EPiC Series in Health Sciences

Volume 1, 2017, Pages 258–263

CAOS 2017. 17th Annual Meeting of the International Society for Computer Assisted Orthopaedic Surgery

Concepts for developing interoperable software frameworks implementing the new IEEE 11073 SDC standard family

Andreas Besting[1], Dominik Stegemann[1], Sebastian Bürgerr[1], Martin Kasparick[2], Benjamin Strahen[3], Frank Portheine[1]

[1] SurgiTAIX AG, 52134 Herzogenrath, Germany

[2] Institute of Applied Microelectronics and Computer Engineering, University of Rostock, Rostock, Germany

[3] Institute of Medical Engineering, RWTH Aachen University, Aachen, Germany

besting@surgitaix.com

Abstract

The long overdue IEEE 11073 Service-oriented Device Connectivity (SDC) standard proposals for networked and surgical devices provide vendor-independent interoperability and therefore room for improved workflow and new functionality in the operating room. Research and development in this domain remain also highly topical in orthopaedic surgery. Due to the novelty and complexity of the SDC standard family, there is currently a lack of open source public implementations. Such implementations have to overcome several non-trivial challenges, mainly because the complexity of the standards has to be reflected in the software design and implementation. The SDC standard family comes in three different parts and all three standard proposals must be considered when designing and implementing standard conform device communication. In this work, we address these challenges and discuss and compare two design approaches for different programming languages (C++ and Java). Suitable software engineering principles are used to ensure a clean design approach. Practical guidelines are given on how to integrate existing third party components and tools in the framework and the development process, respectively. General feasibility is demonstrated by outlining interoperability between two software frameworks developed using different design concepts.

K. Radermacher and F. Rodriguez Y Baena (eds.), CAOS 2017 (EPiC Series in Health Sciences, vol. 1), pp. 258–263

1 Introduction

Research and development regarding networked medical devices remain highly topical in orthopaedic surgery and other domains. Networking devices can simplify existing functionality and create new functionality, based on new possibilities available in a vendor independent system. The foundation for such a system has been laid with the new IEEE 11073 Service-oriented Device connectivity (SDC) standard family. This standard has long been overdue and is crucial to provide plug-and-play functionality for the devices present in the operating room [1], [2], [3].

The SDC standard family contains three different parts [4]. IEEE P11073-10207 is called domain information and service model or Basic Integrated Clinical Environment Protocol Specification (BICEPS). This part describes the message and data model types needed for the device communication, as well as some transport-agnostic rules. The Medical Devices Profile for Web Services (MDPWS) IEEE 11073-20702 standard is a transport-aware extension and constriction of the OASIS Devices Profile for Web Services (DPWS) [5]. The combination of the two standards is described by the IEEE P11073-20701 standard, designated as architecture and protocol binding.

All three standard proposals must be considered when designing and implementing standard conform device communication and new functionality resulting from the devices synergies. However, the standards are inherently complex. This complexity is indeed needed to support the high variety of different medical device types and allow the modelling of complex behavior. The software design and implementation reflects the complexity of the underlying standards. Consequently, it is very difficult to build a reliable and easy-to-use software system.

In this work, we address the problem of designing and implementing standard-conform communication frameworks, by outlining key strategies for arising issues related to semantic- and architectural-related complexity. We present two design approaches resulting in two different programming language implementations and discuss advantages and drawbacks. In the first approach developed in C++, we used almost no external components in favor of creating a clean and compact design approach. In a second approach, developed in Java, we made generous use of available components to quickly generate working results while accepting possible risks of adapting to an unknown software and changing it in order to fulfil all needed requirements. A more detailed description of the first approach is available in [6]. Here, we present a summarized view and focus on the comparison with the second approach.

2 Materials and Methods

In our approach for building a medical software framework, we followed the typical software development process beginning with the gathering of requirements and designing the framework architecture and Application Programming Interface (API). A framework is a component-based, reusable software package that provides general functionality, and can be extended towards a specialized application. One framework has been built from scratch for the most part, while the other implementation is based in large parts on an initial fork of an open source third party DPWS implementation named Java Multi Edition DPWS Stack (JMEDS) [7]. The fork has been reduced to pure DPWS functionality and extended to partly support MDPWS.

For the sake of simplicity, this section describes only the constructive part, leaving the process of requirements gathering aside (a more detailed overview is given in [6]). Most of the functional requirements are defined by the SDC standard documents. SDC-Family conform communication is based on XML messages composed of the Web Service Description Language (WSDL) or SOAP [8]. The main part of a SOAP-message contains instances of the BICEPS message-model and the BICEPS data model. The raw content of XML is sent via HTTP and UDP (see Fig. 1). SOAP and WSDL are

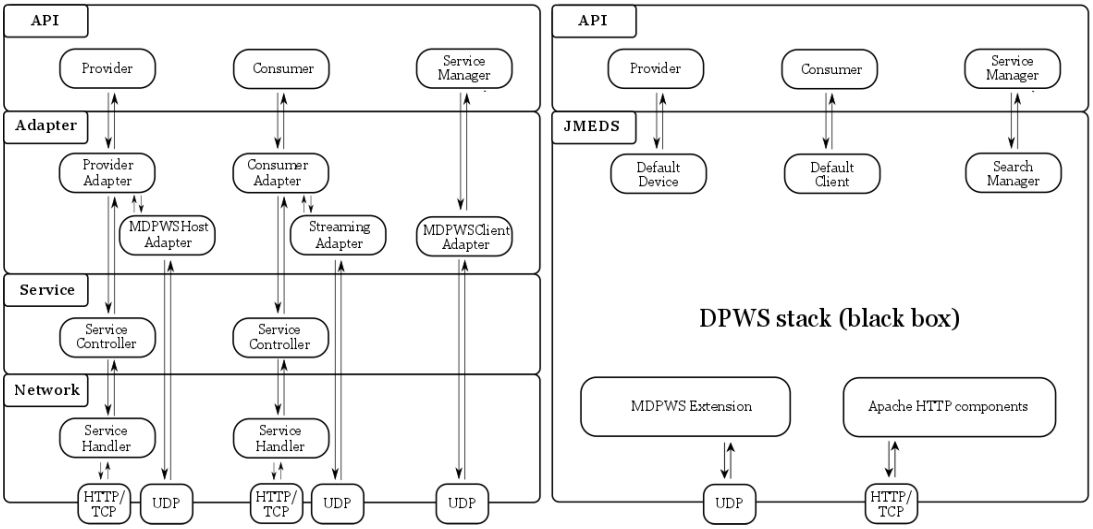

Figure 1: Architectural layers for C++ (left) and Java (right)

technologies reference by DPWS which is a collection of other (non-medical) web-service standards (WS-*). These standards are further extended by MDPWS by defining methods for safe data transmission, compact data transmission and streaming [9].

Figure 1 shows architectural layers for both frameworks, starting with the API layer at the top. The underlying mechanisms remain hidden for the calling component. We designed APIs that follow the Service-oriented architecture communication scenario [8]. We used the publish-subscribe pattern required by the IEEE 11073 standard [4] and expose a Provider class, that is publishing data (e.g. a heart rate value) and a Consumer class (e.g. a monitor that is displaying the heart rate) that registers for value-change events. These events are fired either each time their value is changed by pushing a new value from the application domain to the provider or in periodic time spans. In the top-level API design, we employed the open-close principle, which leaves modules open for extensions but closed for modifications [12]. All classes have certain registration methods for different handlers to interact with the provider or consumer in a very isolated manner. An example is a handler which can be registered at the consumer to observe a certain value. Hence this method is similar to the observer pattern [13]. Another example is a provider handler that can be used to react to value change request sent by a consumer and to update a current value as a reaction to state changes of the physical device. We strictly avoided sharing data between the application domain and the API layer in our design, because asynchronous processing of data creates multi-threading issues.

Below the API layer the two designs are different. In the C++ framework, we designed our own architecture and use custom DPWS and MDPWS functionality, in the Java framework we rely on a read-to-use DPWS component JMEDS 2.0 that we regard as a black box. Since the JMEDS API was also designed following the SOA principals, we were able to map our API components to the JMEDS components (DefaultDevice, DefaultClient and SearchManager). We could also implement MDPWS streaming using certain extension points. For this, we had to extend JMEDS WSDL generation, DPWS metadata exchange and had to send SOAP messages for streaming over UDP sockets.

In the C++ framework, the adapter layer connects the API layer with the service layer. The instances ProviderAdapter and ConsumerAdapter have a role similar to mechanisms described in the mediator pattern [14] as they handle the message routing between the API and service layer. Some mandatory and optional BICEPS services are managed by the provider adapter. In the consumer adapter, messages are stored in an event queue until they are processed by the application domain handlers one after another. The latter is called the reactor pattern [13]. Furthermore, general

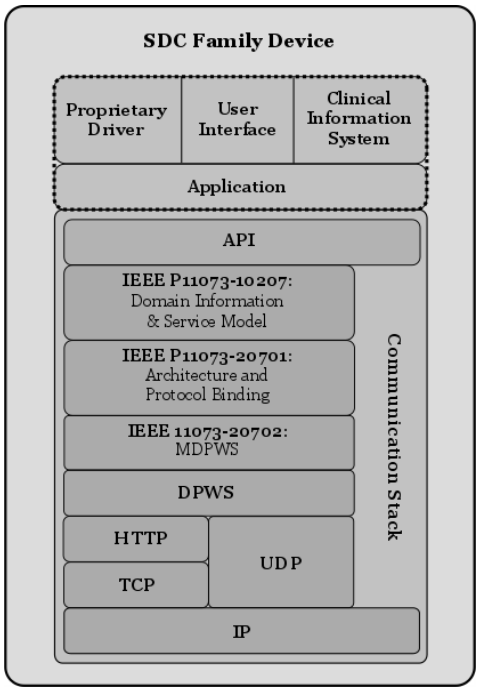

Figure 2: Hierarchical structure of the SDC Family-Connector. The dotted box marks the application domain. The application code can be specialized (top).

(M)DPWS functionality is handled. The service layer connects the service implementation to a specific HTTP request handler. The controller is mainly responsible for providing specific static content (WSDL and XML Schema Definition (XSD) documents) upon request. The last layer (network) contains the handlers which are tightly connected to a HTTP server. A handler is responsible for unpacking HTTP requests and transforming them into SOAP requests. These requests are forwarded to the corresponding service implementations. A similar mechanism is used in the other case. In the Java framework, we replaced the TCP socket management and HTTP server and client implementation in JMEDS by the Apache HTTP components libraries [10] to make use of more efficient implementations. In the C++ case, we used POCO libraries [11], mainly because of the implemented HTTP functionality.

An important aspect of our implementation is the utilization of automatic code generators for integrating new XSD-schemas. This approach eases the development process to a great deal since the adaptation of the current draft is speeded up outstandingly compared to manually maintained source code. In the C++ framework, we used the CodeSynthesis XSD 4.0 compiler for binding XSD-schemas of the SDC Family to C++ classes. The generated code includes serialization and parsing methods based upon the underlying Xerces 3.1 XML-parser. The XSD compiler was also used for generating consistent code for the DPWS and WSDL implementation. In the Java framework, the Java Architecture for XML Binding (JAXB 2.2.8) was used for the generation of Java classes out of XSD-schemas.

In the frameworks we used C++ templates and Java generics respectively to deal with the high quantity of types defined by the SDC Family. This allowed us to address the issues of type-safety and a high amount of redundant source code which would naturally occur in such situations. The high amount of different types generated by the XSD compiler makes it difficult to deal with these types in the architectural layers at runtime. Especially the use of traits [15] in the C++ framework proofed to

be helpful. We used traits to for all SOAP-operations and SOAP-events and tried to bundle all relevant information in one place. This technique is known as the single responsibility principle [16].

3 Results

The C++ framework is called Open Surgical Communication Library (OSCLib), the Java implementation Software for the Integrated Clinical Environment (SoftICE) [17]. Both frameworks implement a conceptually similar API which can be used to translate between the standardized and vendor-independent SDC Family and the proprietary, vendor-dependent protocols, which can be named a connector (see Fig. 2). Such a connector might alternatively be used to interact with a user interface or to report the device's data to the clinical information system.

Besides showing feasibility for different programming languages, external dependencies and tools used for code generation we could demonstrate that both frameworks are able to interact correctly. For each functional case (discovery, device description, getting and setting values, event subscription and streaming) we implemented separate test cases. Our frameworks were also used in several experimental demonstrator environments (e.g. in the OR.NET demonstrator on the conhIT 2016 and conhIT 2017). We could further demonstrate interoperability with the Device and System Connectivity libraries (openSDC) [18].

4 Discussion

Our approaches proof, that the implementation of the 11073 SDC Family drafts in C++ and Java programming language is feasible. While both design approaches led to successful implementations, the first one left us more room to apply certain software engineering patterns. These play an important role in the architectural design, which is particularly important when implementing a large quantity of the SDC Family, which can be regarded as a complex cross-layer protocol (see Fig. 2). Functions which are invoked from the application domain must be transported through the different layers until a message can be serialized and sent over the network. The methods involved are mostly asynchronous, which produces issues that can be solved by patterns.

In case of the Java framework, we relied on a full-featured DPWS stack that reduced our development time significantly. However, we had to apply several patches and bugfixes, since some extension points didn’t work as expected. Furthermore, we have limited knowledge and control of the internal mechanisms, which could be problematic in case of future extensions or changes. The code base of the Java framework is considerably larger because methods for generation and parsing of DPWS messages are manually written, without generics and are based on the Simple API for XML (SAX). In case of C++, we used code generators for all message types and template metaprogramming, resulting in a rather small code base.

The usage of code generators and generic programming in general helped extensively dealing with the fact that SDC Family not only involves layer-related complexity, but also type-related complexity. Since it has been designed to provide medical semantic interoperability, the quantity of various types needed to express this semantics in the data and message model is very high. Transforming XSD-schemas to compliable code speeds up the process of adapting new standard revisions in a less error prone way.

We figured out that the decision to include external dependencies has its own advantages and drawbacks. When building medical software, it should also be considered that such a dependency may have to be treated as Software of Unknown Provenance (SOUP) which requires special

documentation and testing for medical approval [19]. We see the latter as part of our future work, when we'll have finished implementing the SDC standard family in full.

References

[1] R. M. Satava, *The operating room of the future: observations and commentary*, Surgical Innovation, vol. 10, no. 3, pp. 99-105, 2003.

[2] D. W. Rattner and A. Park, *Advanced devices for the operating room of the future*, Surgical Innovation, vol. 10, no. 2, pp. 85-89, 2003.

[3] H. U. Lemke and M. W. Vannier, *The operating room and the need for an it infrastructure and standards*, International Journal of Computer Assisted Radiology and Surgery, vol. 1, no. 3, pp. 117-121, 2006.

[4] M. Kasparick, S. Schlichting, F. Golatowski, and D. Timmermann, *New IEEE 11073 standards for interoperable, networked point-of-care Medical Devices*, Engineering in Medicine and Biology Society (EMBC), 2015 37th Annual International Conference of the IEEE, Aug 2015, pp. 1721-1724.

[5] OASIS, *Devices profile for web services version 1.1*, 2016. [Online]. Available: http://docs.oasis-open.org/ws-dd/dpws/wsdd-dpws-1.1-spec.html

[6] A. Besting, S. Bürger, M. Kasparick, B. Strathen, and F. Portheine, *Software Design and Implementation Concepts for an Interoperable Medical Communication Framework*, submitted, 2017.

[7] *JMEDS (Java Multi Edition DPWS Stack)*, 2015. [Online]. Available: https://sourceforge.net/projects/ws4d-javame/. (Last accessed: Feb. 12th, 2017).

[8] I. Melzer, *Service-orientierte Architekturen mit Web Services: Konzepte - Standards - Praxis*, 4th ed. Springer Spektrum Akademischer Verlag, 2010.

[9] M. Kasparick, S. Schlichting, F. Golatowski, and D. Timmermann, *Medical DPWS: New IEEE 11073 standard for safe and interoperable medical device communication*, Standards for Communications and Networking (CSCN), 2015 IEEE Conference, Oct 2015, pp. 212-217.

[10] *Apache HttpComponents*, 2017. [Online]. Available: https://hc.apache.org. (Last accessed: Feb. 12th, 2017).

[11] *POCO C++ libraries*, 2017. [Online]. Available: https://pocoproject.org. (Last accessed: Feb. 12th, 2017).

[12] B. Meyer, *Object-oriented software construction*. Prentice hall New York, 1988, vol. 2.

[13] J. O. Coplien and D. C. Schmidt, *Pattern languages of program design.* ACM Press/Addison-Wesley Publishing Co., 1995.

[14] J. Vlissides, R. Helm, R. Johnson, and E. Gamma, *Design patterns: Elements of reusable object-oriented software*, Reading: Addison-Wesley, vol. 49, no. 120, p. 11, 1995.

[15] S. Meyers, *Effective C++: 55 specific ways to improve your programs and designs.* Pearson Education, 2005.

[16] R. C. Martin, *Principles of ood*, URL: http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod. (Last accessed: Feb. 12th, 2017), 1995.

[17] *Open surgical communication library and software for the integrated clinical environment*, 2017. [Online]. Available: http://www.osclib.surgitaix.com. (Last accessed: Feb. 12th, 2017).

[18] *Device and system connectivity libraries*, 2017. [Online]. Available: https://sourceforge.net/projects/opensdc/. (Last accessed: Feb. 12th, 2017).

[19] *Software of unknown provenance*, 2017. [Online]. Available: https://www.johner-institut.de/blog/tag/soup/. (Last accessed: Feb. 12th, 2017).