UDAE: Universal Data Access Engine for Sensor Networks

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

Abstract—We present the design and implementation of UDAE, a Universal Data Access Engine for wireless sensor networks. The UDAE allows developers to access data both locally and over the network from a variety of sources, such as sensors, communication links and platform components in a unified manner. It is also fully extensible, allowing data sources and users to be added flexibly at run-time. This enables capabilities for cross-component optimization and run-time configuration. In order to validate our proposed system, we have implemented UDAE for two operating systems (Contiki and TinyOS). While the TinyOS implementation has been done as native code, the Contiki implementation is based on a reconfigurable component-oriented middleware enabling developers to fully benefit from the flexibility offered by the UDAE. Both of these implementations have been made publicly available under an open source license. A detailed performance evaluation carried out with these implementations shows that the overhead induced by the adoption of UDAE in terms of memory, processing requirements and energy consumption is very small and the additional latencies induced by the framework negligible. We also discuss various different application scenarios for UDAE in some detail, particularly cross-component optimization and run-time configuration. Furthermore, a practical case study of cross-component optimization selected from the TinyOS domain is addressed.

I. INTRODUCTION

Wireless Sensor Networks (WSNs) have played a prominent role in the research on small-scale embedded systems for a number of years now. As an outcome of this work a number of platforms have emerged, offering a variety of communication and sensing modalities, and supporting a host of operating systems. Examples of the available communications technologies include the IEEE 802.15.4 (the "ZigBee radio"), Bluetooth, different proprietary low-power radios, and even wireless LANs. The collection of existing sensors used in WSN applications is no shorter, including the usual ones such as temperature, humidity, pressure, noise and magnetic field sensors, together with the more exotic ones such as circuits for detecting the presence of various chemical agents. Because of this emerged heterogeneity there is now a strong need for unifying abstractions, making it possible for the application developers to benefit from vast array of capabilities offered by these various WSN platforms and their components.

In this paper we report on our work with the Universal Data Access Engine (UDAE) which offers a unified view of the various data sources available towards the developer. Examples of the data sources supported include different link-layer technologies, networking protocols and various sensors mentioned above. A number of modes of interaction are supported, including synchronous query of data, asynchronous notifications on the change in the data, and even actuation and configuration of data sources.

The key contributions of this paper are the detailed description of the architecture of UDAE, including interfaces and data model used, together with implementation designs for two major WSN operating systems (Contiki and TinyOS). The UDAE design is unique in the WSN domain by providing a common object-oriented and database abstraction for all types of data sources whether they are present locally on the platform or on the other nodes in the network. The architecture is also completely extendible, allowing for new data sources being introduced at run-time (given a suitable operating system). We also discuss key application scenarios for UDAE in some detail, especially focusing on cross-component optimization which we see as the natural extension of traditional cross-layer optimization into component-oriented systems. The design of UDAE is particularly suited for this task, so we have adopted it as a test case. A full case study on cross-component optimization based on our UDAE implementation is presented together with results of a performance evaluation of our framework. We show that UDAE is indeed lightweight and enables information exchange between components even on timescales of single packet transmissions.

The remainder of this paper is organized as follows. We discuss the motivation of our work and the related background in Section II. We then discuss the UDAE software architecture in Section III. Later we discuss some potential applications which would benefit from UDAE in Section IV. The prototype implementations of our framework are discussed in Section V. Section VI analyzes the two implementations in term of memory footprint, processing delay and energy consumption and validates the UDAE system in practice. We finally conclude the paper in Section VII.

II. MOTIVATION AND BACKGROUND

Writing distributed applications for WSNs has traditionally been a complicated affair. Until recently, it was customary to obtain sensor readings by accessing platform-specific registers and to develop the functions needed to process and exchange the obtained data in application specific manner. Such an approach obviously wastes resources and distracts the developer from focusing on the core functionality of his/her...
application. To remedy the situation, various ways to abstract data present in the sensor networks for easier manipulation and access have been developed. Additionally, unification of data access does not only systematically simplify the development procedure, but also facilitates the capabilities for optimization problems, especially cross-component optimization and run-time configuration. These problems can be partly carried out by having a shared data pool and a common inter-component signaling mechanism either between different components at the intra-node level or between the same components at the inter-node level. In this section we briefly discuss the data and communication abstraction problem for WSNs in general, and then compare the abstraction offered by the UDAE with other solutions presented in the literature.

We distinguish three models of abstraction from the related work, called the neighborhood or link abstraction, sensor data abstraction and flash-based abstraction. The first of these provides access to sensor readings or communication facilities of either immediate radio neighbors or groups of nodes specified according to some flexible criteria. Sensor data abstraction on the other hand focuses on easy access of sensor readings from the network as a whole. A particularly common approach has been to apply the database abstraction, resulting in SQL (or some subset thereof) being used to specify the queries for data. Finally, by flash-based abstraction we mean presenting the available network information as a filesystem. Such approaches have been common in mainstream operating systems (especially various UNIX-variants) but have been found applicable for WSNs as well.

Hood [1] and logical neighborhoods [2] provide a neighborhood-based programming abstraction for sensor networks. Hood and logical neighborhoods use a mechanism which discovers neighboring nodes and shares data among those nodes. Each node can locally access the cached data. The authors of logical neighborhoods have claimed that their system is more flexible than Hood because it was designed for generic purposes. Abstract regions [3] captures local information of nodes within a given region such as link connectivity and geographic location. SP [4] implements a link layer abstraction that offers a unified interface over some different link-layer technologies. Neighbor table and message pool are integrated in the SP design in order to provide Neighbors and Send interface to some network protocols. Nevertheless, these approaches provide only link-related information. They do not generalize a link class as another data class in the way that UDAE does.

TinyDB [5] and Cougar [6] are focused on querying of the sensor network data. The query language is SQL-based. The systems include data aggregation to reduce communication overhead. They provide a query interface to developers that hides away communication and aggregation detail. However, these systems are quite restricted to remote applications running on end-user devices.

DALi [7], flash-based data abstraction layer, provides an abstraction layer between the application and the file system. It aims at organizing storage data and provides three basic services such data search, naming and reduction services. Although the file system abstraction is not naturally encapsulated by the UDAE framework, UDAE foresees this as another well-defined data class. UDAE can reasonably collaborate with DALi to further provide an abstraction of the file system.

In summary, these approaches are only focused on simplifying access to particular type of data. They do not provide a unified way to access information across all data, regardless of component location. The most similar approach to ours is perhaps Neidas (NEighborhood DAta Sharing algorithm) proposed by Lachenmann et al. [8]. Neidas is a general-purpose algorithm that supports local data exchange between software components and data sharing between neighboring nodes. However, UDAE is more flexible than Neidas. UDAE can be used from applications running both locally on a sensor node or remotely running on more powerful devices such as Pocket PC or laptop using the same programming interface. Moreover, UDAE provides a more detailed description of how developers can access and update shared data in a data storage. Data is structurally defined in classes such as link-layer, sensory data, platform classes. The data storage is used to efficiently manage shared data among software components which is required for cross-component optimization. In addition, UDAE design is software and hardware platform-independent.

III. UDAE ARCHITECTURE

The data abstraction offered by the UDAE is a combination of database abstraction and object oriented programming paradigm. Different types of data are grouped as attributes of classes. The instances of the classes are presented as rows of a database. Each class can additionally contain methods used for issuing commands of various sorts. Besides dynamically creating new class instances, also new classes can be introduced at run-time, making the system very flexible. In this section we provide in detail the UDAE design. Later, we explain generic programming interfaces provided by UDAE and how UDAE encapsulates data abstraction and data model.

A. From ULLA towards UDAE

The design of Universal Data Access Engine (UDAЕ) was initially inspired by the Unified Link-Layer API (ULLA) design [9]. Originally, ULLA was designed to abstract away the differences in link-layer technologies. It provides a unified way to retrieve link-layer information in a flexible and usable manner. It was evaluated in several kinds of resource limited devices such as Linux/notebook, Windows CE/PDA and TinyOS/TelosB motes. In addition to ULLA [10], we have further extended the concept of link-layer abstraction to general data abstraction. This concept enables a wider range of applications that can significantly benefit from UDAE.

In the original ULLA approach [10], only lower layer and technology-specific components such as link-layer and sensing ones are employed. In this paper, we present a more generic architecture which also copes with the application data provided by Data User components.
B. UDAE design

The modular components are illustrated in Fig. 1. The main component called UDAE Core is an intermediate component connecting Data Providers (DPs) and Data Users (DUs). DUs are applications that read out data through UDAE Core via Data User interface. DPs are used to provide an abstraction layer to data sources via Data Provider interface and hide the difficulties to retrieve the specific information which ranges from lower layer to application layer information.

At the architecture level, UDAE enforces that all components provide the common UDAE interfaces (Data Provider and Data User interfaces) and receptacles in order to specify their dependencies that are chained to compose applications. This introduces a clear separation between components which are entirely independent to each other. Consequently, the simplification of the developing process would be two-fold. First, DU developers control and retrieve information from the other components at a high level of abstraction. Second, DP developers are more interested in providing a UDAE-equipped software component over their technology or protocol. As UDAE allows developers to access data both locally and remotely over the network, we differentiate here between two kinds of DUs. Local DUs run locally on the node whereas remote DUs are connected to the sensor network via gateway nodes but run on PCs or other devices that also see the network as a variety of sources, such as sensors, communications links and platform components. It is noted that the interfaces and data structures used by both local and remote DUs are identical.

UDAE Core provides three main tasks such as query processing, command processing and event processing. The UDAE Core functionalities are exported to the Data User by means of interfaces and receptacles that can be statically or dynamically linked to the application. Sample UDAE functionalities comprise:

- Synchronous queries
- Synchronous and asynchronous commands
- Event notification: UDAE supports both local and remote notifications of changes in the system. The notification mechanism can also be used to implement access to stream-oriented sensors.

UDAE allows developers to construct applications from different components that provide a subset of data and functionality required by an application. UDAE exposes these data to any interested data user component. Specifically, each component is responsible for a particular functionality. They communicate to each other through UDAE with an interface similar to the Unified Link-Layer API [9]. One of major applications that we shall highlight in this paper is cross-component optimization. In particular, cross-component optimization uses shared data to reduce some development overhead generated by application-specific components which usually leads to inefficient solutions. Thus, a generic programming platform like UDAE is needed not only to trim down development effort but also to make data sharing data sharing more efficient. We will discuss about challenges in cross-component optimization in more detail in Section IV.

We provide several components shown in Fig. 4 for the Contiki implementation such as link-layer adapter (LLA), sensor adapter (SA), query assembler unit (QAU) and UDAE application. Similarly for the TinyOS implementation, depicted in Fig. 5, we provide communication manager, sensor adapter, QAU and UDAE application. Implementations for both operating systems are described in detail in Section V.

The query parser is typically not implemented in the UDAE Core because of the resource constrained nature of wireless sensor nodes. Query processing is split into two phases. At compile time, the query strings are parsed on a PC and then replaced with predefined data structures which are the same structures that the local DUs use. At run time parsed queries are interpreted on a mote. The UDAE Core processes the requested queries and thus selects single DPs which are corresponding the requested data classes as defined in the query strings. We adopted the preprocessing and SQL-based query parsing approach from [10]. Alternately, if a more powerful sensor node such as Imote2 [11] is used, dynamic query processing can be performed on a mote. For example, the implementation overhead of a similar query parser generated by a powerful flex and yacc parser generator and implemented on a Linux prototype [9] requires only about 87 KB of ROM.

In addition, a data storage component is implemented enabling data sharing and more advanced functionality. This can be used for statistical operations or saving an effort of UDAE Core when it needs to frequently retrieve data from the Data Provider. The storage component is also used to maintain data shared among different application components in order to potentially perform cross-component optimization by reducing signaling overheads.
C. UDAE interfaces

The UDAE caters for both data (sensor readings, platform information, link-layer characteristics etc.) and actuator/configuration access. For obtaining data either locally or over the network two models are supported, namely synchronous queries and asynchronous notifications. In synchronous queries an SQL-statement of the form

```
SELECT ⟨attributes⟩ FROM ⟨classes⟩ WHERE ⟨conditions⟩
```

is used to specify the data of interest. The FROM-clause specifies the classes the values of ⟨attributes⟩ are read from and the WHERE-clause can be used to limit the scope of the query. The UDAE programming interfaces allow users to communicate with the data storage using standard SQL syntax. For example, the DU interface includes the function call `udaeRequestInfo()` which is used to retrieve information from the storage using the SQL syntax. Similar SQL-statements are used to set up asynchronous notifications. Notifications are based on the function call `udaeRequestNotification()`. This time the WHERE-clause is used to set up the conditions the fulfillment of which should trigger a notification to the application with an event `udaeRequestNotificationDone()`. Finally, a command interface is offered allowing for local or remote configuration. In the simplest case one can update the values of an attribute via a call to `udaeSetAttribute()`, or invoke a method of the class with the necessary arguments by calling `udaeSetClass()`.

In contrast to the DU interface, the DP interface forms a connection between UDAE and DP components. UDAE calls the function `udaeGetAttribute()` or `udaeGetClass()` to retrieved information from requested DPs.

D. Data abstraction and data model

In general, data abstraction provides a way of organizing data with associated operations, enabling reusability of data/service classes or components in another component. It provides constructive mechanisms for deploying and re-configuring systems. These mechanisms have to be simple, lightweight, efficient and highly tailorable. Typically, data abstraction incorporates a generic programming platform [12] in order to provide a set of configurable parameters/services.

UDAE works with a wide range of data classes. The data model applied in the UDAE framework follows an object-oriented design, exposed to the DU through a data abstraction. Data sources are described using different classes. Examples of data classes are shown in Fig 2. Classes can be categorized into two types: shared classes and specific classes. Shared classes are classes which can be accessed by all components. Examples of shared classes include link, sensor and security classes. Each class consists of attributes and commands. For example, a link class consists of received signal strength (RSSI), link quality indicator (LQI) and neighboring links. A sensor class consists of available sensor types such as humidity, temperature and light sensors. Specific classes, as the name says, are component-specific classes which abstract their own attributes and commands. Examples of specific classes include routing, transport, application classes. The key distinction between the two classes is that attributes and commands of shared classes can be read, written and executed by any DUs while attributes and commands of specific classes can be written and executed only by a cross-component optimizer or a run-time configurator performing as a centralized control component. This way, a control component has the overview of the whole system and avoids a local optimum that can happen if all the components can freely control the others.

DUs can retrieve attributes from any of the available classes in the storage by sending queries to the UDAE Core. For this, DUs call `udaeRequestInfo()` which is given the query string describing which attributes and data class to be requested. The example query string `SELECT LinkId, RSSI FROM LinkClass` results in the link identifier and the received signal strength (RSSI) being returned to the DU. Moreover, DUs can also use conditions to specify more in detail what kind of links they are interested in. For example, the query `SELECT Temperature, Humidity FROM SensorClass WHERE Temperature>20` will only report the temperature and humidity when the temperature is greater than 20 degrees. This query string is encapsulated with the DU interface as previously explained in Section III-C and at compile time the query string is parsed into a data structure of the sensor class. Similarly, dedicated classes can be used to give DUs access to flash memory. The DPs providing such classes can be implemented with an adapter that wraps the flash-based functionality which can be handled through DU commands.

IV. CHALLENGES IN CROSS-COMPONENT OPTIMIZATION AND RUN-TIME CONFIGURATION

Before moving on to the implementation we shall briefly discuss potentially significant applications of UDAE that are inherently architectural in nature, namely cross-component optimization and run-time configuration. Later in Section VI-E we will provide a practical case study of cross-component optimization problem in TinyOS.
A. Motivation

The need for cross-component optimization and run-time configuration arises due to the fact that WSNs usually operate in dynamic environments which impose varying performance and functional requirements. Additionally, the long deployment intervals also increase the probability that the user requirements will change. Therefore, the system has to detect unexpected changes and adapt itself over time to the new conditions, especially without presence of an external or centralized control. WSNs, in particular, should support dynamic adaptation capabilities that enable themselves to be tolerant to abundant operating conditions. Cross-component optimization and run-time configuration in such systems are very challenging owing to limited resources on the sensor nodes. The benefits of cross-layer optimization have been argued to be so significant as to warrant breaking down the layered software model. We argue that UDAE allows similar performance gains from optimization, while preserving clean component interfaces. Moreover, typical WSN applications do not autonomously adapt to environmental changes at run-time and heavily rely on central controlling nodes. Such systems are usually suboptimal and inefficient.

By cross-component optimization we understand the obvious generalization of cross-layer optimization problem into the component-oriented setting. It has been argued by several authors that having dedicated cross-component interfaces for sharing information for optimization purposes does not scale. This is not only because of the quadratic scaling of the number of needed wirings as the function of the instantiated component counts, but also because of the need for customizing the interaction for each pair of components. This problem can be partially solved by having a common inter-component signaling mechanism, for which purpose the UDAE is very well suited. The UDAE Core provides on each node a central place to which components can be wired for accessing and providing signaling information, simplifying the component communication topology and thus reducing overhead. The remaining problem which we shall not tackle in this paper is the establishment of standard data types / class definitions for describing information useful in engineering cross-component optimization mechanisms. For some of the existing work into this direction, see [13], [14]. In practice, cross-component optimization can be performed by tuning of component parameters and component retasking by applying either some rule-based decision making or heuristics algorithms.

If the changing environmental conditions cannot be handled by cross-component optimization alone, an optimization module will form an alternative by performing run-time configuration. Run-time configuration rewires replaceable components in a component pool to form a new component graph. This is not applied at the first place because component reconfiguration is more complicated and probably requires cooperative supports and synchronization at the inter-node level. A set of replaceable components can be stored either in the program memory (ROM) or in an external flash memory (EEPROM).

The system can seamlessly access the needed data via UDAE in order to perform both operations. Another issue is how to decide which mechanism should be used and in particular which parameters should be tuned and which configuration should be set. This is, however out of the scope of this paper and one of our major future research focuses.

Most of the WSN applications described in the literature (see, for example, [11]–[7]) do not feature optimizations across several components, except perhaps as special cases. This tends to lead into increased complexity, memory footprint and power consumption as many of the features best offered by a reusable component are usually replicated in time and across the application components. In the following subsection we shall give some examples in cross-component optimization and run-time configuration aspects in which the adoption of the UDAE framework would make a significant impact.

B. Application examples

In this section we exemplify some UDAE applications in practice. The first two examples describe potential applications for cross-component optimization, namely a context-aware application and knowledge-based application, respectively. The last example is focused on centralized run-time configuration.

The first application we consider in detail as a major potential application is context-aware monitoring system. The system is equipped with components that are dynamically reconfigurable. This application comprises of two components that provide topology control and routing, respectively. Both are implemented based on UDAE and the parameters of both components are jointly optimized. The topology control component connects and continuously maintains the network based on link layer information such as the signal strength or the link quality indicator (LQI). The routing application can also benefit from the UDAE framework by accessing simultaneously topology information and any other link-layer-related data. In addition to the current topology information, the routing protocol also considers the neighboring links in order to reduce the spatial interference and enables dynamic clustering by carefully selecting the cluster heads. A similar example was practically conducted and explained later in Section VI-E using two standard TinyOS applications: link-based routing application and report application.

As a second example we consider here is the bootstrapping of a network which uses distributed source coding (DSC) to save energy and bandwidth usage of the network by exploiting the sensor data spatial correlations [15]. Before data transfer has continued for significant period of time there is little that can be done to estimate the parameters used by the DSC framework without the use of cross-component optimization. Knowledge of typical spatial correlations of the phenomena being measured is not sufficient for this unless the network topology is known. Using UDAE the components responsible for DSC could, however, easily access such knowledge on the topology from clustering or localization components in order to obtain the initial estimates on correlations between sensor readings that are needed to bootstrap the scheme. Similar...
cross-component information exchange can be used to react to significant changes in network topology. Particularly, DSC tunable parameters such as coding scheme, sampling rate, code rate and roles (e.g. compressing, sending side information and decoding) can be adjusted corresponding to these changes. For example, if the node density increases, then the correlation between nodes becomes higher. We can therefore lower the code rate in order to reduce the energy consumption. We expect this avenue of research to significantly extend the network lifetimes of WSNs deployed for large-scale monitoring applications.

The last major application we shall highlight is network management based on the global view of network-wide link and platform information. UDAE provides users a global view of various operating parameters such as network links and lifetime in order to make network management decisions. This eases the burden of managing WSN systems and allows developing network-wide optimization mechanisms. The adoption of UDAE makes it also possible for the developer of the management solution to operate on hardware- and platform-independent abstractions, leaving the hardware and operating system developers the task of providing the appropriate DPs for their platforms. Such a separation of concerns promotes portability of applications as well as enables each party to focus on developing those parts of the system they know best.

V. IMPLEMENTATION

We have implemented UDAE on two operating systems, Contiki [16] and TinyOS [17] as a proof of concept for software platform independence. TinyOS is an event-driven and component-based operating system. Contiki is in many aspects similar to TinyOS, but has an additional support for dynamic linking of code and multithreading. Programming language used in Contiki is C programming which is arguably more familiar to software developers than the nesC programming language [18] used in TinyOS. Our UDAE implementations have been successfully tested on the TelosB sensor platform [19] featuring a TI MSP430 microcontroller, clocked at 8 MHz, 10 kB of RAM, 48 kB of Flash, and the Chipcon CC2420 radio chipset [20]. The implementations are publicly available.

In this section, we provide implementations of the two distinct operating systems in some detail. Additionally, we give some example deployment scenarios in order to better explain the developers how to use UDAE to access information across data abstraction in practice.

A. Contiki

UDAIE components have been implemented on Contiki-2.0 under the RUNES middleware framework described in more detail below. Contiki, together with the RUNES middleware, does not only provide well-defined interfaces but also features dynamic reconfiguration of software components and their connections without the need for a reboot the kernel when a component has been loaded or unloaded [21]. Moreover, it enables support for dynamic adaptation of component’s parameters or dynamic adaptation of replaceable components.

1) RUNES middleware: The RUNES middleware runs on very resource constrained platforms. The components are realized as C macros wrapped around Contiki protothreads [22]. The component model is comprised of the following main elements: components, interfaces, receptacles, connectors and capsules. The elements of the component model and their dependencies are illustrated in Fig. 3. Components are the basic runtime unit and can be loaded (load()), unloaded (unload()) and deleted (destroy()). Via interfaces components can offer their functionality. In case components depend on each other they can express this dependency by means of receptacles. The receptacle of a component has to be connected (connect()) to a corresponding interface of another component before it can be executed. Finally, all components and interfaces reside inside a capsule which serves as a runtime application component offering the API via invoke(). A connection can be established if one component is attached to the receptacle of the other component. Once this has been performed communication starts and each component invokes the functionality of the other component. Hence, the middleware components can communicate with each other via connectors. A connector is a component itself consisting of interface-receptacle pairs and represents a specific behavior (for example, monitoring communication) to be invoked in case a call occurs over one or more of its pairs. The components can also be deployed at runtime on the target platforms in the WSN. The Contiki UDAE components are implemented at the application level.

2) UDAE components and practical deployment: UDAE Core, DPs and DUs depicted in Fig. 4 can be either dynamically or statically linked to applications. UDAE Core comprises four entities: query processing, command processing, event processing and data storage. Queries and commands from the remote DUs are encoded in binary and sent over the network in order that they can be handled similarly to the local DUs.

Queries can be initiated with the function call invoke(udae_r, iudae, udaeRequestInfo((query, queryId)) where udae_r is UDAE’s receptacle and iudae is UDAE’s interface, queryId is an identification given by DUs and query is an SQL-based query as previously explained in Section III-D. For example, similarly to the link abstraction in [10], UDAE Application as a DU/DP can request neighborhood information by calling the above function invoke() with the a query string...
SELECT LinkId FROM LinkClass. Similarly, DUs can request sensory information by calling invoke() with the query string SELECT Light FROM SensorClass WHERE Light>100. The light intensity is returned only if it is greater than 100lux. Complex queries from the remote DUs which do not have a predefined number of attributes and conditions are split into multiple physical packets and gathered in the target sensor node afterwards. In Contiki, the transmission of packets uses the µTCP/IP [24] with packet fragmentation that is managed by TCP/IP. The UDAE Core processes the requested queries and thus selects single DPs which are corresponding the requested data classes as defined in the query string.

As one can see from the examples, the function calls and query format provide a uniform access mechanism to information. The standard classes also provide a great deal of abstraction which keeps attributes, such as BER (bit error rate) invariant in queries regardless of the specific underlying link layer or radio technology. This means that, for example, queries for BER is always exactly the same function call regardless of the radio technology used, or even if it is a query towards neighborhood information. This means, of course, that the code is also more directly reusable and portable between different platforms.

The storage is implemented on the sensor node by statically allocating RAM because a standard database application would require more resources than they can afford. A simple table of supported attributes is created instead.

DP provides the interface dataProviderIf to the UDAE Core. We implemented two components: LLA and SA. LLA uses information retrieved from a CC2420 radio driver code implemented on the Contiki operating system. The driver supports a minimum requirement for CSMA-CA-based MAC protocols which include basic CSMA-CA, retransmission and random backoff exponent mechanism. The packet structure complies with the IEEE 802.15.4 standard. In case LLA needs to update the link class in the storage when there are no incoming packet transmission, it sends out a probing packet using µTCP as a communication mechanism. SA wraps the Hamamatsu [25] driver component which takes readings from two light sensors: Photosynthetically Active Radiation (PAR) and Total Solar Radiation (TSR).

B. TinyOS

UDAЕ has been also implemented in TinyOS 1.1.15 [17]. The TinyOS UDAE implementation was successfully tested on the Telos B platform. We also verified portability to other sensor platforms such as Mica2, MicaZ [11] as well as TOSSIM [26].

1) TinyOS concurrency model: TinyOS does not use traditional concurrency mechanisms such as threads used in Contiki. In contrast, TinyOS uses an event-driven concurrency model [27] that does not allow long running operations which block other operations in the application. Instead, TinyOS uses a split-phase approach meaning that the operation of a request in complete when an event or callback is invoked. For example, an ADC component invokes the getData() command to read data from the sensor and the ADC component signals the dataReady() event when sensor data reading is completed. TinyOS execution model is driven by events generated from either hardware interrupts or software components.

Our TinyOS implementation is fully aware of the TinyOS concurrency model. The main UDAE function calls are implemented as split-phase commands. For example, udaeRequestClass() command is called from a DU to retrieve a data set of a given class name. This can be read either directly from the storage, if not expired, or from the corresponding DP. The DU is then notified with udaeRequestClassDone() when the result tuple is sent.

2) UDAE components and practical deployment: The TinyOS UDAE implementation is illustrated in Fig. 5. UDAE, DPs and DUs are statically linked at compile-time. UDAE in the TinyOS environment has almost the same functionality as in the Contiki environment. One of restrictions is that TinyOS does not naturally allow dynamic rewiring of components at run-time. Complex remote queries are handled similarly to the approach in [10]. The remote query is split into several physical messages. In a destination node, these messages are collected with a query assembler unit (QAU) and reassembled to the original query. QAU keeps the remote query structure flexible by employing dynamic memory allocation offered by
the TinyOS standard component called TinyAlloc [28] and also allows QAU to handle multiple queries at the same time. In brief, TinyAlloc allocates a static chunk of RAM which can be divided dynamically. QAU collects the remote queries and thus uses the function calls provided by the DU interface to communicate with the UDAE Core. An example where we can exploit QAU is the DSC application running as a remote LU on a PC. The DSC application wants to estimate the underlying correlation structure of the network in order to adjust code rate, coding scheme or data sampling rate. The DSC application therefore calls the function udaeRequestInfo(). This query is then parsed into a binary representation and perhaps split into several messages if the query is complex. QAU collects all the sent messages and reformulates an original query on the node. Thereafter, QAU sends the query to UDAE.

**Communication Manager** (CM) wraps the functionality of LLA and a transceiving component which offers the Send and Receive interfaces. The LLA component is based on a standard BMAC [29] protocol implemented in TinyOS. SA wraps all the TelosB sensor drivers provided by TinyOS such as the Hamamatsu sensor drivers (PAR and TSR) as well as the Sensirion [30] sensor drivers (humidity and temperature).

**VI. PERFORMANCE EVALUATION AND SYSTEM VALIDATION**

The main objective of performance evaluation is to illustrate the effectiveness of the proposed architecture. Both Contiki UDAE and TinyOS UDAE which are ported to the TelosB sensor platform are evaluated in the next three subsections in terms of memory footprint, query duration and energy consumption. Eventually, we validate the proposed system deployed in cross-component optimization by porting two standard applications in TinyOS.

**A. Memory footprint**

Table I shows the memory occupied by the evaluated prototype. The data storage is integrated in UDAE by statically preallocating memory for up to 5 links. These three main components feature the basic functionality which includes the query and notification functionality. Overall, UDAE, LLA and SA require only about 9.93 % and 7.78 % of the available ROM and about 4.62 % and 10.4 % of the available RAM on the TelosB platform on Contiki and TinyOS respectively.

The storage implementation in the Contiki prototype is simpler than the TinyOS one. Indeed, the dynamic memory management (DMM) used by the TinyOS prototype is removed from the Contiki prototype since implementation of a static memory management is more efficient than DMM without significantly sacrificing performance. In addition, the current DMM presents the disadvantage of introducing a portion of processing delay that is needed for dynamically handling the storage, especially if the operating system does not naturally support the memory management like TinyOS.

**B. Query duration**

To estimate the delay introduced by UDAE, we measured the query duration for a single standard udaeRequestInfo() call requesting for 6 attributes while five links were present. The experiment was repeated 2000 times in order to obtain an average time used by a single request. The use of data storage potentially reduces a number of probing packets required by each udaeRequestInfo() call. This can be achieved if the attribute expiring period or validity informs the storage unit that the requested attribute has to be refreshed if the attribute is older than a given validity. When requesting eight attributes, the average query duration is less than 3000 µs when using a validity of 200 ms and around 2000 µs when using a validity of 500 ms. The average delay is considered satisfactory when compared to the round-trip time needed for sending a probing packet which is around 4400 µs excluding local processing on a receiver.

**C. Dynamic component loading**

We have also experimented on how much time is consumed by dynamic component loading on Contiki. It took less than one second to locally load local components and approximately 9 seconds to wirelessly upload the 5468-byte LLA component. The file size is larger than listed in Table I because the LLA component was converted into ELF format (Executable and Linkable Format). This reflects how much time is needed to wirelessly load new components. Besides, another factor the developers should also consider is the energy consumed by packet transmissions.

**D. Energy consumption**

Due to the fact that a long battery lifetime is vital for WSNs, it is necessary to evaluate the impact of UDAE energy usage. To determine the energy consumption of our system, we chose one of the measurements performed in the previous section. It includes processing of a udaeRequestInfo() call, checking the storage, transmitting a probing message, receiving reply messages, updating the storage with new information, and then sending results to the Data User. We averaged the energy over 2000 queries to obtain the energy consumed by a single request. A chosen measurement setup requested for six attributes while five links were present and a validity was 500 ms. The energy measurement was done by using one sensor node and one resistor of 3.7 ohms. The energy was calculated through the current over the resistor given by

---

**TABLE I**

<table>
<thead>
<tr>
<th>Component</th>
<th>Operating system</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Contiki ROM</td>
</tr>
<tr>
<td>UDAE Core</td>
<td>2382 B</td>
</tr>
<tr>
<td>UDAE Dynamic Storage</td>
<td>N/A</td>
</tr>
<tr>
<td>CC2420 LLA</td>
<td>1894 B</td>
</tr>
<tr>
<td>Message Broker</td>
<td>3708 B</td>
</tr>
<tr>
<td>QAU</td>
<td>N/A</td>
</tr>
<tr>
<td>SA</td>
<td>404 B</td>
</tr>
<tr>
<td>Total</td>
<td>8388 B</td>
</tr>
</tbody>
</table>
an oscilloscope. The energy consumption includes costs of query processing, data caching and communications between nodes when which are only required when the validity is expired. The averaged energy consumption of a single query was approximately 46.7 nJ which is four times less than that of a single bit data transmission over the radio. This mainly results from the energy being consumed by local processing on the node as UDAE keeps retrieving the information from the storage as long as the validity is not expired.

E. System validation: a practical case study on cross-component optimization in TinyOS

To verify the use of UDAE for cross-component optimization, we reimplemented two standard TinyOS applications, SurgeTelos and MultiHopLQI available from the TinyOS CVS repository [28], as DU and DP components. SurgeTelos is a simple report application which periodically collects sensor readings and delivers them to a base station. MultiHopLQI is an application that handles multihop packet delivery, based on the design of MintRoute [31]. MultiHopLQI combines link reliability parameters of its direct neighbors such as hop count and link quality indicator (LQI) to route to the base station. SurgeTelos usually routes data to the base station with the MultiHopLQI protocol. These two components are generally independent to each other. MultiHopLQI simply offers a Send interface to SurgeTelos. They do not exchange data with each other. This is precisely one of the motivations where UDAE can fill in the gap and subsequently solve the optimization problems by collaboratively sharing resources.

In our implementation, we have inserted the UDAE component in between SurgeTelos and MultiHopLQI as shown in Fig. 6. Both the SurgeTelos and MultiHopLQI protocol function as DUs and DPs at the same time. In particular, when running as a DP, they regularly update the shared storage promptly after they receive an incoming packet. Contrarily, when running as a DU, they firstly probe the shared storage in order to avoid redundant packet transmissions. By default MultiHopLQI broadcasts a neighbor discovery message every 32 seconds while SurgeTelos transmits sensor readings every 2 seconds. With this rate, the link class in the storage is refreshed for every 2 seconds with the message sent by the SurgeTelos component. The UDAE-enabled MultiHopLQI component verifies the validity of the link class in the shared storage before broadcasting the beacon by calling the udaeRequestInfo() function call. As a result, the MultiHopLQI component does not necessarily need to send a beacon message at all. Instead, it constantly retrieves the needed information from the shared neighbor table provided by UDAE.

Similar routing protocols that also profit from this scenario are typically proactive routing protocols. This is the case only if the network operates at relatively high duty cycles where the data delivery rate is greater than the refreshing rate of the routing protocol. Aside from this particular case study, the system can be prosperously extended to be self-adaptive to the changes of data traffic. For example, sensor nodes operate at very low duty cycles. A run-time reconfigurator, if in use, which is fully aware of the current duty cycle, suggests the routing component to adjust its refreshing rate correspondingly. Although this case study is very simple but, indeed, it illustrates a major advantage in adopting UDAE in practice. It clearly leads to fortunate outcomes in adopting UDAE in more complex scenarios.

VII. CONCLUSIONS AND FUTURE WORK

In this paper we have proposed the concept of Universal Data Access Engine (UDAE) into WSNs. UDAE allows application developers to access information via a well-defined query interface either locally on the sensor nodes or remotely in more powerful devices. Generalization and abstraction of data are defined in technology-independent manner that simplify developing processes for the application writers that see WSNs as database. In addition, we have described that UDAE can be potentially employed to obtain cross-component optimization by having a common inter-mechanism signaling mechanism.

The UDAE design and its interfaces are independent of the operating system it is implemented on. We have ported UDAE to two distinct operating systems, Contiki and TinyOS. The implementation on Contiki is fully based on preemptive multithreading design philosophy. Along with the RUNES middleware, we have shown that the Contiki UDAE implementation is very flexible and reconfigurable both on the component level and system level. On the other hand, the implementation on TinyOS benefits from well-defined nesC interfaces and component-based and split-phase mechanism. The performance evaluation results show that the memory footprint, average query duration and energy consumption of the prototyping implementation are definitely acceptable. This indicates that our UDAE design concept is lightweight and can be used in applications targeting the longest deployment times as well. Furthermore, we have exhaustively discussed about challenges in cross-component optimization and run-time configuration using UDAE. A practical case study in TinyOS has been selected to testify the UDAE usage in reality.

Our future work aims at performing cross-component optimization and run-time configuration. The UDAE implementations for Contiki and TinyOS are available from http://udae.sourceforge.net/.
ACKNOWLEDGMENT

This work was financially supported by the European Union (WASP IST-034963, RUNES IST-004536 and earlier GOL-LUM IST-511567) and RWTH Aachen University. Special thanks to Alexandre de Baynat for many thoughtful discussions. We also thank Björn Grönvall and Manish Lad for their support on porting UDAE to the Contiki operating system. We also thank Steve Hailes and Cecilia Mascolo from UCL on collaboration in the RUNES project and making the middleware framework available. We also thank anonymous referees for extremely useful comments and constructive criticisms, which contributed strongly towards the final version of this paper. Our special thanks to Mark Varvis for his insightful comments, suggestions and shepherding this paper through.

REFERENCES


