sDDS: A Portable Data Distribution Service Implementation for WSN and IoT platforms

Kai Beckmann and Olga Dedi
Distributed Systems Lab, RheinMain University of Applied Sciences
Unter den Eichen 5, D-65195 Wiesbaden, Germany
Email: {kai.beckmann|olga.dedi}@hs-rm.de

***Abstract*— In the last years many different protocols, middleware approaches and software stacks were developed for Wireless Sensor Networks (WSN) and the Internet of Things (IoT). Limited interoperability and support for heterogeneous target platforms are often problematic. A good match for this problem domain would be the OMG Data Distribution Service (DDS) standard, which specifies a data-centric publish/subscribe middleware with real-time capabilities and a rich set of Quality-of-Service polices. Although DDS was not intended for embedded systems with limited resources as found in WSN, the authors of this paper propose an approach for a customisable DDS implementation (sDDS) using an model-driven software-development process to tailor the middleware functionality on an individual node level. In this paper, the platform-abstraction layer of sDDS, the platforms supported so far and the effort needed for the porting process are presented. With this approach, it was possible to port sDDS to different platforms, like Contiki, RIOT-OS, FreeRTOS and self-contained ZigBee stack implementations.**

I. Introduction

Wireless sensor networks (WSN) and the Internet of Things (IoT) are becoming real world applications supported by the availability of affordable integrated hardware solutions and driven by the demand for "smart" applications and devices [1]. In the last years, many different protocols, middleware approaches and software stacks were developed to ease application development and to interconnect devices. A problem is the limited interoperability and the pursuit of different approaches for vertical integration. The IP protocol family is often considered the answer for these kind of problems: one protocol to connect them all. But this moves interoperability issues only to the application layer. There is the need for more universal solutions providing interoperability and horizontal and vertical integration, based on free and open standards, independent of vendors, communications technology and hardware platforms. The ease of portability, support of different and heterogeneous platforms are measures for the usefulness of such a solution.

The Data Distribution Standard (DDS) of the Object Management Group (OMG) [2] is a middleware standard for a data-centric, publish-subscribe based communication system, providing real-time and Quality-of-Service (QoS) capabilities. At the core of the DDS standard is the idea of a global data space, where all participants have equal access to the data, enabling transparent horizontal and near-effortless vertical integration.

The characteristics of DDS are a very good match for the requirements of WSN, but originally, DDS implementations have been targeting more resource rich embedded systems with Ethernet-based networks. Therefore, a normal DDS implementation is too large for small sensor nodes with limited memory.

To make DDS applicable in resource limited environments, like WSN, the authors propose the DDS implementation sDDS, which includes a model-driven software development (MDSD) process to tailor and minimise the middleware functionality for each sensor node, depending on the purpose of the node in the network, the resource capabilities of the hardware and the deployment structure. The results are individual middleware implementations with different subsets of the DDS API functionality. sDDS can be used on a wide range of target systems, from 8 bit to 32 bit microcontrollers and PC platforms. The MDSD process was previously presented in [3] and the developed middleware protocol for supporting heterogeneous middleware implementations in a WSN in [4]. The contribution presented in this paper focuses on the approach to minimise the porting effort to and the realisation for different target platforms.

In the following section II, a short introduction to DDS is given. There are approaches to make DDS applicable in WSN, which are summarised in the related work in section III. In section IV, an introduction to sDDS is given, focusing on platform-abstraction layers. The realisations for currently supported platforms are described in section V and the necessary effort in terms of implementation time and lines of code is specified. For these platforms, the memory footprint of sDDS with some example setup is given as well. The last section (VI) gives the conclusion and an outlook on future work.

II. DDS Background

DDS is an open standard for a data-centric publish-subscribe middleware platform with a rich set of real-time and QoS capabilities published by the OMG [2]. The OMG efforts are generally aiming at portable, vendor-neutral standards and therefore the DDS API is expressed in a platform independent and object-oriented manner in OMG IDL. The first version, specifying the API only, was released in 2003, and the current version 1.5 was published 2015. For the interoperability between implementations of different vendors in one network, OMG has standardised the Real-Time Publish-

Subscribe (RTPS) protocol [5] in 2006, which proposes the usage of UDP/IP as network protocol.

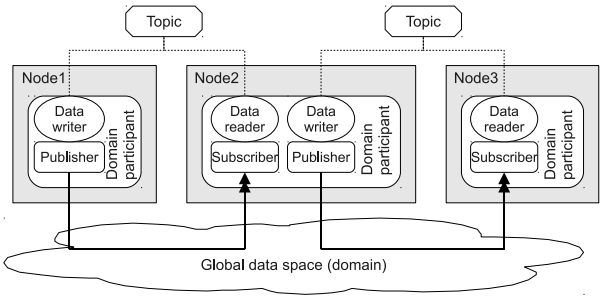

Fig. 1. Essential DDS architecture

The purpose of DDS is summarized by the standard as the "Efficient and Robust Delivery of the Right Information to the Right Place at the Right Time" [2]. The architecture of DDS is based on a data-centric approach and the idea of a global data-space, where typed data can be published or accessed by equal peers. An abstract architecture of an example DDS system is given in Fig. 1. A data-space can be subdivided by so-called *domains*, allowing the separation of interests and communication domains. A *DomainParticipant* connects an application to one specific domain. Each distinguished kind of data is addressed by a *topic*, adding a name and an optional set of mandatory QoS policies to a data-type. A subset of OMG IDL is used to define the data-types, offering complex types like structures and arrays, as well as primitives like integers. From the modelled data-types program code for specific access classes is generated. These *DataReader* and *DataWriter* classes are the interfaces of the applications to receive or publish data from or to the middleware, respectively. The *Subscriber* and *Publisher* classes bundle the DataReader and DataWriter objects of an application within one domain.

The DDS standards defines 22 QoS policies that can be applied to all the architectural DDS class entities, requesting or offering specific QoS properties. To establish a subscription, at least one DataWriter and one DataReader associated with a topic have to be available and they must have compatible QoS settings. The QoS policies manage the properties of the data distribution regarding reliability of communication, bandwidth consumption, latencies and redundancy, as well as resource constrains for the middleware nodes. The policies influence each other significantly and can be changed and adapted during runtime by the application.

The underlying network and the communication relationship between nodes is transparent for a DDS application, they are primarily associated with the data of interest. Nevertheless, the DDS standard defines special *Built-in-Topics* to provide applications with information about the other known entities, like DomainParticipants, DataReader, DataWriter and Topics, in the system. It is possible to get notified when new entities are created or destroyed. The discovery of nodes and DDS entries in the network and the establishment of connections and subscriptions is not part of the DDS standard itself, but covered by the RTPS protocol. RTPS is built on UDP/IP and uses multicast, if possible, to improve the data distribution between publisher and subscriber.

III. Related Work

There are different approaches for enabling DDS for WSNs. One of the first is TinyDDS [6], which is based on the WSN operating system TinyOS. TinyDDS inherits the proprietary nesC programming language from TinyOS and is therefore not API-compatible with the DDS standard. The portability on the application level is thereby lost. TinyDDS implements a limited subset of the DDS functionality, which cannot be changed or adapted to the requirements of the application. As transport protocol, TinyDDS defines its own stack based on the routing functionality of TinyOS. Due to the tight coupling with TinyOS, platform support and portability depend on availability of the operating system.

DDS vendor RTI offers the Connext DDS Micro Edition [7] for smaller target platforms like 16 bit microcontrollers. This product implements a subset of the DDS functionality providing a standard conformant API and implements the RTPS protocol, providing interoperability with normal DDS implementations. However, RTPS needs UDP/IP based on Ethernet and is not suitable for typical WSN network technologies.

Another approach focusing on real-time aspects is µDDS [8]. It implements a subset of the DDS functionality as well and provides a standard conformant C-API. µDDS is implemented on the POSIX 5.1 interface of the PaRTiKle real-time operating system and aims at resource richer target systems than WSNs. µDDS uses the ZigBee protocol as transport layer, which restricts the interoperability with other DDS implementations and the portability to other hardware platforms.

IV. sensornetwork DDS

sDDS ("sensornetwork DDS") is an approach to make DDS available for very small distributed embedded systems like WSN. An MDSD process is used to select a subset of DDS functionality and thereby to generate individual DDS implementations for each node [3]. The required DDS features of the applications and the underlying distributed systems are modelled with domain-specific languages (DSL) separately. This information is combined based on a meta-model covering all these aspects and is used for optimisations, transformation to platform-specific model representations and code generation. An advantage of the MDSD process is the reduced effort for adapting to changes of target hard- or software or the functionality needed. Alternatively, a simpler process with configuration files and scripted code generation can be used as well. The selected subset of the DDS API can be used to implement applications.

To facilitate a system with heterogeneous middleware implementations and respect the special properties of WSN communication protocols, the Sensor-Network Publish-Subscibe (SNPS) protocol was developed for sDDS [4]. It is optimised

for low protocol overhead, small frame sizes and utilises the natural broadcasts of wireless networks for data distribution within the middleware system.

A. Architectural Overview

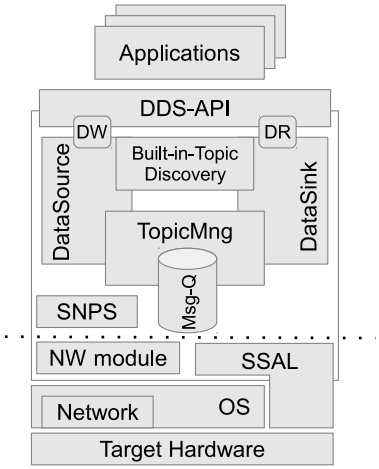

Fig. 2. General sDDS architecture

sDDS uses the class structure defined by the DDS standard as part of its meta model and DSL environment and provides the parts requested by the application developer. But, to reduce the memory footprint and to optimise the processing of data, the sDDS architecture is kept slender and the implementation is generated with less abstraction layers and objects. In Fig. 2, the general architecture is displayed. The structure of sDDS is divided into three parts: the DDS conformant API, the platform-independent middle layer and the platform-specific layer for the network and operation system's abstractions. The latter two layers are completely separated by interfaces and the platform-specific parts are implemented in separate modules. This eases the implementation of sDDS and the porting process at the current development stage significantly, because the possible code variations introduced by the MDSD process are a challenge for maintaining, testing and debugging. The functionality in the middle layer is divided into the tasks of data sending (DataSource), data receiving (DataSink), topic and data management (TopicMng) and Discovery and Built-in-Topic support. Depending on the requested functionality these parts are fully, partly or not present in a generated sDDS implementation. The QoS support is a cross-cutting-concern and impacts different parts of the modules.

B. Platform-specific-Layer

To achieve a general level of platform-independence and compatibility with microcontroller-platforms, sDDS is implemented in plain C in the C99 version with minimal usage of the standard C library and no floating-point arithmetic or variables. The platform-specific part is encapsulated in the network module and the operating system system software abstraction layer (SSAL). The network module is necessary in every sDDS implementation, since it deals with the data transport. The SSAL is modularised and provides interfaces for different system software functionalities; depending on the configured sDDS, the necessary subset is included.

Currently, sDDS and the SNPS protocol depend on a transport layer with a datagram service and routing functionality. To be able to manage and identify connections with other nodes, network addresses are abstracted in locator objects, which are managed by the network module. Locators can be checked for equality, linked in lists and they have a reference counter. The rest of the interface is kept simple: a free frame buffer can be requested, marked ready for sending, and a callback of the middleware is called when a new message arrives. In most cases and platforms, receiving messages is managed by some kind of thread or task within the network module. So far, most effort for porting sDDS to a new platform has to be spent on the network module.

The SSAL currently consists of interfaces for logging, simple memory management, a task concept and mutexes. Logging is implemented by preprocessor macros, supports log levels and can be deactivated completely. Currently, the focus for sDDS are static scenarios, therefore, only a memory allocation functionality is used within the initialisation phase as part of the memory management. The task abstraction allows a function to be registered for periodic or deferred execution. This task model is the common denominator for all platforms considered so far. Mutexes become necessary if multiple applications are present or the internal discovery functionality is selected. The thorough implementation of an option for concurrent access is an ongoing task.

The platform-specific layer is kept simple and lean, therefore, the porting process is straightforward. The network module causes most of the implementation effort, but the existing code of other platforms can normally be used as starting point or template. The implementation time primary depends on the knowledge and experience about the target platform. The logging support is very important for development, and it is unlikely that a hardware or software platform does not support any textual output. In most cases, the default "printf" implementation can be used. Often there is some kind of memory management on the target systems that can be utilised. The implementation of tasks and mutexes is only necessary if some kind of dynamic behaviour or adaptability of sDDS is needed. If there is no system software functionality for tasks or threads, it can be implemented with timers directly. Mutexes would need an atomic test-and-set operation or comparable functionality.

Besides the implementation of the platform-specific layer, the build environment can be the most cumbersome task for porting sDDS. The development environment for most target platforms have their own build system. The integration of sDDS with the code generation process in a complex build environment can be the most time consuming part.

V. Realisation for different Platforms

As presented in the previous section, sDDS does not impose complicated requirements to the system layer. It would be

possible to implement this functionality from scratch for a given hardware platform. For efficiency reasons, sDDS was placed on top of existing software and protocol stacks used in the WSN domain. sDDS was ported to the following platforms so far:

1) GNU/Linux
2) ZigBee stack environment
3) Contiki
4) RIOT-OS
5) FreeRTOS

In the following section, these platforms are introduced and the specific porting activity and lessons learned thereby are detailed afterwards. This section is completed with an examination of the porting effort to and memory consumption of the supported platforms.

A. Supported platforms

For sDDS, the operation system GNU/Linux is used as target and primary development environment. Since sDDS is implemented in plain C and the platform-specific parts are abstracted, it can natively run as a Linux process using BSD UDP sockets over IPv4 or IPv6. The Linux version is primarily used for the development of new platform-independent functionality, given the advanced development and debug options on this platform. Additionally, the Linux platform serves as target system running native sDDS applications and bridging WSN and PC-based systems if the network protocol is IP-based, like 6LoWPAN.

ZigBee is a standard for a WSN protocol stack targeting the home automation domain. It is managed by the ZigBee Alliance, and the protocol is based on an IEEE 802.15.4 network layer. Besides the communication protocol layers, ZigBee defines Profiles for devices and applications. This semantic specification allows the easy pairing of corresponding functionality. There are different ZigBee stacks available, primarily from the hardware vendors manufacturing the integrated microcontrollers for ZigBee appliances. For the two stacks zStack [9] on a CC2430 from TI and BitCloud [10] on a Atmega128RFA1 from Atmel, sDDS was prototypically ported and evaluated. For application development both stacks provide basic system software functionality like memory management and the concurrent execution of application code. The hardware targets were 8 bit microcontrollers with an integrated IEEE 802.15.4 transceiver for 2.4Ghz.

The current main target for development is based on the Contiki operation system [11], a rather popular operating system for WSN. Contiki runs on a wide range of very small hardware platforms like 8 bit microcontrollers and provides basic operating system functionality like threading, events and dynamic program loading. Contiki offers an optional preemptive thread concept and respectively a very lightweight, but stackless protohreads [12] implementation. Furthermore, Contiki provides TCP/IP and 6LoWPAN stacks. To bridge 6LoWPAN and IPv6 networks, Contiki offers a transparent gateway solution, based on a USB device with a 802.15.4 transceiver. As hardware platform the Atmega128RFA1 from Atmel is used. sDDS was ported first to version 2.6 of Contiki and recently to version 3.0.

RIOT-OS [13] is a relatively new operating system for WSNs offering real-time capabilities, real threading and support for 6LoWPAN and IPv6. RIOT is very lightweight in terms of memory footprint. Besides a RIOT-specific API a partial POSIX API is offered for applications. It is possible to run RIOT as a native process under Linux, which was used for porting of sDDS. Unfortunately, there were no embedded hardware platforms with network functionality supported by RIOT available to the authors, therefore evaluation on an embedded target platform is an open task.

FreeRTOS is a popular real-time operating system for embedded devices [14]. Therefore, combined with the community maintained IP-stack lwIP [15], it provides a suitable platform for various IoT applications such as sDDS. For porting sDDS to FreeRTOS the 32-bit RISC based SoC ESP8266 with IEEE 802.11 b/g/n Wi-Fi functionality was used as platform. Hardware platforms supporting Wi-Fi become more common and available. A version of FreeRTOS adapted for this hardware platform was used [16]. IPv6 was not fully supported in this version, therefore, only IPv4 was used.

B. Porting process and lessons learned

The first realisations of sDDS on embedded platforms were based on ZigBee stack implementations, utilising the offered system layer functionality for ZigBee applications. sDDS was placed beside a native ZigBee application, using the ZigBee layer 3 functionality for communication. The available resources were very limited, since ZigBee stack and application consumed most of it. The architecture, API and build system of zStack and BitCloud are very different. While zStack has a procedural interface, BitCloud is event-driven. Therefore, the platform-specific adoption of sDDS differs significantly. Both have in common a steep learning curve before applications can be developed. Another complication for the sDDS realisation for zStack was the need of the IAR Embedded Workbench for development. This environment did not fit well in the Linux-based sDDS development process. Because of the limited functionality, only the network module and the memory abstraction were implemented. The necessary time for implementation was not measured, since these were the first platforms and the basic sDDS functionality was developed in parallel. It was possible to fit sDDS with minimal DDS functionality and a few DataReaders and DataWriters on both ZigBee platforms. Multiple nodes were able to exchange data within the same ZigBee stack environment. An interoperability test was not pursued, because of general interoperability problems at the ZigBee level at this time. After the proof-of-concept implementations, the usage of ZigBee-based systems was stalled, as a consequence of the cumbersome development environments and better interoperability results with 6LoWPAN-based network stacks.

sDDS was first ported to Contiki version 2.6 by a Bachelor student during a course work within two weeks. The sDDS version used was limited to pure data exchange with static

communication relationships and no QoS support. Therefore, only the network module and the memory abstraction had to be implemented for this Contiki version. Contiki protothreads were used for handling incoming messages within the network module and for the application. This version was successfully used on the Atmel Atmega128RFA1 SOC platform in an Ambient Assisted Living (AAL)/Smart Home project [17] connecting a dozen sensor nodes with different sensor configurations and applications to a PC-based service platform. Thanks to the transparent 6LoWPAN-to-IPv6 gateway of the Contiki project, interoperability was achieved, without extra effort, by native Linux sDDS applications. An enhanced version of sDDS with multicast and discovery support was recently ported to the new Contiki version 3.0 within four days by a Master student, who had worked with sDDS before. Building on that work, multicast for 6LoWPAN and a Contiki implementation of the task abstraction layer were added.

For RIOT-OS, the porting process to the POSIX-API and the native Linux environment was very straightforward. RIOT provides a BSD-socket interface, such that the network module from the sDDS Linux platform could be reused significantly. The other parts of the SSAL were ported similarly. A Master student, how has not worked with sDDS before, needed about 14 days for the implementation. A setup with mixed native RIOT and Linux sDDS applications communicating were possible. The RIOT concept promotes a simple transfer of the RIOT applications from the native Linux environment to an embedded platform.

Porting sDDS to FreeRTOS consists of setting up the build environment and implementing the network module and the SSAL modules. This task was accomplished by two Master students within three days, where setting up the build environment turned out to be the most time-consuming part, since each platform comes with an individual makefile which has to be integrated into the existing build environment. One student had worked with sDDS before, the other only had experience with embedded devices in general. The network and SSAL module contain platform specific code for network communication, thread handling and memory management, thus, the implementation effort highly depends on the level of familiarity with the underlying system API.

C. Comparison of effort and memory footprint

As a summary, the platforms presented so far are compared regarding the necessary effort for porting and their resource usage in terms of memory footprint.

The effort for porting sDDS to the presented platforms is summarised in table I. The implementation time for Linux was not measured, because it is the development platform for new functionality. Neither were implementation time data for BitCloud and zStack taken, because they were early target systems. The SSAL part for both platforms does not contain the task and memory module, since they were not used. The Linux network is relatively big, because it contains the implementation for IPv4, IPv6 and IPv6-multicast, which can be selected by preprocessor macros.

platform	time	LoC network module	LoC memory logging	LoC task module
Linux	—	525	86	95
BitCloud	—	145	—	—
zStack	—	241	25	—
Contiki	v2.6 14 days v3.0 4 days	249	92	113
RIOT-OS	14 days	300	87	55
FreeRTOS	3 days	148	87	25

TABLE I
PORTING EFFORT FOR SDDS

To measure and compare the memory footprint of sDDS on the different target platforms the following six small sDDS applications were implemented utilising different subsets of DDS functionality. The data types used for the topics have a size of 8 bytes. The following example applications are using different subsets of DDS functionality implemented by sDDS:

1) simple writer with one topic (static configuration)
2) simple reader with one topic (static configuration)
3) simple reader and writer with one topic each (static configuration)
4) dynamic reader with two topics
5) dynamic writer with two topics
6) dynamic reader and writer with two topics each

The first three example applications have a static communication configuration and no support for discovery is included. The last three have Built-in-Topic support, multicast and a dynamic discovery module. While the first two applications are limited to either sending or receiving data and, therefore, include only the respective functionality, the dynamic versions need both features for the Built-in-Topics and the discovery. The Contiki hardware target was the Atmega128RFA1 and the ESP8266 was used for the FreeRTOS example applications. For RIOT the native Linux environment was used. The results for the flash memory footprint are displayed in table II. The program size increases with the added DDS functionality for the sDDS part, while the platform-specific fraction is constant. The aggregated size is reasonable for small embedded devices and the provided functionality. However, the footprint for the operation system must be added.

VI. CONCLUSION AND OUTLOOK

In this paper, we presented sDDS, an implementation of an operational subset of the DDS standard adapted to small distributed embedded systems as used in WSN or IoT scenarios. The porting process to different target platforms was described. As part of an MDSD process individual tailored sDDS implementations are generated, and a subset of the DDS functionality is selected according to the resource limitations of the target platforms, while retaining API conformance with the DDS standard. The architecture of sDDS emphasizes portability to new platforms. As a result, sDDS, while still under development, was already ported to a wide range of

example platform	1	2	3	4	5	6
Linux						
App	0.9	0.8	1.6	1.5	1.5	3.0
sDDS	8.8	9.0	9.4	15.8	15.9	16.3
SSAL	6.9	6.9	6.9	6.9	6.9	6.9
Contiki						
App	0.7	0.7	1.4	1.3	1.3	2.5
sDDS	5.9	6.0	6.5	11.8	12.0	12.4
SSAL	3.0	3.0	3.0	3.0	3.0	3.0
RIOT-OS						
App	1.5	1.4	2.8	2.5	2.6	5.3
sDDS	15.4	16.5	17.4	28	28.1	28.6
SSAL	6.4	6.4	6.4	6.4	6.4	6.4
FreeRTOS						
App	0.8	0.8	1.5	1.4	1.4	2.7
sDDS	5.9	6.1	6.7	12.0	12.0	12.4
SSAL	2.0	2.2	2.2	2.0	2.0	2.0

TABLE II
FLASH MEMORY FOOTPRINT IN KB FOR EXAMPLE SDDS APPLICATION

different platforms. These platforms were presented and the porting process and effort were described. As conclusion of the implementation up to now, it becomes apparent that familiarity with the target platform is the main factor for the amount of implementation time needed. Especially the build systems differ between platforms and can be quite complex, as was to be expected. With the current set of supported DDS functionality, sDDS does not have complex requirements regarding platform-specific parts. The support of real-time capabilities will probably need more complex interaction with the underlying system software. The DDS Standard only specifies possible QoS policies; the accuracy and quality depends on the middleware implementation. For sDDS, this means that the real-time support depends on the underlying transport layer and its connection to sDDS, the internal real-time aware data organisation and processing, and the intelligent application of the MDSD process, where capabilities and requirements can be joined, load and communication models specified or derived, and, as a result, optimised program code generated.

Currently, DDS functionality supported by sDDS is limited to data exchange and static and dynamic communication relationships. So far, has been focus was on proof-of-concept and basic data exchange. Nevertheless, it was already possible to use sDDS in an AAL/Smart Home research project successfully.

To improve the applicability for this problem domain, the next steps will focus on QoS support for reliable and redundant communication, real-time aspects, the application of the DDS standard extension for security on WSN and a generic gateway to commercial DDS implementations. These new features are necessary to allow an fair comparison with other DDS implementations and middleware approaches for WSNs. For a thorough evaluation regarding performance, real-time capabilities and scalability, the RIOT-platform is currently envisaged, because a real testbed, namely the DES-Testbed of FU Berlin [18] supports this platform and is accessible. Such a test environment can produce realistic and comparable results.

There is the chance that DDS standard itself might become relevant in the industrial automation domain. A testbed for micogrid applications of the Industrial Internet Consortium (IIC) [19] uses DDS as communications middleware. For monitoring equipment and machinery sDDS could be used to bridge sensors and industrial applications.

REFERENCES

[1] S. Li, L. D. Xu, and S. Zhao, "The internet of things: a survey," *Information Systems Frontiers*, vol. 17, no. 2, pp. 243–259, 2015.

[2] OMG, "Data Distribution Service for Real-time Systems," 2015.

[3] K. Beckmann and M. Thoss, "A Model-Driven Software Development Approach Using OMG DDS for Wireless Sensor Networks Software Technologies for Embedded and Ubiquitous Systems," in *Software Technologies for Embedded and Ubiquitous Systems*, ser. Lecture Notes in Computer Science, S. L. Min, R. Pettit, P. Puschner, and T. Ungerer, Eds. Springer Berlin / Heidelberg, 2011, vol. 6399, ch. 11, pp. 95–106.

[4] ——, "A Wireless Sensor Network Protocol for the OMG Data Distribution Service," in *Proceedings of the 10th Workshop on Intelligent Solutions in Embedded Systems (WISES'12)*, Klagenfurt, Austria, 2012, pp. 45 – 50.

[5] OMG, "The Real-time Publish-Subscribe Protocol (RTPS) DDS Interoperability Wire Protocol Specification," 2014.

[6] P. Boonma and J. Suzuki, *TinyDDS: An Interoperable and Configurable Publish / Subscribe Middleware for Wireless Sensor Networks.* IGI Global, Jun. 2010, ch. 9, pp. 206–231.

[7] Rti connext dds micro. [Online]. Available: https://www.rti.com/docs/RTI_Micro.pdf

[8] A. González, W. Mata, L. Villaseñor, R. Aquino, J. Simo, M. Chávez, and A. Crespo, "DDS: A Middleware for Real-time Wireless Embedded Systems," *Journal of Intelligent & Robotic Systems*, vol. 64, no. 3, pp. 489–503, Dec. 2011.

[9] A fully compliant zigbee 2012 solution: Z-stack. [Online]. Available: http://www.ti.com/tool/z-stack

[10] Bitcloud - zigbee pro. [Online]. Available: http://www.atmel.com/tools/BITCLOUD-ZIGBEEPRO.aspx

[11] Contiki: The open source os for the internet of things. [Online]. Available: http://www.contiki-os.org/

[12] A. Dunkels, O. Schmidt, T. Voigt, and M. Ali, "Protothreads: Simplifying event-driven programming of memory-constrained embedded systems," in *Proceedings of the 4th International Conference on Embedded Networked Sensor Systems*, ser. SenSys '06. New York, NY, USA: ACM, 2006, pp. 29–42.

[13] E. Baccelli, O. Hahm, M. Günes, M. Wählisch, and T. C. Schmidt, "RIOT OS: Towards an OS for the Internet of Things," in *Proceedings of the 32nd IEEE International Conference on Computer Communications (INFOCOM)*. IEEE, Apr. 2013, pp. 2453–2454.

[14] Freertos. [Online]. Available: http://www.freertos.org/

[15] lwip - a lightweight tcp/ip stack. [Online]. Available: http://savannah.nongnu.org/projects/lwip/

[16] Open source freertos-based esp8266 software framework. [Online]. Available: https://github.com/SuperHouse/esp-open-rtos

[17] R. Kröger, W. Lux, U. Schaarschmidt, J. Schäfer, M. Thoss, and O. von Fragstein, "The WieDAS AAL Platform: Architecture and Evaluation," in *Wohnen - Pflege - Teilhabe. 7. Deutscher AAL-Kongress mit Ausstellung, 21.-22. Januar 2014, Berlin*. VDE Verlag GmbH, Berlin/Offenbach, January 2014.

[18] M. Günes, F. Juraschek, B. Blywis, Q. Mushtaq, and J. Schiller, "A Testbed for Next Generation Wireless Network Research," *PIK - Praxis der Informationsverarbeitung und Kommunikation*, vol. 32, no. 4, pp. 208–212, 2009.

[19] COMMUNICATION & CONTROL TESTBED FOR MICROGRID APPLICATIONS. [Online]. Available: http://www.industrialinternetconsortium.org/microgrid.htm