Annales Univ. Sci. Budapest., Sect. Comp. 52 (2021) 73–92

UML BASED MODELING AND CODE GENERATION OF NETWORK PROTOCOLS

Máté Cserép and Rudolf Szendrei

(Budapest, Hungary)

Communicated by András Benczúr

(Received May 2, 2021; accepted July 30, 2021)

Abstract. For a high-level description of communication protocols for client-server architectures, we present a modified version of the UML class diagram. This makes it easier to understand the protocol with the help of the diagram, and to automatically generate its code in a specific programming language. Using this method, programmers unfamiliar with network communication can easily and quickly implement a reliable client-server connection in a platform-independent manner. The procedure speeds up the creation of programs, reduces the number of possible errors, and helps programmers focus on the problem to be solved.

1. Introduction

While in the past a large part of Internet data traffic was characterized by stateless communication (e.g. web browsing), a significant progress has been made in recent years to spread stateful communication. The method we present can be applied in both cases. There is plenty of technology background (IDEs, APIs, libraries, methods) available to implement high level stateless communication, but there is much less for the stateful approach.

Key words and phrases: UML, class diagram, communication protocol, code understanding, client-server connection, code generation, websocket.
2010 Mathematics Subject Classification: 68N19, 68M12, 68U35.

Reviewing the current tools, we found that although methods for designing communication protocols with different high-level [3, 19, 21] and low-level [29] descriptions are given, using different diagrams, the implementation should be done at the discretion of the programmer. There are also methods that allow the automatic generation of a code skeleton after the formal description of the given protocol, although in these cases the descriptions mostly involve the creation and editing of text files [8]. However, text files do not provide a visual overview and better understanding of communication.

In this paper, we present a method that is suitable for modeling both stateful and stateless network connections by extending the UML class diagram. However, in addition to visual modeling of the network connection, we also provide an answer to how such a model can be used to automatically generate code skeletons for endpoints for different programming languages.

In Chapter 2, we provide an overview of the state of the art approaches and tools to demonstrate the benefits of our method through them. In Chapter 3 we introduce our concept, which allows the visual design of stateless and stateful communication protocols, as well as the creation of appropriate code skeletons, even in different programming languages. To do this, we will modify the UML class diagram. In Chapter 4 we present a possible implementation for creating a diagram editor for the proposed diagram. In Chapter 5 we present how it is possible to generate a code skeleton from the diagram describing the protocol, demonstrating it through the selected Java and JavaScript programming languages for the server and client parts respectively. Finally, in Chapter 6 we discuss the future direction of our research; then in Chapter 7 we summarize the results achieved and propose the wide application of the benefits of theoretical and practical results.

2. Background and related work

Historically one of the most popular syntactic description model for web services is *Web Services Description Language* (WSDL) [4], which describes web services by separating the abstract functionality offered by a service from concrete details such as how and where that functionality is offered. It supports descriptions for both SOAP-based services and REST API services, and serves as the standard for the former. However it is rarely adopted for the latter as WSDL contains too much technical details, a complex system to learn and difficult to read and understand for humans.

Multiple solutions have been proposed to model web services defined in the WSDL web services, specifically as UML diagrams. De Castro et al. introduced

a UML language extension for WSDL [7], which enables not only the graphic representation of a web service, but also allows the automatic generation of WSDL code from a UML diagram. Based on their work Vara et al. constructed the *MIDAS-CASE* system [32] – a MDA (Model Driven Architecture) tool – which supports the modelling of web services in an extended UML format and the automatic generation of the respective web service descriptors in WSDL format. Another, but similar approach was presented by Skogan et al. [28], that create an extended form of UML activity diagrams from WSDL to ease the design web service compositions.

In their work Jiang and Systä proposed an approach [17] to utilize the UML model representation of WSDL web service descriptions to check the validity of these descriptions with respect to the WSDL structure, SOAP binding, and WS-I Basic profile rules.

The WSDL specification of a web service also enables both the server and the client side code generation, which is widely supported in many professional integrated development environments (IDEs). The automation is often two-way: the WSDL specification can also be extracted from the implemented server-side codebase – then can be used the client-side stubs. As an example, *Microsoft Visual Studio* provides both code generation from WSDL and WSDL extraction from code for the .NET programming language family (primarily C#) [30], while the same can be achieved for Java with *Eclipse* [26, 27]. Beside server and client side stub generation based on the WSDL specification, various research showed the process and algorithms of automatic test case generation based on it for distributed service testing [1, 2, 15, 31]. Most approaches focus on to describe or deduce an interface grammar – a formalism for expressing constraints on sequences of messages exchanged between two components. It can be utilized to generate both a parser, to check the correctness of the messages generated by a web service client; and sentence generator producing compliant message sequences, to verify that the web service responds to the messages according to the interface specification.

In the past years *remote procedure call* (RPC) frameworks like Apache Thrift[1] or Google's gRPC[2] have been introduced and gained significant user base in the software industry. These systems provide more human-readable and editable formats to define the data transfer types and service interfaces, typically supported by code stub generation in various programming languages. While these systems have already obtained integrated support from many popular IDEs, graphic visualization and construction of web services through UML or other type of diagrams is still a missing feature.

In regard of REST APIs, instead of WSDL, more human-readable and easier to use metadata formats have also been introduced more recently, along with

[1] `https://thrift.apache.org/`
[2] `https://grpc.io/`

editors to support developers in the creation of descriptions for REST APIs [8, 6]. Among others, popular description formats are the following:

- *Open API Specification* (OAS)[3] (previously also known as *Swagger specification*), which provides a standard, language-agnostic way of defining a REST API interface based on YAML and JSON.
- *RESTful API Modeling Language* (RAML)[4] is YAML-based language designed to support an API-First top-down development approach. It provides the format for the contract between the API provider and the API consumer.
- *API Blueprint*[5] is a document-oriented language for describing REST API using Markdown syntax. This specification uses Markdown syntax to provide a set of semantic assumptions laid on top of the Markdown syntax.

As a follow-up for WSDL, the *Web Application Description Language* (WADL) [14] was explicitly proposed for API services, which still provides a machine-readable XML format, but addresses the previously mentioned criticism. However while WADL was also proposed for standardisation, this still have not been executed, as there is an ongoing controversy whether something like this is needed at all [6]. A different approach has also been proposed to integrate SOAP services and RESTful services by wrapping the latter into SOAP services automatically. Such a solution is demonstrated by the REST2SOAP framework [22].

For all discussed formats a proper toolset is provided, that is the reason they managed to gather a considerable user-base in the first place. With these tools a developer usually can achieve the following.

- Manually specify the API before the application itself through a text editor with special syntax-highlight and intelligent code completion.
- Generated (either manually or automatically) the API from the an existing application.
- Generate a documentation of the API in various formats, in most cases static HTML web pages.
- Generate static code samples or even an interactive JavaScript client to test the API.

[3] `https://swagger.io/specification/`
[4] `https://raml.org/`
[5] `https://apiblueprint.org/`

- Parse the API specification and generate client stubs and server-side skeleton code in various supported languages that can be further developed.

Beside open-source tools (e.g. for OAS the Swagger product family[6]: Swagger Editor, Swagger UI and Swagger CodeGen), commercials tools like APIMatic[7] or REST United[8] are also available. Without proper UML support current tools still require a considerable effort to visualize and understand what the APIs offer [16].

In their work Ed-Douibi et al. [11] proposed a tool called *OpenAPItoUML*, which generates UML models from OpenAPI definitions, thus offering a better visualization of the data model and operations. The same authors have presented *WAPIML* in their later work [12] as an OpenAPI round-trip tool that leverages model-driven techniques to create, visualize, manage, and generate OpenAPI definitions using an OpenAPI DSL also expressed as a UML profile for a simpler integration with existing modeling tools. WAPIML has been implemented as a set of Eclipse plugins and is available open-source[9].

Koren and Klamma realized in their paper [18] that existing web-based solutions that generate interactive OpenAPI documentation with HTML and JavaScript are far from real-world practices of designers and end-users; instead they are focused on developing and testing the API. In their work they present an approach to overcome this gap, by using a model-driven methodology, the *Interaction Flow Modeling Language* (IFML) as intermediary model specification resulting in state-of-the-art responsive web user interfaces.

The research field to define an executable variant of UML diagrams has been active since the introduction of UML 2.0 [9, 5], and the automated validation and verification of APIs have also gained attention in recent years. Ed-Douibi et al. [10] proposed an approach and provided a proof-of-concept tool implemented as an Eclipse plugin to generate specification-based test cases for REST APIs. These test suites make sure that such APIs meet the requirements defined in their specifications. Sferruzza et al. combined automated code generation and verification and their approach automates both of them [25]. Their tool named *Safe Web Services Generator* (SWSG) is based on a meta-model that allows developers to define implementations of web services, starting from the corresponding high-level contract as expressed by a standard OpenAPI model. Consistency of models can be verified using an operational semantics so that code generated from these models is safe.

[6] `https://swagger.io/tools/`
[7] `https://www.apimatic.io/`
[8] `https://restunited.com/`
[9] `https://github.com/opendata-for-all/wapiml`

As REST APIs are not backed by a uniform standard or specification, but instead several specification formats have been proposed, conversion between them have also created a challange to address. Both professional services such as *APIMatic API Transformer*[10] and free, open-source solutions like the *API Spec Converter*[11] are available to overcome this obstacle.

While there are many approaches proposed to enrich the API description formats discussed so far with semantics, the manual work required to create descriptions, the lack of interoperability standards limited their adoption. Another approach proposed by the semantic web community was to define a global ontology to include model, definitions and descriptions in a coherent system that can be used to make discovery and automatic composition. The most popular proposals were *Ontology Web Language for Services* (OWL-S) [20] and *Web Service Modelling Ontology* (WSMO) [24]. Unfortunately, the expertise required to build and manage such descriptions resulted that nobody actually use them [6].

3. The concept of a high-level description of communication protocols

As reviewed in Chapter 2, current technologies and research are very extensive. However, it can also be seen that the emphasis in describing protocols is mainly on supporting text formats, with (UML) diagram visualization being a possible additional feature without playing a central role in the design and maintenance of the protocols. Our further observation is that nowadays they focus more on stateless communication, while the use of stateful communication can provide significant benefits in real-time applications. For this reason, we want to support both types of communication. Instead of textual descriptions, we consider a description created with a graphic designer, which we imagine in the form of a diagram, to be easier to see and maintain.

One of the main ideas of our article is to apply an object-oriented approach to the high-level description of communication. Based on this, the transmission and reception of data means the sending and receiving of message objects. We also assume that the participants in the communication implement a client-server architecture.

Due to the object-oriented approach, we chose the UML class diagram for this purpose. We will modify this diagram type according to the task later,

[10] `https://www.apimatic.io/transformer/`
[11] `https://lucybot-inc.github.io/api-spec-converter/`

but the benefits are already visible:

- The diagram of the communication protocol can be defined simply with drag and drop technique,
- We can get a visual overview of the whole protocol,
- We can more easily understand the way of communication, the structure of messages.

Before presenting the method for defining communication protocols, we make two important constraints.

The first assumption is that a message sent during communication can be interpreted as an object (we call these from now on simply messages). The second assumption is that our proposed method must adapt to the technologies currently in use. The significance of the latter will be important for code generation.

Considering current technologies, we have found that most communications are consisting of *remote procedure calls* (RPC) and *notifications* (one-way message sending), so we divided the messages into three groups: *request*, *response*, *event*. A request sent by a client can be interpreted as an asynchronous method call to which the server must respond with a response message. The event represents a notification that can be sent by the server (one-way messaging).

Because messages are objects, so it is possible to define their classes. The UML class diagram that defines the messages is modified as follows:

- Only association and inheritance are allowed.
- Multiple generalizations are prohibited (only "trees" can be created).
- Classes should not contain methods.
- The associations are represented with attributes.
- A string value can be assigned (as an annotation) to the class name as a key that will help identifying the message type in the communication.

Due to the above, an association with multiplicity is represented with a collection. Its dimension can be specified as a separate property, which makes it possible to represent arrays and matrices. We also make the following changes (see Figure 1):

- In accordance with the message types, the graph of the diagram consists of three trees, where the root vertices are: REQUEST, RESPONSE, EVENT.

- In the vertices at the first-level of these trees, the selector name must be defined, which is interpreted as a special attribute of the class, and it identifies the super class of the message object during the communication.
- In the vertices at the second and lower levels of the trees, a selector value can be optionally defined, which is related to the selector name in the first-level parent of the vertex. It represents the concrete type of the message object. The above mentioned name - value pair defines the super class of the message, and the concrete type of the message object. This information is used during message dispatching.
- In the case of request classes, an association pointing to a response type class must also be defined.

Although, the above mentioned message types can realize most communication scenarios, our method is free to extend in the future, if a special application would require it.

In most programming languages, the supported basic attribute types in the class diagram are: `null`, `boolean`, `integer`, `float`, `string`. By using these basic types, programmers can create custom types by defining classes, which they can also use to define their messages or even more complex types. Arrays are allowed for class fields. In order to support custom types, the following addition is required to the modified class diagram:

- Make a fourth tree with a root called PLAIN. Call this PLAIN tree for short. Aggregation and multiple inheritance are not allowed here either (see Figure 1).
- Classes are free to reference any class in the PLAIN tree to define attributes. An example of the resulting modified and simplified UML class diagram is shown in Figure 1.

In order to demonstrate the concept, a simple use case of a two player TicTacToe game is shown in Figure 1. A new game can be started by sending a `NewGame` request to the server by both players. Then, the server replies with a `NewGame` response to both players, telling which player is in turn. When a player tries to put its sign on the board, it sends a `PutSign` request to the server, which holds a Coordinate object, that describes the desired place. If the place is already occupied or the player is not in turn, the answer in the response is *refused*, otherwise *accepted*. If the answer was *accepted*, the server also notifies the other player about the changes by sending it an `OpponentPlacedASign` event, which holds a copy of the last received coordinate. For sake of simplicity, we do not discuss here other aspects of the implementation, e.g., visualization, model representation, logic etc.

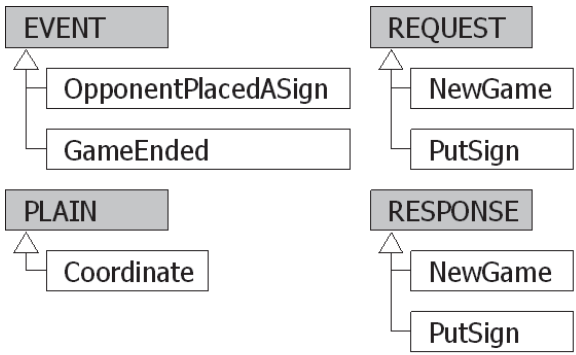

Figure 1. Defining messages of a TicTacToe game using a UML class diagram.

In our implementation we used JSON as the message serialization format, as it is human readable and programming language independent. It is an extremely popular data interchange format among web services, and most modern programming languages provide parsing and writing JSON-format data capability as a standard feature, therefore it simplifies the implementation of our proposed protocol in general.

4. Using the modified UML class diagram

For our modified class diagram, we propose a UML editing tool that enforces the constraints discussed earlier and provides the extra functionality. An additional need is to have a better understanding of the hierarchy of message types represented by classes and to be able to edit diagrams in an intuitive way. We see the reason for this in the fact that there are now a lot of programmers dealing with e.g. web technologies, including network communication, although it is not necessarily their field of expertise. For them, the right tool can help to understand the processes going on in the background.

Based on the above idea, we find such a design interface useful in which classes can be organized as shown in Figure 1. Based on the tree hierarchy constraint, the arrangement of classes in a diagram is simple and does not require complex algorithms. As a possible editing interface for each property of the classes, we recommend the layout shown in Figure 2. Due to the specifics of the task, we consider it expedient to omit the possibilities of the original UML class diagram type that we do not use, such as set the visibility of an attribute or method, and so on.

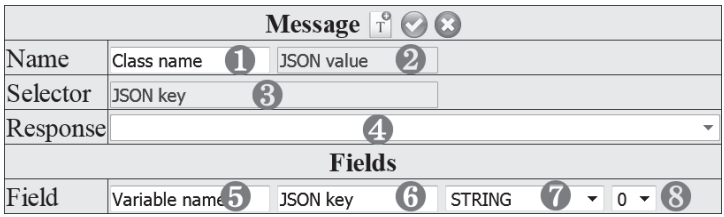

Figure 2. An editing tool of the UML diagram designer for the class. 1) class name, 2) selector value for message delivery, 3) selector key to decide message type, 4) response type for request type message, 5-8) field variable name, JSON name, type, and its dimension in case of arrays

Most programming languages have strict variable naming conventions, while JSON permits any arbitrary string to be an attribute name. To overcome the possible conversion difficulties, we allow to define both the variable name and the related JSON field name for each field. We also use the terminology JSON key for JSON field name, because JSON represents the attributes of objects as key-value pairs.

In Figure 2, Selector is a special field. The name of this field is given in the editor as a JSON key at the first-level of the message class tree. The value of the Selector field is determined in the subclasses by setting the JSON value property in the editor. This key-value pair effectively determines the endpoint and its concrete service on the server during the message dispatching. In practice, the Selector name is often equal to the name of the superclass, while the Selector value is equal to the name of the derived class.

We note that in our implementation the fields of objects are unordered, because we have found that it does not have a meaning in JSON and neither in popular programming languages.

5. Automatic code skeleton generation

By modifying the class diagram, it is possible to automate the production of both classes and program codes related to message sending and receiving. To prove this, we have created a diagram designer and an associated code generator as a web browser application. The two important elements of the diagram designer (creating classes and their hierarchies, and editing classes) are shown in Figures 1 and 2. The code generator is designed to be able to support different programming languages in the future. This is done by developers creating code templates and code libraries for the corresponding

programming language they want to support. We have done all of this for JavaScript and Java so far, which we will present in the next two chapters. For specific applicability, we chose the stateful connection based WebSocket technology for the client-server architecture. We have to note that our proposed method could be applied also to a stateless communications (e.g. REST API can be used instead of WebSocket).

If the generator needs to create interoperable codes in different languages, it is advisable for developers to create a kind of API, which is implemented in a consistent way in different languages (because the generated code skeletons are linked to the technology's own API). The function of the API is generally to perform network management activities, authorization, coordinate message processing, and serialize and deserialize messages.

The code skeletons created with the tool and the accompanying code library together are able to deliver messages between the client and the server in such a way that the developer's next task after creating the diagram is only to implement the body of methods generated to handle each message type.

The following two chapters describe in detail the principles of operation and the usage of the code generated for Java and JavaScript languages.

5.1. Automatically generated Java specific implementation

In Java, both client- and server-side implementations are conceivable, so the generator creates code skeletons for both sides (see Figure 3). The code skeleton consists of three classes:

- An authenticator, which decides whether the new client has permission to connect to the server. For decision making the API provides the username and the hashed value of the password. Developer can turn this feature completely off during the implementation, if authentication is not required.
- In the client class, the declared event processing methods must be implemented, and it must be possible also to create and send request objects. This task depends entirely on the specifics of the particular application.
- The request processing methods must be implemented in the server class. Each method handles a specific request type, for which it provides a response object that the programmer must "fill out". In the same class, the programmer should provide functionality for the user to create and send events.

The classes shown in Figure 3 can be divided into three groups, based on their purposes:

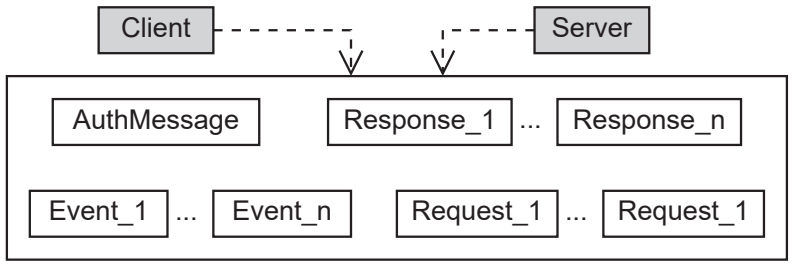

Figure 3. Classes created by the tool related to the Java endpoint.

- The Response, Request and Event classes are the Java classes equivalent of the message classes defined in the diagram designer. These classes contain only those variables that are associated with the annotations specified in the diagram. These annotations as JSON keys connect the variable with the corresponding field of a JSON object.
- Classes marked in grey are code skeletons that contain declarations for all methods to be implemented.

Listing 1 shows how a message class of the TicTacToe game (see Figure 1) is mapped to a Java class. The `JsonField` annotation lets define the relation between a variable and its corresponding JSON field, when the name of the JSON field is not a valid variable name in the programming language.

The template of the declarations of message processing methods located at the endpoint are shown in Listing 2. In practice, the generated code skeleton has all the necessary method declarations, so only the actions based on the received information need to be implemented regardless of the networking tasks.

```
class Request{
  @JsonField(name = "request-type")
  public String requestType;
  @JsonField(name = "message-id")
  public String messageId;
}

class PutSignRequest extends Request{
  public PutSignRequest(){
    requestType = "PutSign";
  }
  @JsonField(name = "x")
  public int x;
  @JsonField(name = "y")
  public int y;
}
```

Listing 1. Example: Automatically generated Java message classes for the PutSign functionality of the TicTacToe game.

```
// Message handler declarations
// at the client side
public void on_<EventName>(<EventName> e);

public void on_<ResponseName>(
    <RequestName> R,
    <ResponseName> r);

// Message handler declarations
// at the server side
public void on_<RequestName>(
    WebSocket w,
    <RequestName> R,
    <ResponseName> r);
```

Listing 2. Declarations of the message processing methods to be implemented.

5.2. Automatically generated JavaScript specific implementation

Since this article is intended to use WebSocket technology and the selected programming languages only to demonstrate the technology, the generator program generates code in JavaScript only for web browsers, although this would be entirely possible for the server side.

The method of generating the client-side code skeleton is the same as that discussed in the Java language. However, it is not necessary to create classes for messages because the language allows message objects to be created directly from a string, that represents a JSON object. The generated class structure is shown in Figure 4, where our API parts are marked in white and the class to be implemented is marked in grey.

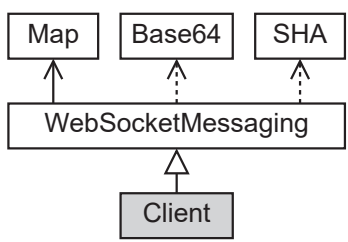

Figure 4. Classes provided by the JavaScript client generating tool.

5.3. The message routing method of the API

Our API provides several classes, that are responsible for message handling at the endpoints of the connection. These classes can be found on Figure 4-5. In grey for Java and JavaScript languages respectively. The main purpose of the API is to convert the arrived message into an object, and forward it to the corresponding user-implemented method. Also, the API converts the message object to JSON format and then forwards it over the network during messaging.

For reasons of expediency, the control of incoming messages is divided into three large groups (*request*, *response*, *event*), the processing of which differs significantly from each other.

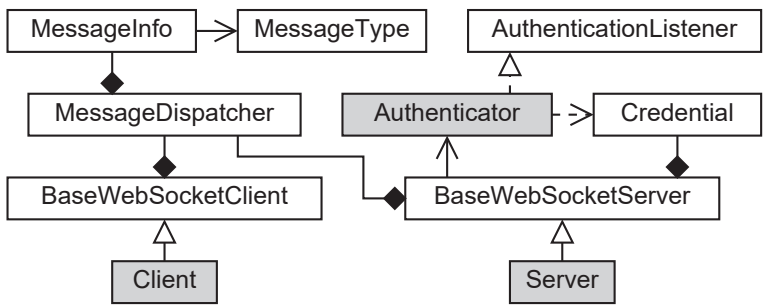

Figure 5. Java related parts of the API.

Sending and receiving of requests and responses (see Figure 6):

- The client provides each *request* with a unique identifier prior to transmission by assigning it to the JSON key named *message-id* of the message object.
- The client stores its sent message until it receives a reply for it.
- The server-side code identifies the type of request received based on the selector key-value pairs specified in the UML diagram. If the message contains a selector key specified in a node at the first level of the request tree, the value of the selector in this subtree clearly determines the type of request received.
- The server creates an "empty" object with the appropriate response type for the request specified in the diagram, and then passes it with the request to the processing method implemented by the user (see Listing 2).
- The request processing method uses a logical value to indicate execution success and, if successful, defines the properties of the response object.

- The server also places in its response the *message-id* key-value pair received in the request and the status of the response (successful / unsuccessful, possible error description) and then transmits it in JSON format to the client.

- The client checks to see if the received message contains a *message-id* key. If so, a previous request will be answered. It retrieves the request sent based on the key from its history and then infers the type of response based on the type of the request. The client creates a response object based on the information above and then passes it to the processing method along with the previous request.

It is important to emphasize that it is essential for the operation of the system that the *message-id* key can only occur in *request* and *response* type messages and should not be used in other contexts.

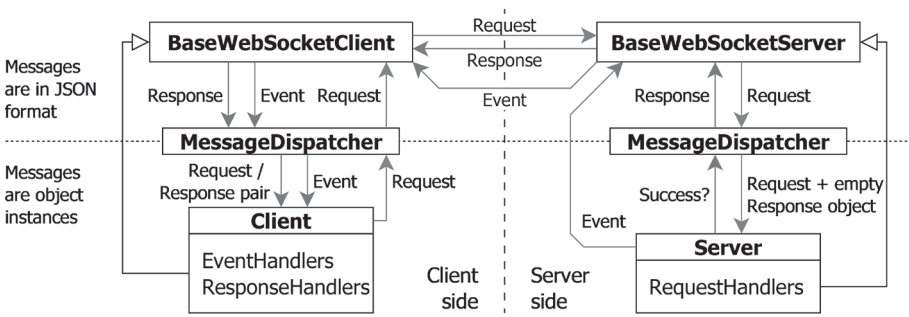

Figure 6. The process of transmitting and processing messages.

Sending and receiving of events (see Figure 6):

- The user creates an object of an event type and determines the value of its fields and then passes it to the server to forward to the client.

- The server converts the received event object to JSON format and then transmits it as text data to the client.

- The client checks that the received message does not contain the *message-id* key (otherwise it interprets the message as a response).

- The client identifies the type of event received based on the selector key-value pairs specified in the UML diagram. If the message contains a selector key specified in a node at the first level of the event tree, the value of the selector in this subtree clearly determines the type of event received.

- Knowing the type of event, the client generates the event object, which is passed to the user-implemented event handling method.

6. Future work

We aim to continue our research in two different directions in the future. Our first goal is to ensure implementation in cooperation with existing standards and systems. One of these would be, for example, support for gRPC and OpenAPI. We aim to do this in such a way that the visual design tool we create will be able to generate code according to an already existing standard from the created diagrams. In essence, we could replace the current text protocol description methods with a simpler, graphical approach, which would allow for faster review and development. The JSON data interchange format could also be replaced with the more state of the art ProtoBuf[12] format, which offers a superior performance compared to binary XML [13] or binary JSON [23] and already obtained remarkable programming language support.

Our second goal is related to the increasingly popular concept of smart cities and smart homes today. Currently, the available devices have a dual task of communicating with the user. On the one hand, embedded devices can easily implement binary data processing and transmission due to limited resources. At the same time, users can communicate in text with the central unit of devices when, for example, they need to access their smart home via a web interface. This raises the need to have a component of the system that performs the necessary conversion in both directions between the two data representations. Our idea is for developers to be able to create an API for their own concept. On the one hand, it is able to support the generation of code skeletons and communication codes from diagrams for both data representations. On the other hand, it is capable of conversion between two data representations. Based on the diagrams, we could then automatically create endpoints that hide the way the data sent and received is described, i.e. the format in which the device actually communicates with the other party.

[12] `https://developers.google.com/protocol-buffers`

7. Conclusion

For a high-level description of the communication protocols used in client-server architectures, a modified version of the UML class diagram has been introduced. This new version has a number of advantages over implementing protocols directly in a given programming language. First of all, it provides an opportunity to easily review and understand the messages of the protocol, as well as to define them in a language-independent way.

It has been shown in previous chapters that the concept allows the automatic generation of code skeletons for specific languages based on the protocol given in the diagram. By preparing the code generator program to support multiple programming languages, we can get a cross-platform solution.

Because network communication programming requires deeper knowledge and error detection is more difficult, the generated message delivery and processing codes reduce the amount of errors that a programmer can make. A further advantage is that on cross-platform systems, programmers have so far had to rhapsodically perform the generator-induced work over and over again for each platform separately. The method presented avoids that messages are defined differently in different programming languages if a well-designed API and code template store is provided.

It is worth noting that programmers unfamiliar with network communication can easily and quickly implement a reliable client-server connection with the tool. To do this, it is sufficient to define the types of messages and to "fill in" the message objects with the appropriate values in the methods declared to them.

Last but not least, the work done algorithmically by programmers so far has been automated. This not only effectively reduces the time it takes to build programs and the amount of potential errors, but also helps the programmer focus on the essence of the problem.

References

[1] **Bai, X., W. Dong, W.-T. Tsai and Y. Chen,** WSDL-based automatic test case generation for web services testing, in: *IEEE International Workshop on Service-Oriented System Engineering* (*SOSE'05*), IEEE, 2005, pp. 207–212.

[2] **Bartolini, C., A. Bertolino, E. Marchetti and A. Polini,** Towards automated WSDL-based testing of web services, in: *International Conference on Service-Oriented Computing*, Springer, 2008, pp. 524–529.

[3] **Bauer, B., J.P. Müller and J. Odell,** Agent UML: A formalism for specifying multiagent software systems, *International Journal of Software Engineering and Knowledge Engineering*, **11(3)** (2001), 207–230.

[4] **Christensen, E., F. Curbera, G. Meredith and S. Weerawarana,** Web services description language (WSDL) 1.1, tech. rep., World Wide Web Consortium (W3C), 2001.

[5] **Ciccozzi, F., I. Malavolta and B. Selic,** Execution of UML models: a systematic review of research and practice, *Software & Systems Modeling*, **18(3)** (2019), 2313–2360.

[6] **Cremaschi, M. and F. De Paoli,** Toward automatic semantic API descriptions to support services composition, in: *European Conference on Service-Oriented and Cloud Computing*, Springer, 2017, pp. 159–167.

[7] **de Castro, V., E. Marcos and B. Vela,** Representing WSDL with extended UML, *Revista Colombiana de Computación*, **5(1)** (2004), 1–15.

[8] **De, B.,** API documentation, in: *API Management: An Architect's Guide to Developing and Managing APIs for Your Organization*, Springer, 2017, pp. 59–80.

[9] **Dévai, G., T. Gregorics, B. Németh, B., B. Gregorics, D. Németh, G. Kovács, Z. Gera and A. Dobreff,** Novel architecture for executable UML tooling, 2018.

[10] **Ed-Douibi, H., Izquierdo, J. L. C. and Cabot, J.,** Automatic generation of test cases for REST APIs: a specification-based approach, in *2018 IEEE 22nd International Enterprise Distributed Object Computing Conference* (*EDOC*), pp. 181–190, IEEE, 2018.

[11] **Ed-Douibi, H., J.L.C. Izquierdo and J. Cabot,** OpenAPItoUML: a tool to generate UML models from OpenAPI definitions, in: *International Conference on Web Engineering*, Springer, 2018, pp. 487–491.

[12] **Ed-Douibi, H., J.L.C. Izquierdo, F. Bordeleau and J. Cabot,** WAPIml: towards a modeling infrastructure for web APIs, in: *2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems Companion* (*MODELS-C*), IEEE, 2019, pp. 748–752.

[13] **Gligorić, N., I. Dejanović, and S. Krčo,** Performance evaluation of compact binary xml representation for constrained devices, in: *2011 international conference on distributed computing in sensor systems and workshops* (*DCOSS*), IEEE, 2011, pp. 1–5.

[14] **Hadley, M.J.,** Web application description language (WADL), tech. rep., Sun Microsystems, Inc., 2006.

[15] **Hallé, S., G. Hughes, T. Bultan and M. Alkhalaf,** Generating interface grammars from WSDL for automated verification of web services, in: *Service-Oriented Computing*, Springer, 2009, pp. 516–530.

[16] **Ivanchikj, A., C. Pautasso and S. Schreier,** Visual modeling of restful conversations with restalk, *Software & Systems Modeling*, **17(3)** (2018), 1031–1051.

[17] **Jiang, J. and T. Systa,** UML-based modeling and validity checking of web service descriptions, in: *IEEE International Conference on Web Services* (*ICWS'05*), IEEE, 2005.

[18] **Koren, I. and R. Klamma,** The exploitation of OpenAPI documentation for the generation of web frontends, in: *Companion Proceedings of the The Web Conference 2018*, 2018, pp. 781–787.

[19] **Kumar, B. and J. Jasperneite,** UML profiles for modeling real-time communication protocols., *J. Object Technol.*, **9(2)** (2010), 178–198.

[20] **Martin, D., M. Burstein, J. Hobbs, O. Lassila, D. McDermott, S. McIlraith, S. Narayanan, M. Paolucci, B. Parsia and T. Payne,** OWL-S: Semantic markup for web services, *W3C member submission*, **22(4)** (2004).

[21] **Parssinen, J., N. von Knorring, J. Heinonen and M. Turunen,** UML for protocol engineering-extensions and experiences, in: *Proceedings 33rd International Conference on Technology of Object-Oriented Languages and Systems TOOLS 33*, IEEE, 2000, pp. 82–93.

[22] **Peng, Y.-Y., S.-P. Ma and J. Lee,** REST2SOAP: A framework to integrate SOAP services and RESTful services, in: *2009 IEEE international conference on service-oriented computing and applications* (*SOCA*), IEEE, 2009, pp. 1–4.

[23] **Popić, S., D. Pezer, B. Mrazovac, and N. Teslić,** Performance evaluation of using protocol buffers in the internet of things communication, in: *2016 International Conference on Smart Systems and Technologies* (*SST*), IEEE, 2016, pp. 261–265.

[24] **Roman, D., J. Kopeckỳ, T. Vitvar, J. Domingue and D. Fensel,** WSMO-Lite and hRESTS: Lightweight semantic annotations for web services and RESTful APIs, *Journal of Web Semantics*, **31** (2015), 39–58.

[25] **Sferruzza, D., J. Rocheteau, C. Attiogbé and A. Lanoix,** A model-driven method for fast building consistent web services from OpenAPI-compatible models, in: *International Conference on Model-Driven Engineering and Software Development*, Springer, 2018, pp. 9–33.

[26] **Simpkins, N.** Block 3 part 1 activity 5: Implementing a simple web service, in *T320 E-business technologies: foundations and practice*, Open University, 2008.

[27] **Simpkins, N.,** Block 3 part 2 activity 2: Generating a client from WSDL, in: *T320 E-business technologies: foundations and practice*, Open University, 2008.

[28] **Skogan, D., R. Grønmo and I. Solheim,** Web service composition in UML, in: *Proceedings. Eighth IEEE International Enterprise Distributed Object Computing Conference, 2004. EDOC 2004.*, IEEE, 2004, pp. 47–57.

[29] **Thramboulidis, K. and A. Mikroyannidis,** Using UML for the design of communication protocols: The TCP case study, in: *International Conference on Software, Telecommunications and Computer Networks, SoftCOM. Januari*, 2003.

[30] **Troelsen, A. and P. Japikse,** *Pro C# 7 with .NET and .NET Core.* Apress, 2017.

[31] **Vanderveen, P., M. Janzen and A.F. Tappenden,** A web service test generator, in: *2014 IEEE International Conference on Software Maintenance and Evolution*, IEEE, 2014, pp. 516–520.

[32] **Vara, J.M., V. De Castro and E. Marcos,** Wsdl automatic generation from UML models in a MDA framework, in: *International Conference on Next Generation Web Services Practices* (*NWeSP'05*), IEEE, 2005, Volume 1, pp. 319–324.

M. Cserép and R. Szendrei
Eötvös Loránd University
Faculty of Informatics
Budapest
Hungary
mcserep@inf.elte.hu
swap@inf.elte.hu