Programming Sensor Networks with State-Centric Services

Andreas Lachenmann, Ulrich Müller, Robert Sugar,
Louis Latour, Matthias Neugebauer, Alain Gefflaut

European Microsoft Innovation Center GmbH, Aachen, Germany
{andreasl, ulrichm, rsugar, llatour,
mattneug, alaingef}@microsoft.com

Abstract. This paper presents the uDSSP ("micro DSSP") programming model which simplifies the development of distributed sensor network applications that make use of complex in-network processing. Using uDSSP, an application is composed of state-centric services. These services interact by accessing the state of other services or by subscribing to changes of that state. uDSSP supports heterogeneous networks that consist of PCs, resource-rich sensor nodes, and resource-limited nodes with just a few kilobytes of RAM. The evaluation uses a non-trivial application to compare it to Abstract Regions and Tenet.

1 Introduction

In the future, sensor network applications will emerge that do more complex in-network processing than the simple aggregation (min, max, average, etc.) of most applications today. For example, in an elderly care application a body-worn sensor node determines the patient's activity level whereas nodes in the kitchen and in the dining room cooperate to infer the activities of daily living such as cooking and eating. Together with other nodes throughout the home they form the application that keeps track of the patient's activities. This information is useful to detect changes in the patient's behavior, which could be an indication of health problems.

Creating such distributed applications with complex in-network processing is still a difficult task. If such an application is – like most real-world applications today – directly developed on top of a sensor network operating system, the developer becomes easily distracted by low-level details such as sending packets and message formats. Furthermore, since application components are often tightly coupled with static references to specific nodes, the reusability of individual parts of the application is very limited. Therefore, in this paper, we present the uDSSP ("micro DSSP") programming model and middleware that addresses these problems. With its runtime system and a set of standard services, it relieves the developer of many recurring tasks such as dealing with communication or discovering nodes.

In our terminology, an *application* spans the whole network and provides the desired functionality to the user. It can include different classes of devices such as TelosB and Imote2 nodes as well as PCs. Such an application consists of services that are distributed over different nodes. The application is formed by combining services

and letting them interact. A uDSSP *service*, which is of similar granularity as a web service, provides a part of the application's functionality like sampling data or inferring the activities of an elderly person. Such a service is state-centric, i.e., it is built around its state and exposes this state to other services. An example for the state of a service is the data sampled and the result of the computation on that data (e.g., the activity of an elderly person). A service communicates with other services by invoking operations such as subscribing to state changes or retrieving the state.

By storing the results of their computation in their state and by exposing this state, the services facilitate a loosely coupled, subscription-based form of interaction. Thus, a service does not have to broadcast its results to all nodes or include static references in the code where to send them. This subscription-based interaction reduces coupling between nodes and fosters software reuse. It fits many sensor network applications that are event-based and transmit data only when it changes.

This model has been inspired by the ideas of the Decentralized Software Services Protocol (DSSP) [1] but has been tailored to the special characteristics of sensor networks. DSSP is a SOAP-based protocol with some pre-defined operations for state-centric services. uDSSP implements a subset of the DSSP operations. Instead of using SOAP, however, it encodes its messages in a small, binary representation.

We have implemented uDSSP for several platforms, including the .NET Framework on PCs, the .NET Micro Framework (e.g., for Imote2 nodes), as well as Contiki [2] and Mantis [3] (e.g., for TelosB nodes). In both .NET implementations and in Contiki, uDSSP uses 6LoWPAN for a seamless integration in existing IPv6 networks. Since no implementation of 6LoWPAN is available for Mantis yet, these nodes are limited to a custom mesh protocol. This protocol is also supported by our uDSSP implementations for .NET. All platforms share the same concepts and a common (application-level) network format. Therefore, it is possible to create distributed applications that incorporate different classes of devices.

The contribution of this paper is a novel, state-centric programming model to create sensor network applications with complex in-network processing. This model simplifies the development of such applications by fostering the reuse of parts of the application, by providing easy-to-use mechanisms for communication between services, and by including common functionality (e.g., to discover services). As shown in the evaluation, efficient real-world applications can be implemented with uDSSP.

The rest of this paper is structured as follows. Section 2 presents related work. In Section 3 we give an overview of our overall system. Section 4 details the use of the programming model. In Section 5, we then present evaluation results, including case studies of two non-trivial applications. Section 6 concludes this paper.

2 Related Work

With its network data types, nesC [4], the programming language used for TinyOS, supports the developer in creating distributed applications that incorporate different types of devices. However, the interaction is not handled by the runtime system but requires a custom implementation by the application developer – often with static references to the data's destination. A more high-level, network-wide programming abstraction is offered by database approaches such as TinyDB [5]. However, such an

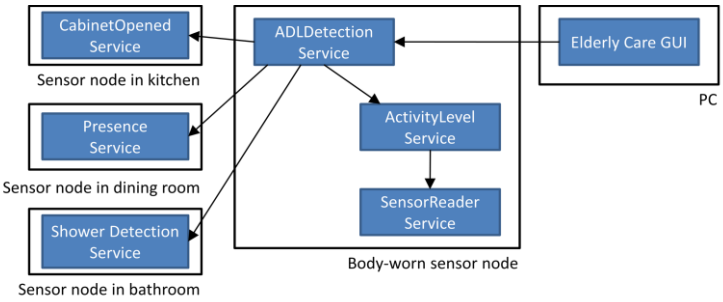

Fig. 1. Architecture of the elderly care application

approach is limited to simple queries. Li *et al.* [6] added support for events to DSware, another database-like middleware. However, they are limited to simple events that can be expressed in an SQL-like syntax. Macroprogramming languages such as Kairos [7] provide a way to create a network-wide program that is then executed in a distributed way in the sensor network. Unlike our approach, Kairos splits up a central program into individual threads and distributes them.

With its mechanism to register for certain data, TeenyLime [8] is similar to our subscription-based approach. Instead of exposing services as a structuring element, it shares data via a flat tuple space. Abstract Regions [9] uses a similar abstraction to exchange data. It forms groups of nodes based on properties such as hop distance or location. However, its data exchange mechanism is pull-based, i.e., nodes have to query the data of other nodes instead of being notified when it changes.

In Tenet [10], the sensor network consists of resource-limited motes and less constrained master nodes. This is similar to the network architecture of uDSSP. Unlike uDSSP, with Tenet, the application on the motes is written in a simple tasking language that is interpreted by the runtime system.

A design alternative for uDSSP would have been to use web services or Devices Profile for Web Services (DPWS) [11]. With its standard services, DPWS is similar to uDSSP but has not been designed for as resource-constrained devices as uDSSP: It requires the use of verbose XML messages. Tiny Web Services [12] have shown that web services can be implemented on resource-limited sensor nodes, though at the cost of some overhead. Instead of composing an application of services as with uDSSP, the architecture of Tiny Web Services seems to be targeted towards interaction with clients external to the sensor network.

3 System Overview

3.1 Architecture

Our network model is different from the early ideas of sensor network research with a flat, large-scale network of resource-limited nodes. Like most deployments today, the networks we target do not consist of thousands of nodes but rather of tens or hundreds. Furthermore, they can be heterogeneous and consist of different classes of devices. They include resource-constrained sensor nodes such as TelosB nodes, more

Table 1. Operations supported by uDSSP

Operation	Description
CREATE	Creates a new service instance
DROP	Removes the service instance
LOOKUP	Retrieves the partner services
GET	Retrieves the complete state
QUERY	Retrieves parts of the state
REPLACE	Replaces the complete state
UPDATE	Replaces parts of the state
SUBSCRIBE	Subscribes to state changes
SUBMIT	Calls an exposed function

Table 2. Example service definition

Item	Values
Service name	ShowerDetectionService
Service ID	0x45218A74
State	Bool Showering UInt16 HumidityLevel
Functions	-
Partner services	-

powerful nodes such as Imote2 nodes as well as PCs or servers. In this model, PCs are not just data sinks but fully participate in the network.

In our architecture, an application is composed of instances of services that communicate with each other. Each service instance has its state, a set of partner services, and functions that can be called by other services.

The overall application can be viewed as a graph of service instances, as shown in Fig. 1 for the elderly care application that will be used in Section 5.3.1. The arrows in the figure show the relationship to partner services, which tell a service which services it should, for example, subscribe to. The partners can be located on the same node or remotely in the network. In such an application, there is not necessarily a need for a single node to keep track of all service instances. Therefore, each service only has to have local knowledge about the services it interacts with.

The explicit definition of the service state distinguishes uDSSP services from web services. We think that exposing state fits well with the typical services we expect to find in a sensor network. For example, such services can make available their latest computation results on sensor data. Especially with our subscription mechanism, this model helps to compose the application from individual, loosely coupled services.

3.2 Runtime System

The main tasks of the runtime system are to handle requests, manage subscriptions, and deal with communication between services. There are two layers of the runtime system. The first one is service-specific and generated from the service definition. To access other services, this layer also includes generated proxies. We tried to keep this layer minimal to avoid duplicate functionality. The second layer is independent of the service. It dispatches messages to services and manages subscriptions.

In uDSSP, services invoke operations on other services. These operations are a subset of the operations specified by DSSP. Table 1 gives an overview of them. As described in Section 4, almost all operations are handled by the runtime system; the developer does not have to implement these standard tasks.

The CREATE and DROP operations are used to create a new instance of a service and to remove an existing one, respectively. The LOOKUP operation retrieves the set of partners of the service instance. GET, QUERY, REPLACE, UPDATE, and SUBSCRIBE all deal with the state of the service. These operations can be used to retrieve or modify either the complete state or parts of it. Furthermore, the SUBSCRIBE operation is used to subscribe for a notification when the state changes. This operation has a bitmask parameter, where each bit corresponds to a state variable. A bit set in the

mask indicates that the client wants to be notified upon any modification of the corresponding subset of the state. Such a bitmask is also used to select variables with QUERY and UPDATE. Because of the limited capabilities of sensor nodes, we selected this simple but efficient model. The last operation is the SUBMIT operation. It is used to call a function that has been exposed by the service which is not directly related to the service's state. This mechanism is similar to remote procedure calls and web services.

3.3 Message Encoding

uDSSP messages are encoded in a binary format that is based on the service definition. Since we assume that two communicating services share knowledge about the service definition, the messages do not include any metadata but are just the concatenation of the data fields. In fact, the payload of some responses can only be decoded if the service specification and the original request are known. Compared to binary encodings of XML such as CBXML [13] this scheme reduces data size even more.

For instance, the size of an example QUERY message (excluding the headers of layers below) is just 14 bytes. In contrast, a SOAP-based DSSP message with similar functionality has more than 900 bytes even if each identifier consists just of a single byte (as suggested to reduce the size of web service requests [12]).

4 Using the Programming Model

In this section, we describe our programming model in more detail. First, we explain how a service is defined, what the compile-time tools generate from that definition, and how the service is implemented. Then we describe uDSSP's standard service and how an application is composed.

4.1 Defining Services

For our programming model, it is important to explicitly specify the state and interface of a service. The format of this description, however, is not a key component of our approach. In our implementation, a service is defined in a custom XML format.

Table 2 shows an example for the information contained in such a service definition. The name of the service ("ShowerDetectionService") is used to refer to the service in the source code. The ID, in contrast, is a (random) 32-bit value that is used to identify the service type in network messages. Both the name and the ID have to be unique in an application. While the ID identifies the service type and the corresponding message format (i.e., its interface), the service name is used to identify the implementation in the source code. At runtime, there can be several instances of a service running on a single node. Each instance is identified by the combination of the node's address, the service ID, and an automatically assigned instance number.

Besides simple data types such as integers of various lengths, uDSSP currently supports structs, arrays, and strings. In the example, the state consists of the "Showering" and "HumidityLevel" variables. No functions are exposed by this service.

```
public class ActivityLevelService : ActivityLevelServiceStub {
   /// Called when the service instance is created
   override protected void OnCreate(ServiceReference[] partners) {
      srProxy = new SensorReaderServiceProxy(this);
      srProxy.Bind(LOCAL_HOST, 0);
      srProxy.Notification += new SensorReaderServiceEventHandler
          (SensorData);
     SensorReaderServiceMask mask = new SensorReaderServiceMask();
     mask.HaveAccelX = true;
     mask.HaveAccelY = true;
     mask.HaveAccelZ = true;
     srProxy.Subscribe(mask, 0, 0, 100);
   }

   /// Notification called with new sensor data
   void SensorData(SensorReaderServiceProxy sender,
       SensorReaderServiceEventArgs e) {
     activityLevel = ComputeActivityLevel(e.State.accelX,
       e.State.accelY, e.State.accelZ);
     ActivityLevelServiceMask mask  = new ActivityLevelServiceMask();
     mask.HaveActivityLevel = true;
     NotifyUpdate(mask);
   }
   ...
}
```

Fig. 2. Sample of a .NET service implementation

4.2 Code Generation

A compile-time tool generates source code (C or C#) from the service definition. First, it generates the service-specific part of the runtime system. This code is responsible for message encoding and decoding as well as for handling all requests. If possible, it replies to the request without requiring interaction of user-created code.

Second, the code generator creates a client proxy for the service. Another service can bind this proxy to a specific instance and invoke operations on it. As the counterpart of the service-specific part of the runtime system, the proxy deals with communication. The user of the service only has to call a simple function of the proxy.

Finally, the code generator creates a template of the service implementation. The developer just has to fill the function prototypes given there. Those include the functions exposed in the service definition as well as callbacks that are invoked by the runtime system when some operations are performed. For example, the service can react to the case when no subscriber is present any longer and stop sampling data. If these callbacks are not needed, they can be left empty.

4.3 Implementing a Service

Reacting to events from the runtime system, adding the functionality of the service, and updating the state are the only parts the developer has to write. Common tasks, such as communicating with other services are handled by the runtime system.

Fig. 2 shows parts of a service implementation for the .NET Micro Framework. The implementations for Mantis and Contiki vary slightly because they are not object-oriented. The service is the ActivityLevelService, which determines the activity of an

elderly person based on accelerometer data. The "OnCreate" function is called when instantiating the service. As specified with the "mask" variable, it subscribes to the three axes of acceleration data of the SensorReaderService on the same host.

"SensorData" is the function registered for receiving the notifications of the SensorReaderService. Here the service computes the activity level and notifies its subscribers of the state change. The "mask" parameter tells the subscribers which part of the state has been modified. Rather than sending automatic notifications after each change, the developer has to call the "NotifyUpdate" function in order to make sure that subscribers are only notified when the data is in a consistent state. The user code just modifies local variables and calls functions of the runtime system and proxies.

To simplify application development, all calls to other services are blocking. Therefore, the developer does not have to deal with replies that arrive asynchronously. The matching to the request is done by the runtime system. If no reply arrives, the blocking functions return with an error after a timeout. Since .NET and Mantis support multithreading, the implementation of synchronous communication is straightforward there. For Contiki, each such call is a so-called protothread [14].

4.4 Composing an Application

An application is composed by combining instances of services running throughout the network. As shown in Fig. 2, a connection between two services is created by instantiating a proxy and binding it to a service instance. The example refers to a static node address (in this case the local node). However, using the Discovery and Metadata Services (see Section 4.5), it is also possibly to find such partner services at runtime. Then it can, e.g., subscribe to all temperature services in the living room.

Other than binding to an existing instance, a service can use the proxy to create a new service instance – if the code for executing it is installed. In that case, it can pass references to other services as parameters. This way a generic service can be reused unmodified without the need for adding references to specific nodes in the code.

4.5 Standard Services

A key component of uDSSP is that it includes a set of standard services that we expect to be useful for many applications. We identified the following services by building several non-trivial applications:

Discovery Service: The Discovery Service detects nodes that have joined or left the network. Subscribers of this service can react to changes in the network and query more information from new nodes using the Directory Service and the Metadata Service. The Discovery Service is fully implemented on nodes with sufficient resources. On resource-constrained Mantis and Contiki nodes, it refers to another node.

Directory Service: The Directory Service keeps track of all service instances running on the local node. It returns their service IDs and instance numbers. More information can be retrieved by sending a LOOKUP request to the service instance.

Table 3. Memory size of the runtime (in bytes)

	Program memory	RAM
Mantis	10,180	2,188
Contiki	13,055	1,108
.NET MF	44,808	5,346

Metadata Service: The Metadata Service links a node to the physical environment it is deployed in. It provides information about the sensor node's identity, capabilities and location. The user can set the metadata upon deployment.

Deployment Service: The Deployment Service is used for installing new services on a node at runtime (assemblies for .NET or dynamically linked ELF files for Contiki and Mantis). Since each node potentially executes different services, uDSSP cannot leverage existing code distribution mechanisms.

5 Evaluation

In this section, we present evaluation results from experiments with real sensor nodes. We present results about the memory footprint of uDSSP and its performance. Furthermore, we describe how uDSSP can be used in real-world applications and how it compares to other approaches.

5.1 Memory Footprint

Table 3 shows an overview of the memory consumption of uDSSP. These numbers are just the size of the runtime itself; they exclude the operating system, the network stack, and services. The Mantis and Contiki implementations consume just 10-13 KB of program memory and 1.1-2.2 KB of RAM. The Mantis implementation consumes more RAM since this number already includes the reserved stack space for the uDSSP threads. In Contiki, our implementation is based on stack-less protothreads [14]. Instead, since protothreads cannot use local variables, some (reentrant) functions use pointers to store their state in variables passed as parameters. This and the use of IPv6 with its longer addresses increase the code size of the Contiki implementation. For the .NET Micro Framework implementation size limitations are less stringent because it is executed on less constrained devices. Even there, uDSSP also needs just a few KB.

If services are added, the service-specific part of the runtime system consumes at least 2-3 KB of program memory. However, the actual size and also the size in RAM largely depend on the service itself. For example, the CabinetOpenedService, which will be described in Section 5.3.1, needs 2.6 KB of program memory on Mantis. If not the full functionality of uDSSP is needed, both the size of the runtime system and the size of the generated code can be reduced further by deactivating optional functionality with *ifdefs*.

5.2 Performance of the Runtime System

Although we expect that services typically do not send requests at a high rate, the time for processing requests gives a good indication about the performance of

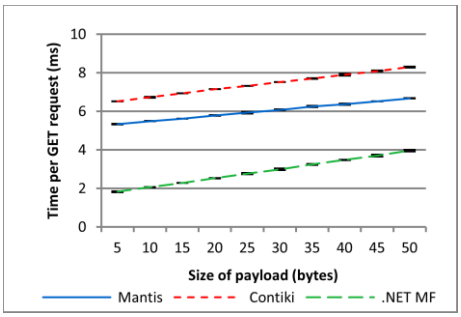

Fig. 3. Time in milliseconds for processing a local GET request, including (almost invisibly small) 95% confidence intervals

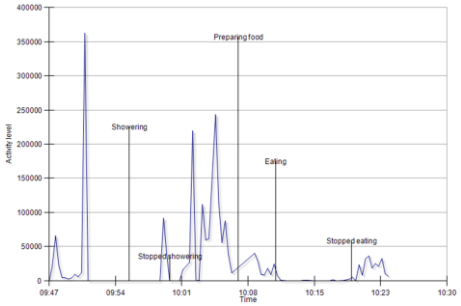

Fig. 4. Activities detected by the elderly care application

uDSSP's runtime system. To get these numbers, we repeatedly executed a GET request for arrays of different sizes. Although the results shown here are limited to GET requests, the performance of the other request types supported by uDSSP are similar. For example, a local QUERY request takes about 1% longer than the GET request since the bitmask has to be included in the request and evaluated by the receiver.

Fig. 3 shows the results for requests within a single node. The Mantis and Contiki services were run on a TelosB node whereas the .NET Micro Framework implementation was executed on an Imote2. With increasing sizes of the state, the delay grows slightly by approximately two milliseconds. We attribute the difference between Mantis and Contiki to the use of IPv6 addresses in the Contiki implementation. In a real application that includes some other functionality, the difference would be smaller.

For remote requests, the processing time largely depends on the bandwidth of the radio channel and the MAC layer protocol. Since uDSSP is independent of those low-level communication mechanisms, these results are not meaningful to evaluate the performance of our runtime system. To give a general idea of the performance, depending on the platform and the size of the request, the processing time for a request to a node in the local neighborhood was measured between 29 and 43 ms.

Using Contiki's online energy estimation mechanism [15], we measured the energy overhead of an application on TelosB that sends a notification to a neighboring node every minute. Compared to a highly optimized, non-uDSSP application that implements the same functionality with a simple best-effort network protocol, the overhead of uDSSP is less than 0.07 mW, which should be negligible for most applications.

5.3 Case Studies

In this section we present two applications we have implemented with uDSSP. We use the first one to compare uDSSP with other programming models and the second application to show that it can be used to implement a wide variety of real-world applications. Both scenarios have been taken from the WASP project [16].

5.3.1 Elderly Care

With an aging population, improving care in the home of elderly people – instead of having them move to a nursing home – becomes more and more important. By monitoring their activities of daily living, sensor networks can help elderly people living alone and give their relatives the comforting information that they are doing well.

Fig. 1 in Section 3.1 shows the general architecture of the corresponding uDSSP application. The patient has a body-worn sensor node with an accelerometer to monitor the overall activity (e.g., attached to a wireless emergency button around the neck). We have selected an Imote2 for this part because it has more processing resources. Besides the body-worn node, the application consists of additional sensors nodes that are deployed throughout the home (e.g., TelosB nodes). We assume sensor nodes to be deployed in the kitchen, the dining room, and the bathroom. Using the input from these sensors, our application can detect if the patient is preparing a meal, eating, or showering. With the modular approach of uDSSP, this application could be easily extended with additional services.

The CabinetOpenedService sends a notification when the fridge or a cabinet in the kitchen has been opened. Our implementation of this service simply uses the light sensor which is available on most sensor nodes today. The PresenceService uses a pressure-sensitive foil on a chair to detect if somebody is sitting there. The ShowerDetectionService is deployed on nodes in the bathroom. If the humidity level is above a threshold, it assumes that somebody is having a shower. All of these services just notify their subscribers of state changes and do not send their raw data. Unlike scientific monitoring applications such as habitat monitoring, the users of this application are not interested in the raw sensor data. Therefore, it is sufficient if the services send messages when they detect an event and can truly benefit from in-network processing.

The application makes use of uDSSP's discovery functionality to incorporate several instances of each of these services. Using the standard services such as the Directory Service and the Metadata Service (see Section 4.5), it discovers services (e.g., the CabinetOpenedService) on all nodes and determines in which room the nodes have been deployed. Only if the location fits the expected room (i.e., in this case the kitchen), it subscribe to this service.

On the body-worn node, the SensorReaderService provides the data from the sensor board – in our case the acceleration values only – to the subscribers. The ActivityLevelService samples data at a frequency of 10 Hz, adds the values for the three axes of each sample together, and periodically computes the variance of the values. The results give a good indication of the patient's level of activity [17]. Finally, the ADLDetectionService uses the activity values and information from sensor nodes in the apartment to determine the patient's activities of daily living. For this purpose, it implements some simple rules such as if the fridge and the cabinets in the kitchen have been opened, the patient probably prepares a meal. The GUI subscribes to services running on the sensor node worn by the patient and displays its results.

We deployed this application in an apartment and monitored the activities of the person living there. Fig. 4 shows an example of the morning activities detected by the application in this deployment. Using the sensor nodes deployed throughout the house, the application detects the activities of showering, preparing a meal, and eat-

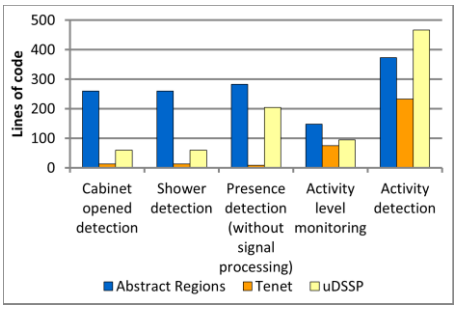

Fig. 5. Lines of code for the elderly care application

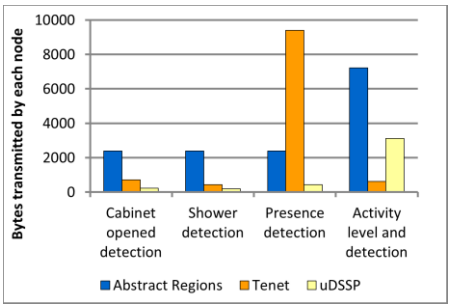

Fig. 6. Bytes transmitted in the elderly care scenario

ing. Furthermore, it shows the activity level of the person with the body-worn sensor node. For example, the activity level also shows the reduced movements when the person sits at the table to eat breakfast. Even with the simple sensors we use, the application can give useful hints about the patient's status and activities.

To compare uDSSP with other approaches, we have implemented (almost) equivalent applications in other programming models: Abstract Regions [9] and Tenet 2.0 [10]. We tried to optimize all versions as much as possible regarding lines of code and bytes transmitted. The Abstract Regions version does not include support for multihop routing and discovery of other services; each node announces itself only in its radio range. In the Tenet implementation, cabinet and shower monitoring, which just check the threshold of a periodic sensor reading, can be easily implemented in Tenet's tasking language. However, these small programs are already very close to the maximum program size supported by Tenet. The complexity of presence detection, which requires some processing of the sensor signal, was too high to implement in this language. Therefore, the raw data has to be transmitted to the less constrained master node for processing in a C application. Finally, since the activity detection requires input from other nodes, this functionality is also not supported by Tenet's tasking language and has to be implemented on the master node in C. Unlike the size limitations, which could probably be modified, it is a fundamental principle of Tenet that only the less-constrained master nodes can process input from other nodes.

In Fig. 5 we compare the lines of user-written code to implement the application. For Abstract Region, these numbers do not include changes that were necessary to its runtime components in order to support nodes that are part of several regions. For Tenet, the numbers are quite low on the sensor nodes because the pure functionality of some nodes is readily available in its tasking language. However, as described above, some functions cannot be implemented in this language and have to be run on the master node. Furthermore, since Tenet's language requires the use of numbered attributes instead of meaningful variables, writing applications is more error-prone than the lines of code suggest. uDSSP is somewhere between Abstract Regions and Tenet. However, compared to Tenet, it provides access to the full functionality of a node and includes additional support for discovering new nodes. This discovery mechanism is mostly responsible for uDSSP's higher numbers for activity detection.

Fig. 6 compares the number of bytes sent in the first 40 minutes in an exemplary run of the sequence shown in Fig. 4. These numbers only include the bytes sent on the application layer because the underlying protocols are not important for this compari-

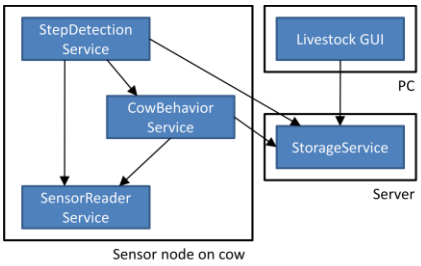

Fig. 7. Architecture of the livestock application

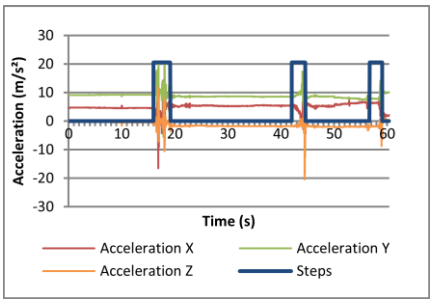

Fig. 8. Acceleration readings and steps detected

son. For Abstract Regions and uDSSP, the activity node transmits many bytes since the activity level is sent frequently to the GUI. These numbers are comparatively small for Tenet because they include only the pure application data in the messages sent by the C application. Unlike with uDSSP, the GUI application cannot benefit from the programming model and has to parse the messages manually. On the presence detection node, the numbers are significantly bigger for Tenet because the node has to transmit its raw data for processing to a master node.

Abstract Regions has significant overhead for two reasons. First, although we increased the interval from 1 s to 30 s, it still sends periodic beacons to announce itself to the other nodes in the neighborhood. Second, due to its pull-based data sharing approach, a node interested in data has to periodically query the data sources. In an application where data like the result of the shower detection changes very infrequently, the subscription-based model of uDSSP is preferable.

For uDSSP, these numbers include the overhead for discovering services and for subscribing to them (about 150 bytes for the cabinet node). In a static network, this is a one-time overhead. Without that, the shower detection node, for example, just sends 34 bytes. This number is comparable to an optimized manual implementation.

We are convinced that these results can be transferred to other applications. Tenet is suited well if the resource-limited nodes do only very simple processing and if the developer switches to its new, unfamiliar programming language. The pull-based model of Abstract Regions is of advantage if the observed data changes faster than updates are needed by the nodes interested in this data. uDSSP supports this use case also well by setting a minimum interval between notifications. Furthermore, it offers a good compromise between expressiveness of the programming language, complexity of the source code, and high efficiency for infrequent events.

5.3.2 Livestock Monitoring

Dairy farmers have to deal with claw health problems of their cows. If these problems are detected early, they can be treated before the cow is seriously impaired. However, to reduce costs, many farmers have to increase the size of their herds. Therefore, they have less time to spend with each cow. By continuously monitoring the activities of the cows with a wireless sensor network, less monitoring by the farmers is needed.

In this application, we focus on two aspects: the proportion of the time the cows are standing or lying and the number of steps they take. For this purpose, we attach a

sensor node with an accelerometer to a leg of each cow. Using the accelerometer as a tilt sensor, we distinguish between cows that are standing and lying. Furthermore, by computing the variance of acceleration readings, we detect the steps of a cow.

Fig. 7 shows an overview of the services running in this application. There can be many sensor nodes present that run such services for one cow. The SensorReaderService interfaces with the node's sensor board. Two services subscribe to the acceleration data provided by this service: the CowBehaviorService, which monitors if the cow stands or lies, and the StepDetectionService, which determines the number of steps of the cow. These two services subscribe to the SensorReaderService at different notification rates (1 Hz and 50 Hz, respectively). When the cow is not standing, no steps have to be detected and the subscription of the StepDetectionService can be temporarily released. The SensorReaderService is notified of the currently needed maximum rate and can adjust its sampling rate accordingly. Both processing services deliver their results to a generic StorageService outside the sensor network. A GUI application retrieves the data from the storage service and subscribes to it in order to be notified when new data from a cow arrives. Alternatively, the GUI or an application-specific storage service could directly subscribe to the cow services.

To show the practicality of this application, we performed some experiments on a farm. Since there is always the risk of injuring the animal when attaching or removing the sensor node, in this application it is important that software updates can be installed wirelessly. With uDSSP, this is the task of the Deployment Service (see Section 4.5). Fig. 8 presents some results of the StepDetectionService and the corresponding raw acceleration data. In this example, there were three steps detected during one minute. It should be noted that the raw data shown here has been made available for testing. To reduce network traffic and energy consumption, the application only provides access to the behavior and number of steps detected.

6 Conclusions and Future Work

As we have described in this paper, uDSSP provides a programming model and the corresponding middleware to create applications that do complex in-network processing like the applications in Section 5.3. We are convinced that such applications will be among the first sensor network applications to be widely used. Using the abstraction of state-centric services, parts of the application can be developed independently and later be combined. Services do not have to deal with the users of their data. Other services that are interested in the data will be notified automatically or can retrieve it when needed. With its higher level of abstraction, uDSSP helps the programmer to focus on the actual functionality of the application.

The runtime system of uDSSP is small and efficient. Furthermore, the evaluation shows that it is possible to develop a wide range of applications with uDSSP. Compared to other approaches, uDSSP offers a good compromise of flexibility and efficiency.

With the same programming interface and communication protocols available on PCs, sensor nodes are no longer simple data suppliers attached to a serial port: They can invoke services outside the sensor network and fully participate in a network consisting of sensor nodes and IPv6-capable computers. Therefore, uDSSP will enable exciting new sensor network applications that integrate with other networks.

Regarding future work, we are planning to add security mechanisms that restrict the access to nodes and services. Especially if sensitive data like in the elderly care application is transmitted, this is necessary for the system to be accepted by the users.

Acknowledgments

This work is partially financed by the European Commission under the Framework 6 IST Project "Wirelessly Accessible Sensor Populations (WASP)".

References

1. Nielsen, H. F., Chrysanthakopoulos, G.: Decentralized Software Services Protocol – DSSP/1.0. [Online] http://purl.org/msrs/dssp.pdf.
2. Dunkels, A., Grönvall, B., Voigt, T.: Contiki - a Lightweight and Flexible Operating System for Tiny Networked Sensors. In: Proc. of the Worksh. on Embedded Netw. Sensors. (2004)
3. Bhatti, S., et al.: MANTIS OS: An Embedded Multithreaded Operating System for Wireless Micro Sensor Platforms. Mobile Networks and Applications, 10, (2005) 563-579.
4. Gay, D., et al.: The nesC language: A holistic approach to networked embedded systems. In: Proc. of the Conf. on Programming Lang. Design and Impl. (2003) 1-11.
5. Madden, S. R., et al.: TinyDB: An acquisitional query processing system for sensor networks. ACM Trans. Database Syst., 30, (2005) .
6. Li, S., et al.: Event Detection Services Using Data Service Middleware in Distributed Sensor Networks. Telecommunication Systems, 26(2-4), (2004) 351-368.
7. Gummadi, R., Gnawali, O., Govindan, R.: Macro-programming Wireless Sensor Networks using Kairos. In: Proc. of the Conf. on Distrib. Comp. in Sensor Syst. (2005)
8. Costa, P., et al.: Programming Wireless Sensor Networks with the TeenyLime Middleware. In: Proc. of the Int'l Conf. on Middleware. (2007)
9. Welsh, M., Mainland, G.: Programming Sensor Networks Using Abstract Regions. In: Proc. of the 1st Symp. on Networked Systems Design and Implementation. (2004) 29-42.
10. Gnawali, O., et al.: The Tenet Architecture for Tiered Sensor Networks. In: Proc. of the Conf. on Emb. Netw. Sensor Syst. (2006) 153-166.
11. Chan, S., et al.: Devices Profile for Web Services. (2006)
12. Priyantha, N. B., et al.: Tiny Web Services: Design and Implementation of Interoperable and Evolvable Sensor Networks. In: Proc. of the Conf. on Emb. Netw. Sensor Syst. (2008) 253-266.
13. Conner, M.: CBXML: Experience with Binary XML. In: W3C Workshop on Binary Interchange of XML Information Item Sets . (2003)
14. Dunkels, A., et al.: Protothreads: Simplifying Event-Driven Programming of Memory-Constrained Embedded Systems. In: Proc. of the Int'l Conf. on Emb. Netw. Sensor Syst. (2006) 29-42.
15. Dunkels, A., et al.: Software-based On-line Energy Estimation for Sensor Nodes. In: Proc. of the Worksh. on Embedded Networked Sensors. (2007)
16. WASP consortium: WASP project web site. [Online] http://www.wasp-project.org/.
17. Lo, B., et al.: Real-Time Pervasive Monitoring for Postoperative Care. In: Proc. of the Worksh. on Wearable and Implantable Body Sensor Networks. (2007) 122-127.