In Proc. Work-in-Progress Session of the 1st European Workshop on Wireless Sensor Networks (EWSN)
Technical Report TKN-04-001 of Technical University Berlin, Telecommunication Networks Group, pp. 41-45, Berlin, Germany, January 2004.

Structuring the Information Flow in Component-Based Protocol Implementations for Wireless Sensor Nodes

Andreas Köpke, Vlado Handziski, Jan-Hinrich Hauer and Holger Karl
Telecommunication Networks Group
Technische Universität Berlin
E-mail:{koepke, handzisk, hauer, karl}@tkn.tu-berlin.de

Abstract

Protocol implementations for wireless sensor networks have particular requirements. We analyze shortcomings of existing architectures and propose the separation of packet flow and meta information as a key abstraction for a new architecture. To handle meta information, we use a publish/subscribe-based, anonymous and asynchronous paradigm, for which we describe a blackboard-based implementation.

I. Introduction

Wireless sensor networks (WSN) are challenging because of their severe resource constraints, their requirements for highly optimized performance, especially concerning communication protocols, and their vast range of different applications with very different trade-offs. This last requirement implies that any single, standardized protocol stack is unlikely to be able to provide the required efficiency, unless it were a full superset of all potentially conceivable protocol mechanisms – such a "superstack" would be a nightmare to develop and maintain and impossible to fit into the tight resource limitations of a WSN node. On the other hand, forgoing standardized protocol stacks and embracing a full custom design for every different application is also too expensive and time-consuming.

Hence, a way of structuring and implementing communication protocols is required that combines flexibility with manageability and efficiency. For such a purpose, the deficiencies of the traditional layered model have long been a focus of interest. Initially, it revolved mainly around efficiency issues, as the newer and faster networking technologies exerted pressure on the software running on the end systems [1]. More recently, the pressure comes from the migration of functionality from the end-systems back into the network (middleboxes like firewalls, NATs, web caches, etc.)[2], or form the incompatibility of the wireless medium with protocols like TCP (e.g., [3]).

Another example is the need to violate layering principles to assist network-layer mobility schemes with link or physical layer information such as the received signal strength indicator (RSSI) for faster handoff times [4]. In a WSN, this RSSI information would be used by potentially many different entities, e.g., it can be used to computed location, to assist MAC protocols, to determine neighborhood information, or to find stable routes in the network layer. It is not clear how such an information flow between vastly differently entities that do not really care about their mutual existence and operational details should be organized.

While the layered approach has proved a very valuable tool for promoting modularity, reuse, and standardization, the previous examples show that it is problematic when applied to WSNs. It falls short in efficiency as it severely limits the flexibility of information exchange between different entities. A new structure for this information flow is required.

Some newer architectures support such a more flexible, less rigidly designed way of information exchange. We give an overview of such approaches and their shortcomings in Section II. We go beyond these proposal by defining an architecture that separates two main forms of interaction between entities. This architecture is described in Section III; Section IV discusses its prototypical implementation.

II. Component-based architectures

One viable alternative to the traditional layered architecture is the use of the component model where the functionality of the monolithic layers is broken up in several smaller, self-contained building blocks that interact with each other via clearly defined interfaces [5]. The hiding of the implementation behind well-defined interfaces still preserves the modularity of the solution and promotes reuse. At the same time, the component model supports richer interactions between the building blocks. The interaction is no longer in a strict up/down nature, but starts to resemble a graph. This enables the extraction of common functionality and definition of complex relationships that would clearly require a layering violation in the traditional case.

This model represents an especially good fit to the specific requirements in WSNs [6]. Their event-driven nature and the constrained resources require a code organization that is very well covered by the component paradigm. The thin hardware wrappers, the communication primitives and the sensing tasks can all be naturally abstracted in the form of components. The power of this approach is best evidenced by the apparent success of *TinyOS* [7], the component-based operating system

for WSNs developed at the University of Berkeley, and its supporting language *nesC* [8].

Equally important to the type of modularization is the nature of the supported interactions between the components. The main concern here is the asynchronous and event-driven type of exchange that occurs not only in the communication context, but also in the user space, as the applications in WSNs are tightly coupled with the environment and usually perform processing as a reaction to some sensed event.

The TinyOS components interact with each other via bi-directional interfaces that support invocation of *commands* and signaling back *events*. The applications are composed by "wiring" together the necessary building blocks. This entails explicit specification of the components together with the involved interfaces and their role (provider/user) in the information exchange.

We believe that this type of interfacing may not be the optimal solution for several types of interactions that frequently occur in the protocol stacks for WSNs.

To illustrate our point, let us return to the RSSI example in the introduction. The nesC interfacing approach demands that we explicitly connect the component providing the RSSI data (for example the PHY component) to all of the other components that need to receive it (MAC, Network, Neighborhood, Location components). This will couple them tightly (as their identities are explicitly stated in the configuration files), and will impede or at least complicate future revisions of the affected interfaces.

The identity coupling increases the complexity of the interaction graph, hinders the modularity and complicates the reuse of the components. We claim that this class of information exchange is much better supported by an anonymous and data-centric approach which we detail in the next section.

III. Support for component interaction

The explicit "wiring", like in TinyOS, of components is beneficial for some kinds of interactions, specifically the "push" interaction type, where e.g. a packet is pushed towards another component.

But it is not well suited for the "pull" interaction type. In the explicitly wired Received Signal Strength Indicator (RSSI) example in the previous section, the receiving components get called immediately when new information is available, at a point in time when they do not need it. In order to access it when they need it, each component stores it somehow. A better solution for this is a blackboard[1]: when a new RSSI reading is available, it is stored on the blackboard. This way, the provider of this information does not know about the users and the users do not know about the provider: the information exchange between the components is *anonymous*. This enables loose coupling between the components.

However, when a component waits for a certain information to become available, polling is not very efficient, a notification is better. It is not easy to decide at compile time when a component wants to be notified and when it wants to just read the latest value when necessary, because this can depend on the internal state of a component. Therefore, the blackboard should not be a dumb shared memory, but also provide an interface where components can register their interest in notification events. This interface is the *control interface* of a blackboard. For a data-centric structure like a blackboard, a data-centric control interface is a natural choice. Hence, we use a publish/subscribe [10] control interface.

The provider or publisher of an information only needs to know which information is potentially useful for others (called subscribers) and how to publish it on the blackboard. All the subscribers need to know is how to (un-)subscribe notification events and how to read the information from the blackboard.

This part of the architecture is the main contribution of this paper and an efficient implementation is described in the next section.

Apart from this asynchronous and anonymous exchange of (meta-)information – RSSI values are a typical example – also the handling of packets has to be supported in such an architecture. There are some innovative ideas where even the packet handling is fairly decoupled between the individual components (e.g., the "protocol heaps" as described in [11]); we are currently intending to handle actual packets more along the lines of simple configuration languages between the different components, as similarly done e.g. by TinyOS.

IV. Implementation

A. Overview

The main objectives for an implementation of the blackboard in a WSN context are twofold: first, it must be very memory efficient, especially with respect to random access memory (RAM), which is often a scarce resource in a WSN node when compared to ROM or FLASH memory, and second, it must allow subscribers to dynamically (un-)subscribe to notification events.

In our approach we assign a unique number (the notification event number) to each information published on the blackboard. For simplicity and speed reasons, publishing an information means that the publisher writes variables on the blackboard and tells the blackboard about it by calling a publish function with the number of the information as a parameter. This number is used by the blackboard to figure out which components are currently subscribed for a notification event. A straightforward solution is to keep a list of all currently active subscribers and a list of all currently published events. However, this implementation consumes a large amount of precious RAM. It also requires a dynamic memory management that might have too high an overhead or be too complex. In our solution, a subscriber declares at compile time in which events it is *potentially* interested. This information is gathered by a small script that constructs *static* data structures. These structures, placed into the flash memory of the node, enable the blackboard to notify a subscriber that new data is available. To allow dynamic (un-)subscriptions and

[1]For an overview of blackboard concepts and terminology cp. e.g. [9].

also to track published events, the blackboard maintains a *bit field* in the RAM of the node.

B. Data structures used by the blackboard

The blackboard uses three tables, called index table, subscriber table and subscriber flags; they are shown in Figure IV-B.

The index table is a constant table which has an entry for each declared event. Each entry consists of two values: The "count" value specifies the amount of subscribers declared for this event. The "offset" value is an offset into the subscriber table. The subscriber table contains the addresses of the functions to be called by the blackboard in order to notify a component that new data is available.

Note that these lists do neither contain information about which subscriber has currently an active subscription for a certain event nor for which subscriber an event has been published; they are static and stored in the flash-memory of the node.

The blackboard uses the subscriber flags table to keep track of dynamic subscriptions and recently published events; it is the only structure that changes at run time and has to be placed into RAM. Its structure matches the structure of the subscriber table, but instead of a list of subscriber addresses it stores a list of flag-pairs. For each subscriber in the subscriber table two flags are stored, S-flag and P-flag. The S-flag is set if the corresponding subscriber is currently subscribed to the event, or reset if not. The P-flag is set if an event has been published for the subscriber, otherwise it is reset. The P-flags are set by the publish function of the blackboard.

Publishing an event is done by accessing the index table to acquire the offset-value for all related subscribers. The offset-value can be transformed to an offset into the subscriber flags (by a division and modulo operation). The count-value represents the number of flag-pairs whose P-flag must be set if the subscriber has currently subscribed to the event (has the S-flag set).

When new data is available, the blackboard has to find out which subscribers need to be notified. It simply scans the subscriber flags for a pair of flags with both S-flag and P-flag set. Since the structure of subscriber flags and subscriber table matches it can directly access the corresponding subscriber address in the subscriber table without further overhead. The index to the flag-pair in the subscriber flags is the same as the index to the corresponding subscriber in the subscriber table.

Subscribing to an event is done by scanning all subscribers for this event in the subscriber table to find the exact offset into the subscriber flags. Then, the corresponding S-flag in the subscriber flags can be set to indicate that the subscriber has now subscribed to the event. Unsubscribing is done by resetting the S-flag.

The amount of memory consumed for the internal data structures depends on the amount of total events (E) and the number of events each subscriber is interested in. Let N be the number of subscribers and E_i the number of events for which the ith subscriber ($i \in [1..N]$) has declared its potential interest, the following holds for a 16 bit processor, if $E \leq 255$ and $\sum_{i=1}^{N} E_i \leq 255$:

$$\begin{aligned} memory(indextable) &= 2E\,[byte] \\ memory(subscribertable) &= 2\sum_{i=1}^{N} E_i\,[byte] \\ memory(subscriberflags) &= \tfrac{1}{4}\sum_{i=1}^{N} E_i\,[byte] \end{aligned}$$

Example: 60 subscribers, potentially interested in three events each, 100 events in total lead to 560 bytes of constant memory (flash) usage and just 45 bytes of dynamic memory (RAM) usage.

C. Properties

The implementation described here has the following properties:

Memory overhead The consumed dynamic memory (RAM) per subscriber is only 2 bits for each associated event.

Guarantees It guarantees that a subscriber is not informed about events that were published prior to its subscription, hence a causal order of subscription and event delivery is maintained without using timestamps or queues.

Speed The time needed to find a list of associated subscribers for any event is constant and amounts to one memory read and one addition instruction (16-bit architecture and not more than 255 events and 255 subscriber assumed).

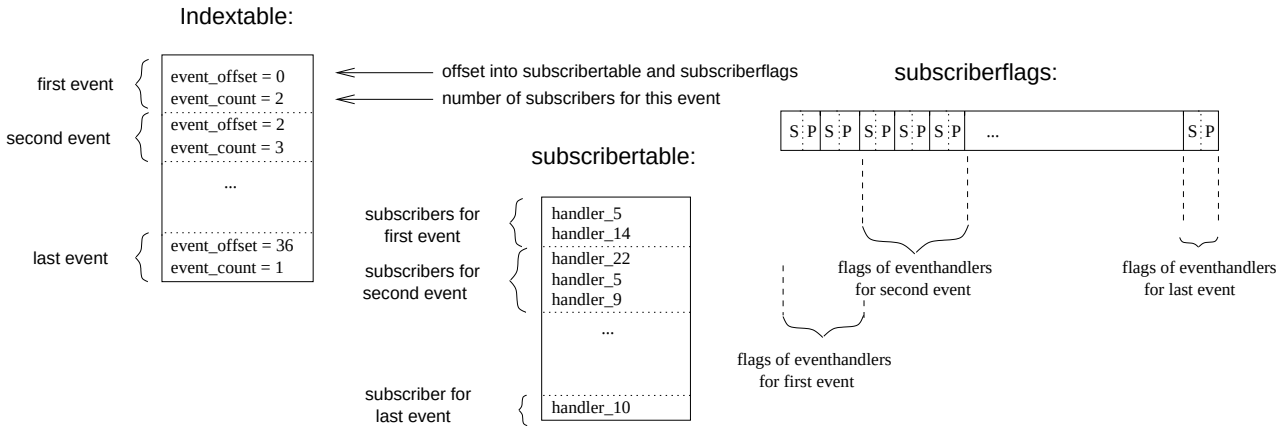
Flexibility Event handlers are not limited to subscribe to one event only, they can subscribe to multiple events and distinguish the source by their parameter. Also, an event can be delivered to more than one subscriber without any problems.

Stability Only one event can be pending per event handler and associated event at a particular time, i.e. any consequent events will be overwritten. This keeps the time needed by the blackboard to process events bounded, it can not be drowned in an event storm.

The implementation does not address the issue of safely accessing global data structures and avoiding race conditions. Whether this is an important problem depends on the Operating System (OS). If the OS does not interrupt subscribers, then this is only a problem in connection with Interrupt Service Routines (ISRs). Otherwise, standard techniques like semaphores can be used. Another approach are TinyGUYS [12]; it could be implemented with little overhead: Instead of writing directly to a global data structure, the data is written to a buffer (there is a buffer for each critical data structure). Before the blackboard would call any subscribers it would copy any updated buffer content into the corresponding global data structure. However, this approach assumes that a publisher is not interrupted itself.

V. Conclusion

In this paper, we have identified the need to distinguish between two main types of interaction between building blocks for WSN protocol implementations. One type is the actual passing of messages of packets between these blocks, which is relatively well covered by existing configuration languages. The other type is the less structured exchange of



meta information, like RSSI, between these blocks. While for this second type, also some solutions exist, but they should be complemented by the asynchronous, anonymous, publish/ subscribe style of interaction described in this paper.

This interaction style is powerful and can actually be efficiently implemented. The described implementation enables dynamic publishing and subscribing of events with little memory overhead. In future work, we plan to extend our implementation with configuration languages similar to TinyOS and integrate our concepts with the existing TinyOS implementations.

REFERENCES

[1] D. D. Clark and D. L. Tennenhouse, "Architectural considerations for a new generation of protocols," in *Proceedings of the ACM symposium on Communications architectures and protocols*. ACM Press, 1990, pp. 200–208.

[2] R. Braden, T. Faber, and M. Handley, "From protocol stack to protocol heap: role-based architecture," *ACM SIGCOMM Computer Communication Review*, vol. 33, no. 1, pp. 17–22, 2003.

[3] H. Balakrishnan, V. N. Padmanabhan, S. Seshan, M. Stemm, E. Amir, and R. H. Katz, "TCP improvements for heterogeneous networks: The Daedalus approach," in *Proc. 35th Ann. Allerton Conf. on Communication, Control, and Computing*, Urbana, IL, October 1997, http://daedalus.cs.berkeley.edu/publications/allerton.ps.gz.

[4] A. Festag, "Optimization of handover performance by link layer triggers in ip-based networks; parameters, protocol extensions, and apis for implementation," Telecommunication Networks Group, Technische Universität Berlin, Tech. Rep. TKN-02-014, July 2002.

[5] F. Bachmann, L. Bass, C. Buhman, S. Comella-Dorda, F. Long, J. Robert, R. Seacord, and K. Wallnau, "Volime ii: Technical concepts of component-based software engineering, 2nd edition," Software Engineering Institute, Carnegie Mellon University, Tech. Rep. CMU/SEI-2000-TR-008.

[6] E. A. Lee, "What's ahead for embedded software?" *IEEE Computer*, vol. 33, no. 9, pp. 18–26, 2000.

[7] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister, "System architecture directions for networked sensors," in *Proceedings of the ninth international conference on Architectural support for programming languages and operating systems*. ACM Press, 2000, pp. 93–104.

[8] 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 *Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*. ACM Press, 2003, pp. 1–11.

[9] D. D. Corkill, "Blackboard systems," *AI Expert*, vol. 6, no. 9, pp. 40–47, Sept. 1991.

[10] B. Oki, M. Pfluegl, A. Siegel, and D. Skeen, "The information bus: an architecture for extensible distributed systems," in *Proceedings of the fourteenth ACM symposium on Operating systems principles*. ACM Press, 1993, pp. 58–68.

[11] R. Braden, T. Faber, and M. Handley, "From protocol stack to protocol heap: role-based architecture," *SIGCOMM Comput. Commun. Rev.*, vol. 33, no. 1, pp. 17–22, 2003.

[12] E. Cheong, J. Liebman, J. Liu, and F. Zhao, "Tinygals: A programming model for event-driven embedded systems," in *ACM Symposium on Applied Computing*, 2003, to appear.