Using Web Service Gateways and Code Generation for Sustainable IoT System Development

Till Riedel*, Nicolaie Fantana**, Adrian Genaid*, Dimitar Yordanov*, Hedda Schmidtke*, Michael Beigl*,
*TecO, Karlsruhe Institute of Technology **ABB AG Corporate Research Germany

Abstract—Wireless Sensing and Radio Identification systems have undergone many innovations during the past years. This has led to short product lifetimes for both software and hardware compared to classical industries. However, especially industries dealing with long-term support of products, e.g. of industrial machinery, and product lifetime of 40+ years may especially profit from an Internet of Things. Motivated by a practical industrial servicing use case this paper shows how we hope to make equally sustainable IoT solutions by employing a model driven software development approach based on code generation for multi-protocol web service gateways.

I. INTRODUCTION

Information sources from an Internet of Things (IoT) are especially valuable for industrial field service [1] and maintenance engineers [2]: new IoT technology can help to complete maintenance tasks, react more precisely to faults and thus extend the lifetime of machines. We already described potentials and challenges of using IoT systems in industrial field service in [1]. Service engineers are experts in connecting information about the product life cycle and its current state using data sources ranging from knowledge bases and Internet resources to machine diagnostic information and in-situ collected measurements – not to forget the sensory inputs and knowledge of the engineer himself. In an engineer’s toolbox wireless sensing and RFID systems complement traditional measuring equipment and industrial IT. However, those new tools prove only powerful if they integrate with existing tools and work flows. This imposes a problem in industries where devices have a lifetime of often more than 40 years, and old tools may especially valuable if paired with years of experience.

Within the Aletheia project (www.aletheia-projekt.de) we started looking at an integrated approach making information from the IoT available as federated product and process information sources. Also motivated by the economical risk to bet on one IoT technology, we focused on a platform independent solution. Typical high level IoT standards like Application-Level Events (ALE) were not intended to capture low-level semantics that are often important for industrial measurements. Existing low-level wireless networking and sensing standards did not match our effort to support diverse applications and setups (like WirelessHART) or targeted only lower network levels of communication (like 802.15.4). Practically we are facing a vast number of application specific protocols, runtime systems and APIs. This is a natural development for an area where the most powerful systems operate at limits in terms of cost and run on very constrained resources and optimized miniaturized hardware. Some of those specialized technologies already face widespread implementation in industry. In a few years Generation 1 RFID tags, e.g. used in name plates, may prove to be the Philips screw of the IoT: although better drive types (i.e. interfaces) may exist it will be wise to integrate the fitting driver into your tool box.

Our challenge is to provide a tool box that include such technologies while reducing its overall (future) weight in both figurative and practical sense. In this paper we report on applying state of the art theoretical findings as well as best practices from other areas into a novel system, which we apply in an industrial servicing use case. The implemented system consists of two parts for which we present our design decisions and whose components we describe in this paper. A flexible run-time gateway architecture abstracts basic network services and translates them to a standardized interface based on the Device Profile for Web Services. The second part the design section directly provides the necessary tools to develop platform optimized services by means of code generation. We practically integrate a model driven software development framework to bring web service processing to any device: it is designed so as to enable even the smallest devices such as resource-constraint wireless sensor nodes or even passive RFID tags to implement document based Web Services. Based on the implementation of the industrial service use case we are able to show that clear high level abstractions and low level optimizations are desirable goals for a any IoT infrastructure that do not contradict each other.

II. MULTIPROTOCOL GATEWAY DESIGN

Numerous works have been presented that employ Web Services directly on embedded devices, e.g. recently [3],[4]. They are mostly relying on the valid argument that Web Services are needed at some point for coupling embedded systems with state of the art distributed architectures. In particular, the Device Profile for Web Services (DPWS) is targeted towards bringing Web Services to devices.

It provides a basic profile for Services that is practically motivated by the application on devices and successful integration of embedded systems, such as machine and control elements, into a service landscape using DPWS has been shown [5].

The general problem of using Web Services is that even lightweight implementations are too resource heavy for many IoT systems. That is why it makes sense to use optimized gateways to bridge enterprise and personal computers with the IoT. However, there is currently no satisfactory practical solution on how to develop a large number of different IoT gateways as a software product line with minimal adaption while preserving platform specific optimizations.
With the gateway depicted in figure 1 we present a piece of software and hardware that translates between multiple IoT systems with only small adaption. One objective of the gateway design was it can directly run on a number of embedded hardware systems, such as an RFID Reader, a Wireless Sensor Network Bridge, or any mobile device that has the physical interfaces needed to connect to local IoT devices. Many previous implementations of Web Service gateways either instantiate a proxy service on the gateway or leave it completely to the IoT Node to handle the Web Service information [6],[3]. However, there is a tradeoff between gateway scalability and node complexity.

Both solutions are non-optimal extremes of a complicated practical optimization problem. The proposed framework directly addresses this issue by allowing a developer to choose how much of the state needs to be handled in the gateway. In contrast to other systems, our system operates transformation driven, i.e. like a style sheet: it converts incoming information and passes it on. We thus exploit a design aspect of Web Services, namely that all necessary information is contained in the message. In order to reduce the load of the IoT system, the gateway may transform the messages in a target optimized representation. The gateway can further optionally handle a request in proxy by holding state on the gateway or leave it completely to the IoT Node to handle the Web Service information [6],[3]. However, there is a tradeoff between gateway scalability and node complexity.

In [7], we introduced the concepts of semantic and syntactic transformations for including smart items into ERP systems. The semantic translation matches different network semantics and translates them. To do this a set of network primitives need to be implemented for the platform. The syntactic translation of the message of the payload. Because the payload is application-specific and the concrete application is probably not known when building a gateway this part is handled by automatically generated code which is again interfaced by a few simple primitives. The according components of the gateway are depicted in figure 1. In the following section we describe the overall design and the basic component functionality. A concrete implementation for the technology mapping is described in the use case presented in section §III.

A. Semantic Transformations

The idea of the semantic transformation is to map certain parts of the rather verbose SOAP binding to the semantics of a concrete networking platform. For RFID technology, e.g., it translates the semantics of an RPC to a reader query or transfers a read to an event.

1) Lifecycle Management and Identification: The primary concern of the lifecycle management component is to manage the underlying IoT network that the gateway proxies. Depending on the concrete network this can be implemented by reader queries or network joining procedures depending on the network topologies. The WS-Addressing standard allows specifying addresses at different networking layers. This is exploited by giving each IoT device a unique logical address. This URN can embed other address spaces containing other types of unique ids, i.e. it may include a hash code of an Electronic Product Code for RFID or an 802.15.4 MAC address for a wireless sensor node. This way a node stays unique across multiple gateway instances and can physically move between gateways.

The life-cycle management component deals with announcing nodes and services running on the nodes. We decouple the Web Service Discovery from the IoT discovery, in order to lower the load on the IoT network, especially in the presence of multiple client applications, and to mitigate the effects of fluctuating networks like RFID on the client logic. The gateway can use static knowledge about devices, e.g. retrieved from the unique numbering scheme of a MAC address or EPC code, and services, which can be provided by the node.

2) Address Transformation and Routing: The definition of hosting relationship in DPWS metadata maps service instance to devices. We use endpoint addresses to store all necessary dispatch information needed to route a message from a WS client to an IoT node. The first part of the URL, the host and port number, ensures that a request to the node service reaches the correct gateway that handles the node. The rest of the endpoint address specifies the dispatch path of the message towards the IoT node. This allows us to transfer this part of the routing state towards the client, so that messages can be routed into the IoT completely stateless.

The routing component of the gateway is used to deliver IoT events by push delivery to Web Service clients using the WS-Eventing protocol. Eventing call backs and network-specific event addresses are stored in a local address lookup table that is queried when a message enters the gateway from the IoT. Furthermore specifying custom WS-Eventing filters exposes a way to translate arbitrary network-specific subscription patterns to the Web Service world.

B. Syntactic Transformations

When a message is transferred into an IoT network we support a second step of transformation that is rather important for optimizing the communication on the upper network layers. DPWS

The payload of a message is transformed from verbose XML to an optimized representation and back. In order to
minimize the overhead on the gateway, the structure of the message is retained and the transformation works on the syntactic structure of the message and the encoding of its elements. For recoding and optimizing the message, knowledge about the Web Service Message structure is necessary to reduce complexity. For each message, we dynamically load a recoding function that can map a Web Service message into an optimized message format.

As the number of recoding functions required would equal the number of Web Services times the number of devices existing in a system, implementation by hand is not feasible in truly heterogeneous IoT landscapes. We propose a Model Driven Software Development (MDSD) approach with code generation for the Web Service Description Language (WSDL) for development of encoders and decoders.

1) Model Driven Software Development: Supporting different source and target encodings as well as execution platforms and programming languages easily leads to redundant overhead of code and consequently less manageable code. Introducing a common intermediate representation for both the data format and the encoders and decoders addresses this problem. Considering the formality, consistency and tool support for data meta-models we decided to use the Essential Meta-Object Facility (EMOF) (ISO/IEC 19502:2005). To map WSDL or rather XML Schema to EMOF, we rely on existing work[8]. Although formality and tool support is a big advantage, EMOF is a superset of context-free grammars[9] and thus traditionally difficult to handle on devices with limited resources. Therefore we use only a subset of EMOF as our intermediate representation. This subset is also known as the class of regular nested word languages [10], which coincides with the structure of XML described by an XML Schema. This class of languages includes non-regular, context-free languages while being as robust as the class of regular word languages [10]: deterministic nested word automata have the same expressiveness as non-deterministic ones, they can further represent both linear as well hierarchical relations between language elements, and the class of nested word languages is closed under intersection and union. These properties make it particularly suitable for representing XML structures and for supporting modular design.

On this basis of those meta-models we have already sketched a model-driven work flow within the Eclipse Modeling Framework for developing message translation for IoT subsystems [11] that largely automates the process of developing new syntactic transformation for our gateway system as well as message binding for the IoT platform. This model-driven development process starts with a high-level WSDL description of all data to be communicated from which it automatically constructs an abstract model. This model is transformed into a target model, which in turn is used for code generation. Up to this point the development process is platform independent and needs no further intervention as long as standard WSDL is used (later we will describe possible transparent annotations to XML Schema that can be used to further optimize target encodings.) In contrast to many other Web Services frameworks like AXIS, gssoap or WCF, the code generation is open in terms of programming language, underlying network layers, and data representations. Figure 2 outlines the proposed model driven software development workflow according to [12] that we proposed to develop encoding translators.

For implementing communication of structured messages on an arbitrary channel a linear encoding is needed rather than an abstract representation. In our case an encoding can be accepted by a special class of pushdown automata called visibly pushdown automata (VPA). Automata are a good compromise between an abstract and formal representation of data formats and a low-level, imperative solution to describe a computer system. They can be easily constructed on the basis of the abstract grammar of the language they should accept and can be mapped efficiently to soft and hardware. Especially embedded designs are traditionally affine to automata because of their efficiency on non-parallel, pipeline-less MCU and their deterministic behavior. The meta-model and algorithms to build VPA are based on [10] and were implemented as EMOF meta models for our MDSD process. Visibly pushdown automata are the basis of our code generation and can be used to accept, generate, rewrite and query nested word languages such used in typed Web Services.

One of the downsides of restricting ourselves to such a simple automata model is that not all semantics of the EMOF message meta model can be captured by the VPA. However we do not lose this information for later code generation steps, as the original model is linked via references in all automata transitions. Traditionally this information can be used for syntactical runtime validation of messages in the gateway. More importantly information captured in the data model can be used to optimally encode message data. We e.g. exploit data type informatino (integer, timestamp, string, etc.) to select a value encoder/decoder. We also support arbitrary minimum and maximum values as well as precision information: a temperature value with a range from $-20$ to $80^\circ$C with a resolution of $0.1^\circ$C is automatically encoded as 10 bits. Code generation makes this scheme extensible for arbitrary XML datatypes.

While by design all encodings can be accepted and emitted by the same type of automata to interface those automata different API types can be suported. For example we can support event, document model and pull API based messaging processing. Voelter [13] shows how different software configurations can be efficiently be generated using an aspect oriented code generation. We adopt this approach for our purposes using the xPand template language to implement the code generator. All refinements are encoded as advices of the basic automata.

![Model driven message exchange scheme](image-url)
control flow templates. Different input and output advices can be added to the transition templates of the pushdown automata via a configuration file. This allows the on-demand creation of specific eclipse code generator plug-ins for any combination of model source and target platform. While code generation always requires a onetime effort per target, it eliminates the need for re-implementation for different message protocols. The advantages of removing such redundancies are in our experience manifold beyond saving routine work. Especially code maintenance can be done much better: possibilities for bugs are reduced and improvements propagate more quickly.

Compiler and framework support is offered via eclipse builders that can be customized directly as part of a plug-in work flow. Triggering code generation automatically generates static or dynamic link libraries for inclusion into the target system. We have currently created code generators that can produce stubs for the C, C# and Java programming languages on the Windows, Linux, Symbian, Particle[14] and Contiki[15] operating systems using DOMs both untyped or typed and typed and untyped SAX style events: the code is always optimized to fit the needs of the target system.

Compiler and framework support is offered via eclipse builders that can be customized directly as part of a plug-in work flow. Triggering code generation automatically generates static or dynamic link libraries for inclusion into the target system. We have currently created code generators that can produce stubs for the C, C# and Java programming languages on the Windows, Linux, Symbian, Particle[14] and Contiki[15] operating systems using DOMs both untyped or typed and typed and untyped SAX style events: the code is always optimized to fit the needs of the target system.

The gateway and the development tool chain were designed in a way that they can implement a number of different concrete systems and support a number of use cases. As a prototypical representative of ad-hoc IoT use cases we present an Internet of Things application from an ongoing project. In the following we show that a derived instance of the designed architecture functionally fully maps the requirements of this use case. This shows the practical relevance as well as the expressiveness of the model in terms of this application class. By implementing gateways for multiple IoT subsystems, we further demonstrate some practical implication of choosing this development approach. Besides reporting our experience with practically implementing and using the system as well as giving a qualitative evaluation of our work, we further experimentally show some important quantitative aspects of our system design based on the implementation of this use case.

III. MOBILE SERVICE ENGINEER USE CASE

Mobile service worker Internet of Things application When servicing equipment, historical and real-time information is a valuable good. Often the information gap between the information system and the reality or missing details from site prevents optimal reactions to equipment malfunction. Any effective maintenance strategy where condition or risk is considered relies upon the availability of data and knowledge[1]. In a modern industrial environment a lot of data is available in plant automation and maintenance systems, which are using fixed installed sensors and asset supervision systems. However, flexible on-site, ad hoc data collection by the individual service worker proves helpful for diagnostics, increases the information amount and supports better diagnostic or decisions. Permanently installed RFID tags as well as ad hoc sensor networks can add valuable information in this process. The ultimate value of all technology comes from the integration and intercommunication of all systems including the service engineer himself. Today, field service engineers often have limited information available on-site to perform service tasks. For instance information is needed to find the right equipment on-site, to analyze the application environment, to diagnose the root-cause of the problem, and to repair the faulty equipment. The envisioned Internet of Things system improves information availability dramatically: it provides the mobile service engineer with context-specific information via real-time transactions with the back-end information sources or a dedicated knowledge base has been send up front to the mobile device of the service engineer, e.g., together with the service ticket.

The design decisions in the previous section are all made on a very abstract level without any technology binding. The gateway and the development tool chain were designed in a way that they can implement a number of different concrete systems and support a number of use cases. As a prototypical representative of ad-hoc IoT use cases we present an Internet of Things application from an ongoing project. In the following we show that a derived instance of the designed architecture functionally fully maps the requirements of this use case. This shows the practical relevance as well as the expressiveness of the model in terms of this application class. By implementing gateways for multiple IoT subsystems, we further demonstrate some practical implication of choosing this development approach. Besides reporting our experience with practically implementing and using the system as well as giving a qualitative evaluation of our work, we further experimentally show some important quantitative aspects of our system design based on the implementation of this use case.

A. Integration

A mobile service platform serves as an information hub for the service engineer. The target platform is a portable personal device with display and input capabilities (running a Windows Operating System). Currently the gateways run on the same mobile device with hardware interfaces to interact with both sensors and identification tags directly attached. However, settings with the gateways either running on external readers, base stations or dedicated gateway devices are possible using the same components. We initially started off with a measurement application where we continuously integrated new RFID and sensor network interfaces. After only a few years, the interface code outweighed the functional code of the application, which supports standard diagnosis workflows. Using web services now allows discarding most of the platform specific parts. The software already works the .NET environment and nicely integrates any sensor node via the Windows Communication Foundation. Listing 1 demonstrates how comfortable the integration of IoT systems can be done in only 3 lines of code for anyone used to traditional sensor network communication. Besides automatically generated code this is the only interfacing necessary.
Practically we learned the hard way that this is only part of the story when integrating heterogeneous systems via Web Services. While adding a DPWS client to the C# application took us about 10 minutes, it took us a few months to understand the problems that sometimes occurred in the interaction between the gateway and the mobile device each using different Web-Service stacks – WS4D-gsoap and WCF. (This additionally supports our decision not to fully implement SOAP Web Services on each IoT device) Although widely used and quite matured both implementations did not support the standard completely, WS4D-gsoap did not understand client generated header parameters and the Microsoft WCF did not correctly translate a valid WSDL issued by the DPWS gateway; both without giving any explicit error. Debugging the running system also practically proved that the verbosity of XML did not help with messages quietly disappearing in transparent dispatching code. It seems especially problematic to us that different client behavior increased the gateway complexity and requires unnecessary resources on the server, which is an issue for practical scalability. Nonetheless full support of Web Service standards pays off when considering all possible network interactions within the use case: By supporting routable Web Services as well as multicast announcements we have successfully tested can support complex topologies where service engineers can interact with the system locally and remote support is done over the Internet.

B. Functional Expressiveness

While the expressiveness in terms of communication patterns and different modes of operations are guaranteed by the working DPWS interface it was crucial to validate that all functional aspects and work flows of our use case can be expressed by the system. This includes identification, placement, configuration, measurements, storage and analysis [1]. In order to do this we took an existing measurement application (figure 3) that implements the servicing process and ported it to the new interface. In various experimental settings we confirmed the completeness of the interface.

1) Identification: One of the primary concerns of industrial measurement is the unambiguous association of data to data sources may that be pre-deployed measurement units or RFID name plates. He needs to scan the landscape for existing IoT devices and to add to devices like sensor nodes, if he needs to gather certain physical parameter. WS-Discovery is a very simple standard that allows ad-hoc scanning for devices and services. It can further be extended to search for devices with custom criteria. Most importantly it works infrastructureless using multicast queries and announcements on peer to peer basis, which can be easily translated to RFID batch readings or discovery protocols in wireless sensor networks.

2) Placement: The DPWS device based metadata exchange clearly identifies model and device on the basis of common attributes. Because the WS-MetadataExchange is extensible, the Lifecycle management component can augment further information that is important to the use case. An example is location information, which is an important part of the identification data. The discovery protocol further allows extending the discovery process by using custom query expressions to select nodes.

3) Configuration: Once the service engineer has an overview of the number and types of the systems in place, he informs himself about the current state and the capabilities of all systems. In our case the functional capabilities and the interface of a node are encoded in WSDL as a service. The service interface contains all the static functional capabilities of a node, i.e. what information it can provide. Furthermore dynamic capabilities can be configured via RPC. Also more complicated configuration tasks that involve changing the measurement service interface by e.g. adding processing tasks that change the representation of data from the time to the frequency domain, can be realized using DPWS by announcing the new interface via the discovery service.

4) Measurement: The most important part of any sensing system is the acquisition of measurement data. Document based (in contrast to RPC style) Web Services provide a very fit tool for those needs. WS-Eventing is sufficient to handle most aspects of starting or selecting event sources (we additionally implement inter-measurement synchronization as an additional service). We use the WSDL/XMLSchema interface to describe various aspects of a measurement value like min and max values. Using annotation like precision we can further add information to measurement data. We also started to annotate the WSDL with ontologies like UnitML, which allow us to even better describe measurement data, other information is added as documentation into WSDL file.

5) Storage: Documenting the measurement and the process of measurement is as important as the measurement itself especially when talking about digital signal processing systems. Comtrade is a classical format for transient recording[16]. It specifies three files to be stored with each measurement the data set itself, a description and the configuration parameters. We emulate those separate data sets by serializing the communication data, the metadata and all configuration calls to a node into XML documents. Because all exchanged data is well-typed and refers to an interface or grammar description, this data can be reinterpreted at any point. No additional storage formats are needed. The same is true for the binary storage of sensing data on the nodes. Using document based WS and accordingly crafted interfaces, we can use the same data typing and encoding for local storage as for communication. For the RFID, we also use the same data model for on chip storage as for communication. The model driven transformation allows a different representation of the data based on the local needs.
C. Efficient Development

One of our design assumptions was that a model driven software development process helps us to make the development of a whole software product line [13] of gateways more efficiently. By implementing this design process for ourselves we are able to show that those software technical innovations can effectively be applied to the IoT domain. We implemented (and are continuing to implement) a number of gateways for both productive and experimental purposes:

- Particle (binary encoding, AwareCon MAC, DOM-based code, new interface for existing pPart node)
- 6LoWPAN (binary encoding, 6lowpan/802.15.4 MAC, event-based code, new Contiki Node)
- “Dummynode” (various encodings, IPv4, dom-based, PC-based test environment)

As well as three prototypes we are currently working on:

- RFID (application specific encoding, reader protocol, no code, RFID storage prototype)
- Microstrain (vendor specific encoding, serial AT, closed source system, prototype for lifting non-service nodes)
- Java Node (binary encoding, SAXReader implementation, IPv4, java sensor nodes or mobile devices)

For the Particle platform, which is a ultra-low power wireless sending system with a 8-bit PIC18f6720 MCU and ultra low power unbuffered 869MHz TR1001 OOK transceiver, we compared the generated platform stubs on a quantitative level with the existing hand optimized minimalistic ConCom API [14]. We could confirm our claim that using high level design abstraction we do not sacrifice performance. This is especially interesting because obeying models adds design constraints so that theoretically some optimal manually written code exists. In practice the fact that knowledge is better represented to the developer seems to outweigh this even on a mature system. We could confirm that the generated code added no overhead to the code size, both programs uses 40 kByte of the available 128 kByte of code memory. Concerning memory usage we could see a slight shift towards statically allocated memory in the program using the generated code, which 604 instead of 560 byte of RAM. At the same time the maximum amount memory allocated dynamically on the stack during runtime was reduced from 118 to 80 byte in comparison to the manually optimized code. However, we discovered that a unexpectedly high overhead was introduced when using bit-packed encodings because the PIC MCU only supports a 1 bit rotate operation for shifts. We ultimately discovered that a unexpectedly high overhead was introduced when using bit-packed encodings because the PIC MCU only supports a 1 bit rotate operation for shifts. We ultimately decided that the added computational effort, was compensated for by the savings in the over the air encoding and memory savings by the inline structure bits. However, this is typical point where it might be necessary where optimizing the system requires a change in the message encoding which is supported by the development process without any explicit effect on the application. In figure 4, we compared the size and transmission power needed to transfer need to a single packet. In a addition to the original semistructured and untyped ConCom format, we compared the generated the document produced by the generated encoder to a “naive” encoding as XML as well as packed XML documents using GZIP and the XML-aware XMll compressor. Furthermore we added a comparison to VTD-XML that provides support effective processing of received data for embedded systems.

Another proof of flexibility is our implementation for a 16 MIPS Jennic JN5139 SOC based sensor node that runs the Contiki Operating system using 6lowpan communication. The system varies from the Particle Sensor Nodes in various aspect besides using a 2.4 GHz 802.15.4 MAC layer. It has low-power 32bit OpenRISC MCU at its core that using big-endian byteorder and aligned memory access. In contrast to e.g. using packed structures the generated binary en-/decoder could automatically handle network to host byte-order conversions and alignment corrections. Secondly the Contiki is build around protothreads, which provide coroutine like behavior within the operating system. By adding yielding in the automata we execute the automata concurrently with the applications that consume the data. This allows us to even receive large structured messages with theoretically infinite size (i.e. streams) requiring only memory in the size of the largest simple data type contained in the message and a symbol stack that can be bounded for most message models. Writing the code generators we largely reused the generator java SAX-Events. Although the concrete programs look quite different due to different programming languages and data structures, similarities in the control flow are very significant. The code generator requires only about 150 lines of platform specific generator code.

D. Gateway performance

An important practical aspect of our work was that the system was able to scale with a growing number of deployed nodes and services in the IoT. Although we are happy with the general resource consumption in normal operation, rather than presenting very application specific results, we like to report some of the more critical finding. To illustrate this we present some experimental measurements taken from simulated IoT nodes with the gateway and the DPWS client on the same machine. One of the positive scaling features of a gateway based approach is that each gateway only needs to handle the network traffic of one IoT platform. The traffic behind a single network is bounded, the overall throughput actually decreases due to coordination overhead in the network. For our test we generated events with a constant rate 100ms from a varying number of node. Figure 5 shows a near constant throughput from 20 nodes onwards because the sensor network bandwidth was reached. We can see a slight increase on the WS-Eventing side as additional subscription data was exchanged. The system performed well up to 50 nodes after
this point the WS client was not able to handle the push HTTP push delivery dispatching fast enough (The traffic increase afterwards is caused by the connection cleanup). This shows a bottleneck in Web Service callback communication to the client. Performance increases can actually only be expected by optimizing the Web Service clients which was out of scope in our case. Yazar argues in [4] that RESTful Web Services may solve performance issues, however, in our experiments performance losses occurred due to slow TCP connection setup, that affects SOAP (using http) and RESTful Web Services equally.

The problem becomes even more apparent in Figure 6. Here we scaled up the number of clients. Because WS-Eventing uses unicast to each client the connection could not be pooled. We used only a single sensor node that transmitted events with 4kbyte/s, this output was transformed and multiplexed to the clients. The figure shows nicely that the resource usage on the sensor nodes does not scale with the number of clients. However, we can also see that only after 5 clients the event delivery rate on the WS-Eventing site stagnates at approximately 250kbyte/s. All performance measurements were dominated by TCP setup times and connection handling on the Web Service side. The actual transformation steps were below the measurement resolution at the rates that were achievable through both the sensor network and the event delivery. Nonetheless especially in the target scenario either only two or three clients consume fast events or the nodes are in low-duty cycle operation anyway for long term monitoring. We further hope to improve our results by optimizing the connection handling towards the WS client and by moving the gateway to separate hardware.

IV. DISCUSSION AND RELATED WORK

We have shown that using Web Service based interface descriptions paired with a model driven approach we can achieve a high flexibility at a low runtime overhead when designing message based communication within an Internet of Things. We could show that the system perfectly fits our use case and we think the approach is applicable to many other areas with similar requirements. However, there are various limitation to a general applicability of the approach. One principal concern we cannot address with our work is that by limiting our design space via meta-models, we can never get better performance than the best hand-optimized code or message encodings. This might be critical in rare cases, where resources are extremely critical. In those cases a different description (in the worst case executable code) is needed to describe the mapping to an abstract message format. The same is true if a standard compliant encoding is required, that cannot be represented as a visibly pushdown language. However, a look at practically used systems quickly shows that by far most commonly used encoding fall into the class of tree languages we are dealing with in our work.

Another expected long-term result from working strictly top down is that we are transferring large parts of the complexity formally captured in run-time middleware and protocol frameworks into code generation. This leads to a higher level of abstraction and simpler sources at design time, but can lead to a similar complexity at compile and run time. The system even encourages the use of diverse proprietary message encodings. However, we believe that much like with todays programming language compilers can generate code for different processors, coprocessor extensions and optimization goals, only few experts need to be aware of this low level architecture. Additionally all communication can be traced back to the abstract representation, that can be debugged with a common set of tools. We believe that the risk of designing IoT systems is rather that they cannot be understood on a global level. One of the consequences of such a top-down approach is that our current communication architecture only guarantees that a message can be understood if the abstract type is known. This is a closed world assumption and in parts contradicts the principle of self-descriptive messages as proclaimed by modern REST [17] architectures. XML and for XML binary encodings like EXI [18] allow a structural decoding at any point in the system. We purposefully require much of the intelligence of the system only at compile time while not ruling out classical self-descriptive message or just-in-time compilation of acceptors. With EXI we already mentioned a different approach on handling XML efficiently.

[19] recently has compared different binary encodings for use with Web Service on sensor nodes. This paper is not about the implementation of a single combination of encodings and platform interfaces. We rather designed a flexible framework that uses open technology and standards to support a novel development approach. Our explicit goal was to integrate as much as possible from the previous work and allow developers to adjust systems to their requirements. This work therefore builds upon earlier works that bridge sensor networks like uMiddle [20] or the CoBIs UPnP gateway architecture [7] that address network convergence by providing manual message mappings between wireless networked systems. Although both system also a different standard, UPnP, the main difference is that this work uses automatically generated translation that
only require the original web service description. Translation is not specified per message but per platform using model-to-text transformations in a model driven workflow.

A similar system was sketched in [21] based on Ada for heterogeneous military applications. In recent years, however, much progress has been made in terms of standardization and advances of model driven software development tools and techniques. This paper exploits those techniques with advances in the formal modeling and parsing of structured data formats and applies it to the IoT domain. It provides well-defined models and meta-models for describing both data-structure of messages and the execution structure for de-/encoders and builds a code generator toolchain on top that is usable in an industrial context.

V. CONCLUSION AND FUTURE WORK

Automatically generated DPWS gateways for sensor nodes presents a novel practical approach to handling the task of integrating multiple concurrent IoT systems. Model driven code generation techniques and highly efficient communication bindings to arbitrary wireless sensor network services via a Web Service interface can help us to design sustainable IoT systems. Based on our experiments with an industrial servicing use case we could show both optimization and better platform integration. Although our study only has exemplary nature, we think by showing a vertical integration we could underline the potential of applying state of the art in model driven development and formal language techniques to the domain of information exchange in an Internet of Things on a very practical basis.

We were surprised that even using a minimal web services and standard HTTP communication we ran into performance issue on the client side early on. This confirms earlier results where also that classical ERP systems have difficulties scaling with sensor network events under dynamic load conditions [7]. From a different perspective it just underlines how efficient communication can be handled inside IoT networks. We still believe that Web Service add to a real IoT because they enable flexible architecture and new interaction schemes. Especially if using conceptionally powerful SOAP-based standards it is important that the associated overhead is not off-loaded to the IoT devices. This is what we have achieved with our gateway approach. Our Web Service gateways scale well with local IoT networks. An open problem that our experiments expose how a gateway may handle a large number of arbitrary clients. Luckily the industrial service use case does not have this problem and thus we are confident that our approach will help us to integrate further IoT technology at a higher pace without losing manageability.

We are currently working on integrating new RFID reader and sensor network platforms using our gateway design. In the long run we hope to remove an often artificial technological classification of networked sensing systems and enable truly diverse Internet of Things landscapes. By adding IoT technology to the tool box of the industrial service engineer, we already see many innovative practical applications on the horizon. Abstracting from technology is an important step towards focus on functionality.

VI. ACKNOWLEDGMENT

This work was partially funded by the German Federal Ministry of Education and Research (BMBF) as part of the Aletheia project.

REFERENCES


