Available online at www.sciencedirect.com

SciVerse ScienceDirect

Procedia Computer Science 19 (2013) 1121 – 1128

Procedia Computer Science

The 8th International Symposium on Intelligent Systems Techniques for Ad hoc and Wireless Sensor Networks (IST-AWSN)

Connecting Wireless Sensor Networks to the Robot Operating System

Philipp M. Scholl[a], Brahim El Majoub[b], Silvia Santini[c], Kristof Van Laerhoven[d]

[a] *scholl@ess.tu-darmstadt.de, Embedded Sensing Systems, Technische Universität Darmstadt*
[b] *bmahjoub@stud.tu-darmstadt.de, Embedded Sensing Systems, Technische Universität Darmstadt*
[c] *santinis@wsn.tu-darmstadt.de, Wireless Sensor Network Lab, Technische Universität Darmstadt*
[d] *kristof@ess.tu-darmstadt.de, Embedded Sensing Systems, Technische Universität Darmstadt*

Abstract

Robot systems largely depend on embedded systems to operate. The interfaces of those embedded systems, e.g. motor controllers or laser scanners, are often vendor-specific and therefore require a component that translates from/to the Robot Operating System (ROS) Middleware interface. In this work we present an implementation and evaluation of a ROS Middleware client based on the Contiki operating systems, which is suitable for constrained embedded devices, like wireless sensor nodes. We show that in-buffer processing of ROS messages without relying on dynamic memory allocation is possible. That message contents can be accessed conveniently via well-known concepts of the C language (structs) with negligible processing overhead compared to a C++-based client. And that the message-passing middleware concept of ROS fits nicely in Contiki's event-based nature. Furthermore, in order for an environment enriched with wireless sensor network to help robots in navigating, understanding and manipulating environments a direct integration is mandatory.

Keywords: wireless sensor networks, middleware, robot operating system, code generation

1. Introduction

Exploring previously unchartered and possibly risky environments with technology is the focus of several research initiatives. In numerous disaster recovery and monitoring scenarios for instance, robots are envisioned as a reliable instrument to enter and survey dangerous territory without risking human lives (as for instance during the Fukushima incident [1]). An alternative approach uses large distributed networks of wireless sensors that could be deployed by air to cover the area of interest, as for instance proposed by [2] where small and cheap wireless sensor nodes can support first responders in disaster mitigation. Both robot and wireless sensor network (WSN) approaches have their advantages and disadvantages: The former generally lends itself well in circumstances when actuation could prove vital during the exploration but takes time to cover and observe larger areas, the latter tends to be faster in capturing signals from a larger territory but does not include means to take significant action. This has led to several approaches enabling communication across nodes from a wireless sensor network and mobile robots active within the same environment.

doi:10.1016/j.procs.2013.06.158

For example, the sensor nodes dropped by a mobile robot [3] could not only provide sensor measurement but also an ad-hoc deployed wireless communication infrastructure.

Middleware to enable such direct integration of WSN and robot systems is still rare. In this paper we describe the direct integration of WSN platforms into the popular Robot Operating System [4]. ROS provides a message-passing publish/subscribe communication mechanism. The problem at hand is how to efficiently exchange those messages and how to create a simple, yet robust resource discovery mechanism. We decided for a code generation approach to translate ROS messages to a machine-native C-struct memory layout and show that a simple centralized proxy approach to resource discovery can achieve reasonable and robust synchronization despite the unreliable transmission links in wireless sensor networks. Compared to the standard ROS-client, this allows to use platforms which have no C++-support, where dynamic memory allocation is prohibitive or support for predictable execution speed (like real-time application) is necessary. While we evaluated our system [1] on the Contiki operating system, it is general enough to be also applied on other operating systems like TinyOS for example.

Several publish/subscribe middleware systems have been proposed for the specific needs of wireless sensor networks but with different goals in mind. LooCI, proposed by Hughes et. al. [5], supports many useful features like run-time introspection and dynamic reconfiguration and has a similar approach to defining messages. However these features are already available in ROS and therefore an adaption would have been too expensive. Active Messages in TinyOS [6] also define messages at compile-time, as well as the wiring of components. While this allows for optimizations it is not applicable to the dynamic nature of the ROS environment. The Constrained Application Protocol (CoAP) [7], an IETF effort to standardize a webservice-like middleware system for contrained devices, also provides similar mechanisms to the publish/subscribe ROS system. It however lacks a type system for the exchanged messages but might prove to be an alternative for resource discovery. ROSSerial [2] is the most similar effort but aims for general embedded systems, like the Arduino. There is however only partial support for resource discovery in wireless sensor networks and networked connections in general. We compare our implementation of the message en-/decoders to the ones generated by ROSSerial.

2. System Design

The governing goals of our design are to avoid unnecessary runtime and API complexity while maintaining a low memory/code footprint, portability and robustness against transmission failures. These goals stem mainly from the special requirement of the embedded platforms typically used in wireless sensor networks. Memory constraints, in the scale of a few kB and the overhead of dynamic memory allocation are the toughest challenges posed by these microcontrollers. In section 2.1 we present the design of an “in-buffer” en-/decoding scheme for ROS messages. Another important factor of the design is the unreliable nature of wireless communication, which not only influences the actual data transmission but also the overhead of maintaining a congruent view of the available resources in the network. We show two strategies for facing this challenge in section 2.2.

In order to ease the transition for programmers with a ROS background, we map most of the ROS concepts to our implementation. ROS networks are organized as *components*. These are executable processes that are connected to other components via topics or services. *Topics* provide a publish/subscribe mechanism which enables many-to-many one-way transports. *Services* on the other hand provide one-to-one remote procedure call interactions based on the publish/subscribe mechanism. Both interactions are based on the exchange of *messages*, which are defined as nested data structures containing primitive types (integers, floating points, ...) and arrays. These messages are defined during compile-time and attached to

[1] The proposed system is publicly available at `https://www.github.com/pscholl/contiki_rosnode`

[2] During preparation of this paper a further alternative ROS-Client became available, also relying on dynamic memory allocation despite aiming for embedded systems. It is freely available at `https://github.com/openrobots-dev/uROSnode`

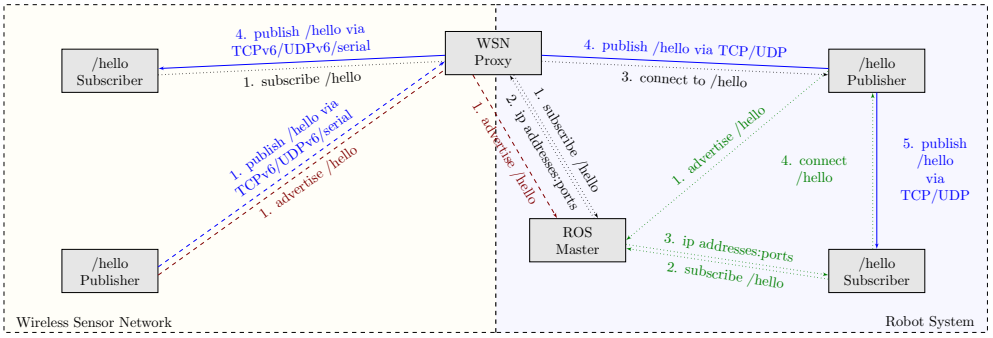

Fig. 1: This figure shows the resource discovery and connection establishment of ROS components. Each block represents a component running either on a PC System (right side) or on a wireless sensor node (left side). Dotted lines represent the exchange of (ordered by number) management information. Black colored components shows the subscription establishment for PC components and green for subscriptions. Yellow and red show the same for a wireless sensor node, i.e. all communications runs through the proxy component. Blue lines represent the actual data exchange on established connections.

topics/services during run-time, which leads to a statically typed message exchange. Topics, services and message alike are identified by strings, which also allow for hierarchical ordering. Following the taxonomy of Eugster et. al. [8] we can see that this design can be coined as a hierarchically-ordered Topic-Based Publish/Subscribe system with static component deployments.

2.1. En-/Decoding ROS messages in C

A central part of our proposed ROS middleware client is the encoding and decoding component of ROS messages. Since ROS messages are defined during compile-time we adopted a code generation approach to map the on-wire ROS message format to the machine-dependent C struct memory layout. The generated code tries to minimize the number of additional memory needed to hold the message and the memory required during en-/decoding. To achieve this, especially in the absence of dynamic memory allocation, most operations are done inside the transmission buffer with only a small fraction done via memory allocated temporarily on the stack. Such a code generation approach allows to adapt the endianess of primitive types as well as aligning the message to the machine's natural data access width by reordering the objects inside the message.

The ROS on-wire message encoding works by sequencing the objects of a message one after another. Primitive types (like integers, floats ...), arrays of fixed size, nested messages, arrays with a dynamic size as well as strings are supported. Dynamic arrays and strings are the exception, since they are prepended with their length, i.e. the pascal string encoding. An example of an encoded message can be seen in Fig. 2. The left hand side of this figure contains the message declaration of two nested messages, the shaded areas in this figure shows the message as it would be encoded in a transmission buffer. One can see that objects are put into the buffer one after another, while the structural information has been stripped.

The straightforward ROS message encoding is not directly mappable to a C-struct memory layout, because there is no language construct to allocate memory for dynamic sized arrays (and strings for that matter) inside of structs. However, if we allow for a slight increase in storage space the memory can be reorganized to allow for natural access (in the sense of available C language concepts). To achieve this we conceptually split the message into a static and dynamic part. As its name implies the static part holds all members, for which the size is know during compile-time, i.e. primitive types, static arrays, and additionally indirections to the dynamic sized object. The data of these dynamic objects will be moved to the dynamic part (i.e. to the tail) of the memory buffer and can only be accessed through aforementioned indirections. This approach allows to define a ROS message in the most natural C-language struct, which can be seen in Fig. 2. The

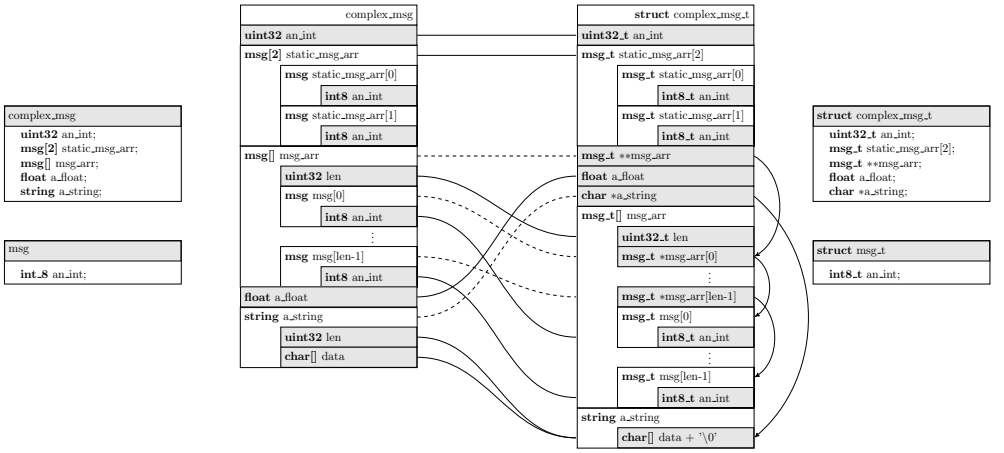

Fig. 2: The memory layout (and declaration) of an on-wire ROS message and the respective C struct layout (and its declaration). Shaded areas depict actual used memory, while non-shaded areas depict structural information that is not stored/transported. Lines in this figure constitute the operation during en-/decoding, solid lines are copy operations while dashed line show when management information needs to be added (and accordingly further memory needs to be allocated) or can be removed.

right-hand side of this picture shows the respective C-struct declaration and its memory layout, the figure also shows where necessary indirections are created. So, all primitive types and static arrays will end up in the head part of the message buffer, dynamic sized objects will be replaced by indirections and their data moved to the tail of the message buffer. It should be noted that this approach can not be easily applied to nested messages, since these can contain dynamic sized objects too and therefore their size is not known during compile-time. The code generator takes care of this by "unrolling" nested message definitions. So for each nested message, there is a special part of code which en-/decodes a particular nested message inside the containing message. By reorganizing the static objects to the head of the message buffer, moving dynamic objects to the tail of the buffer and adding according indirections it is possible to map ROS messages to a natural C struct memory layout.

In order to decode a ROS message to a C struct we propose the following two-step algorithm, which only temporarily allocates memory on the stack and places the result in the original transmission buffer:

1. Allocate memory for all static members from the stack and copy all static members into this memory region. While copying, alignment and endianess of primitive types are fixed. Additionally, indirections to dynamic arrays and strings are initialized since information on their size is now available.
2. In the second step, we iterate over all message member in reverse order. The data of each dynamic member is then *moved* to its respective position at the tail of the message buffer. Once this movement has been completed, the static members are copied from the stack into the transmission buffer and the decoded message is returned.

In step 1 the overhead in storage space is introduced. Compared to the ROS encoding, the C struct encoding needs additional management information in order to support dynamic members. For dynamic arrays the overhead equals the size of $n + 1$ indirections, where n represents the number of elements in the arrays and one additional indirection is needed for storing the position of the arrays inside the message buffer. For strings the overhead can be calculated with the following formula: $sizeof(char*) - sizeof(uint32\_t) + 1$, which represents the size of the indirection needed to access the string and the space reduction by converting from a length-prefixed string to a null-terminated C-string.

Encoding messages from a C-struct representation to the ROS message representation takes less effort. Simply iterating over all members in the message and copying their data, again fixing endianess and alignment issues, and prefixing by length for dynamic members does the job. This of course only works when transmission buffer and message buffer do not overlap. However, we have seen that by allowing a small overhead in memory consumption allows to access the static and dynamic members of ROS message with C constructs.

2.2. Resource Discovery

In this paper the term Resource Discovery is used to address mechanisms which allow string descriptions to be resolved to actual data sources. In the case of our proposed ROS client, strings can be used to address *topics* as well as *services*. The central ROS master keeps track of the networking addresses of individual nodes, of the respective *topics* they publish on/subscribe to and their advertised *services*. In our implementation the WSN proxy performs this task for the WSN. This centralized proxy is used to keep transport-agnostic information about the advertisement/subscription of the connected WSN nodes. For example, whenever a WSN node wants to subscribe to a specific *topic* it sends a request to the WSN proxy, which then negotiates this request with the OS master, creates a connection to the remote node and brokers the data exchanged on this topic. Fig. 1 shows the message exchange of the WSN proxy, ROS master and participating nodes during topic negotiation (i.e. subscribing to/publishing on a *topic*) inside the ROS network, and over the WSN boundary.

While a centralized approach to resource discovery, like the one presented here, constitutes a single point of failure it has several advantages that make it a viable solution. First of all, multiple different transport protocols like TCP/IPv6, UDP/IPv6, serial transmission lines or the XBee protocol can be supported. While for most of the thinkable protocols a direct connection between the nodes of the ROS network and WSN network is not possible (e.g. one node communicates via the XBee protocol), it is still possible to directly connect those that do. For example, if both a WSN node and a ROS node communicate on the same IP link a direct connection can be established instead of brokering all message through the proxy, mitigating the single point of failure. The same applies to connections inside the WSN network as well, in which case the WSN proxy only provides a central register of resources and their respective network addresses.

In order to increase the portability of the proposed system we decided to adopt a stubborn message exchange strategy for resource discovery. Furthermore, assuming an unreliable transport channel allows to cover a large number of different transport channels at the expense of increased implementation complexity - resource discovery needs to be as reliable as possible. To keep this complexity on a manageable level we adopted a stubborn communication strategy, i.e. nodes communicate their advertised/subscribed topics and services to the WSN proxy in a periodic fashion. This strategy does not scale well and is not very resource-efficient, but can be customized to the specific environment by adapting its periodic. Such a periodic exchange of resource information provides a robust discovery mechanism and allows to guarantee a discovery latency with high probability for unreliable channels as can be seen in Section 3.3.

3. Implementation and Evaluation

Our implementation is based on the Contiki [9] Operating System. Contiki provides the means to communicate with PCs through its TCP/IPv6 stack via 6LoWPAN. The ROS Publish/Subscribe middleware concept fits nicely into the event-based nature of Contiki - messages published on a topic are delivered to Contiki processes as an event and are also published to topics in such manner. It should be noted that while our implementation is based on Contiki, it is also general enough to be ported to other OSs like TinyOS [10] or even use it as an alternative ROS client on PCs.

In this section we compare our implementation to the ROSSerial [3] implementation, which also targets embedded platforms. We quantify the performance and memory overhead of both solutions, as well the

[3] available at: `http://www.ros.org/wiki/rosserial`

Table 1: This table shows the code size in bytes, as reported by the compiler-generated memory map file, of the ROSSerial and our implementation on various WSN platforms.

	C++ ROSSerial			C our solution			
	discovery	de-/encoder	total	discovery	de-/encoder	total	compiler
TMote Sky	996	2502	3498	806	3774	4580	msp430-gcc 4.6.3
Zolertia Z1	1356	2332	3688	806	3814	4620	msp430-gcc 4.6.3
RedBee EconoTag		-		936	3052	3988	arm-gcc 4.3.2
Jennic-based jNode		-		788	3583	4371	ba2-gcc 4.1.2
Intel Core i7	1205	2015	3220	1074	2222	3296	gcc 4.7.2

delay of our resource discovery strategy in presence of unreliable transmission links. Our WSN proxy (called hector_serialization) [4] implementation is compatible to the ROSSerial one and can be used in place.

3.1. Memory Overhead

We evaluated two different types of memory overhead. The first one originates from the necessary code to handle resource discovery and to de-/encode the ROS message format. Table 1 shows the size of our proposed solution split into a discovery and a de-/encoder part compared to the C++-based ROSSerial solution. The specific messages for which these de-/encoders were generated for are shown in Fig. 2. We compiled our solution for multiple typical WSN platforms that are supported by Contiki, like the TMote Sky, Zolertia Z1, RedBee EconoTag, the Jennic-based jNode and for a native environment based on an Intel Core i7. It can be seen that our C-based solution consumes more code memory for the de-/encoder when compared to the C++-based ROSSerial solution. This increase in memory consumption is mainly due to the fact that our solution supports arrays of nested messages, which needs code to unroll this nesting. This feature is not supported by the ROSSerial implementation and leads to a lower code memory consumption. For messages which are not using these nested arrays similar space is required for both solutions. It can also be seen that discovery has comparable code complexity.

The second overhead we can look at is the memory required to store the message. For encoding messages this overhead is nearly the same, in both cases the message needs to be stored either on the stack or in static buffers. Additional memory, i.e. information that is not transferred, is needed for accessing strings and dynamic arrays via indirections. This additional memory is also needed when storing a decoded message. Our solution however decodes in the transmission buffer, which allows to save the memory for almost the whole message. If we let n_s be the number of strings, n_a the number of dynamic arrays, m be the number of elements in an array with nested messages, and $sizeof(x)$ be the bytes required to store an object without string and array data, we can calculate the memory required for storing a decoded message. For ROSSerial this equals $sizeof(msg) + n_s * sizeof(char*) + n_a * sizeof(void*)$, i.e. the whole message plus additional space for storing indirections to strings and arrays. For our solution the additional required space is only $n_s * sizeof(char*) + (n_a + m) * sizeof(void*)$. So our solution requires less additional storage space, while also supporting arrays of nested messages.

3.2. Performance Overhead

We define performance overhead as the time needed to translate a message from the ROS on-wire format to the memory layout used by the programmer's code. In our implementation this overhead stems mainly from the adjustment of the memory layout of ROS messages to the architecture-dependent C-struct layout. Since there is no concept of in-place strings or dynamic arrays in C, these need to be shifted to the tail of the message buffer in order to be accessible from the message header. Integral types like floats and integers have to be copied as well in order to convert them to their machine-native format.

[4] available at: `https://github.com/tu-darmstadt-ros-pkg/hector_serialization`

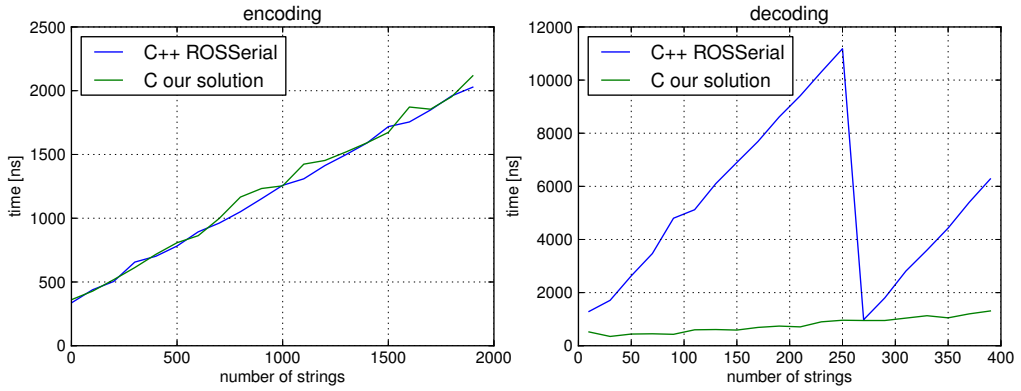

Fig. 3: Time needed for the two en-/decoder implementation to encode/decode a ”worst-case” message on a PC. It’s clearly visible that the use of dynamic memory management (via realloc) is a performance problem during decoding.

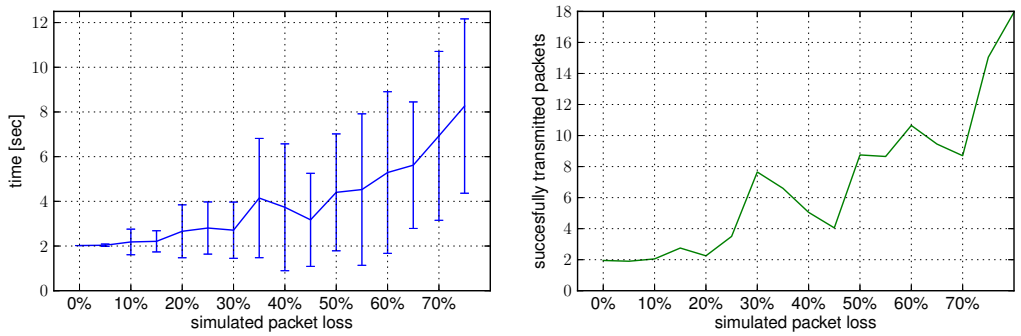

Fig. 4: Required time and number of succesfully transmitted packets to subscribe to a topic under simulated packet loss.

To quantify this overhead we measured the CPU time difference required to encode/decode the two messages shown in Fig. 1 with the C++-ROSSerial implementation and our proposed C-based implementation. We ran these experiments on a linux machine with a quad-core Intel Core i7 CPU running at 2.8GHz. Each measurement was repeated 10000 times and, since ROSSerial does not support nested dynamic arrays, we varied the length of the included array of strings as the worst case scenario for both implementations. Fig. 3 shows the result of this experiment. While the encoder performance is virtually the same, the ROSSerial decoder shows almost a 10-fold increase in runtime compared to our C-based solution. This mainly stems from the use of dynamic memory management (via realloc) in this decoder to support dynamic arrays. In our solution this is solved by reordering the memory layout and moving dynamic arrays to the tail of the buffer. One can also see when the memory needs to be moved around during the *realloc* operation and when enough memory is still available (the sharp decline in time consumption during decoding for ROSSerial is an artifact of this). Our solution can therefore decode messages faster while not relying on the availability of a dynamic memory management unit.

3.3. Discovery Latency

To validate our resource discovery approach, we emulated an increasing packet loss and measured the time and the number of exchanged packets required to subscribe to a single topic. For this we setup we

ran our Contiki implementation on a standard linux machine. Communication was done via a TUN/TAP device, which emulates network interfaces on linux. Subscription requests were sent every 250ms. To simulate various random packet losses we employed the NetEm [11] package available on linux. We varied the packet loss from 0 – 80%, measured the walltime and number of exchanged packets needed until the WSN proxy acknowledged the subscription on this topic, and repeated each measurement 20 times. The result of this experiment can be seen in Fig. 4. We can see that the delay is only slowly increasing while the number of exchanged packets shows an increased effort. While interpreting the number of transmitted packets it should be kept in mind that the dropped packets are not included in this figure. So for example under a packet loss of 60% 10 packets have been successfully sent from the node to the proxy until the acknowledgment was successfully received. However even under high packet loss rates the subscription delay is reasonably small, showing that our simple stubborn resource discovery approach is robust.

4. Conclusion and Future Work

Connecting Wireless Sensor Network to the Robot Operating System is hindered by missing middleware for constrained devices like WSN nodes. Together with ROSSerial this paper presents first steps towards integrating the Wireless Sensor Network world with the Robot Operating System on the application layer. We have shown that despite the non-fitting design of the ROS message encoding (by simply sequencing all message members) an integration is achievable and that even a simple resource discovery approach can achieve robust functionality.

It remains open however whether our approach is feasible for large networks, leaving specifically the question whether research in WSN protocols can be applied fruitfully. Future work should look into different resource discovery schemes, for example the topic-based publish/subscribe concept could be mapped to multicast approaches. Especially multicast DNS for topic resolving and multicast communication for message exchange could prove to be a viable alternative to a centralized approach. Furthermore, a message encoding which explicitly separates static and dynamic parts of a message might prove to be an important optimization.

Acknowledgments

The authors would like to acknowledge funding by the DFG/GRK1362 - Cooperative, Adaptive and Responsive Monitoring in Mixed Mode Environments.

[1] K. Nagatani, S. Kiribayashi, Y. Okada, K. Otake, K. Yoshida, S. Tadokoro, T. Nishimura, T. Yoshida, Koyanagi, et al., Emergency response to the nuclear accident at the fukushima daiichi nuclear power plants using mobile rescue robots, Journal of Field Robotics.
[2] V. Kumar, D. Rus, S. Singh, Robot and Sensor Networks for First Responders, Pervasive Computing, IEEE 3 (4).
[3] T. Suzuki, K. Kawabata, Y. Hada, Y. Tobe, Deployment of Wireless Sensor Network using Mobile Robots to Construct an Intelligent Environment in a Multi-Robot Sensor Network, Adances in Service Robotics, Austria.
[4] M. Quigley, et al., Ros: an open-source robot operating system, in: ICRA workshop on open source software, Vol. 3, 2009.
[5] D. Hughes, et al., Looci: a loosely-coupled component infrastructure for networked embedded systems, in: ACMCM, ACM, 2009.
[6] P. Buonadonna, J. Hill, D. Culler, Active message communication for tiny networked sensors.
[7] M. Kovatsch, S. Duquennoy, A. Dunkels, A low-power coap for contiki, in: Mobile Adhoc and Sensor Systems (MASS), 2011 IEEE 8th International Conference on, IEEE, 2011.
[8] P. T. Eugster, P. a. Felber, R. Guerraoui, A.-M. Kermarrec, The many faces of publish/subscribe, ACM Computing Surveys 35 (2).
[9] A. Dunkels, T. Voigt, Contiki - a Lightweight and Flexible Operating System for Tiny Networked Sensors, Computer Networks.
[10] P. Levis, S. Madden, J. Polastre, R. Szewczyk, K. Whitehouse, A. Woo, D. Gay, J. Hill, M. Welsh, E. Brewer, et al., Tinyos: An operating system for sensor networks, Ambient intelligence 35.
[11] S. Hemminger, et al., Network emulation with netem, in: Linux Conf Au, Citeseer, 2005.