Andreas Reinhardt, Parag Mogre, Ralf Steinmetz: Lightweight Remote Procedure Calls for Wireless Sensor and Actuator Networks. In: Workshop Proceedings of the 9th Annual IEEE International Conference on Pervasive Computing and Communications (PERCOM), p. 116--121, EDAS Conference Services, March 2011. ISBN 978-1-61284-936-2.

Lightweight Remote Procedure Calls for Wireless Sensor and Actuator Networks

Andreas Reinhardt, Parag S. Mogre, Ralf Steinmetz
Multimedia Communications Lab, Technische Universität Darmstadt
Rundeturmstr. 10, 64283 Darmstadt, Germany
Email: {andreas.reinhardt, parag.mogre, ralf.steinmetz}@kom.tu-darmstadt.de

***Abstract*—This paper presents S-RPC, a lightweight framework for remote procedure calls on wireless sensor and actuator nodes. It enables seamless interoperability between nodes by offering a common interface to remote method invocations. In the past, communication between these embedded devices has been realized over proprietary protocols specifically tailored for the given use case. This, however, strongly limits the possibilities to integrate new functions into the network and extend it by new sensing and actuation devices. In contrast to proprietary solutions, S-RPC is a framework which allows unified access to functions on remote devices. S-RPC uses an efficient data representation scheme to achieve small message sizes, and permits dynamic integration of new devices with different functions into the network during runtime. We show its real-world applicability through a reference implementation, which occupies less than 4% of a TelosB's resources, and increases the energy consumption per invocation by just 0.11%.**

I. Introduction

Wireless Sensor and Actuator Networks (*WSAN*s) augment the capabilities of pure sensor networks by the possibility to interact with the physical world [1]. For example WSANs for smart buildings may allow actuation (e.g. turn a fan on) based on sensor (e.g. temperature) measurements. Traditionally, WSAN communication protocols define packet structures statically at compile time due to efficiency reasons, i.e. each of the fields in a radio message is hard-coded to contain a particular type of data [2]. While this is efficient with regards to the computational complexity (and thus the demand for energy) of accessing the data in the packet, applications must be adapted and re-deployed on the nodes whenever new sensor or actuator types are introduced into the network. Current WSANs are thus confined to a static set of sensor and actuator types only; adding new types to the network at runtime is impossible.

To gain the flexibility of accessing sensing and actuation functionality offered by remote WSAN nodes and at the same time cater for the dynamic extensibility of the network, a generic approach without static field allocations is required. The paradigm of Remote Procedure Calls (*RPC*s) [3] is well established to access functions on remote hosts on the Internet, and we believe that it is optimally suited for the given application scenario. Current WSAN applications are however generally implemented in a monolithic way without support for RPCs. In contrast, RPCs are prominent in the area of Web Services [4], where they enable loose coupling of components. The applicability of existing implementations on embedded systems is strongly limited, as WSAN nodes can often not even transmit packet payloads of more than 127 bytes at a time (assuming operation over IEEE 802.15.4 [6], commonly used in WSANs). To allow access to sensing and actuation capabilities over an RPC interface, a lightweight message format as well as an efficient serialization mechanism for messages is necessary.

In this paper, we present S-RPC (Sensor-RPC), a lightweight RPC framework for WSAN nodes, which addresses the shortcomings of existing approaches. Each method provided on the node (e.g. access to sensor readings, processing, or actuation capabilities) can be registered to the S-RPC framework during runtime, enabling its access through remote invocations. Our contributions are:
(1) We define a generic and efficient RPC message format, catering for the extensibility of the network by additional devices with new sensors or different actuator controls.
(2) We present our modular parameter serialization concept, enabling to confine S-RPC instances to the required data types to cover a heterogeneous set of devices.
(3) We validate the applicability of our S-RPC implementation on real hardware, analyzing its resource consumption, message sizes, invocation delays, and energy demand.

We address these contributions as follows: In Sec. II, we motivate the need for lightweight remote procedure calls. The detailed concept and implementation of S-RPC are presented in Sec. III, and its performance is evaluated in Sec. IV. We summarize related work in Sec. V and conclude this paper in Sec. VI.

II. Bridging Device Heterogeneity using RPCs

Radio communication is an inherent characteristic of WSANs, and many functions frequently used in current applications rely on the exchange of data packets. In current WSAN operating systems like TinyOS and Contiki, tasks like neighbor discovery, periodic data reporting, or the activation of actuators are generally realized by exchanging data in statically defined message structures. To distinguish incoming packets and cater for correct interpretation, type fields or port numbers are used. Messages addressed to invalid ports or with unknown packet types are silently discarded, effectively disabling the incorporation of new functionalities into the network.

The documents distributed by this server have been provided by the contributing authors as a means to ensure timely dissemination of scholarly and technical work on a non-commercial basis. Copyright and all rights therein are maintained by the authors or by other copyright holders, not withstanding that they have offered their works here electronically. It is understood that all persons copying this information will adhere to the terms and constraints invoked by each author's copyright. These works may not be reposted without the explicit permission of the copyright holder.

A. Remote Procedure Calls

In contrast to static allocation of message types or ports, the use of RPCs allows to extend a WSAN by new functions at any given time, as the format of exchanged messages is unified. After a node has published the description of a new procedure interface, any other node can invoke the method through means of the generic message format.

The general process of invoking a method on a remote device is shown in Fig. 1. The optional first step of publishing the interface description can be omitted in systems where interfaces are known a priori, or when additional means for obtaining information about these interfaces are employed, e.g. service discovery [5] or introspection [7]. In the subsequent step, the remote procedure call is made, requesting the provider to invoke the referenced method. If required, the parameters to be passed to the method are also provided along with the RPC request. On the provider side, the procedure is invoked locally with the given parameter set, and the result returned to the requester in a response.

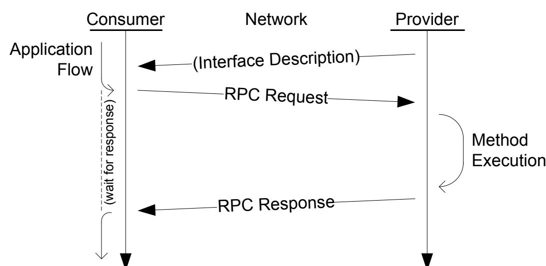

Figure 1. Simplified flow of a remote procedure call

Traditionally, RPC calls exhibit synchronous behavior, i.e., they block the invoking application until a result has been obtained. To cope with provider and connection failures during invocations, specific error semantics are used (e.g. at-least-once or at-most-once). While being irrelevant for idempotent methods (operations that do not incur a state change, e.g. polling a sensor in WSNs), these semantics become relevant when actuators are controlled. We discuss the semantics supported in WSANs in Sec. III-C.

To transport RPC messages over the network, all parameters need to be serialized into a sequence of bytes. While the serialization process of primitive data types often equals the concatenation of the individual data bytes, the representation of complex data structures and objects as a series of bytes is often confined to platforms, programming languages, or comparably verbose, thus strongly hampering the applicability of RPCs on resource-constrained devices.

B. Lightweight RPC for WSANs

In contrast to smartphones or laptop computers, WSAN nodes incorporate embedded microcontrollers and are generally operated on sources with a finite energy budget like batteries [8]. Besides the use of energy-efficient hardware, simultaneously energy-efficient algorithms are applied to maximize node runtimes. To adapt to these limitations of WSANs, a solution to perform RPCs must be sufficiently lightweight to be supported even on the smallest participating platform. We have compared a representative set of WSAN platforms in Table I, illustrating the discrepancy between device capabilities. Especially the limited resources of the TelosB platform are significantly exceeded when using full-flavored RPC implementations, such as CORBA [9].

Table I
SELECTION OF PLATFORMS ENVISIONED IN WSANS

Device name	CPU clock	RAM	Operating system
TelosB	8 MHz	10 kB	TinyOS / Contiki
SunSPOT	180 MHz	512 kB	Java / SquawkVM
Gumstix Verdex	600 MHz	256 MB	Linux

While full RPC implementations are designed to support serialization of a broad variety of data types, the actual number of different data types in a WSAN is usually limited. Especially, the absence of complex objects leads to a small number of data types, e.g. signed and unsigned integers of different sizes, strings, and arrays of primitive data types. Providing the RPC implementation with generic data serializers leads to increased resource consumption, although the implemented functionality is never being used.

As mentioned in the preceding section, RPCs are performed in a synchronous manner and block the application while waiting for the response. However, especially when WSAN nodes are connected over wireless links, packet losses may occur. Simultaneously, high roundtrip times may be introduced due to the use of energy-aware medium access control protocols. To differentiate lost packets from high latency in the transmissions, dedicated mechanisms are required. We target to keep our protocol compatible with existing solutions to reliably transport data over unreliable channels (e.g. [10]), although details are beyond the scope of this paper. In summary, the following three factors influence our RPC design:

1) Confining the resource demand for encoding/decoding and buffering of data to a minimum.
2) Finding a trade-off between the data types supported and the corresponding resource consumption.
3) Defining a compact data representation scheme to limit the additional overhead on packets.

III. DESIGN OF THE S-RPC FRAMEWORK

The S-RPC framework has been designed following the structural diagram depicted in Fig. 2, in which the sequence numbers indicate the typical flow of data. Remote methods can be invoked by the consumer node through a dedicated interface offered by the S-RPC framework. Invocations are encoded by the S-RPC data representation layer (*DRL*). All parameters are serialized by dedicated modules specifically designed to efficiently encode the given data type. The encoded message is transmitted to the method provider,

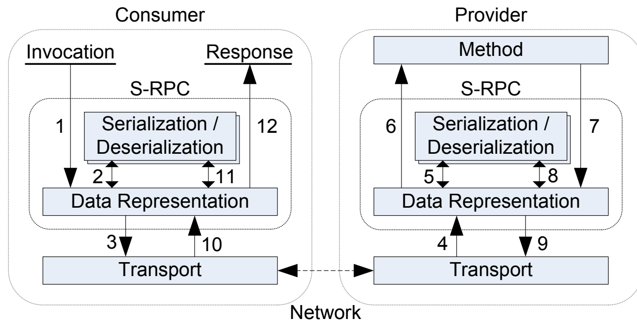

Figure 2. Layers of the S-RPC framework

where the DRL caters for invoking the necessary deserializer modules, invocation of the local method, and generation of the S-RPC response. On reception of a reply, the return value is deserialized and returned to the consumer application.

We have chosen to uncouple data serialization and representation to allow for a flexible configuration of the data types supported by each participating node. Each device can be fitted with the data types its own procedures consume or generate instead of enforcing support for a network-wide selection of relevant data types. As the use of complex serialization engines would disqualify S-RPC for resource-constrained systems, the use of modular serializers even allows more powerful platforms to support a larger set of serializers while small systems can be confined to just the minimum set.

A. The Data Representation Layer

The DRL is the core component of S-RPC, and fulfills the task of creating S-RPC messages from invocation calls as well as returning the responses to the requester. For methods on remote hosts that require parameters, the DRL checks the set of available serializers for their support of the given data type. If present, the corresponding serializer is invoked, returning the serialized values as well as a unique data type identifier. Otherwise, an error code is returned immediately to the invoking application. Once all parameters have been serialized, the resulting byte sequences are concatenated by the DRL and prefixed by the S-RPC header.

The header follows the structure shown in the upper four bytes of Fig. 3 for a message with three parameters, and its fields are used as follows: The **message type** (MsgType) field is two bit wide and specifies if the following message represents a request (binary value `00`), a response (`01`), or an error (`10`). The binary value of `11` is reserved for future extensions of S-RPC. In the **parameter count** field, the number of data type identifiers following the second header byte is defined. Being six bits in size, it allows to indicate between zero and 63 parameters. The **invocation sequence number** is used to distinguish multiple calls between identical consumer and provider nodes due to the asynchronous character of S-RPC, discussed in more detail in Sec. III-C. Finally, the **data type identifiers** indicate the types of the

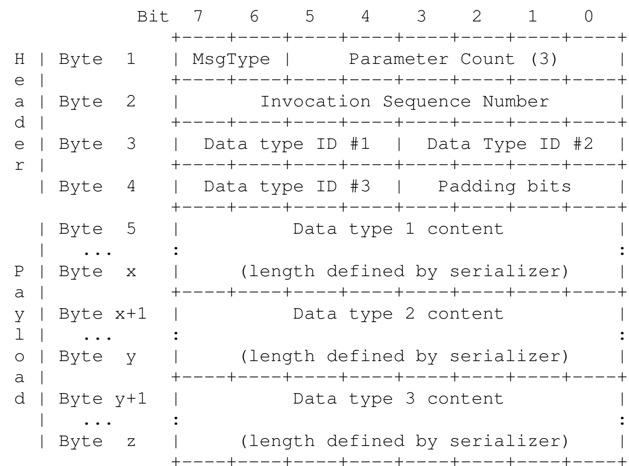

Figure 3. S-RPC representation of a message with three parameters

data fields that follow the header. If the number is odd, four padding bits are inserted. For RPC request messages, the first parameter contains a byte array indicating the name of the function to call. Replies are generally identified through the invocation sequence number, but can optionally also carry the name of the invoked function.

The design decision to specify the data types in the header has been made to allow verification of the availability of the required deserializer module before trying to deserialize the data. Knowledge about the packet structure also allows invoked methods to verify if the provided parameters match the required parameter set before executing the call. Additionally, using the structure of contained data types as the similarity metric for stream-oriented data compression, presented in our previous paper [11], becomes a viable way to allow packet size reductions and thus preserve energy.

B. Serialization Modules

As a core design decision, we have decided to use dedicated serialization modules for each data type anticipated to be present in the network. Instead of utilizing a single serializer/deserializer component, S-RPC employs a modular serialization concept, which allows the developer to fit nodes with serializer components on demand. Each serializer bears a unique identifier for the data type it represents, allowing the decoder to interpret the corresponding content correctly. As shown in Fig. 3, the identifier fields in our reference implementation have been defined to be four bits wide, so that up to 16 data types can be represented in total.

Given the typical application domain of WSANs, where packet payloads are mostly comprised of boolean fields and integer values of different sizes, the comparably small number of supported data types proves to be sufficient. Each serializer is responsible for performing efficient conversion of its input data, i.e. means towards lossless data reduction (such as run-length encoding) amongst array elements, or the application of efficient string encodings. If further data

Table II
DATA TYPE ASSIGNMENTS

ID	Data type	Size	Remarks
0	null	0 bytes	indicates void results
1	boolean value	1 byte	
2	8bit unsigned integer	1 byte	
5	32bit signed integer	4 bytes	
8	byte array (string)	variable	output prefixed by length
9	array of other types	variable	types defined in subheader
14	Java serialized object	variable	uses Java serializer
15	reserved	n/a	allows future extensions

types, such lists of key-value pairs, need to be transferred between applications, corresponding serializer/deserializer components can be included. The efficient data representation of S-RPC can also be used to encapsulate calls to RESTful web services. In our reference implementation, we have defined data type assignments as shown in Table II. Some type identifiers have been intentionally left unassigned to allow adaptation to the intended application.

As shown in Fig. 3, the length of serialized values is determined by the data type itself and is either fixed or stored in the serialized implementation. Each deserializer thus returns both the value stored in the S-RPC data type field as well as the pointer to the beginning of the subsequent entry in the sequence. Obviously, when deserializer modules are unavailable, the correct offset cannot be returned, possibly leading to incorrect deserialization. However, if data types are unknown to a node, it can generally be assumed that no local method makes use of them, and thus that an error message can be returned immediately.

C. Message Transport and Error Handling

As RPCs are traditionally executed in a synchronous manner, a low delay link is mandatory to avoid unnecessary waiting periods of the consumer node. However, to achieve long node lifetimes, energy-efficient data transport protocols are mandatory, which often introduce additional delays. Hence, synchronous method invocation is infeasible when addressing such devices. In consequence, S-RPC has been designed to work in an asynchronous manner.

Sequence numbers are added to all S-RPC messages to link corresponding response and request messages. To recover from remote node failures, packet losses, or excessive delays, timeout mechanisms are integrated into S-RPC. If the provider nodes get disconnected from the network, an error message is generated at intermediate nodes to provide the caller with details about the problem and thus terminate the call. As a result, only best-effort execution and at-most-once semantics are possible.

D. Method Implementation from a Developer's Perspective

The S-RPC framework caters for the communication and provides access to methods for serialization/deserialization of messages. To integrate a new method into the framework, the corresponding function must be programmed, and its metadata must be registered to the framework. The basic implementation of a method to add two integer numbers in our TinyOS version is shown in Code Fragment 1, wherein the helper functions to convert integer values to and from their serialized form are provided by the S-RPC framework.

Code Fragment 1 Implementation of the `add` function

```
char* add(char* i1, char* i2) {
  return int2srpc(srpc2int(i1) + srpc2int(i2));
}
```

Besides its implementation, the function must be registered to the S-RPC framework by defining its metadata and adding it to the list of supported functions. This process is indicated in Code Fragment 2 for the function implemented above. The `methodStruct` prototype is provided by the framework, and must be instantiated for every function available for remote access. Registration of the function is completed after the `add_srpc_method` call, while unregistration (e.g. due to a lack of energy) is possible at any time using the `remove_srpc_method` function.

Code Fragment 2 Registering the function to the framework

```
const struct methodStruct add_func = {
  .name = "add";
  .paramCount = 2;
  .parama = IntegerDataType;
  .paramb = IntegerDataType;
  .returnType = IntegerDataType;
  .function = &add;
};
add_srpc_method(add_func);
```

Both presented implementations are based on our TinyOS implementation, however the corresponding code in our Java implementation running on SunSPOT [12] nodes is analog and thus omitted.

IV. PERFORMANCE EVALUATION

To validate the applicability of S-RPC on WSAN nodes, we have implemented S-RPC using the TinyOS operating system on the TelosB platform. To allow for comparison, we have additionally developed a reference implementation with the same functionality, where a dedicated type field determines which function to execute. The following eight methods were implemented in the both the reference implementation as well as our S-RPC version, and used in all further analyses:

`char[] cat(char[] a, char[] b)`	concatenates strings
`void ledsOn(void)`	turns node LEDs on
`char[] ping(void)`	returns string `pong`
`int[] echo(int[] a)`	echoes input array
`int add(int a, int b)`	returns $a + b$
`int diff(int a, int b)`	returns $a - b$
`bool xor(bool a, bool b)`	returns $a \oplus b$
`int sum(int[] a)`	returns $\sum a$

Table III
RESULTING SIZES FOR THE TINYOS IMPLEMENTATION OF S-RPC

	No methods		8 methods	
	ROM	RAM	ROM	RAM
Ref.	16720 bytes	1210 bytes	17372 bytes	1374 bytes
S-RPC	17376 bytes	1348 bytes	19256 bytes	1578 bytes
Diff.	656 bytes	138 bytes	1884 bytes	204 bytes

A. Size of the S-RPC framework

As the TelosB platform [13] is limited to 48kB of application ROM and 10kB of RAM, the implementation of S-RPC must expose a sufficiently small footprint in both regards to remain applicable. We have thus compiled both S-RPC and the reference, and show the resulting ROM and RAM footprints in Table III. When incorporating the S-RPC stack without any functions offered, a mere size increase of 656 bytes application ROM as well as 138 bytes of additional memory are occupied. Even when all eight functions are included, only 1.8 kilobytes of ROM and only 204 additional bytes of RAM are required, representing less than 4% of the available application memory and only 2% of RAM.

B. RPC Message Sizes

The message sizes to invoke the functions presented in the preceding section are compared in Table IV. All carry the full name of the called method, as well as all parameters including their types. Similarly, all responses contain the corresponding invocation sequence number as well as a type field indicating the type of the returned value. Overall, only $\lceil 2 + \frac{\#parameters}{2} \rceil$ bytes of overhead are generated by S-RPC, an amount well tolerable in most WSANs.

Table IV
MESSAGE SIZE COMPARISON

Method call	Request size	Response size
`cat("foo", "bar");`	15 bytes	8 bytes
`ledsOn();`	9 bytes	2 bytes
`ping();`	12 bytes	7 bytes
`echo(1, 2, 3);`	21 bytes	15 bytes
`add(1024, 2148);`	15 bytes	6 bytes
`diff(728, 8210);`	16 bytes	6 bytes
`xor(true, false);`	9 bytes	3 bytes
`sum(3, 2, 1);`	20 bytes	6 bytes

C. Invocation Delay and Energy Consumption

To monitor the time required to process invocations, we have implemented a function to invoke each of the eight methods shown in Table IV locally at the provider node. The encountered delays of these local invocations show a measurable discrepancy on the TelosB: Less than 20μs are required to execute the actual processing, while the time required to deserialize the S-RPC data, invoke the contained method, and serialize the result ranges from 360–375μs.

In order to verify the real-world impact of this observation, we have subsequently set up a practical experiment, in which a dedicated TelosB node emitted S-RPC requests over the radio channel whenever a reply for the preceding

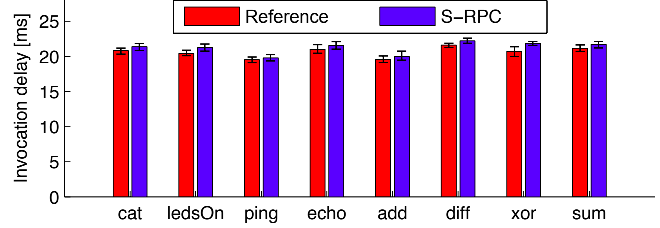

Figure 4. Experienced delays for remote method invocations

invocation was received. The experiment has been conducted for each of the functions, whereby the average delays measured for remote invocations as well as the minimum and maximum values are depicted in Fig. 4. In contrast to local method calls, it becomes clear that the processing duration becomes almost negligible when compared to radio communications. A supplementary energy simulation in COOJA has shown that the energy consumed to process a single call on the MCU increases by 9.4% from 2.56mJ to 2.81mJ. With regard to the node's total energy requirement however, the additional expenditure to process S-RPC requests reduces to an 0.11% increase of the total consumption only.

V. RELATED WORK

With the rise of computer networks and the Internet, the way has been paved for the invocation of methods and services on remote machines. Approaches towards lightweight implementations include RESTful web services [14] and the use of compact data representations in SOAPjr [15]. Both do however not focus on the efficient serialization of parameters. Addressing the efficient encoding of messages, the kXML [16] parser for resource-constrained devices, provides support for the WAP Binary XML (WBXML) protocol [17]. However, as all data fields are encoded using string representations, the encoding is still not optimal in terms of size. Specifically adapted to embedded sensing systems, the TinyRPC stack, allowing for static remote procedure calls on sensor network nodes, has been presented in [18]. Utilizing a compiler extension, all methods offered for remote invocation must be defined at compile time. Later, Whitehouse et al. have developed Marionette, a version of Embedded RPC [19], which adds RPC functionalities to nodes in sensor networks to increase the debugging capabilities of the network. All interfaces are extracted during compile time as well, thus later changes or additions to the set of interfaces are unsupported.

Special focus on the efficient serialization of parameters contained in RPC messages is given in [20]. The serialization preprocesses radio message definitions at compile-time and performs optimizations, e.g. limiting the number of bits required to transfer an integer if its values range between previously known bounds. Recently, Moritz et al. have presented means for encoding and compression of SOAP messages [21]. Similarly, in their implementation the interface definition step must precede compilation to adapt to

the message contents. The initial design target of RPCs, as presented by Birrell and Nelson in [3], was transparency, i.e. make remote calls indistinguishable from local calls. In the case of WSANs however, the overhead introduced by calling local functions through the RPC framework should be avoided. We have thus chosen to allow access to local functions directly and only use S-RPC when invoking methods over the network. Opposed to static sensor network deployments, we assume that nodes may join the network at a later time, or leave during runtime due to a depletion of their energy budget. None of the presented solutions has been designed to adapt to these dynamics. We have therefore addressed these shortcomings in this paper.

VI. Conclusion

In this paper, we have presented S-RPC as a lightweight and extensible mechanism to invoke remote methods in wireless sensor and actuator networks. Specifically designed for resource-constrained node platforms, S-RPC features a modular serialization concept, allowing to configure it to the relevant data types. To cater for correct interpretation of the exchanged messages, S-RPC uses a lightweight data representation layer, which only adds a few bytes of overhead to all packets. Our experimental validation has shown that S-RPC requires less than 0.5ms to process incoming requests, resulting in an additional energy demand of just 0.11%. S-RPC can be run over existing WSAN transport protocols like 6LoWPAN. It is thus well suited to unify the access to the heterogeneous set of embedded systems envisioned to comprise WSANs and the Internet of Things.

Acknowledgment

This research has been supported by the German Federal Ministry of Education and Research (BMBF).

References

[1] I. F. Akyildiz and I. H. Kasimoglu, "Wireless Sensor and Actor Networks: Research Challenges," *Ad Hoc Networks Journal*, vol. 2, no. 4, 2004.

[2] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister, "System Architecture Directions for Network Sensors," in *Proceedings of the 10th Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2000.

[3] A. D. Birrell and B. J. Nelson, "Implementing Remote Procedure Calls," *ACM Transactions on Computer Systems*, vol. 2, no. 1, 1984.

[4] G. Alonso, F. Casati, H. Kuno, and V. Machiraju, *Web Services – Concepts, Architectures and Applications*. Springer, 2003.

[5] R. Marin-Perianu, J. Scholten, P. Havinga, and P. Hartel, "Cluster-based Service Discovery for Heterogeneous Wireless Sensor Networks," *International Journal of Parallel, Emergent and Distributed Systems*, vol. 23, no. 4, 2008.

[6] IEEE Std, "802.15.4 Part 15.4: Wireless Medium Access Control and Physical Layer Specifications for Low-Rate Wireless Personal Area Networks," 2006.

[7] B. Henderson, "XML-RPC Introspection," Online: http://xmlrpc-c.sourceforge.net/introspection.html, 2007.

[8] D. Estrin, L. Girod, G. Pottie, and M. Srivastava, "Instrumenting the World with Wireless Sensor Networks," in *Proceedings of the International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 2001.

[9] S. Vinoski, "CORBA: Integrating Diverse Applications Within Distributed Heterogeneous Environments," *IEEE Communications Magazine*, vol. 35, no. 2, 1997.

[10] Y. Sankarasubramaniam, Ö. B. Akan, and I. F. Akyildiz, "ESRT: Event-to-Sink Reliable Transport in Wireless Sensor Networks," in *Proceedings of the 4th ACM International Symposium on Mobile Ad Hoc Networking and Computing (MobiHoc)*, 2003.

[11] A. Reinhardt, M. Hollick, and R. Steinmetz, "Stream-oriented Lossless Packet Compression in Wireless Sensor Networks," in *Proceedings of the 6th Annual IEEE Communications Society Conference on Sensor, Mesh and Ad Hoc Communications and Networks (SECON)*, 2009.

[12] Sun Microsystems Inc., "Project SunSPOT - Sun Small Programmable Object Technology," Online: http://www.sunspotworld.com, 2008.

[13] Memsic Corporation, "TelosB Datasheet," Online: http://www.memsic.com, 2010.

[14] L. Richardson and S. Ruby, *RESTful Web Services*. O'Reilly Media, Inc., 2007.

[15] R. Manson, "SOAPjr," Online: http://soapjr.org/, 2008.

[16] S. Haustein, "kXML v2," Online: http://kxml.sf.net/, 2005.

[17] Wireless Application Protocol Forum, Ltd., "Binary XML Content Format Specification, Version 1.3," 2001.

[18] T. D. May, S. H. Dunning, and J. O. Hallstrom, "An RPC Design for Wireless Sensor Networks," in *Proceedings of the 2nd IEEE International Conference on Mobile Ad-hoc and Sensor Systems Conference (MASS)*, 2005.

[19] K. Whitehouse, G. Tolle, J. Taneja, C. Sharp, S. Kim, J. Jeong, J. Hui, P. Dutta, and D. Culler, "Marionette: Using RPC for Interactive Development and Debugging of Wireless Embedded Networks," in *Proceedings of the 5th International Conference on Information Processing in Sensor Networks (IPSN)*, 2006.

[20] D. Pfisterer, M. Wegner, H. Hellbruück, C. Werner, and S. Fischer, "Energy-optimized Data Serialization For Heterogeneous WSNs Using Middleware Synthesis," in *Proceedings of the 6th Annual Mediterranean Ad Hoc Networking Work-Shop (Med-Hoc-Net)*, 2007.

[21] G. Moritz, D. Timmermann, R. Stoll, and F. Golatowski, "Encoding and Compression for the Devices Profile for Web Services," in *Proceedings of the 5th IEEE International Workshop on Service Oriented Architectures in Converging Networked Environments (SOCNE)*, 2010.