Energy-optimized Data Serialization For Heterogeneous WSNs Using Middleware Synthesis

D. Pfisterer, M. Wegner, H. Hellbrück, C. Werner and S. Fischer
Institute of Telematics, University of Lübeck,
D-23538 Lübeck, Germany,
{pfisterer, wegner, hellbrueck, werner, fischer}@itm.uni-luebeck.de

Abstract—Developing applications for resource-constrained devices is an intricate task in itself and additionally requires in-depth domain expertise to optimize aspects such as communication overhead, resource usage and energy consumption. Frequently, these refinements are omitted because they are time-consuming, laborious and error-prone. Hence, automating these aspects lets developers and applications intrinsically benefit from the wisdom of experts. In this paper we propose a combined approach to WSN development that is comprised of our novel data type serialization scheme microFibre and our previously published work FABRIC to generate custom tailored, lean code for heterogeneous devices. We present measurements showing that microFibre clearly outperforms existing well-known solutions in encoding quality while only moderately increasing the application’s footprint.

I. INTRODUCTION

Nowadays, embedded systems are constant companions of our daily life. Countless of everyday applications rely on these unimposing helpers. Mostly invisible, they are the core of nearly every device ranging from simple washing machines to game consoles. The advent of wireless sensor networks (WSNs), which are in essence tiny embedded systems augmented with sensors and a wireless interface, has enabled a fundamentally new class of applications. And, while software development for customary embedded systems is already a challenging task in itself, the unique characteristics and requirements of WSNs further aggravate this situation.

Sensor networks may consist of hundreds or thousands of sensor nodes while an individual node may be only a few cubic millimeters in size [1]. The tiny form factor imposes strict constraints on the size of sensors, memory, processor and energy supply. As a consequence, developing applications for these massively distributed systems with only very scarce resources requires entirely new paradigms to counter the arising challenges. Software for WSNs must be optimized towards energy consumption to maximize the lifetime of the network. Additionally, software requirements include small footprint and minimal resource usage of memory and CPU. A typical WSN deployment also comprises gateway nodes and backend computers which are usually more powerful computing devices. Hence, heterogeneity, energy awareness and harsh resource constraints are constantly hanging over the heads of developers like the sword of Damocles [2], [3]. Hiding these intricate communication aspects from the developer is beneficial in order to reduce the overall design complexity of applications.

In this paper we propose microFibre, an optimized data type serialization scheme for heterogeneous WSN applications. It automates the tedious process of data encoding and data type implementation for various target platforms and programming languages. We have implemented microFibre as a module for FABRIC [4], our data type-centric middleware synthesis framework.

This combined approach offers two benefits: First, an application’s data types are defined globally using the widely accepted XML Schema standard. This specification is then used to synthesize code for multiple target hardware platforms. Second, our serialization scheme exploits the formal and detailed definition of the data types. The resulting code is lean and custom-tailored to match the resource constraints of WSNs. Compared to related schemes, microFibre yields optimized data type encoding at bit-level. This level of detail achieves considerable energy savings, since receiving and transmitting data are the most energy-wasting operations in a WSN [5], [6].

The remainder of this paper is structured as follows. We first discuss related work and motivate the need for our novel scheme in Section II. Section III briefly introduces FABRIC, Section IV explains microFibre and Section V underlines its applicability by presenting experimental measurements. The paper is concluded by a summary.
II. RELATED WORK

A unique property of WSNs is the fundamental requirement of wireless communication to achieve global objectives like creating a temperature map or tracking a person. Hence, a common task is to serialize some local state information of a node to network messages. To implement this crucial conversion from in-memory data structures to network messages, two fundamentally different approaches exist. In the following, we will illustrate these approaches and introduce related projects. Furthermore, we discuss their drawbacks that motivate the need for our novel microFibre scheme.

In the first approach the developer handcrafts data structures, network messages and the mapping between them. In the context of wireless sensor networks, the application’s data types are habitually implemented as nested data structures directly in the target programming language [7], [8]. Network messages are an exact, bitwise copy of the in-memory representation on the sensor node. Although widely used due to its simplicity, this approach has various severe drawbacks. It relies on the assumption that different compilers for different devices represent data structures in exactly the same manner so that data serialized from memory on one device can be easily de-serialized on another.

The network types [8] technique addresses this problem by proposing a programming language extension to nesC [9] that augments data types. It specifically enables the cooperation of those sensor nodes that are programmed using nesC by mitigating the effects of different memory alignments and endianness. Yet, the designed data structures are still confined to the nesC programming language. Hence, this approach is neither suited for heterogeneous WSNs nor for the integration with traditional networks.

As applications and protocols evolve over time, enhanced versions frequently require changing the application’s data types. Manually updating the various implementations for the different programming languages and node architectures is a tedious and error-prone process. Finally, using the same representation in memory and on the wireless interface yields only suboptimal encodings in terms of bit-length and hence energy consumption.

The second approach is to describe the application’s data types in a platform-independent manner and to transform this specification into platform-dependent code for multiple target platforms and languages. This generated code comprises the language and architecture specific data types and routines that (de-)serialize them from and to a common encoding. A number of these data type and data binding specifications such as XDR [10], ASN.1 [11] and XML Schema [12], [13] exist and are widely deployed.

XDR specifies an architecture independent data encoding to ease the data exchange between heterogeneous computers that is not optimized for size. It mainly targets desktop and server class devices and is therefore not suited for resource constrained devices. ASN.1 defines several encoding schemes, amongst them the Packed Encoding Rules (PER) [14] which yield a very effective and bit-length optimized encoding. Nevertheless, as we show in Section V, ASN.1 does not inherently target resource-constrained devices. XML Schema as the latest and most generally accepted standard commonly uses verbose, human-readable XML as the encoding format that is unsuitable for serialization in WSNs. Even so, approaches enabling XML processing on sensor nodes exist but they neither address heterogeneity nor compact encodings. “<<ASTAX” [15] for instance presents an event-driven programming scheme for WSNs that resembles the Simple API for XML (SAX).

We are convinced that starting from a high-level data type specification to generate platform-dependent data type and (de-)serialization code is the most suitable approach. However, to the best of our knowledge, existing schemes do not support heterogeneity, extremely resource-constrained devices and automatic optimized encoding at bit-level for network messages.

III. FABRIC ARCHITECTURE

To address exactly this problem, this section presents the architecture of our proposed approach that is comprised of two distinct building blocks. Both are contained and valuable in themselves, yet they unleash their full potential when they join forces.

The first part (FABRIC) that is presented in this section focuses on relieving WSN application developers from dealing with low-level networking aspects and heterogeneity. Instead of handcrafting data types and serialization code repeatedly for each application, hardware platform and programming language, our approach is based on a platform-independent data type specification augmented with annotations. These allow the developer to parameterize the following code synthesis process in a straightforward and convenient manner.

The second part (microFibre) specifies our data type serialization scheme that we have integrated as a code generating module in FABRIC. It acts on behalf of the application developer and fabricates the concrete
Furthermore, as XML Schema documents are expressed in a platform independent manner, this has led to an adoption by researchers, companies and software developers. (de-)serialization code. *microFibre* is presented in detail in Section IV.

To allow the framework to optimize the generated code, the application developer’s input needs to be as precise as possible. For this reason, a formal description of the data types is a must. We have chosen XML Schema with its powerful expressions, pre-defined data types, and the compelling extension mechanisms. XML Schema allows describing data types hierarchically including choices, optional elements and additionally supports annotations for the treatment of data as we will see later. A simple example will illustrate the basic idea of the input requirements for a lean sensor network implementation, the steps performed by our system and the resulting output.

Imagine an application developer designing a simple sensor network that helps controlling the heating of a building. Here, sensor nodes that are scattered across several rooms measure the temperature (in either degrees Celsius or Fahrenheit) and detect motion. The developer will carefully specify the two data types along with their required accuracy which results in two integer temperature types and the motion detection as a simple Boolean value (cp. Figure 1). According to Shannon’s information theory [16] the entropy defines the number of bits necessary for encoding data. That means if the required range for the temperature values is known, we can reduce the uncertainty and thereby save bits in the corresponding message. So, the application developer should limit the range of the room temperature (e.g., °C from 0 to 50 and °F from 32 to 122) and thereby implicitly supply additional valuable input to the automatic generation process.

![Fig. 1. Visual representation of an application-specific data type comprised of a choice between two range-restricted integers and one Boolean value](image)

Advantages of using XML Schema are the worldwide adoption by researchers, companies and software developers as the primary means to describe application data in a platform independent manner. This has led to a countless number of software tools that support XML and XML Schema and thus ease their use significantly. Furthermore, as XML Schema documents are expressed in XML, they benefit from the inherent validation and error-checking features present in XML.

Figure 2 shows an overview of FABRIC’s architecture. It has been designed to support multiple hardware platforms, heterogeneous environments and the interconnection of them by synthesizing application-layer gateway functionality. We distinguish two roles: an application developer and a framework developer. An application developer’s primary interest is an easy to use, flexible system while the framework developer customizes the generic FABRIC system and implements modules that provide the functionality available to the application developer. A code generation process transforms the type annotations into customized middleware source code such that the resulting code complies with the given target specification.

![Fig. 2. Overview of FABRIC’s architecture](image)

FABRIC intrinsically links the annotations with the data type definitions by using a feature defined in the XML Schema standard that allows amending nearly all schema elements with two different types: human- and machine-readable information. Annotating a data type in the context of FABRIC therefore means attaching a simple XML document to each data element definition in the XML Schema. Using XML again to specify the annotations is beneficial for a number of reasons. First, it does not introduce another language to be learned by the developer. Second, the grammar of the embedded XML document can also be specified as an XML Schema.

Figure 3 depicts the XML Schema of the data type shown in Figure 1 and illustrates the attached annotations. The XML Schema elements are prefixed with *xs* whereas the elements containing the type annotation start with the prefix *fabric*. The application’s data element “sensordata” is defined as a local complex type that is comprised of a choice (“temperature”) between two local simple types (the restricted integer value “celsius” and...
Fig. 3. XML Schema data type for the exemplary heating control application and its annotation

“fahrenheit”) and a Boolean value “motiondetection”.

FABRIC groups conceptually similar annotation aspects into domains and the set of annotatable aspects is freely customizable, yet for a concrete FABRIC instance it is predefined by the framework developer. In this example “compact” from the domain “serialize” and “reliable” and “unreliable” from the domain “tx” are annotated. In addition to the annotated data types, the system needs a target specification in order to produce the adequate code for a specific device (platform, programming language, etc). From these annotations and the target specification, FABRIC chooses the best modules that will participate in the code synthesis process. Note that the annotated data types are only needed for the code generation prior to the application’s compile-time and do not have to be stored on the devices at run-time. FABRIC then synthesizes custom-tailored, application-specific middleware code that features exactly the required functionalities for the defined data types. For further details on FABRIC’s internal mode of operation and the tasks of the framework developer, please refer to our previous work [4].

Finally the application developer writes the code for his target application and benefits from the type-specific API of the middleware. Without the need to change a single line of application code, a data type itself or the treatment of a data type can be altered by simply changing its specification or annotation. This allows for easy application development where the developer is relieved from dealing with error-prone issues like serialization.

IV. THE microFibre SERIALIZATION SCHEME

As stated in the introduction, it is an intrinsic property of WSNs that data is processed on a variety of heterogeneous devices along its way to a destination. The majority of the devices are extremely resource-constrained sensor nodes with only a few ten kBytes of memory. Since the key ingredient in WSNs is the exchange of wireless messages, it is a major issue that receiving and transmitting messages dominates the energy consumption of sensor nodes [5], [6].

Integrating microFibre as a FABRIC module addresses both, heterogeneity and energy consumption. It lets developers inherently benefit from an advanced encoding mechanism without the need to handcraft the data types and the serialization code. This is especially indispensable because this optimization is regularly omitted as discussed in Section II. A superior encoding of these network messages implies reduced energy consumption on the sending as well as on the receiving devices. Furthermore, is cuts the bandwidth requirements and hence the probability of collisions.

To achieve an optimized encoding for network messages, microFibre incorporates additional knowledge on the devices to reduce the uncertainty of received messages. Reducing uncertainty directly translates to fewer bits needed to encode the resulting network messages. This knowledge is derived from the data type definitions and manifests itself in the generated optimized and data type specific serialization code.

In contrast to traditional XML Schema based serialization schemes, we do not (de-)serialize XML document instances but instances of data types. This relieves our approach from the need to encode opening and closing tags or attributes explicitly since the sequence of data may be chosen by microFibre where the schema does not dictate a fixed order since all devices have knowledge of this order. In the following, we first describe the structure of XML Schema and how it enables us to optimize the
resulting encoding and illustrate how these messages are composed. Finally we present excerpts from the generated C code as this is the most ubiquitously used language in the context of WSNs while our current implementation can also generate C++, Java and nesC code.

At the core of XML Schema there are two different kinds of data type specifications: simple and complex types. Eighteen built-in simple types are standardized that can be categorized into integer and real numbers as well as strings. These built-in types can be further restricted to user-defined simple types (e.g. by limiting their range) through so-called facets. When dealing with numerical data types restrictions may be applied by specifying a minimum and maximum value. Both the lower and upper boundary can be either defined as inclusive (facets minInclusive and maxInclusive) or exclusive (minExclusive and maxExclusive) boundaries. Strings, on the other hand, can have a restricted length. Similar to the numerical values a minimum and a maximum or a fixed length (facets minLength and maxLength or length) can be set.

Complex types may be composed of several simple and complex types. XML Schema supports three different types of compositions: sequence, choice and all. A sequence is a data type where the order of the contained elements is fixed. For each element the number of minimum and maximum occurrences may be specified (facets minOccurs and maxOccurs). A choice is similar to a sequence, but contrary to the sequence, only one of these types may be present in any given instance. The all composition is somewhat special since the ordering of the contained data types is arbitrary in instance documents but each element may occur at most once. In addition to the definition of simple and complex types, an XML schema also defines top-level elements to be of a certain type (e.g., element sensordata on Figure 3). These then actually constitute the root elements of an XML document or in our case, message types.

It is the application developer’s task to define the data types as concise as possible allowing microFibre to generate an optimized encoding at bit-level. On the one hand, they must match the application’s requirements while on the other hand as restricted as possible to enable optimizations. In the following we highlight were optimization potentials exist and discuss how they are realized in microFibre.

Without any restrictions, all built-in signed and unsigned integers have a fixed size of 8 bits for byte, 16 bits for short, 32 bits for int and 64 bits for long. When restricting the range of these types, serialization can be compacted. It makes sense, for example, that integer numbers ranging from only 0 to 50, as in our example above, should require less bits than an unsigned byte ranging from 0 to 255. According to Shannon, the number of bits $n$ necessary for transmitting a restricted integer value can be calculated as

$$n = \lceil \log_2(max - min + 1) \rceil$$

When using exclusive boundaries instead, these can easily be transformed into inclusive boundaries by adding/subtracting one. When serializing a restricted value it is normalized by subtracting the lower boundary which is added again on de-serialization. In the above example 6 instead of 8 bits are required saving 25% of the transmitted data.

Real numbers require 32 bits for single (float) and 64 bits for double-precision (double) as defined by IEEE 754. Reducing the required bits is difficult because of the nature of floating point numbers where small numbers are encoded with a higher precision than larger numbers. Hence, restricting the range does not decrease the required bits for serialization. Nevertheless, optimizing the serialization of real numbers is still possible by using XML Schema’s totalDigits and fractionDigits facets that restrict the number of positions before and after the decimal point. These can then be encoded as two restricted integer values that use fewer bits than the standard representation.

Finally there are string values which are in general encoded as one byte per character. Yet, as WSNs usually run only a single application, additional knowledge about the application’s data is available at design time. For instance, the required alphabet and text samples are available prior to deployment. Hence, if this data is included into the annotations, it is used to construct a Huffman tree [17] that is available on all devices and hence does not have to be included in the network message. The tree contains the necessary characters and reflects their frequencies by assigning shorter codes to more frequent ones. Besides the character data, the string’s length must be encoded as well. For applications where the string lengths are bounded to a minimum and maximum value and it can be encoded using the same rules as stated above for integers. A special case is a string with a fixed length where the length may be omitted completely.

Transmitting elements of complex types is accomplished similar to transmitting the length of a string. If the number of occurrences is not fixed, i.e. the values for
**minOccurs** and **maxOccurs** differ, then the actual number of elements needs to be transmitted before the elements themselves. If these elements in turn are complex types, serialization is done recursively.

Besides a compact serialization of data types it is equally important to create optimized data structures for a particular platform since sensor nodes have scarce memory resources. Figure 4 shows an excerpt of the C header file generated by our framework from the schema document shown in Figure 3. In lines 1–6 the data type declaration can be seen, followed by the declarations of the functions which perform the (de-)serialization of the data type. When translating data types defined in the schema, most of built-in schema data types can be mapped directly to their C counterparts. However, in some cases it is important to take care of the differing sizes of these types (e.g., long is 64 bits in XML Schema but usually only 32 bits in C). Restrictions of integers are easily mapped to bit fields where the number of necessary bits is again calculated as given by Equation 1. Other types such as **boolean** and special types like date and time do not have exact counterparts in C but they are generated as bit-fields too.

```c
enum fabric_LocalComplexType10_element_selector {
  CELSIUS, FAHRENHEIT
};
typedef struct fabric_LocalComplexType8 {
  union {
    char celsius:6; char fahrenheit:7; } temperature;
  int temperature_selector;1;
} fabric_sensordata;
unsigned int fabric_serialize_sensordata {
  (fabric_sensordata* sensordata, char* buffer);
void fabric_deserialize_sensordata {
  (char* buffer, fabric_sensordata* sensordata);
```

![Fig. 4. Excerpt from the generated source code](image)

Complex types in XML Schema are converted to the well-known language constructs available for structures in C. While both **sequence** and **all** are represented by a **struct**, a **choice** is represented by a **union** in C. It is necessary for **microFibre** to encode which of the elements in a **union** has actually been set at runtime (“celsius” or “fahrenheit” in the above example). For that purpose, an enumeration containing the choices and a (restricted integer) selector variable is generated automatically. As mentioned before, child elements of complex types can have a specified minimum and maximum occurrence. If one of these occurrences does not equal 1, the child elements are represented as an array of the element’s type. If the minimum and maximum occurrences differ, an additional variable is added containing the actual number of occurrences at run-time.

![Fig. 5. Encoded example message for two distinct cases](image)

The encoding step that serializes instances of the data structures from Figure 4 into messages is shown exemplarily in Figure 5. The Boolean value at the beginning requires only 1 bit, the following selector variable also requires only 1 bit since it encodes two possible choices. Finally, depending on the selector variable, one of the two integer values is encoded. Note that the value for “celsius” is normalized to the minimum value of 0 while the normalized value of the “fahrenheit” variable is 47 because the minimum value given by the schema is 32.

### V. Evaluation

To demonstrate the real-world applicability of our approach, we evaluate **microFibre** by using the **QueryMessage** data type from the prominent and widely deployed TinyDB [18]. To allow comparison with other approaches, we have converted QueryMessages’s C data structures to XML Schema. In the following we present experimental measurements that compare our approach with the TinyDB implementation and ASN.1’s Packed Encoding Rules (PER)\(^1\).

To give a first impression of the amount of compiled code and data, Figure 6(a) compares the size of the resulting binaries for simple (de-)serialization programs and the in-memory sizes of the data structures using a standard x86 gcc compiler. It is clearly visible that Fabric’s generated type definition is roughly the same size as the handcrafted TinyDB data structure. Obviously, the ASN.1 compiler output is not optimized for WSNs as the data structures are bulky and the resulting serialization code is approximately 11 times larger than the one generated by **microFibre**.

To demonstrate that the overhead introduced by **microFibre**’s serialization code increases linearly with the

\(^1\)Implementation note: ASN.1 compiler: http://sf.net/projects/asn1c, XML Schema to ASN.1 converter: http://asni.elibel.tm.fr/tools/xsd2asn1
Table 1. Comparison of code and data size (using a x86 gcc)

<table>
<thead>
<tr>
<th>Code</th>
<th>Type</th>
<th>Size [kByte]</th>
<th>Size [byte]</th>
</tr>
</thead>
<tbody>
<tr>
<td>FABRIC</td>
<td></td>
<td>7.4</td>
<td>55</td>
</tr>
<tr>
<td>TinyDB</td>
<td>n.a.</td>
<td>84.6</td>
<td>164</td>
</tr>
<tr>
<td>ASN.1 (PER)</td>
<td></td>
<td>42.0</td>
<td>84.0</td>
</tr>
</tbody>
</table>

(a) Simple (de-)serialization (b) Code size in relation to the number of elements in the schema

Fig. 6. Comparison of code and data size (using a x86 gcc)

number of data types and elements, we have created a simple data type in XML Schema that is comprised of only a single integer element. Step by step, one further integer variable has been added to the data structure. Figure 6(b) shows the size of the generated object files versus the number of data elements. The basic functionality required 1.4 kByte and each additional element only adds 68 byte to the compiled (de-)serialization code.

Besides the code size, the resulting network message lengths are of foremost interest. The QueryMessage data type is composed of nine different complex types that can be assembled to 78 unique combinations. To compare microFibre with the encoding quality of TinyDB and ASN.1 (PER), we have created an instance of each of them. The integer values were randomly assigned as they do not affect the encoding length. For the string values, we have chosen a typical English text to demonstrate the potential of optimized string encoding when the developer provides additional knowledge. Here, microFibre has been parameterized to use the character frequencies of the English language. Figure 7 shows the number of bytes required by TinyDB, ASN.1 (PER) and microFibre to encode these test cases.

The comparison is slightly biased in favor of TinyDB since some variables are generated redundantly by ASN.1 (PER) and microFibre. This is due to the fact that the QueryMessage data type already contains variables to encode the existing choices. ASN.1 (PER) and microFibre must generate their own variables since – unlike TinyDB – they have no application specific knowledge of the existing ones. This explains, why ASN.1 (PER) requires more bytes to encode the message than TinyDB’s structure. Please note that we do not compare the number of bits but the number of bytes as all encoding schemes must pad to byte boundaries.

![Figure 7. Size of the encoded QueryMessage network message](image)

TinyDB’s implementation copies the data structure directly to the network and hence always requires 42 bytes independent of the test case. The comparison against ASN.1 (PER) can be classified into three categories: (1) Test cases 1–12: microFibre yields exactly the same encoding size. This is due to the fact that only (restricted) integers are encoded and both schemes achieve an optimal encoding. (2) Test cases 13–66: an improvement of 1–6 bytes can be observed. Here our encoding of choices and optional elements needs fewer bits and the further byte savings stem from the presence of one string. Here our advanced encoding of strings by using a Huffman tree proves beneficial. (3) Test cases 67–78: 15–17 bytes less bytes are needed to encode the same message. This is essentially the same case as the second, but here three string values are encoded.

To summarize, Figure 8 shows that the encoding produced by microFibre is in average 28.85% (TinyDB) and 12.20% ASN.1 (PER) smaller while only introducing a minimal overhead in terms of code size.

![Figure 8. Comparison of the size of the encoded messages](image)

VI. CONCLUSIONS AND FUTURE WORK

In this paper we present a combined approach of FABRIC and microFibre that offers various benefits to ease WSN application development by relieving the developer from dealing with intricate networking. First, FABRIC radically simplifies and accelerates the development process as the developer maintains only a single, abstract data type definition instead of keeping multiple
instances in sync for several programming languages and target platforms. This allows incorporating application knowledge into the data type definition resulting in lean and custom-tailored code that only contains the required functionality. Second, it addresses heterogeneity since the serialization code is generated for a variety of platforms and it does not rely on platform-specific assumptions such as memory alignment or endianness. Hence, the binary network messages can be exchanged seamlessly between sensor nodes, gateways and backend servers. Third, we demonstrate that FABRIC is beneficial for WSNs and that it performs excellent in comparison to traditional programming schemes. microFibre achieves considerable energy and bandwidth savings and helps to prolong the network’s overall lifetime.

Our current implementation supports the complete XML Schema standard and generates code for C, C++, nesC and Java that runs on a multiplicity of sensor nodes, gateways and backend computers. We have implemented FABRIC and microFibre in Java and integrated it into the Eclipse framework as a plug-in to support the rapid development of sensor network applications. Future work will focus on optimization of the footprint as well as the further integration with simulation and visualization tools. Another interesting aspect is to integrate FABRIC with external XML-based applications. Currently, FABRIC uses XML Schema as a description language for data types during development time, but no XML is used at runtime due to the resource constraints in WSNs. Anyhow, for future applications it might be valuable to foster the integration of functionalities in the WSN with external infrastructure elements such as SOAP web services. To avoid data conversions on gateway nodes it might be interesting to use compact XML serializations for messaging also in the WSN at runtime. In previous work we have developed the very lightweight state-machine based XML-processing component XENIA [19]. We will integrate FABRIC, microFibre and XENIA and evaluate how XML messaging inside a WSN is feasible.

REFERENCES