification of significant changes. Figure 1 illustrates how the components interact and how internal driver/MAC states or metrics are provided to other Linux system components.

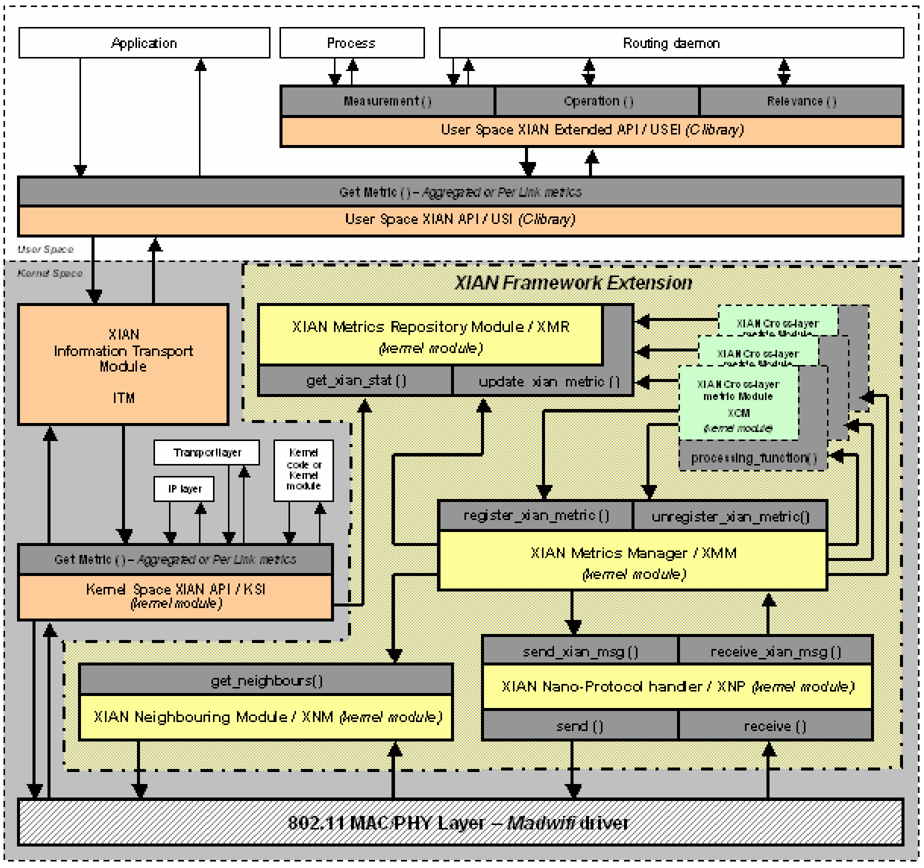
Application
Process
Routing daemon
Measurement ()
Operation ()
Relevance ()
User Space XIAN Extended API / USEI (C library)
Get Metric () – Aggregated or Per Link metrics
User Space XIAN API / USI (C library)
User Space
Kernel Space
XIAN Framework Extension
XIAN Information Transport Module
ITM
XIAN Metrics Repository Module / XMR
(kernel module)
get_xian_stat ()
update_xian_metric ()
XIAN Cross-layer metric Module
XIAN Cross-layer metric Module
XIAN Cross-layer metric Module
XCM
(kernel module)
processing_function()
Transport layer
IP layer
Kernel code or Kernel module
register_xian_metric ()
unregister_xian_metric()
XIAN Metrics Manager / XMM
(kernel module)
Get Metric () – Aggregated or Per Link metrics
Kernel Space XIAN API / KSI
(kernel module)
send_xian_msg ()
receive_xian_msg ()
XIAN Nano-Protocol handler / XNP (kernel module)
send ()
receive ()
get_neighbours()
XIAN Neighbouring Module / XNM (kernel module)
802.11 MAC/PHY Layer – Madwifi driver

Figure 1. XIAN Framework software architecture.

From the developer's point of view, the two XIAN APIs (i.e. KSI and USI) are identical. The information exchanged through the ITM and accessible via the USI and the KSI are of two kinds: (1) basic metrics extracted from specific structures/states maintained by the *MadWifi* driver, and (2) new metrics integrated to the framework thanks to the software components described in the following section.

2.2 Extension to integrate and manage complex cross-layer metrics

XIAN offers mechanisms implementing bidirectional exchanges of metrics among neighbouring nodes (such as those classically needed by [6, 9, 10, 11]). This makes possible the creation and the integration within the XIAN framework of metrics that require information from neighbouring nodes (e.g. the number of packets that have been received by a neighbour). Implementing a new metric involves the definition of the *metric calculation formula*, i.e., it defines how to combine readings from several individual PHY/MAC layer metric values to obtain a new compound one. Obtaining the individual values involves configuring the XIAN framework, by setting which metrics to read from the node's neighbours and at which frequency. The implementation of this major extension to XIAN is available in the new release on the official project web site [7].

As shown Figure 1, four software components, all implemented as Linux kernel modules, support the integration of such complex metrics within the XIAN framework:

- The ***XIAN Metric Manager*** (XMM), in charge of the registration and unregistration of the new cross-layer metrics instantiated as ***XIAN Cross-layer Metric modules*** (XCMs – An XCM mainly implements the *metric calculation formula* and *configures the protocol exchanges* involved in the calculation of the metric in terms of type of information and frequency – see section 3.2).
- The ***XIAN Nano-Protocol handler*** (XNP), which handles the *XIAN Nano-protocol* messages containing the values of the metrics exchanged between neighbouring nodes. The XNP is implemented between the *MadWifi* driver and the Linux kernel implementation of the IP layer. Its role is to extract the *XIAN Nano-protocol* messages from in-coming 802.11 frames and to create and send out-going *XIAN Nano-protocol* messages to the desired neighbours.
- The ***XIAN Metrics Repository*** (XMR), which is responsible of recording the calculated values of the new cross-layer metrics introduced within the XIAN framework. Once recorded by the XMR, the metric values become accessible (via the KSI or the USI – see section 2.1) to other operating system components, along with all *MadWifi* driver information reported by default by XIAN.
- The ***XIAN Neighbouring Manager*** (XNM), which detects neighbour nodes and triggers updates necessary to automate *XIAN Nano-protocol* message exchanges.

To enable the exchanges of metrics between neighbouring nodes, we propose a simple MAC layer-oriented protocol, called the *XIAN Nano-protocol*. It is described in the following section.

2.3 The XIAN Nano-protocol

The *XIAN Nano-protocol* is responsible of exchanging the *XIAN Nano-protocol* messages. Figure 2 gives an example of such a message. It contains a *XIAN Nano-protocol Metrics Reports* made up of (1) a header, which mainly indicates information about the report object itself and (2) a payload, which can embed several *XIAN Nano-protocol Metric Objects* that provides metric values and associated meta information. Note that *XIAN Nano-protocol* messages are encapsulated into unicast or broadcast 802.11 data frames.

The *XIAN Nano-protocol Metrics Report* is composed of the following fields:

- `Version` indicates the used version of the *XIAN Nano-Protocol*.
- `Sequence` is the identifier of the *XIAN Nano-protocol Metrics Report*.
- `Length` indicates the number of XIAN Nano-protocol Metric Objects contained in the XIAN Nano-protocol Metrics Reports.
- `Payload` contains a set of XIAN Nano-protocol Metric Objects.

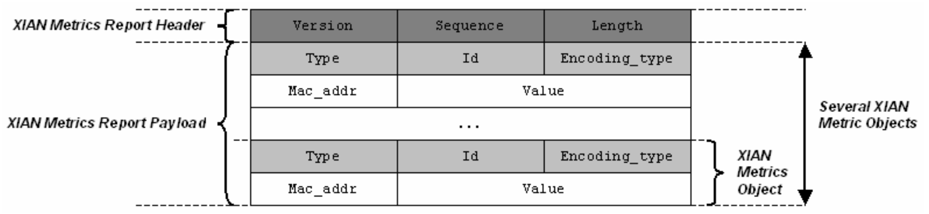

Figure 2. Metrics Report format of the XIAN Nano-protocol.

A *XIAN Nano-protocol Metric Object* is composed of the following fields:

- `Type` indicates the identifier associated to a metric.
- `Id` identifies a reference to the associated XCM (see section 3.2).
- `Encoding_type` indicates how the metric value is encoded (e.g. integer or float).
- `Mac_addr` indicates the MAC address to which the metric value belongs to.
- `Value` contains the effective numerical value of the metric.

Moreover, in order to optimize the number of messages sent to a given neighbour node, note that the ***XIAN Nano-Protocol handler*** (XNP) is able to aggregate several *XIAN Cross-Layer Metrics Object* of different types in a single *XIAN Metrics Report*.

3 XIAN programming interfaces

The previous section described the main building blocks of the XIAN architecture. This section details the programming interfaces offered by XIAN to implement view to access, refine, design and integrate MAC-oriented cross-layer metrics.

3.1 Accessing MAC-oriented basic metrics

XIAN eases the access to a large set of basic metrics offered by the *MadWifi* driver. The most important ones can be divided into two groups:

- ***Global metrics***, similar to counters, this kind of metrics provides global statuses on the use of the 802.11 network interface. The reported information can be: the number of received frames dropped or with wrong BSSID, the number of transmitted frames with CTS or with RTS enabled, the relative signal strength indicator (RSSI) of the last ACK received, the number of failed receptions (due to queue overrun, bad CRC, PHY errors or decryption problems).

- ***Per link metrics***, which stores per-neighbour information. For instance, this kind of metric reports the number of received/transmitted data frames or bytes, the relative signal strength indicator (RSSI) or the number of transmission retries.

Depending on the type of reported metric (aggregated or per-neighbour/link), the prototype of the function (for a given metric named `metric_name`) is defined as follows:

- For a global metric:

```
u_int32_t                                 /* returned metric */
get_metric_name(char * dev_name,          /* Interface name */
                unsigned int * code_err); /* Error code */
```

- For a per-neighbour/link metric:

```
u_int32_t                      /* returned metric value */
get_node_metric_name(
      u_int8_t * macadd,       /* Neighbour node’s MAC address */
      char * dev_name,         /* Interface name */
      unsigned int * code_err); /* Error code */
```

About 180 such basic metrics are supported in XIAN, obtained from converting the corresponding readings of the local *MadWifi* driver. 40 of these measurements are given on a per-neighbour basis, the remaining ones being global values for the node.

3.2 Design and integration of new complex metrics

New cross-layer metrics are implemented in XIAN as ***XIAN Cross-layer Metric modules*** (XCMs). XCMs are usual Linux kernel modules which use specific APIs mainly defined by the ***XIAN Metric Manager*** (XMM). To define both the *metric calculation formula* and the protocol configuration, the XMM proposes the following interface to register a new XCM within the framework:

```
void
register_id (unsigned int id,                   /* Conf. identifier */
             unsigned int metric,               /* New metric id. */
             char dev[IFNAMESIZ+1],             /* Net. interface */
             char mac[IEEE80211_MAC_ADDR_LEN+1], /* Report dest. MAC */
             unsigned int freq,                 /* Reports frequency */
             void * pf)                         /* Processing func. */
```

Once called, the XMM introduces the new metric identified within XIAN under the reference `metric`. The XMM will automatically send a report about this cross-layer metric (through *XIAN Nano-protocol Metrics Reports*) on the network interface identified as `dev` by the Linux kernel each `freq` milliseconds. Moreover, when a *XIAN Nano-protocol Metric Object* of type `metric` is received, the XMM calls the associated processing function `pf` to compute the new value of the metric (typically implementing the *metric calculation formula*). Note that the field `id` allows to uniquely identify an XCM to allow reuse of the same cross-layer metric for different processing or configurations.

Note that with such a design, experimenters do not have to take care about the *XIAN Nano-protocol* exchanges, but just have to configure its behaviour, meaning: the frequency of the metrics reports (i.e. the parameter called `freq`), what are the nodes involved in the calculation of the metric (i.e. the parameter `mac`) and the formula to calculate the metric (i.e. the processing function `pf`).

In addition to the function `register_id()`, the XMM proposes another complementary interface which allows to un-register an XCM:

```
void unregister_id(unsigned int id,     /* Conf. Identifier */
                   unsigned int metric) /* Metric Id. */
```

This function un-register the metric identified within XIAN by its reference `metric` and its associated XCM (or configuration) identifier `id`.

Therefore, as mentioned, a typical XCM implementation looks like to a classical Linux kernel module with specific calls to the functions `register_id()`and `unregister_id()`. In this way, a typical XCM would implement the following functions:

- An ***initialisation function***, as for Linux kernel modules implementation, which is necessary to load the XCM into the Linux kernel and to register the new metric (or several) within the XIAN framework thanks to the XMM's function called `register_id()`.
- A ***processing function***, which is specific to XIAN and enables the reception of *XIAN Nano-protocol Metric Objects* in view to extract the metrics values and to apply the associated formula before updating the metric values through the XMR, the metrics repository of the XIAN framework.
- A ***cleanup function***, as for Linux kernel modules implementation, which is important to unload properly the XCM from the Linux kernel and to un-register the given cross-layer metric (or several) from XIAN thanks to the XMM's function called `unregister_id()`.

Since the ***processing function*** performs operations on the metrics exchanged through the *XIAN Nano-protocol* messages, it follows a specific prototype:

```
void processing_function(unsigned char *saddr,      /* Source MAC */
                         struct metric_msg *metric) /* Metric object */
```

Thanks to this function, once the XCM is inserted inside the Linux kernel and registered within the XIAN framework, the XMM is able to pass the received *XIAN Cross-Layer Metric Object* in `metric` to the right XCM. Moreover, note that the XMM provides an additional information to the given XCM: the MAC address of the source node (indicated by the variable `saddr`), which sent the received *XIAN Cross-Layer Metric Object* (i.e. contained in `metric`).

3.3 XIAN User Space Extended Interface

Several metrics may have to be further combined or refined in order to be meaningful or at least more useful for specific system components. For example, the number of transmitted MAC frames (in bytes) would not be an interesting metric if no time-correlation is introduced to reflect how this metric evolves during the life-time of the wireless communication system. In other cases, the average value of a given metric is more meaningful than an instantaneous measurement. Therefore, XIAN complements the interfaces introduced in Section 3.1 and 3.2 with another API, available in user-space, called the ***User Space XIAN Extended Interface*** (USEI), which provides:

- ***Measurement functions***, which compute metrics resulting from the combination of several elementary metrics taken directly via the USI or from the refinement of an elementary metric (e.g. average values).

- ***Operation functions***, which implement the corresponding mathematical operator required by the new defined calculated metrics (e.g. min or max functions).
- ***Relevance functions***, which implement the corresponding comparator indicating if a significant difference occurs between two calculated values (typically between the previous and the new ones).

The USEI has been implemented as a usual C library to facilitate its use and integration with user space processes. Based on the description of the XIAN architecture and programming interfaces, the following section exemplifies their use with the implementation of the ETX (*Expected Transmission Count* [6]) metric as a ***XIAN Cross-layer Metric module*** (XCM) available as an example of complex metric in the new release of XIAN.

4 Implementation of a complex metric within XIAN

Several cross-layer metrics have been proposed in the literature. De Couto et al. [6,8] have proposed the *Expected Transmission count* (ETX) which measures the bidirectional packet loss ratio of link. Awerbuch et al. [9] have introduced the *Medium Time Metric* (MTM) that selects high throughput paths. Déziel et al. [10] have defined the *available bandwidth*. Iannone et al. [11] have combined the packet success rate, the interference level, and the physical bit rate.

In this section, we specifically illustrate how the new set of XIAN functionalities presented in this paper enable experimenters to easily implement the ETX metric. First, we recall the definition of this metric. Then, we show how to implement this metric within the XIAN framework.

4.1 The Expected Transmission count (ETX)

The *Expected Transmission count* (ETX) calculates the expected total number of packet transmissions (including retransmissions) required to successfully deliver a packet to the ultimate destination. ETX predicts the number of transmissions required using per-link measurements of packet loss ratios in both directions of wireless links.

The ETX value for a route is the sum of the link metrics. ETX is a combination of two measurements: (1) the *forward delivery ratio*, called D_f, which is the measured probability that data packets successfully arrive at the recipient, and (2) the *reverse delivery ratio*, D_r, which is the probability that ACKs packets are successfully received. The probability that a transmission is successfully received and acknowledged is $D_f \ x \ D_r$. Then, ETX is expressed as follows:

$$ETX(link) = 1/(D_f \ x \ D_r).$$

The main issue when it comes to ETX implementation is that D_f and D_r are not directly available locally. The only solution is to exchange delivery ratios among neighbours. These delivery ratio can be determined in various manners: (1) by looking at statistics reported in a per-neighbour fashion by XIAN (e.g., number of data transmission trials, number successful data transmissions), (2) by comparing the number of broadcast packets that have been received from neighbours to those that should have been received, assuming that nodes send broadcast messages every *T* milliseconds.

In this work, to demonstrate how ETX can be implemented as a *XIAN Cross-layer Metric modules*, nodes exchange their delivery ratios for all their out-going links using *XIAN Nano-protocol* messages. They also take advantage of these message exchanges to calculate these delivery ratios using the solution (2) explained in the previous paragraph.

4.2 ETX implementation in XIAN

ETX has been implemented within the XIAN framework as an XCM, called `ETX_XCM`. At the load of the `ETX_XCM`, we configure the XNM to send every T milliseconds the metric D_r in a broadcast message. Thus we create a new configuration identifier and a new metric code for it. The parameter `pf` given to the function `register_id()` is a function called at each reception of the metric ETX, which has the following prototype:

```
processing_ETX(unsigned char*saddr, struct metric msg *metric).
```

The broadcast message contains the values of D_r for the list of MAC addresses provided by the function `insert_mac_to_broadcast()` which is used for each received broadcast message when the MAC address is not present. At the reception of those messages, the `ETX_XCM` stores the D_r metric's value thanks to the XMR by using its interface: `update_xian_stat()`. Then, `ETX_XCM` increments a counter and records it in the XMR. Thanks to D_r and D_f, we can calculate ETX. The calculated ETX values are stored within the XMR in order to provide the metric and the possibility to estimate the quality of a link.

In parallel, a thread is wake up every W milliseconds to recover the counter and to calculate the D_f, thanks to the counter and the expected number of received messages during W milliseconds. D_f is then recorded within the XMR and the counter reset to zero. At last, ETX is recalculated with the new value of D_f and updated within the XMR.

Note that, since the Linux kernel does not manage float values, all the metrics within the kernel space are under the format `struct xian_float` and accessible by the function `struct xian_float get_xian_stat()`, added to the KSI. In the user space, the results are available under the format `float` with the similar function `float get_xian_stat()`, offered by the USI. Moreover, to perform operations on `struct xian_float`, the addition, the multiplication and the division are available as functions provided by XIAN.

Based on this implementation of ETX, the following section explains how the `ETX_XCM` has been used to perform measurements over a real 802.11 ad hoc testbed.

5 Experimental testbed and results

To illustrate the benefit of XIAN to facilitate cross-layering experiments, we present experimental measurements oriented on QoS routing and performed with the `ETX_XCM`, the implementation of ETX with the XIAN APIs.

The testbed we setup was composed of 3 machines equipped with Cisco Aironet Wi-Fi cards equipped by the Atheros chipset and configured to use the 802.11b standard at the bit rate of 11 Mbits/s in ad hoc mode (without RTS/CTS). On each machine, we used `iperf` [12] to generate TCP and UDP traffic and the STAF/STAX [13] framework to automate the experiment runs. We also use a simple program that logs, for each link, the metrics accessed through XIAN. We logged each of the following metrics every δ seconds:

- The ***RSSI*** (Relative Signal Strength Indicator): the wide-band received power within the used channel;
- The ***throughput***: the sum of total data bytes received and sent at the MAC layer within the last δ seconds;
- The ***ETX*** metric measured over 10 beacons (*XIAN Nano-protocol* messages) sent every 200 milliseconds.

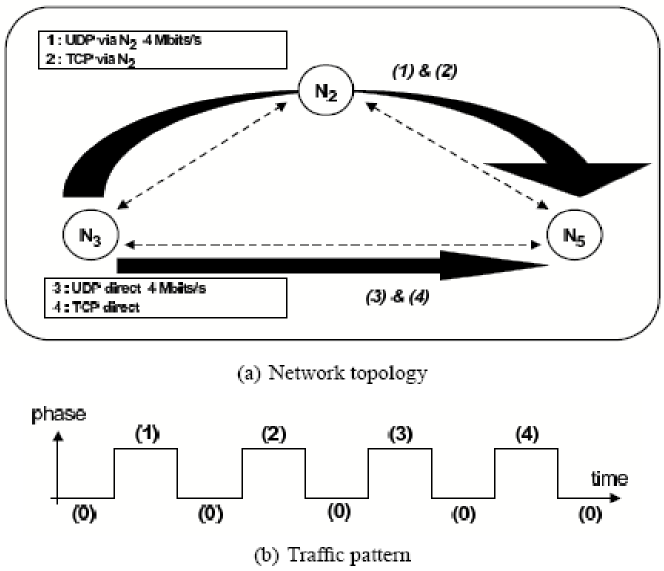

Figure 3. Experimental setup and traffic generation.

Figure 3 presents the triangle network topology that we considered and the different flows that we have generated. The link between the machine N_2 and N_5 is significantly longer than the other ones. The traffic generation sequence used (depicted in Figure 3(b)) is composed of the following phases each lasting 24 seconds:

- ***Phase (0)***: no traffic.
- ***Phase (1)***: UDP traffic between N_3 and N_5 via N_2 at 4 Mbits/s.
- ***Phase (2)***: TCP traffic between N_3 and N_5 via N_2.
- ***Phase (3)***: UDP traffic between N_3 and N_5 at 4 Mbits/s.
- ***Phase (4)***: TCP traffic between N_3 and N_5.

Figure 4 presents the measurements that we have performed. More specifically, Figure 4(b) and Figure 4(e) present the throughput respectively for links $N_3 \rightarrow N_2$ and $N_3 \rightarrow N_5$. We can see that when UDP and TCP flows issued from N_3 are passing via N_2 to reach N_5, high throughput are achieved with average values equal respectively to 3.4 Mbits/s (saturation is achieved) and 2.4 Mbits/s. Whereas when UDP and TCP flows issued from N_3 are passing directly over the link $N_3 \rightarrow N_5$, the average values are equal respectively to 0.1 Mbits/s and 0.1 Mbits/s. These results show that in our case, routing decisions based on hop-count can fail. Cross-layer metrics are then required to achieve acceptable performances.

Figure 4(a) and Figure 4(d) look at RSSI values respectively for links $N_3 \rightarrow N_2$ and $N_3 \rightarrow N_5$. We can see that due to the higher length of the link $N_3 \rightarrow N_5$, RSSI values are lower on this link than those or the link $N_3 \rightarrow N_2$ with average values equal respectively to 12.5 Db and 36.1 Db. An average RSSI of 33.4 Db have been observed for the link $N_2 \rightarrow N_5$. In our case, we could have used this metric to make efficient routing decisions. However, the RSSI does not capture information regarding the congestion and contention levels at MAC layer and thus may not work in larger and more realistic conditions.

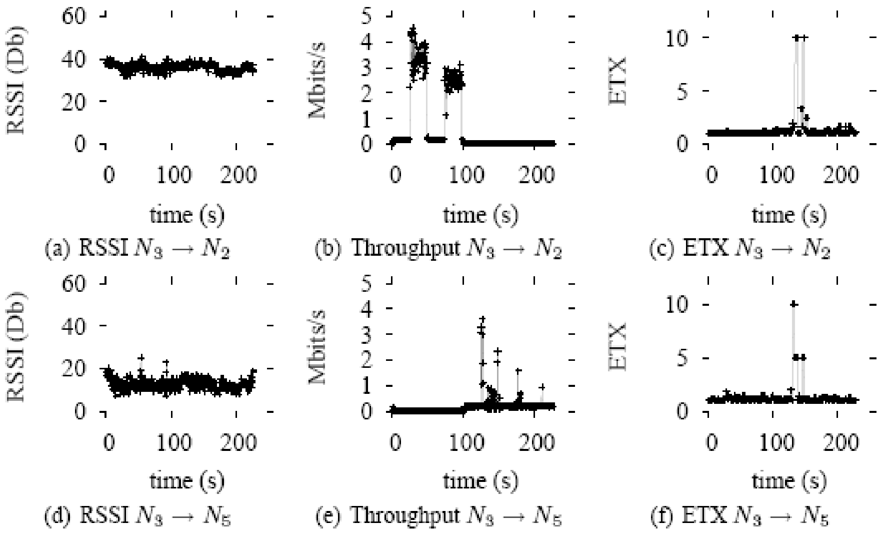

Figure 4. Metric values measured with XIAN.

Figure 4(c) and Figure 4(f) present the ETX values respectively for links $N_3 \rightarrow N_2$ and $N_3 \rightarrow N_5$. We can see that the link $N_3 \rightarrow N_5$ plots ETX values continuously higher than the one of the link $N_3 \rightarrow N_2$ (see especially values for the different phases (0)) which denotes the fact that ETX captures the poor quality of links. In phases (3), as the UDP traffic is totally not able to be transferred over the link, node N_3 runs out of buffer memory. This results in a large amount of drops for the beacons used in ETX computation and thus to high values of the metric (this impact can also be seen for the ETX value of link $N_3 \rightarrow N_2$). Some of the loses of beacons might have also caused by collisions over the wireless medium. We have seen here that ETX captures two crucial factors: (1) the weakness of links due to low RSSI values and (2) the local and neighbouring traffic load.

This experiment illustrates how the ETX metric, implemented within the XIAN Framework through automated *XIAN Nano-protocol* exchanges, introduced in this paper, allows to make efficient QoS routing decisions.

Note that the purpose of this work was to show how to implement the original ETX specifications in XIAN. With XIAN it is moreover possible to explore variants of this protocol, for example by monitoring more exactly the real traffic to estimate transmission count in place of the dedicated ETX probes, thus taking into account the packet size distribution.

6 Conclusions and future work

This paper presented XIAN a cross-layer interface implementation specialized in the building of experimental setups for validating a large variety of use cases of IEEE 802.11 cross-layering and its extensions to support complex user-defined metrics. The extension proposed in this paper turns XIAN into an extensible framework dedicated to the creation of complex cross-layer metrics. Moreover, the decomposition of cross-layer metric definitions in *metric calculation formula* and *protocol exchanges configuration*, translated into software components, facilitates their design and their integration within the XIAN framework.

It is thus possible to define and to implement cross-layer metrics as new components in XIAN that can use both local and neighbouring measurements transparently exchanged with the *XIAN Nano-protocol*. We exemplified its use with the implementation of the ETX (Expected Transmission Count) metric and we provided experimental demonstration of its potential benefits. Finally, we released our code that can be downloaded from [7].

Future work along these lines would include the development of interfaces working in a publish/subscribe manner. This kind of interface may improve further the integration of the MAC and routing layers as it would allow, for instance, reporting of link up and link down events and help the system react more quickly to topology changes. Finally, one could wish to extend the generic APIs to support other chipsets than Atheros in the spirit of the Wireless Tools.

References

1. Corson S., RFC 2501, Mobile ad hoc networking (MANET): Routing protocol performance issues and evaluation considerations. IETF (January 1999)
2. Conti M., Maselli G., Turi G., Giordano S., “Cross-layering in mobile ad hoc network design”, IEEE Computer (February 2004) 48–51
3. Kawadia V., Kumar P.R., “A cautionary perspective on cross layer design”, IEEE Wireless Communication Magazine (July 2003)
4. Aïache H., Conan V., Leguay J., Levy M., “XIAN: Cross-layer interface for wireless ad hoc networks”, In Proc. Med-Hoc-Net. (2006)
5. MadWifi. `http://www.madwifi.org`
6. De Couto D.S.J., Aguayo D., Bicket J., Morris R., “A high-throughput path metric for multi-hop wireless routing”, In Proc. MobiCom. (2003)
7. XIAN. `http://sourceforge.net/projects/xian/`
8. De Couto D.S.J., Aguayo D., Chambers B.A., Morris R., “Performance of multi-hop wireless networks: Shortest path is not enough”, In Proc. HotNets, ACM SIGCOMM (2002)
9. Awerbuch B., Holmer D., Rubens H., “High throughput route selection in multi-rate ad hoc wireless networks”, In: Proc. WONS. (2004)
10. M. Déziel, L.L., “Implementation of an IEEE 802.11 link available bandwidth algorithm to allow cross-layering”, In: Proc. WiMob. (2005)
11. Iannone L., Khalili R., Salamatian K., Fdida S., “Cross-layer routing in wireless mesh networks”, In Proc. ISWCS. (2004)
12. Iperf. `http://dast.nlanr.net/Projects/Iperf/`
13. STAF: Software Testing Automation Framework. `http://staf.sourceforge.net`