This full text paper was peer reviewed at the direction of IEEE Communications Society subject matter experts for publication in the ICC 2007 proceedings.

Unified Link-Layer API Enabling Portable Protocols and Applications for Wireless Sensor Networks

Krisakorn Rerkrai, Janne Riihijärvi, Matthias Wellens, and Petri Mähönen
Department of Wireless Networks, RWTH Aachen University
Kackertstrasse 9, D-52072 Aachen, Germany
Email: {kre, jar, mwe, pma}@mobnets.rwth-aachen.de

***Abstract*— Considerable amount of research work has recently been performed in the area of wireless sensor networks. In this paper we address the diversity of programming interfaces for link layer information as well as sensor measurement data as one of the few critical problems that have not in our opinion been addressed thoroughly enough. We propose the use of the Unified Link-Layer API (ULLA) for wireless sensor networks. It offers a common interface to retrieve link layer information independently of the deployed radio technology considerably simplifying development process of link-aware protocols and applications. The ULLA can be used not only by applications running locally on sensor nodes but also remotely by applications running in an end-user device. Additionally, ULLA allows to retrieve sensor readings via the same convenient interface. The interface offers a powerful notification mechanism, which can be used to monitor the sensor network and trigger certain actions when either the communication environment changed, for instance an ad hoc route disappeared, or the sensor readings pass a threshold, e.g. usable for fire-alarm. These features essentially generalize the approach taken by TinyDB to include link-layer information as well, and further extend the abstraction of the sensor network as a database. A prototype implementation of ULLA on Telos B motes has been made to demonstrate the feasibility of the ULLA approach. The footprint of the implementation is very acceptable, and performance offered sufficient even for protocols taking per-packet decisions based on link conditions. The ULLA implementation is available from http://tinyos.cvs.sourceforge.net/tinyos/tinyos-1.x/contrib/rwth/mobnets/**

I. Introduction

During the last years wireless sensor networks have become an area of intense study. Numerous protocols and algorithms have been developed, simulated and implemented for all layers of the protocol stack. However, especially most of the implementation and prototyping efforts have been very platform-specific in nature. While basic unifying abstractions for functions such as message sending or receiving have emerged (for an overview of these, see [9]), especially techniques for accessing link-layer information have remained specific to particular wireless technologies used. In practise this means that most of the implementations of important clustering, link-aware routing, and topology control algorithms, to name a few, are not portable across different platforms and link-layer technologies. In fact, their implementation on present software platforms requires register-level programming involving knowledge on the specific radio chips used. In our opinion this is a major roadblock for emergence of "real-life" wireless sensor network applications.

In this paper we introduce a version of the *unified link-layer API* (ULLA) for wireless sensor networks, offering technology-independent way to access and control link-layer information. ULLA has originally been designed in the European Union research project GOLLUM [1], [14], [5], where the focus has mostly been towards more high-end embedded devices, but, as we shall show, it can be modified to be a powerful tool for WSN use as well. It should be emphasised that ULLA is, as the name implies, an *API*. We specifically offer a way to abstract away the differences between present and upcoming link-layer technologies, and do *not* attempt to define a complete *unified link-layer* (for an example of the latter approach see [7]). From practical point of view several different link-layer technologies for WSN use will in any case exist, and in our opinion offering a unified way to build link-layer aware protocols and applications on top of those while ensuring portability is the problem to tackle.

In addition to the ULLA presented in [5] we also describe some extensions to the basic architecture to allow the integration of queries for link-layer information with queries for actual sensor data in TinyDB [8] fashion. This kind of unification retaining the "sensor network as database" point of view should considerably simplify the work of the application writers by reducing the number of abstractions they would have to be familiar with. ULLA takes this unification to its logical conclusion by allowing same programming primitives to be used both for code running on a single node, as well as for code accessing the sensor network as a whole from a gateway device. Use of a single API for these tasks also inherently improves the possibilities of making heterogeneous sensing and link-layer technologies to work together. To summarize this discussion, the contributions made by our sensor ULLA design are:

1) Generalization and abstraction of link-layer information in technology-independent manner.
2) Integration of link-layer and sensor information into a single database abstraction.
3) Introduction of remote access to link-layer information as a strong enabling technology not offered by earlier approaches such as TinyDB.

1-4244-0353-7/07/$25.00 ©2007 IEEE

The prototype implementation shows that all of these can be introduced without high overhead and that good performance can be achieved.

The rest of the paper is organized as follows. We start with giving some more insights about our motivation for the presented work in section II. In section III we describe the general ULLA architecture which is common to all platforms it has been implemented on. The prototype implementation of ULLA on TinyOS is then described in detail in section IV together with some modifications to the basic architecture to make the implementation for WSNs feasible. The last part of section IV presents first but promising performance evaluation results. Finally, in section V we discuss some potential application domains which would benefit from ULLA, before concluding the paper in section VI.

II. Motivation: Why a Unified Link-Layer API for Sensors?

Before going into the implementation details of the Unified Link-Layer API for TinyOS, we shall briefly discuss the motivation in providing a powerful API like ULLA to access link-layer information. After all, sensor networks being very resource constrained all the functionality and portability offered by an API like ULLA might seem like an overkill. Additionally, present WSN nodes typically have only one radio interface, reducing the functionality required from a link-layer API even further.

We argue strongly that even in the single-radio case adoption of ULLA would bring significant benefits. The technology-independence of the API allows link-aware programs to be written in a portable manner, significantly reducing the overhead of porting a link-aware protocol implementation from one platform to another. Additionally, availability of the same API both in "high-end" wireless devices (such as laptops, PDAs and smartphones) and sensor platforms reduces the learning curve for programmers switching from one platform to another. In commercial development settings this kind of advantage should not be underestimated. We also argue that WSN platforms are not going to be exclusively using a single radio interface. Compact gateway designs with multiple wireless interfaces are already being introduced [4], [10], and the suggestions on using separate wake-up radios to coordinate the medium access amongst the nodes before utilizing the "main" radio for actual data transfer are gaining momentum.

Finally, in many cases we expect the adoption of ULLA as part of the functionality offered by the operating system to actually improve performance instead of degrading it. This is because many of the powerful functions offered (such as asynchronous notifications) would most likely be implemented by programmers themselves using application-layer primitives due to their necessity. Letting a single operating system entity handle these functions makes it possible to optimize the implementation of these features to the fullest, for example exploiting different hardware-specific solutions such as programmable hardware timers if available.

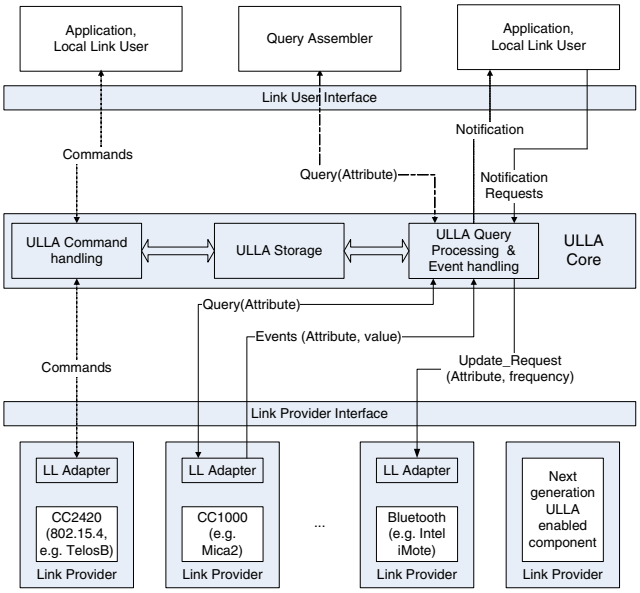

Fig. 1. Architecture of the ULLA implemented on sensor nodes.

III. ULLA architecture

The ULLA was initially prototyped on more powerful devices but characteristics of embedded devices were considered throughout the whole design process. We shortly introduce the main ULLA architecture for convenience of the reader and continue with the WSN-specific adaptations. Further architectural details and design rationales are available from [5]. In this paper we focus specifically on the extensions and enhancements developed to apply the ULLA concept to WSNs.

ULLA design follows a modular approach as shown in Fig. 1. The main component in the ULLA framework is *ULLA Core* in the middle of the figure. It is an intermediate entity connecting *Link Providers* (LPs), an abstraction of the sensor radio interface, and *Link Users* (LUs), the applications[1] taking advantage of ULLA. The two interfaces between these three architectural levels form the Unified Link-Layer API.

Link Layer Adapters (LLAs) are the major part of the software implementation of an LP. They implement the LP interface towards ULLA Core and cope with technology-specific ways to retrieve lower layer information. In the first step LLAs will be implemented as wrapper units utilizing proprietary functions offered by existing link layer and radio components. Later on ULLA-enabled components might incorporate the LLA-functionality from the beginning avoiding the need for an extra block.

In case of sensor networks the application using the sensor readings will often be placed in a PC with the gateway node connected to it instead of running locally on the nodes. In order to use ULLA also in such scenarios we differentiate

[1]In this context we do not only foresee link users being applications working on layer seven but also entities such as routing daemons may benefit from ULLA.

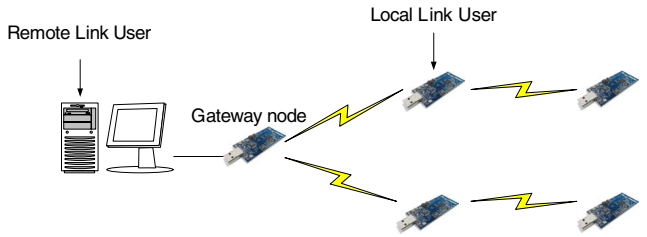

Fig. 2. ULLA system structure and the two types of supported Link Users.

between two kinds of LUs as shown in Fig. 2. *Local LUs* run as the name says locally on the node and represent LUs as used also with the ULLA on desktop systems. The newly introduced *Remote LUs* are connected to the sensor network via gateway nodes but run on PCs or other devices including an end-user interface. Remote LUs can retrieve information from all nodes participating in the connected WSN or from single devices by addressing certain nodes. The addition of remote access to the ULLA is an enhancement compared to the versions available [5], [13] for high-end devices tailored towards the special needs of wireless sensor nodes.

A. Data model

ULLA works with *Links* which are described in classes following an object oriented approach. The most important base class *ullaLink* has to be supported by every LP and groups common attributes in an abstract way. For instance, throughput or latency are available in each L2 technology and thus are included in the mandatory base class. More technology-specific attributes, such as the preamble length used in B-MAC, that might only be available in one system, are put to separate classes. LUs can choose the abstraction level to work on, e.g., ad hoc routing agents might only be interested in a generalized link quality metric, which is part of ullaLink, but not in the specific implementation details, which are hidden in the LLA. Other diagnosis tools will be aware of additional technology-specific classes and use those still utilizing the same interface.

Attributes which are not configurable per link but more related to the hardware, such as transmit power, are part of LP-classes leading to the second mandatory base class *ullaLinkProvider*. One important attribute in that class, the *lpId*, is used as address inside the ULLA-enabled WSN. It is initialised as the *nodeID* and can be used by Remote LUs to query measurements taken at certain nodes.

In addition to networking information the extended WSN-ULLA also allows to access the sensor measurement data via the same interface. Specific classes are again introduced that model the sensor readings. Standard measurements, such as temperature or humidity, are part of the base class *sensorLink* which can then be extended with additional classes to provide access to sensors to present in common platforms. As the sensor configuration differs between nodes and platforms none of those measurement classes are mandatory. However, each node has to support one mandatory class that includes an attribute describing the supported sensing capabilities classes in form of a simple but efficient bitmask.

The object oriented model inherently ensures extensibility as the classes need not to be known at compile time. The Link-Layer adapters can register new class descriptions at runtime, ensuring that the implementation of the basic ULLA components need not be constantly updated as new sensors or wireless technologies come available. Additionally, the combination of link layer and sensing information in one interface using a single data model drastically eases the application development.

B. Link User interface

The initial ULLA design [14], [5] was focused on a very extensible architecture enabling the dynamic introduction of new technologies and classes without any change to the API-calls. Therefore, also the extension of the data model towards sensing information does not require any change in the API itself. We present the two main parts, LU and LP interface, in concise form in order to show the API-flexibility and as background for preceding implementation and application sections.

The LU interface offers three main features: Queries, Notifications and Commands. Queries are used to retrieve single pieces of information using the *ullaRequestInfo()*-call. Since we have adopted the database view both in the node and throughout the network, the basic argument of the query is an SQL query string. Since full SQL is rather complex, for parsing and processing simplicity we have specified a compact subset of SQL to be used in the queries, called the ULLA Query Language, or UQL for short. An example query might be *SELECT temperature FROM sensorLink WHERE lpId=4*, which could be used by a Remote LU to read out the temperature-measurements taken at sensor four. Local LUs might be interested in networking information: *SELECT linkId, rxSignalStrength, rxLinkQuality FROM ullaLink* results in a list of signal strength and link quality values for each available link. Using such a query the best available link could be chosen.

Notifications are closely related to queries because they also use UQL-statements, which usually incorporate a condition. LUs use the *ullaRequestNotification()*-call to ask for notifications. These notifications can either be *event*-based, or *periodic* for updating measurement values at certain intervals. Pending route changes could be detected by registering for a notification using a query such as *SELECT linkId, rxLinkQuality FROM ullaLink WHERE (rxLinkQuality $<$ 40) AND (linkId = 5)*. Remote LUs might want to be notified if the humidity measured by any node in the network is above a threshold: *SELECT humidity FROM sensorLink WHERE humidity $>$ 80*. As no lpId is given the notification is flooded in the whole WSN and all nodes are evaluating it each time when new measurements are available. In order to avoid overloading the network such operations should be carefully used and cancelled as soon as not needed anymore.

The ULLA storage is also used to optimize the efficiency of the query processing for standard queries as well as for notifications. Several consecutive requests asking for the same attributes in a short time can be answered using values saved in the ULLA storage. For this purpose the whole ULLA implementation offers one *validity*-parameter that describes the allowable staleness of any measurement result. This feature might be extended so that the timeliness can be given as an additional query-parameter or simply specified in the WHERE-clause of the UQL-query.

The last feature offered via the LU interface are commands which enable LUs to configure LPs but also to start procedures such as *scanAvailableLinks* or *connect*, the former one being an LP-command and the latter one being a link-command. ULLA includes the *ullaDoCmd()*-call as part of the LU interface for issuing such commands.

C. Link Provider interface

The LP interface is structured similarly as the LU interface but works on the level of single attribute values instead of rows in tables. LUs requesting latency will get one value per available link, which means all entries in the latency-row in the ullaLink-table. The respective LP interface *getAttribute()* is addressed to one LP, asks for one single link and finally results in one attribute value. Thus, ULLA Core performs queries, which do not limit their scope using the WHERE-clause, by going through all available links using separate *getAttribute()*-calls.

Also, each attribute involved in a notification is handled separately. ULLA Core calls *requestUpdate()* in order to ask the respective LPs to signal events when new values are available. Changes in such single attributes may at the end not trigger a notification because the involved condition is only half-fulfilled.

Commands, the *methods* in the object-oriented class abstraction, are handled likewise. LUs issue the command using *ullaDoCmd()* and ULLA Core forwards it to the requested LP utilizing the *execCmd()*-call, which is part of the LP interface.

IV. Implementation

We have implemented ULLA in TinyOS 1.1.15 [2] using the nesC programming language [16] in order to evaluate the performance of the proposed design. We ported ULLA to the event-based TinyOS architecture by defining respective events for completion of most API-calls. For instance, an LLA will signal the event *getAttributeDone()* when the requested attribute is available.

We successfully tested our ULLA implementation on the Telos B platform [12], which uses a TI MSP430 microcontroller, clocked at 8 MHz, 10 kB of RAM, 48 kB of Flash, and the Chipcon CC2420 radio chipset [6]. To verify portability we checked that ULLA compiles for Mica2 platform [3] as well as the TOSSIM [15] environment. The implementation is available from http://tinyos.cvs.sourceforge.net/tinyos/tinyos-1.x/contrib/rwth/mobnets/.

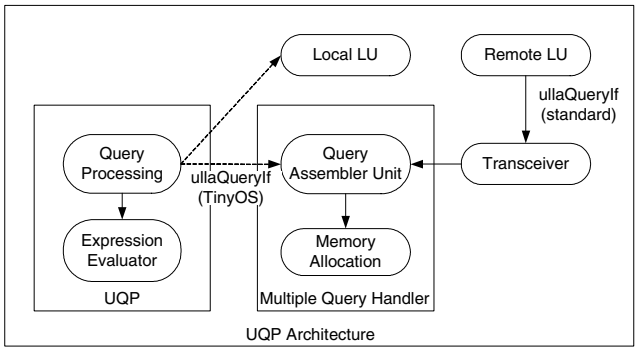

Fig. 3. UQP architecture.

A. ULLA Core

Due to the resource constraints of the motes, the original design of ULLA had to be trimmed down to a reduced version which has a small footprint and lightweight features. 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 two components can communicate through are well defined interfaces. The application can use ULLA by simply wiring the ULLA component in its configuration file. The ULLA Core and the Link Providers are statically linked into one application during the compilation of the TinyOS runtime. The application does not need to know which Link Provider it has to use. Instead, the appropriate Link Provider will be automatically wired to the ULLA Core depending on the selected platform. The Local LU is wired to the ULLA Core via the *ullaQueryIf* and *ullaCommandIf* TinyOS interfaces while the Remote LU can communicate with the ULLA Core via the standard query and command interfaces offered on the end-user PC.

The ULLA Query Processing, or UQP for short, handles queries and notification requests sent from both Local LU and Remote LU. The only difference is that the size of a query sent from the Remote LU is undefined and thus has to be kept flexible. This is achieved by splitting a single query or notification request into multiple messages. Therefore, on the destination mote we use a Query Assembler Unit (QAU) to gather these messages and reassemble the original query. The QAU is used to allocate the memory, gather the messages and check whether or not the query has been completed. Although dynamic memory allocation is not supported in TinyOS, it is allowed to allocate a static chunk of RAM which can be parceled out dynamically. This way, UQP can handle multiple queries at the same time. QAU uses the ullaQueryIf interface and then passes the queries to UQP. From this point, QAU can be seen as another Local LU which uses the same interface provided by the UQP. The respective architecture is depicted in Figure 3.

The ULLA storage is implemented on the motes by statically allocating RAM since a database application requires much more resources than are affordable. Simple tables of

supported attributes are built up instead.

B. UQL Parser and Preprocessing

Although the structure of TinyOS ULLA function calls differs a bit from that of standard interfaces, ULLA allows both Remote and Local LU to use the same UQL syntax. The UQL parser is needed to support this flexibility. It will parse the query strings into some comprehensible data structure which can be understood by the application. For example, parsing UQL strings from a Remote LU can be simply done in real time in a high-end device. What comes out of the parser is the binary representation of the query strings. On the other hand, to parse queries from a Local LU the UQL parser requires a large amount of memory footprint and processing power which cannot be implemented on each mote. The UQL parser implementation is left to developers' choice of design as it is not the focus of our architecture. We consciously implemented the UQL parser in a more powerful device, such as a standard PC. In addition, the UQL parsing phase has to be preprocessed before compiling and linking of an application because the nesC compiler does not understand the UQL syntax. Therefore, we need to translate and replace the query with the nesC data structure before compiling with the nesC compiler. During such a text replacing step also typechecking, one usual advantage of static interfaces, can be performed. If the parser knows the type of each requested attribute the correct handling can be ensured in case of both, Remote and Local, LUs.

Figure 4 shows the architecture of the query processing for Local as well as Remote LU. The first case is considered as a real-time query processing while the second case is considered as a static query processing. In a Remote LU, the UQL translation can be performed during the runtime in the desktop computer. However, in order to support flexible queries which do not have a predefined number of attributes and conditions, a mechanism of splitting a single query into multiple physical messages as previously mentioned is used. On the destination mote, all the messages are gathered into the original query with the QAU.

On the other hand, in a Local LU, the translation process is done before the TinyOS compile time. One of the solutions is to use a source code replacing method which is called before compiling and downloading the codes into the motes. This way, the UQL query will be transparent to the nesC compiler as well as the use of the ULLA interface will be transparent to the user.

C. CC2420-BMAC LLA

The LLA component is based on the B-MAC code which is a standard MAC algorithm in TinyOS that supports the CC2420 radio offering the IEEE 802.15.4 physical layer and framing support. CC2420Radio provides flexible basic interfaces to control the B-MAC by higher layers, namely CC2420Control, MacControl and MacBackoff. This allows an LLA developer to write a wrapper interface to control the radio such as setting the transmit RF power.

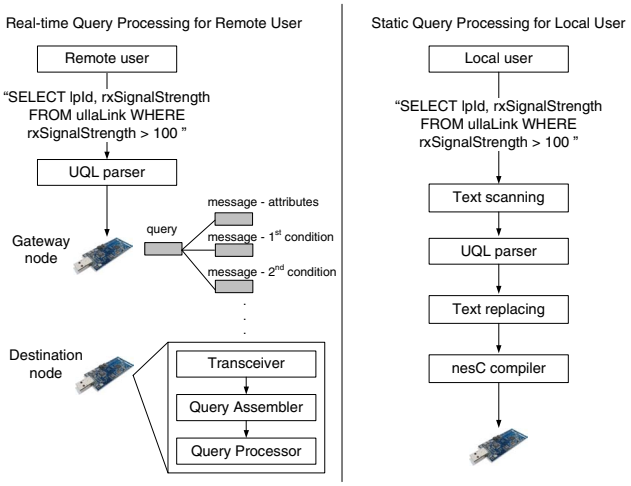

Fig. 4. Query handling in Remote and Local LU.

The LLA provides the interface *linkProviderIf* to the ULLA Core and on the other hand uses the interface *ullaEventIf* which is provided by the ULLA Event Processing in the ULLA Core. Attributes can be retrieved by calling *getAttribute()* or *requestUpdate()* from the LLA using one of the following three approaches.

- The attribute values are known by the driver component. There is no need to ask the firmware or do any complex computation. The link identifier is a typical example.
- The attribute values are calculated from measurement results such as the Packet Error Rate.
- The attribute values are obtained by either retrieving the information from the firmware if the beacon-network is enabled or sending one or more probe messages if the beacon-network is disabled.

The difference of retrieving attributes by using these approaches will be hidden from the LU because the LLA provides the same interface to the ULLA Core. However, differences in performance should be noticeable by the Link User since each approach has a different set of procedures which means different levels of complexity. For instance, in the last approach if the user uses CC2420-BMAC LLA which does not provide any beacon mechanism, the LLA needs to send probes in order to retrieve the link information that takes longer time than fetching the values from the firmware. We shall leave the freedom of choosing the MAC protocol to LLA developers.

The performance can be improved if the optional ULLA storage is implemented. It first checks the attributes in the storage. If the attribute is newer than the given validity, described in section III-A, it can be sent back to the LU. If the attribute is expired, we need to probe neighboring nodes in order to get refreshed information. A new attribute value will be used to update the ULLA storage afterwards.

TABLE I
MEMORY FOOTPRINT OF ULLA [BYTES].

Component	ROM	RAM
ULLA Core	3748	791
CC2420 LLA	734	245
ULLA Storage	88	607
Total	4570	1643

D. Evaluation

This subsection presents the evaluation of ULLA in terms of memory footprint and query duration in order to explore the tradeoff between flexibility and cost of implementation complexity and performance. Table I shows that usage of ULLA is feasible on motes which usually have limited resources especially in terms of memory. Taking the example of a Telos B mote ULLA requires only about 9.5 % of the available ROM and about 16.4 % of the available RAM. As TinyOS does not support dynamic memory allocation we preallocated memory for up to 25 attributes for each of up to 10 links. If the memory footprint is too large the storage size can be lowered or even taken out completely because ULLA storage is only an optional component.

However, it can save otherwise required probing messages as we show with our next experiment. We measured the query duration for a single standard *ullaRequestInfo()* call asking for different number of attributes as part of a Local LU. We chose attributes that require probing to evaluate the worst case. During our measurements five links were present and we repeated the request 2000 times. The results are shown in figure 5. When querying only two attributes the average query duration when using a validity of 200 ms is around 1050 μs and the average time when using a validity of 500 ms is around 850 μs. Comparing these results to latency values measured for, e.g., the B-MAC protocol [11] shows that the latency induced by ULLA is acceptable even for algorithms taking decisions on a per packet level. The impact of the number of requested attributes and the required validity is obvious making clear that careful selection of what is requested and how up-to-date the results have to be is important during development of WSN applications.

Introducing another entity between the application and the module interfacing to the radio chipset in principle increases the system complexity. However, the added latency is in practice negligible and from our point of view, the improved flexibility and the added features offered by ULLA are definitely well worth it. As no complex processing is performed in ULLA Core we also do not expect considerable increase of power consumption but detailed measurements to confirm this are a major aspect of our plans for future work.

V. APPLICATIONS

Applications of the described sensor ULLA are as manifold as WSN-applications themselves. The additional abstraction is extremely useful for Local LUs. Routing agents require

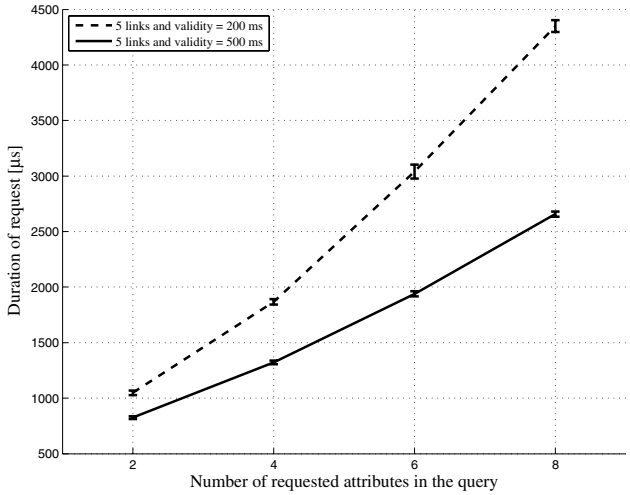

Fig. 5. Measurement results of query duration.

knowledge about the radio environment for optimizing the route selection process. Telos B motes offer a link quality indicator [6], which combines packet error rate and received signal strength already in the firmware; in contrast Mica2 motes do not offer such a value. An LLA for Mica2s could simply combine the two single metrics and offer the same abstract attribute. This allows the developer of the routing agent to work on a higher abstraction level still using link layer information but not requiring deep knowledge about the deployed technology. Porting protocols as well as applications becomes an easier task. Advanced approaches benefitting from information about the network topology can be implemented without limiting the deployment to one WSN platform or family of platforms.

A typical example for a Remote LU is a smart home monitoring tool (SHMT) that gathers environmental measurements from the sensor network to collectively display them in a user interface. Additionally, the same information can be used to control diverse functionalities such as heating, air condition or lighting. Compared to legacy WSNs ULLA-enabled systems offer a much more convenient way of writing such applications.

At the beginning the SHMT uses one query to retrieve information about all motes present in the network:

```
SELECT lpId FROM ullaLinkProvider
```

The query is flooded in the whole network because no lpId is specified in the WHERE-clause. After retrieving messages from all motes in the WSN, the SHMT knows how many motes are present. In the next step the SHMT is interested in the capabilities of the motes and thus which sensing capabilities they have. The used functionality is similar to a reflection interface but its complexity is clearly lowered due to the limited resources of motes. The tool retrieves another attribute providing a bitmask that describes which predefined classes are supported, as explained in section III-

A. As the support for this additional attribute is mandatory for all ULLA-enabled motes it can be read out from all. The class descriptions for classes including sensing attributes are already known by the SHMT so that it can afterwards query for specific attributes.

```
SELECT supportedClasses, lpId
FROM sensorDescription
```

In the next step sensor readings can be requested. As an example the SHMT asks for the temperature and the light intensity measured at a single mote. Similar queries can easily be constructed also in a dynamic way based on the knowledge collected during the former steps.

```
SELECT temperature, lightIntensity
FROM sensorMeter WHERE lpId = 3
```

In order to do smart monitoring the notification functionality should be used with, e.g., one of the following queries:

```
SELECT lightIntensity FROM sensorMeter
WHERE (lpId = 5) AND
(lightIntensity < lightIntensityThreshold)
```

```
SELECT temperature FROM sensorMeter
WHERE (lpId = 5) AND
(temperature > temperatureThreshold)
```

Both requests address mote number five, which was arbitrarily chosen for this example, and ask for notifications when the environmental conditions pass a certain threshold. These are examples for queries that for example an air condition control system or automatic lighting system might use.

If the SHMT should constantly monitor environmental conditions also periodic notifications could be used. The tool would receive latest sensing information each time the period elapsed which is another parameter for the *ullaRequestNotification* function.

All these activities are supported by ULLA and are implemented with a small number of flexible and powerful TinyOS interfaces. Additionally, the characteristics of the underlying sensor network technology are completely hidden as long as the user limits itself to attributes and commands that are common to all WSN-platforms, although their implementation might still be completely technology-dependent.

VI. Conclusions

In this paper we presented the architecture and the implementation on wireless sensor nodes of the Unified Link-Layer API (ULLA) as developed by the EU-project GOLLUM [1] as an extension to the existing ULLA running on more powerful devices [5]. It enables application development for sensor networks independently of the finally deployed sensor platform both for applications running on the nodes but also remotely in end-user devices. ULLA supports the retrieval of link layer information as well as sensor readings via a convenient query interface that follows the object oriented approach ensuring extensibility. Abstract class definitions ensure that the API can be used in technology-independent manner, and the extension of the database abstraction to cover link-layer information as well is powerful feature with numerous potential uses. The offered notification mechanism allows flexible definition of conditions and supports periodic updates of attribute values. The memory footprint of the prototyping implementation shows that the overhead added for the ULLA framework is acceptable also for small and embedded devices. Also the query duration tests lead to the conclusion that protocols taking decisions on per packet level can be improved using ULLA. Our future work will extend this performance evaluation, especially to consider energy consumption issues. The ULLA implementation for TinyOS-1.1.15 is available from http://tinyos.cvs.sourceforge.net/tinyos/tinyos-1.x/contrib/rwth/mobnets/.

Acknowledgment

We would like to thank DFG, European Union (GOLLUM- and RUNES-projects) and RWTH Aachen University for the financial support. We would also like to thank the GOLLUM research team for fruitful discussions.

References

[1] *The GOLLUM-project website*, http://www.ist-gollum.org [Cited on: 23-02-2007].
[2] *TinyOS website*, http://www.tinyos.net [Cited on: 23-02-2007], 2004.
[3] *Crossbow website*, http://www.xbow.com [Cited on: 23-02-2007].
[4] *Lippert AG Cool Master website*, http://www.lippert-at.com/index.php?motemaster [Cited on: 23-02-2007], 2006.
[5] M. Bandholz, A. Gefflaut, J. Riihijärvi, M. Wellens, and P. Mähönen. Unified Link-Layer API Enabling Wireless-Aware Applications. In *Proc. of PIMRC '06*, Helsinki, Finland, September 2006.
[6] ChipCon. CC2420 data sheet. Available at: http://www.chipcon.com/ [Cited on: 23-02-2007].
[7] J. Polastre *et. al.* A unifying link abstraction for wireless sensor networks. In *Proc. of SenSys'05*, 2005.
[8] S. Madden, R. Szewczyk, M.J. Franklin, and D. Culler. Supporting aggregate queries over ad-hoc wireless sensor networks. In *Proc. of WMCSA'02*.
[9] P. Levis *et. al.* The Emergence of Networking Abstractions and Techniques in TinyOS. In *Proc of NSDI'04*, 2004.
[10] P.-H. Persson, P. Kannegiesser, and M. Johansson. RUNES D3.4 HW Platform Definition. IST RUNES consortium, http://www.ist-runes.org [Cited on: 23-02-2007], January 2006.
[11] J. Polastre, J. Hill, and D. Culler. Versatile low power media access for wireless sensor network. In *Proc. of SenSys'04*, Baltimore, USA, November 2004.
[12] J. Polastre, R. Szewczyk, and D. Culler. Telos: Enabling ultra-low power wireless research. In *Proc. of IPSN/SPOTS'05*, Los Angeles, USA, April 2005.
[13] J. Riihijärvi, M. Wellens, P. Mähönen, and A. Gefflaut. Implementation and Demonstration of Unified Link-Layer API. In *Poster/Demonstration session of INFOCOM'06*, 2006.
[14] T. Farnham *et. al.* Toward Open and Unified Link-Layer API. In *Proc. of the IST Mobile and Wireless Summit*, Dresden, Germany, 2005.
[15] P. Levis, N. Lee, M. Welsh, and D. Culler. TOSSIM: accurate and scalable simulation of entire tinyOS applications In *Proc. of SenSys'o3*
[16] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler. The nesC language: A holistic approach to networked embedded systems. In *Proc. of PLDI'03, ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 2003.