Unified Link Layer API: Design and Initial Implementation Results

T. Farnham, M. Sooriyabandara, C. Efthymiou, M. Wellens, K. Rerkrai, M. Bandholz, J. Riihijärvi, P. Mähönen, D. Melpignano, D. Siorpaes, V. Gutiérrez, A. Gefflaut, A. van Rooijen

***Abstract*— This paper describes the design and initial results from implementation work carried out in the European GOLLUM project developing an open, extensible, and unified Application Programming Interface (API) for accessing, querying and controlling all types of wireless links. Key features of this API include a general querying mechanism based on database technologies, and methods for setting up asynchronous notifications regarding changes in link conditions, in a technology-independent manner. Detailed design aspects are described here together with information on the various implementations so far. Initial results show that the proposed ULLA design provides an extensible, scalable, platform independent and power efficient framework enabling seamless link access in various types of systems including embedded devices with limited resources.**

***Index Terms*—abstraction layer, API, link-layer events**

I. Introduction

THE main goal of the GOLLUM project [1] is to propose and develop a software API that will hide connectivity standards heterogeneity behind a common set of functionality applicable to all types of link technologies. The API is termed Unified Link Layer API (ULLA). It is envisaged that such an API stands as a big step towards intelligent radio aware software that will be able to accommodate multiple radio standards in a seamless way.

ULLA will help to resolve the complexity and interoperability problem related to the large number of different APIs [2] and methods used for accessing communication interfaces, especially in the embedded domain. It provides real and useful triggers and handles for different smart, context sensitive and link/network aware applications, thus enabling the development of "cognitive applications".

The ULLA provides an abstraction from specific link technologies to the applications or other link users (LUs) by regarding a link to be a generic means of providing a communication service. In this context, links are made available and configured through link providers (LPs) to permit abstraction from specific platforms and technologies. Link users that benefit from ULLA services include any higher layer protocols, middleware or application software. The high level requirements and main functionalities of ULLA are elaborated further in [1] where the design rationale and requirements are presented in more detail.

The structure of this paper is as follows: Section II describes the overall design of ULLA, with details of its architecture, interfaces, and functionality. Section III includes details of the various ULLA implementations that have been already produced together with some initial performance evaluations while Section IV presents the conclusions and further work.

II. ULLA Design

A. Overview

On top of providing a uniform API, the architecture of the ULLA has been designed to fulfill the following requirements:

1) *Extensibility:* the proposed architecture should be able to easily integrate new link layer technologies, possibly providing new features.
2) *Platform independence:* the proposed architecture should be able to be integrated on multiple software and hardware platforms.
3) *Scalability:* the proposed architecture should be light enough to be used on very limited platforms such as sensor devices.
4) *Power consumption efficiency*: the proposed architecture should not be a major source of battery drain.
5) *User transparency:* the user should also be able to interact with the same API, irrespective of the subjacent technology.

This work was supported in part by DFG, RWTH Aachen, and the European Union (GOLLUM-project, IST-511567).

T. Farnham, C. Efthymiou and M. Sooriyabandara, Telecommunications Research Laboratory, Toshiba Research Europe Ltd, 32 Queen Square, Bristol, BS1 4ND, UK (e-mail: {tim.farnham, costas, maheshs}@toshiba-trel.com). (corresponding author: M. Sooriyabandara, phone: +44 117 9069 830; fax: +44 117 9060 701; e-mail: maheshs@toshiba-trel.com).

M. Wellens, K. Rerkrai, M. Bandholz, P. Mähönen and J. Riihijärvi, Dept. of Wireless Networks, Aachen University (RWTH), Kackertstrasse 9, D-52072 Aachen, Germany (e-mail: {mwe, kre, mba, pma, jar}@mobnets.rwth-aachen.de)

A. Gefflaut, European Microsoft Innovation Center (EMIC), Ritterstrasse 23, D-52072 Aachen, Germany; (e-mail: alaingef@microsoft.com).

D. Melpignano, D. Siorpaes, Dept. of Advanced System Technology, STMicroelectronics v. Cardano, 1 - 20041 - Agrate Brianza - Italy (email: {diego.melpignano, david.siorpaes}@st.com).

A. van Rooijen, Materna GmbH, Vosskuhle 37, 44141 Dortmund, Germany (email: arthur.vanrooijen@materna.de)

V. Gutiérrez, Dept. of Communications Engineering, University of Cantabria, Avda de los Castros S/N, 39005 Santander, Spain (e-mail: veronica@tlmat.unican.es)

ULLA provides an abstract view of link layers to ULLA clients (link users) to facilitate a uniform way to access the wide range of existing link layers, independently of their implementation. In order to manipulate these abstractions, a specific query language (i.e. ULLA Query Language (UQL) – a subset of SQL) is used. A ULLA query specifies a request made by an application to retrieve information about a link layer synchronously (i.e. information request) or asynchronously (i.e. request notification). Finally a command is a request specifying an action that should be executed to modify a link layer state. An overview of the basic elements composing the ULLA architecture is presented in this section.

B. *ULLA Interfaces & functionality*

ULLA provides a set of unique and well defined functions used by the Link Users (LUs) to access relevant information and issue commands to Link Providers (LPs). It is worth noting that ULLA currently defines interfaces with both LUs and LPs and core functionality (namely, command processing, event processing and query processing) in order to achieve the above objectives (as shown in Fig. 1). This kind of separation makes the distinction between different software domains, which are uncorrelated with each other, clear: one domain is intended for application developers, who are willing to control wireless adapters as a whole with a very high level of abstraction. The second domain includes device manufacturers who are more interested in providing a ULLA compatible software layer (as an LLA – Link Layer Adaptor) over their technology and OS specific device drivers. In the LLA context, the API definition is more tightly coupled with the concept of a LP mapped to a physical network adapter.

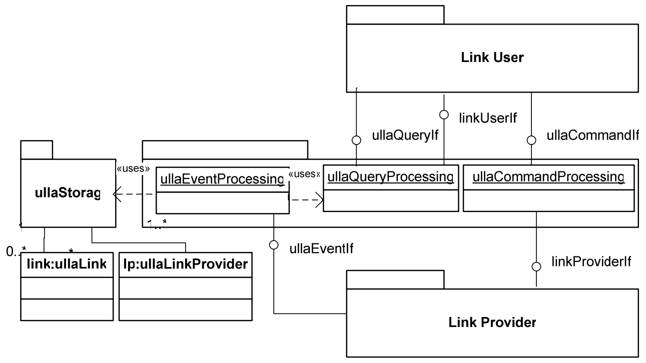

Fig. 1. The ULLA Architecture

The interfaces provide functionalities that can be logically split into the following:

1) Sending commands to the ULLA Core (either synchronously or asynchronously)
2) Request notifications from the ULLA Core
3) Querying the ULLA storage system for retrieving information.

The fine grained definition is still under discussion, but general requirements and design have been identified. Commands directed to the core aim to satisfy some high level application requirements (i.e. bandwidth, delay, link quality etc.). Such commands may be sent in a synchronous or asynchronous way in order to accommodate every application need and design. In particular, asynchronous commands are suited to long running tasks such as the request for wireless LAN infrastructure scanning.

LU interfaces offer the means to notify the application when certain conditions are met. Such conditions are specified via UQL, whereas asynchronous notifications are realised by means of a callback mechanism: the LU provides ULLA with the implementation of a well prototyped function which is called by ULLA upon satisfaction of a registered request. This relieves the application from needing to resort to polling mechanisms, which are inefficient and consequently have high resource consumption.

C. *Link User Interface*

In order for the LU to use ULLA it needs to register itself with it. The function *registerLu* will be used to pass a *LuDescr* structure (containing name, description and version) to ULLA. ULLA will return a unique LU ID (*luId*) upon successful registration. A LU must call the *unregisterLu* method when it stops using ULLA functionality. It needs to pass the *luId* it received when registering. It is recognized that the LU registration procedure has critical platform security implications, which are currently being discussed.

1) Command Handling: The LU has the ability to send commands to the LPs and Links via the ULLA Core. The *doCmd* method is used to execute a synchronous command. The LU needs to pass its *luId* and a *cmdDescr* command description structure. This structure contains the name and parameters of the command to be executed. It also passes the ID of the LP and Link that has to execute the command.

The *requestCmd* method is used to execute an asynchronous command. In addition to the previous parameters there is also a need to pass command descriptor *requestCmdDescr*. This structure contains a pointer to a callback routine to handle the result of an asynchronous command. In order to execute a command periodically a count and a time interval can be set in the *requestCmdDescr* structure. The function will return a unique *cmdID* upon successful queuing of the command. The LU can also cancel a queued command by using the *cancelCmd* method. It needs to pass the *luId* and the *cmdId* as parameters.

In case several LUs issue conflicting commands to the same link, an arbiter called a Link Manager (LM) can apply suitable policies to resolve conflicts. An interface is defined in the current design to ease the insertion of a third-party LM in the ULLA SW architecture. LM details are however outside the scope of this document.

2) Query Processing: The LU can retrieve information from the ULLA Core or LPs by doing an information request. This is done with the *requestInfo* method. With the *requestInfo* method, the LU needs to pass the *luId*, a query string and a pointer to an *ullaResult* handle. The data in the result handle can be retrieved using special *ullaResult* assessor functions. After all the data has been retrieved the LU needs to call the *ullaResultFree* method to free the memory. Sometimes it is necessary to be informed when a certain attribute of a Link is changed. This can be done by requesting a notification. This works in a similar way to the *requestCmd* method and the *requestNotification* method is being used for this. In addition to these parameters, a *RequestNotificationDescr* structure has to be passed, which contains a pointer to a callback function, a count and period. The function will return a *requestNotificationID* when the notification has successfully been queued. This notification request can be cancelled by the *cancelNotification* method.

D. Link Provider Interface

This interface provides the means to notify link events received from the LP to ULLA and to pass commands received from LUs to LPs. When registering using the *registerLp* method it needs to pass the structure *LpDescr*. This structure will contain information about the LP and also a pointer to its interface. Upon successful registration the ULLA Core will return a unique *lpId*. LP registration does not automatically imply that there are also active links. The ULLA Core needs to do an active scan to find all available active links. When a LP is unloaded it must call the *unregisterLp* method. It needs to pass the *lpId* it received when registering.

1) Command Handling: Commands are passed (by the ULLA Core) from the LU to the LP. The *execCmd* method is used to pass a synchronous command to the LP along with the link that has to execute the command. The ULLA Core uses the *lpId* in the *cmdDescr* to address the appropriate LP. The *cmdDescr* contains all other information the LP needs. In the current API definition, the LP has no special function to pass an asynchronous command. The handling of asynchronous commands is completely handled by the ULLA Core. From the LPs point of view the *execCmd* method is used in both cases.

2) Query Handling: The ULLA Core will normally first retrieve data from the ULLA storage. When newer information than that which is stored is needed, ULLA Core can retrieve this information by using the *getAttribute* method. As parameters, ULLA Core needs to pass the Link ID, the identifier of the attribute, a data qualifier and a pointer to where the result has to be stored. With the data qualifier the LP can tell if the value should be a *HARDCODED*, *THEORETICAL*, *ESTIMATED* or *EXACT* value. The memory to store the attribute is allocated by the LP. This memory then has to be freed by the ULLA Core with the *freeAttribute* method when no longer required.

When the LU requests a notification, e.g. when a certain attribute has changed, the ULLA Core will call *requestUpdate*. The ULLA Core needs to supply an event handler to process all the update events. With the *RequestUpdateDescr* structure, information regarding the requested attribute, the count and time interval the update needs to be sent is passed to the LP. The update request can be cancelled with the *cancelUpdate* method. In order to track all different update requests sent to the LPs, ULLA will generate a unique ID for all requests.

III. Implementation Results

In order to validate the ULLA design four distinctive implementations have been developed for different application usage scenarios. The implementation is still ongoing and the initial results will be used to refine the ULLA design and further validation and evaluation will take place. The ultimate goal of ULLA evaluation will be to validate it against all known requirements and to assess the capabilities and limitations with respect to different application domains. Each platform uses a different ULLA core implementation and these are:

1) *Real-time multimedia platform:* Linux user-space based using SQLite [6] database backend
2) *Multimedia streaming platform:* Linux user-space based using Postgres [7] backend
3) *Connection management platform:* Windows CE kernel space using proprietary storage
4) *Wireless sensor network platform:* Implementation in TinyOS on Telos motes with proprietary database.

Various lessons have been learned from this implementation experience for the requirements of the different applications and these are discussed further for each platform.

A. Real-time Multimedia Platform

The real-time multimedia platform uses IEEE 802.11g and Bluetooth link technologies with a Voice over IP (VoIP) application. This application can jointly adapt RTP packetization, vocoder parameters, max number of MAC retries and application-level Forward Error Correction (RTP-FEC, RFC2733) to the link characteristics. The platform is based on the Nomadik™ ARM-based application processor and the low-power STL4970 WLAN chipset for mobile phones, both available from ST Microelectronics [4].

The ULLA Core and the Link Providers are in the user space. A so-called “ULLA library” is provided to applications that wish to use ULLA services. Such library communicates with an ULLA daemon, which is responsible for loading the link providers. The communication between the ULLA library and the ULLA daemon uses System V IPC, in particular one message queue for commands and notifications. The ULLA daemon runs as super user and dynamically loads link providers. For ULLA storage functionality, two database backends exist: mySQL [5] and SQLite [6]. The measured performance of this prototype shows that querying link information takes around 2ms.

Ullad is the ULLA daemon, whereas the ULLA Core includes the libraries *libullacommon* and *libulla*. The DB backend accounts for the most memory occupation, but this can be optimized if needed. A library named *liblpcommon* has been created to ease the adaptation of existing drivers to the ULLA architecture. Based on this library, a WLAN Link Layer Adapter has been developed in just 16 kBytes.

The lessons learnt from this implementation are that database backends have to be selected carefully for embedded systems where memory footprint is an issue. MySQL does not

TABLE I
CODE FOOTPRINT FOR ULLA IMPLEMENTATIONS

	Real-time Multimedia Platfrom Linux – Nomadik ™	Multimedia Streaming Platform Linux – PDA / Laptop	Connection Management Platform Windows CE - smartphone	Wireless Sensor Network Platform Telos Motes – sensor device	
Enhanced Functionality	Multiple LLA	Asynchronous commands, join queries, multiple LLA, inheritance, Java API and statistical operations.	Multiple LLA	ROM	RAM
ULLA User Library	43	46 - LU and LP interface	19.5		
ULLA Core	37 – ulla daemon 406 – sqlite database	107 – postgres database 52 – System V IPC libs (Postgres backend 2.6M)	72.0	3.6	0.6
IEEE 802.11 LLA	16 lla & 26 lib	51 lla & 25 wireless extensions lib	43.5		
Bluetooth LLA			30.0		
GPRS LLA			25.5		
IEEE 802.15.4 LLA				1.5	0.1
TOTAL	528k	281k plus database (2.9M)	190.5k	5.1k	0.7k

support multiple processes using the same DB and requires at least 10 MB memory even for an empty DB. Polling link information in order to gather statistics is a processor intensive task which needs to be designed carefully. It is easy to end up with a system that calculates complex statistics that no link user will ever use. It is preferable for LUs to provide requirements to ULLA with respect to the type of measurements and statistics that need to be computed.

B. Multimedia Streaming Platform

The multimedia streaming platform uses IEEE 802.11 between client and server devices and configures the radio channel (and other link settings) as well as video transmission rate according to the observed performance and channel monitoring information. The platform is currently based on laptop PCs running Linux, but will also use Windows CE and Linux embedded platforms based on the Intel PXA255 processor chipset. The performance in terms of ULLA command execution overhead (for a synchronous command) for a Toshiba Tecra M3 laptop is at worst case 280µs and for a 400MHz PXA255 is 1120µs. The reason for this overhead is the fact that the command involves a context switch between the application process and the LP process. This context switch is required as the LP process runs with sufficient privilege to perform the command (usually root privilege) and the application process runs with basic user privilege. The ULLA command processing functionality permits the application to execute the LP command only if it is sufficiently privileged to do so. In the current Linux implementation a System V IPC mechanism is used for this purpose. More detailed performance assessment of information retrieval and statistical operations is ongoing, but typical queries have worst case delays of less than 1ms.

For development of the multimedia streaming platform, it is not appropriate to rely solely on instantaneous values of performance (such as activity and noise levels); these can be misleading as they can fluctuate very widely. It is also undesirable to have generic statistical parameters without knowing how the statistics are computed as these too can be ambiguous. Instead it is necessary to control how parameters are measured with a reasonable degree of flexibility. For example, for assessing channels or links to use for dynamic channel or link selection schemes it is necessary to determine the suitability of the channels and links (such as their available throughput and quality of service) over some period of time in order to determine the best channel or link.

Channel objects were created to handle the channel information obtained by monitoring. This information consists of activity (number of link layer packets transmitted), total data bytes transmitted, etc. This information was useful for dynamic channel selection in order to avoid congested or noisy channels. However, reliably calculating the available bandwidth on a shared link is still an ongoing issue. In decentralised access control schemes (such as the IEEE 802.11 CSMA/CA MAC), passive observation of the traffic may not reflect the actual available throughput as other devices can be forced to defer their transmissions whilst still maintaining fairness. Therefore, channel activity is only an approximate indication of the available throughput.

Database triggers are used in this implementation to allow asynchronous notifications. Each time a database update occurs, a trigger function is called to check whether a notification should be sent. Most databases support this concept. Postgres also supports dynamic loading of trigger functions and can allow for more complex statistical or other data manipulation functions to be added relatively easily. However, the Postgres database has a large footprint that is not suitable for smaller embedded devices and so the Polyhedra ™-Flashlite database is being considered as an alternative (having a code footprint of 500-600 kBytes) [3].

C. Connection Management Platform

The connection management investigates the usage of the ULLA for intelligent and automated connection management. Establishment of network connections becomes more and more complicated for end users as the number of supported network connection increases on mobile devices. It is now current to see devices with at least 3 possible wireless technologies (GPRS, Wifi and Bluetooth). The targeted platform supports a variety of different link technologies (GPRS, UMTS, IEEE 802.11 and Bluetooth) and uses the ULLA to access link characteristics. The link characteristics are matched against application requirements in order to select the most appropriate technology at any point in time and hence guarantee the best networking experience to users.

From a hardware point of view, this platform uses a Pocket PC running windows mobile 2003 or windows mobile 2005. The code could also run on a Smartphone class devices. The ULLA core is implemented as a stream driver that is easily accessible by all applications running on the device. The ULLA user library provides the defined Link User API and maps the function to IO/Control calls to the driver implementing the ULLA core. In the ULLA core, 3 libraries implement respectively the UQL parser, the storage and the Link Provider interface. Asynchronous notifications are supported by message queues running in the Link User application and handled by a dedicated thread.

Measurements on this device show that request duration increases from 500 to 2500µs, depending on the number of attributes in the request, the number of available links and the validity used in the storage. As an example, a request returning 4 attributes, at a 100 ms validity and 5 links present has been measured at around 500µs. For the notification latency, measurements have shown that almost 90% of the notifications have a latency of less than 1.5 ms. This result is quite encouraging since the CPU is only clocked at 400 MHz. Finally, the measurements have also shown that ULLA has a negligible impact on the battery consumption of the device, even in the case of frequent notifications. The main reason for this behaviour is that the wireless interface itself dominates by far the amount of energy consumed.

D. *Wireless Sensor Network Platform*

The Wireless Sensor Network (WSN) platform consists of wireless sensor devices (motes [8]) and gateway sensor nodes responsible for the connection to the end-user PC. The applications built on top of the ULLA can either be seen as local Link Users (running on the motes) or remote Link Users (running on the end-user PC). The selected applications on this platform are a routing agent (local user) and a smart home monitoring (remote user) application. The routing agent could use the ULLA Storage to save link quality values in order to use them as valuable information for a routing protocol. The smart home monitoring uses the gateway nodes to enable access to the WSN measurement information while each node is running the intelligent routing agent.

The platform is based on the TinyOS operating system. The targeted wireless sensor devices are Telos and MicaZ which use IEEE 802.15.4 link technology and Mica2 which use ChipCon1000 link technology. Further information on this platform and its components can be found in [9]. Due to the resource scarcity in the motes, a reduced version of ULLA Core architecture which has a small footprint and lightweight concurrency was implemented. Some of the important characteristics of TinyOS which make it lightweight are the lack of dynamic linking and memory management. Every module is seen as a component and the only way to communicate two components is through well defined interfaces. The manner the component joining skeleton (through interfaces) is defined, is specified by configurations schemes. Therefore, the ULLA Core and the Link Providers are considered as the components which will be statically linked into one application during the compilation of the TinyOS runtime. In addition, LLAs cannot be dynamically loaded on this platform. These will be independent modules, for instance LLAs for two different technologies which depending on our configuration will be added or not. Although dynamic memory allocation is not allowed in TinyOS, it is allowed to allocate a static chunk of RAM which can be parceled out dynamically. In this sense, the size of a query sent from a remote user can be flexible by splitting one query into multiple messages. Similarly, multiple queries can be allocated in the motes dynamically.

One of main issues in the WSN area is the memory constraint. Table I shows that the ULLA is suitable for memory constraint devices. For example, the mentioned Telos motes have 48 kBytes ROM and 10 kBytes RAM.

IV. Conclusions

This paper has focused on the design of ULLA and initial implementation results from the various ULLA implementations developed within the GOLLUM project. The specific observed advantages of the ULLA approach is that basic link characteristics are abstracted at the same level for all links so that an application program does not need to use any link specific command or information requests. Asynchronous notifications are supported to enable notification to the applications about degrading performance (for example signal strength measurements), information that could be used to trigger handovers or further radio resource management operations such as rate adaptation and channel selection. Overall the initial results have shown that the ULLA design described in section III meets the basic design requirements: extensibility, platform independence, scalability, power consumption efficiency and user transparency.

Further work is ongoing to evaluate performance and refine the ULLA definition to meet all of the requirements. The final specification will be made publicly available together with a partial reference implementation for selected platforms.

References

[1] T. Farnham, A. Gefflaut, A. Ibing, P. Mähönen, D. Melpignano, J. Riihijärvi and M. Sooriyabandara; "Toward Open and Unified Link-Layer API"; IST Mobile Summit 2005, Dresden, Germany, June '05

[2] "Generic Open Link Layer API for Unified Media access", D2.1 state-of-the-art http://www.ist-gollum.org/docs/Gollum_D21_State_of_the_Art_Public.pdf

[3] Polyhedra flashlite www.enea.com [to be released in 2006]

[4] Nomadik www.st.com/stonline/books/pdf/docs/9306.pdf

[5] mySQL website: http://www.mysql.com/ [visited 17/01/2006]

[6] SQLite website: http://www.sqlite.org/ [visited 17/1/2006]

[7] PostgreSQL website: http://www.postgresql.org/ [visited 17/01/2006]

[8] moteiv website: http://www.moteiv.com/ [visited 18/01/2006]

[9] J. Polastre, R. Szewczyk, D. Culler, "Telos: Enabling Ultra-Low Power Wireless Research", Proceedings of the Fourth International Conference on Information Processing in Sensor Networks: Special track on Platform Tools and Design Methods for Network Embedded Sensors (IPSN/SPOTS), April 25-27, 2005