A Lightweight Web Service Approach for Querying Sensor Networks using the example of ZigBee

Helmut Dispert, Johannes Bönniger, Alkje Kalies
Jonas Kaufmann, Jan Küting, Sebastian Lampe
Michael Lodemann, Justus Rogowski, Benjamin Widmann

Kiel University of Applied Sciences, Germany
Faculty of Computer Science and Electrical Engineering
{johannes.boenniger, alkje.kalies, jonas.kaufmann, jan.kueting}@student.fh-kiel.de
{sebastian.lampe, michael.lodemann, justus.rogowski, benjamin.widmann}@student.fh-kiel.de
helmut.dispert@fh-kiel.de

ABSTRACT

We present the design and implementation of a unified access platform to sensor networks based upon *Web Services*. This allows other systems in heterogeneous environments to easily consume and process sensor network data transmitted by sensor nodes. Clients can subscribe to different node or value types while data filtering is performed on the server side. The system design is modular and all components are exchangeable. The prototype version uses *ZigBee* as an example wireless communication technology. To demonstrate the capabilities of the system, four different clients were developed that demonstrate different access strategies in homogeneous and heterogeneous environments.

Categories and Subject Descriptors

C.2.0 [**Computer Systems Organization**]: Computer Communication Networks.

General Terms

Design, Experimentation, Languages, Standardization

Keywords

ZigBee, Wireless Sensor Networks, Web Services

1. INTRODUCTION

When designing *Sensor Network Systems*, there are often proprietary methods for querying sensor nodes. On the other hand, a simple integration of sensor networks in heterogeneous systems might be useful for various applications. A commonly used way of sharing data is *Web Services*, which provides method invocation and data exchange across different platforms. To get these two aspects together, it would be convenient to be able to query sensor networks through *Web Services*. In contrast to existing solutions like TinyDB [1], the approach is not limited to a specific platform. Such a system would also efficiently decouple data acquisition from data processing through business rules and data presentation. Further, multiple applications would be able to consume data from the same sensor network.

We present a prototype that uses a lightweight protocol for querying *ZigBee* networks. However, the design is modular to support seamless integration of multiple sensor networks. The implementation strategy behind querying methods - based on *Web Services* - is very lightweight. We implemented basic access methods that are absolutely necessary for querying the network. All other methods are considered application-specific and therefore should be implemented by the business tier.

Similar work is done by Woo [2]. There the main attention is on creating an embedded component that provides a rich *API* based on *Web Services*. We chose to use common PC infrastructure together with an *USB* dongle to *sniff* network information. This greatly improves extensibility and makes the use of other Sensor Network types much easier using less development time.

2. SYSTEM ARCHITECTURE

The system architecture represents the concept and idea of reading wireless sensor-network data, filtering and dispatching it to multiple client applications for further use. The following explains the different layers of the system, how data-flow is handled and storage is realized. This section describes the system architecture from a conceptual and therefore a technology-independent point of view.

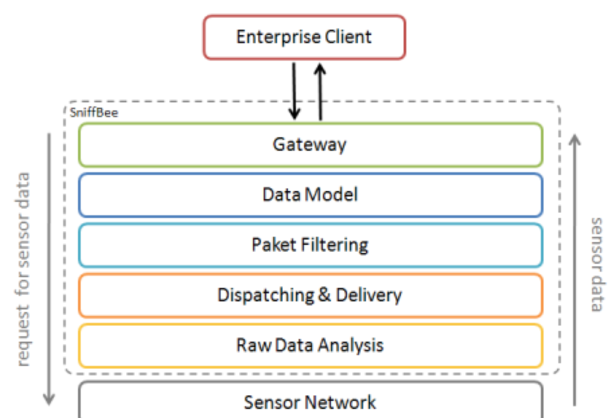

Figure 1: System architecture (conceptual view)

2.1 Raw Data Analysis

The lowermost level of the technological environment is represented by the sensor network. Sensor network nodes are able to send many kinds of frames including command-frames, beacons, ACK or data-frames of different size.

Regarding the system's *Raw Data Analysis* layer, network packets are intercepted during continuous observation of the sensor network by using an adequate hardware interface. Having promiscuous mode functions, the hardware interface must not necessarily be an active node of the sensor network. Therefore, in its range, it has the ability to *sniff* network packets. This way it is even possible to run different analysis components for different technologies at the same time.

After intercepting a packet from the network, it is classified in order to filter only required packets. Packets are in a different format depending on transmission technique, protocol and sensor type. So an analysis component is introduced which is able to extract all relevant information out of the packet's header and body. This information is transformed into a technology-independent representation format which contains predefined meta-data for example a sensor name or sensor profile corresponding to the systems Data Model (see chapter 2.4).

The system is designed to be modular containing highly exchangeable components. The *Raw Data Analysis* layer follows this directive in order to provide flexibility. The system is able to host different components - even at the same time - to provide network connectivity for example to *Bluetooth* or other wireless technologies.

2.2 Dispatching and Delivery

To support multiple endpoints such as session storage, message queues, databases or logging systems, the *Dispatching and Delivery* layer offers an abstract definition of a delivery strategy. Following a strategy, sensor data packets are routed to the designated endpoint specified in the concrete implementation.

The implementation follows a technology-independent approach which enables the system to dispatch data to multiple targets by following a concrete delivery strategy per target endpoint. A plug-in model might provide the opportunity to add or remove other implementations of delivery strategies in further versions (see chapter 5).

2.3 Packet Filtering

Packet filtering allows clients to specify which data they are interested in by using a simple query language. Query statements are then used by the service to filter and transform sensor data packets. A more sophisticated way to implement such behavior might be to reconfigure the sensor network in a way that would 1) satisfy all client needs and 2) optimize the overall power consumption by eliminating duplicate packet transmission (see chapter 5). The current system just receives anything transmitted by the sensor network. Therefore, serving client requests is performed by filtering incoming packets based upon query compilation and execution as described in section 3.3. The following section is intended to give the reader a definition of the proposed query language.

Definition

A query Q is defined as $Q = \{s_1, s_2, \ldots, s_n\}$ where s_i is one separated query statement which is defined as $s_i = \{k, t, c\}$ where k defines what kind of sensor type s_t is requested, t represents any kind of transformation function (to transform requested sensor data value v) and finally c is any condition that must be satisfied in order to let a sensor data packet passing the filter.

Each statement of a query Q is applied on each incoming sensor data value v and its sensor type s_t in the following way: The packet is forwarded to the client if $s_t = k$ for at least one s_i in Q and the corresponding condition $c(v)$ are both true. The condition c might reference the original value v or its transformed representation $t(v)$. In this case $c(t(v))$ must be true. Any sensor data packet that passes the filter contains the transformed data value $t(v)$. If no transformation has been defined, t is defined as $t(x) = x$. If no condition is defined, c is defined as $c(x) = true$.

Examples

Listing 1 shows a query example that receives any temperature value from a device with the identifier 0000. This query statement defines k as *temperature*, t as $t(x) = x$ and c as *deviceid* $=$ 0000. Because there is no transformation function defined, the actual temperature value as received from the sensor network is delivered.

Listing 1: Simple query statement

```
select temperature
    where deviceid = '0000';
```

The second example as shown by Listing 2 is a similar query statement, but it contains a transformation function. Furthermore the transformed value is declared as *temp* in order to reuse the transformed value again inside the condition. The transformation function can include any construct the query language supports such as mathematical functions like $\ln(x)$, x^y or $\sqrt{x}$. The term temperature can occur multiple times inside t and c, but mixing different sensor types inside the same transformation function is not allowed since a query statement is applied for each single sensor data packet. We assume that each packet only contains one type of sensor value (see section 2.4).

Listing 2: Query statement with transformation

```
select transform_function(temperature)
       as temp where temp > 0;
```

2.4 Data Model

The *Data Model* layer represents the data model for single sensor packets used for the system-internal communication and realizes a session concept to manage multiple client connections. A single sensor data packet consists of a header and a body part. The header holds some meta- data about the *source*, *creation time*, *type* and how to interpret the attached data in the body. The body contains the sensor data payload represented as a floating point value.

Devices and sensor profiles of the monitored sensor network have to be defined in the server context (see chapter 3.4).

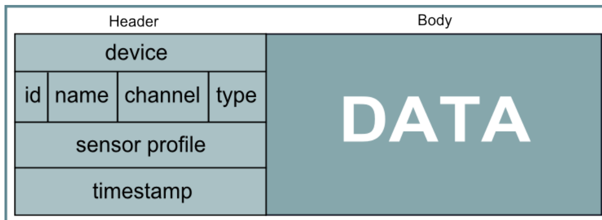

Figure 2: Structure of an internal data package

Depending on the stored sensor profile, the packet's content can be interpreted, according to scale factor and dimension, e.g. as a photo sensor measurement value of *100* luces from node *0x1F*.

As already mentioned, *SniffBee* supports multiple listening clients. By introducing a session model it can be assured that sensor data is assigned to a particular client. Since the Session model follows the paradigm of the pull concept, it is also necessary to have a store to cache data until it is requested by the client (not applicable for message queue adapter). All data-request related operations are bound to a session. Monitoring a sensor network with *SniffBee* requires opening a session. As long as a session is valid and active, all packets matching the filtering rules are cached and delivered. Closing the session stops this process.

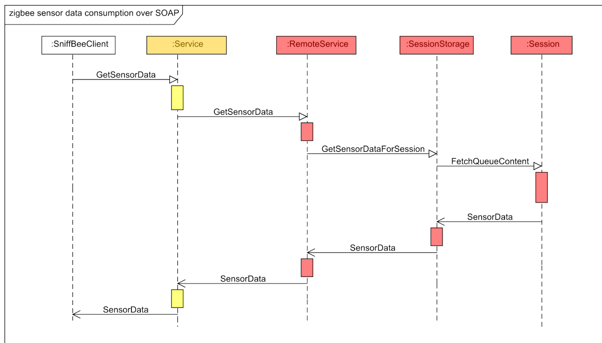

Figure 3: Client/server data exchange

2.5 Gateway

The *Gateway* layer provides a standardized interface to access data from sensor networks. This is important for the integration of those networks in enterprise environments. However, the architecture is designed to support different interfaces.

The difference between those interfaces is the method of data access. We provide *push* and *pull*-based mechanisms. By using push-based mechanisms like *Message Queues*, clients receive new data as soon as it becomes available. In contrast, pull-based mechanisms like *Web Services* have to be queried periodically. In push-based solutions, a client connects to the data source and receives data as a constant stream. There is no need to manually care about session data as the underlying technology like *Message Queues* handles those concerns. In contrast, pull-based solutions have to track session information for all queries. First, a client has to open a session. Second, a client might request meta-information about the network or specific devices and their abilities. Finally, a client requests data that has been cached since the last request periodically.

3. IMPLEMENTATION

In relation to the system architecture in figure 1, the following figure shows the implementation of the presented system.

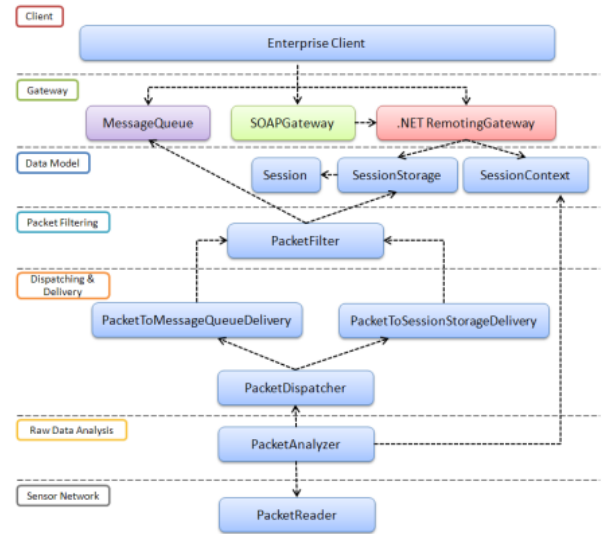

Figure 4: System implementation

The following section discusses all components in bottom up order, which is coherent to the direction in which data flows.

3.1 Raw Data Analysis

The *PacketReader* component in this concrete implementation communicates with a *ZigBee* peripheral unit (a *USB*-dongle). The component interacts with the dongle's native driver in order to configure its behavior. For example, the USB-dongle can interact as a router or end device. It even can be used as a *ZigBee* coordinator in order to manage a whole *ZigBee* network.

According to the *ZigBee* specification [3], the dongle is a fully configurable *ZigBee* sensor node which can operate as a *FFD*[1] or *RFD*[2]. An important ability of the dongle is to operate in *Promiscuous Mode* as well. In this mode, the device received all network traffic. Therefore all packets are received without any further differentiation.

For data analysis issues, raw data packets are passed to the *PacketAnalyzer*. Packets are classified into one of the following categories:

- **Stack-Command-Packages** represent internal command packets defined by the dongles' software stack. They are not transmitted within the sensor-network.
- **Command-Packages** are used for network-management issues like adding and removing devices or coordinator and router alignments.

[1] Full Function Device
[2] Reduced Function Device

- **Beacons** are packets that are sent periodically by special network nodes like routers or coordinators in order to announce their presence to other nodes.
- **Data-Packets** are the relevant packets for this application. They contain sensor data.

Actually, this list is incomplete, but missing categories are not relevant for the application, like *ACK*-packets for example.

After a packet has been identified as a data packet, the *PacketAnalyzer* extracts the sensor data value out of the packet's payload. The *ServerContext* (see chapter *sec:serverContext*) holds a list of *SensorProfiles* which is used to determine which kind of sensor data is located at which bit-position of the payload data. After the information has been extracted, the concrete sensor data and its sensor profile are repacked into a technology-independent format (*SensorDataPacket*). Finally, this new packet is passed to the upper next software layer - the *PacketDispatcher*.

3.2 Dispatching and Delivery

As mentioned before, the *Dispatching and Delivery* layer consists of two components to enable the multiple strategy approach: *PacketDispatcher* and *PacketDeliveryStrategy*. The *PacketDispatcher* reads the *SensorDataPackets* and distributes the data to the designated targets by calling the corresponding method implemented in the concrete *PacketDeliveryStrategy*. It acts as a data sink to the lowermost system layer which writes packets into a queue. Using a threaded loop, the dispatcher reads and removes the stored data and pushes it to the strategy-related targets. Therefore, the dispatcher holds a list internally containing the strategies available at run-time - calling each one for processing.

A *PacketDeliveryStrategy* implements the provided interface *IPacketDeliveryStrategy*. Thereby, the implementing class controls how to and where to publish the data. At this stage of development the system makes use of two strategies (see figure 5). The *PacketToSessionStorageDelivery* transmits incoming packages to a *SessionStorage* instance (which offers session management and per-session data caching functionality). The second one, *PacketToMessageQueueDelivery*, delivers the data to a Microsoft Message Queue.

As shown in the class diagram (see figure 5), the layer implements the following interfaces to fulfill the desired functionalities of routing sensor data packets:

- **IPacketDispatcher:** Multiplexing and forwarding of incoming packages into one or more packet delivery strategies.
  - *AttachDeliveryStrategy(IPacketDeliveryStrategy)*: Attaches the given delivery strategy to the dispatcher. As soon as a delivery strategy is attached, the dispatcher forwards incoming packets to the strategy.
  - *DeliverPacket(SensorData)*: Called by the packet analyzer to forward packets to this layer. Each packet is then forwarded to all attached delivery strategies.
- **IPacketDeliveryStrategy:** Interface to be implemented for a concrete endpoint. The *PacketFilter* verifies if the sensor data value satisfies a filter condition, before it is passed on.
  - *DeliverPacket(SensorData)*: Called by the dispatcher to deliver a package due to this delivery strategy.

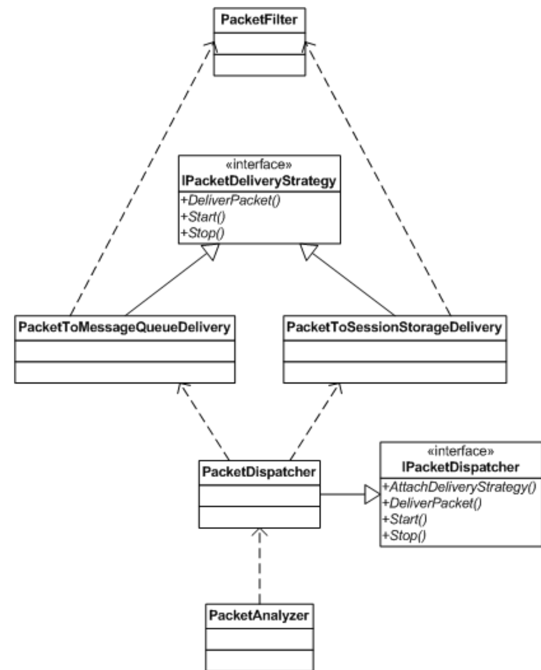

Figure 5: Dispatching a package

3.3 Packet Filtering

Any client might provide query statements during session creation. A client might define what kind of sensor data it is interested in and can expect data delivery that satisfies those needs. Packet filtering has been implemented server-side by query compilation and execution.

During session creation the server compiles any given query Q into a set of instructions forming the program P. The instruction set provides value comparison and transformation, higher-level mathematical functions and flow control instructions. We use a simple one-phase compiler. Instructions are implemented as being instances of corresponding classes for each specialized instruction type.

Program P is executed by a virtual machine (*VM*) for any incoming sensor data packet and receiving client respectively. The program input is the actual sensor data value v and its sensor type s_t. The output is the result of $c(v)$ and $c(t(v))$ respectively and the transformed value $t(v)$. Input and output data exchange is handled by making use of four registers the *VM* provides. The program executes all instructions in P in sequential order under consideration of flow control instructions.

To execute an instruction, the method m of the corresponding class for that instruction is called. The method m has access to control flow and registers through the instance of the *VM* that is passed to it.

System performance might be improved by compiling Q into native byte code. This strategy might benefit from new features such as support for expression trees which had been added to the *.NET platform* recently through the *2008* release.

3.4 Server Hosting and Configuration

Designed as an enterprise application, the *SniffBee* server is implemented as a Windows service. This allows the automated execution of the application. For testing purposes, there is also a console version of the server available (*SniffBee.ServerTTY*). To use the message queue as the delivery strategy corresponding to the *PacketToMessageQueueDelivery*, the *Microsoft Message Queue* must be installed and a message queue called *SniffBee* has to be created manually at this stage of development.

The configuration of the sensor network includes the definition of the basic *ZigBee* network infrastructure and the configuration details of each node. A simple network is defined by a network data object which holds the unique network id and the devices it contains. Possible devices of a ZigBee network are coordinators, routers and sensor nodes. A device object is defined by its *type*, *id*, *name*, *channel* and its assigned *sensor profiles*. The sensor profiles hold the relevant information needed to extract the data the clients are interested in from the raw data. This includes the unique name of the sensor, the dimension unit and the bit position, bit width and scale factor. A sensor profile could be assigned to various devices.

This configuration is specified in an XML file, which represents the serialization of the server context used by the system. Its location is specified in the application settings of the server.

For ease of creating and editing the network details, a configuration tool (Figure 6) is provided. It shows the network as a tree of devices and its assigned sensor profiles.

3.5 Gateway

We provide three different gateway interfaces: *Web Service* (pull-based), *Message Queue* (push and pull-based) and *.NET Remoting* (pull-based). While sensor data is pushed directly into the message queue by a specialized delivery strategy (see also chapter 3.2), *Remoting* and *Web Service* rely on a *SessionStorage* object which realizes packet queuing and storage and holds meta-information about a session.

Access to *SessionStorage* is realized through *Web Services* and *.NET Remoting*. However, *.NET Remoting* is limited to the *.NET platform*, *Web Services* is also suitable for heterogeneous environments. Three methods of pull-based querying can be distinguished: session management, meta-information and data retrieval. The Web Service methods as we implemented them shall be presented below.

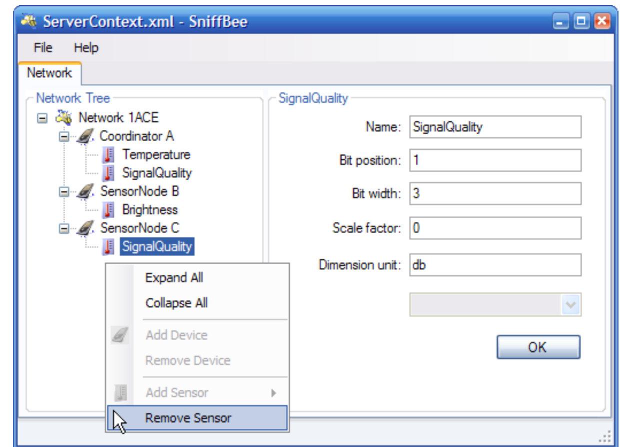

Figure 6: SniffBee Network Configuration Tool

- **Session Management** deals with establishing and closing a connection to the server and starting a session during which data can be received.
  - OpenSession
  - CloseSession
  - OpenQuerySession
- **Meta-information:** The following methods are used for querying the status and the composition of the network. With these methods, a client can receive meta-information about all available networks, their devices and their capabilities.
  - GetNetworkList
  - GetDeviceList
  - GetDevice
  - GetSensorProfile
- **Data Retrieval:** After opening a session, data can be received by using the following methods.
  - GetPacketCount
  - GetSensorData

4. DEMO APPLICATION

There is a high number of possible applications that could make use of the solution presented in this paper. Service Oriented Architecture (SOA) plays a growing role in today's industry for building enterprise applications. Accessing sensor networks through *Web Services* simplifies the integration of sensor networks into enterprise applications without any need to harmonize interfaces and communication. In production processes there is usually a gap between the business application and the technical application that controls and monitors the production process. Crossing this gap by using *Web Services* is an important step towards integrating two totally different views on production processes. There are several suppliers providing a middleware to integrate sensor values (whatever sorts) into the business application but they are all focused on *wired* technology and they rarely provide *Web Services* to access these measurements.

Table 1: Client overview

Client	Technology	Data Source	Used Sensor Data	Visualization	Purpose
1	.NET	SOAP Gateway	can be freely chosen via GUI	time-graph, that refreshes every second	real time curves of measured values
2	.NET	SOAP Gateway	luminosity values of a predefined sensor	color changing user interface reacting when a threshold exceeds	monitoring of values - reacting with alerts
3	Java	Microsoft Message Queue	all the values from the message queue	direct output of values to standard output	logging of measured values
4	Java	SniffQL	SniffQL query	command-line based input and output	specialized manual queries to sensor network

To demonstrate how the integration has been simplified, we provide four demo clients using the different interfacing methods.

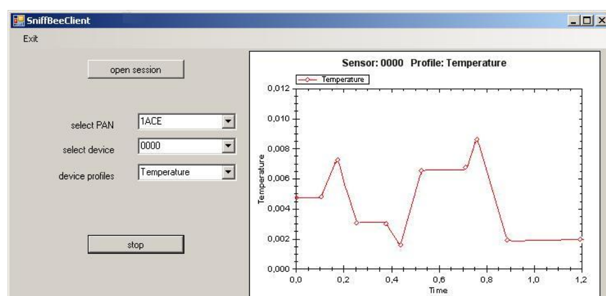

Figure 7: Example of a GUI that access the sensor network by using Web Services

5. FUTURE WORK

To allow multiple delivery strategies to be added to the system, we propose a plug-in model which makes it easy to implement the provided interface as well as build and integrate it into the running system.

While the system is at the moment passively sniffing all available data as well as the sensor nodes transmit all their sensor values, lots of overhead traffic is generated. Energy could be saved if the system would use configuration data and query information to configure sensor nodes in a way that satisfies the requirements of all clients. Furthermore we are interested in supporting actuators as well.

Finally, we would like to provide a programming framework to simplify the integration of diverse sensor technologies via plug-in development. As we have several adaption and extension abilities it seems plausible to provide an implementation framework with a clear rule-set on how to develop plug-ins to evolve the strategy portfolio or connect a further sensor technology approach.

6. CONCLUSION

In this paper, a middleware solution is presented to acquire data from wireless sensor-networks through *Web Services*. To archive flexibility, we provide interfaces for data collection and data distribution. Therefore, besides the presented delivery strategies further could be implemented. The same applies to the integration of further sensor-technologies. To proof the general concept we provide an end to end demo-system which uses *ZigBee* network communication and clients consuming the collected data via the developed strategies. Furthermore, to enhance usability, a user interface is introduced which allows easy configuration. Finally, a query language is presented to filter and manipulate sensor data in a familiar (database-like) way.

7. REFERENCES

[1] S. R. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong. Tinydb: an acquisitional query processing system for sensor networks. *ACM Trans. Database Syst.*, 30(1):122–173, 2005.

[2] A. Woo. A new embedded web services approach to wireless sensor networks. ACM, 2006.

[3] ZigBee Alliance - http://www.zigbee.org. *ZigBee Specification 4th quarter 2007.*