Modeling Cloud Messaging with a Domain-Specific Modeling Language

Gábor Kövesdán, Márk Asztalos and László Lengyel

Budapest University of Technology and Economics, Budapest, Hungary
{gabor.kovesdan, asztalos, lengyel}@aut.bme.hu

Abstract. This paper introduces a domain-specific modeling language (DSL) for modeling application-level network protocols. Application-level messages may be expressed in object-oriented general-purpose programming languages as classes. Instances of these classes can be sent through the network with the help of a customized serialization process. However, protocols have several special characteristics that do not fit easily into this abstraction, for example, bitfields or specially encoded lists. Furthermore, the limitations of generic serialization frameworks inhibit using them for this purpose. These factors suggest creating a DSL that more easily expresses these protocols and allows for code generation to support application-level messaging. Application-level messaging is a crucial part of cloud services that follow the Software as a Service (SaaS) paradigm and it must be implemented at both clients and servers. A DSL that allows for efficient modeling of the messages and generating implementation code significantly simplifies the development of cloud applications.

Keywords: Modeling ▪ Domain-Specific Languages ▪ Code Generation ▪ Protocols ▪ Cloud

1 Introduction

Nowadays, there are several high-level communication standards that allow for network communication between two pieces of software. One group of these technologies consists of object-oriented remoting standards, like *Common Object Request Broker Architecture (CORBA)* [1] or Java's *Remote Method Invocation (RMI)* [2]. The other kind of commonly used technologies includes variants of Web Services, namely, the *Simple Object Access Protocol (SOAP)* [3] and *RESTful Web Services* [4]. Despite the availability of these mechanisms, still numerous software vendors decide to develop a lightweight binary application-level protocol that has a lower network footprint and does not require depending on resource-intensive libraries and application servers. However, when it comes to developing such a protocol, developers do not get too much help. The *Specification and Description Language (SDL)* defined by *ITU-T Z.100* [5] allows for describing system behavior in a stimulus/response fashion and thus, it can also be used to specify network protocols. However, SDL has a wide scope and focuses on the stimulus/response relations and does not capture message structure. In theory,

code generators can be developed for SDL to generate implementation but the generated code will cover only the stimulus/response relations. The *Protocol Implementation Generator (PiG)* [6] is a domain-specific modeling language [7] [8] that is designed for code generation but this solution also focuses on interactions. We have not found a domain-specific language with code generator that allowed for the modeling of message structure. In cloud services, especially in those that follow the Software as a Service (SaaS) paradigm, the message structure has more importance than communication states and interactions. First, these systems do not maintain a permanent connection and their messaging is often limited to notifications and request-response messages. Secondly, lower level protocols hide the establishment and the closing of connections, which in turn, involves communication states and interactions. Because of these factors, the development of SaaS messaging primarily consists of determining the message structure and developing the supporting code. Using binary messaging is more challenging to implement than relying on commonly supported formats, such as XML or JSON, that have extensive support in third-party libraries. However, this is the most concise form and thus it generates less network footprint and it is faster to parse. This suggests investing in a DSL and code generator to facilitate this development task. The messaging logic needs to be developed for both the client application and the cloud server. If they do not run on the same platform, the supporting code cannot be shared and has to be developed twice. A DSL and code generation techniques can remedy these difficulties. A code generator can be constructed that uses the model of the message structure and generates the supporting classes and the boilerplate code, even for multiple platforms, if necessary. Such a tool facilitates development and can ensure that the implementations in different languages are consistent.

In this paper such a DSL is presented. Our solution, *ProtoKit*[1], is a lightweight framework that focuses on modeling message structure and generating code to manipulate messages. It encompasses a metamodel that can be used to describe a wide variety of features that can be encountered in application-level messaging. The DSL syntax is similar to Java class definitions because the message structure shares some commonalities with them. The tool targets the use of binary messages. This format is the most concise and helps to save bandwidth, although it does not support well versioning and maintaining backward compatibility. As mentioned before, *ProtoKit* focuses on message structure since it is the most important factor in cloud messaging. It does not deal with modeling interaction: that is simply left for the application developers. We believe that if individual messages can be handled easily, dealing with simple interaction scenarios from handwritten code is easy. However, we may decide to implement modeling interactions in later versions of *ProtoKit*. The tool primarily targets cloud service providers because they always have to implement messaging at the server side and they often also provide the client software or the client library. In this way, they facilitate the use of their service to the consumers and they do not have to publish the protocol specifications. In this scenario, cloud service consumer indirectly benefit from *ProtoKit* as well. Additionally, if there is no appropriate client software or library but the protocol

[1] See http://gaborbsd.github.io/ProtoKit/.

specification is available, the service consumers may also use the tool to develop their own solution.

The rest of this paper is organized as follows. In Section 2 the motivation for creating a protocol modeling language is explained. Section 3 introduces the metamodel that was used to describe the problem domain. Section 4 explains the concrete syntax and show its grammar. Section 5 gives a detailed explanation of how the *ProtoKit* language was implemented. Section 6 presents a case study in which the solution is evaluated and Section 7 concludes. Despite not being a cloud protocol, the *Domain Name System (DNS)* protocol will be used as an example throughout the paper. The DNS protocol is well-known, has a wide variety of features that has to be handled in the metamodel and has similar characteristics as cloud messages: it lacks complex interactions and uses a simple request-response communication model. We are working on other applications that use *ProtoKit* and offer cloud services but they are still in an early phase of the development. Therefore, we have chosen to use the caching DNS server as an example.

2 Motivation

Protocol message types are very similar to classes in object-oriented programming languages: both notions define a complex data type with some properties that hold values. For example, a protocol message type may hold a transaction number of integer type, the identifier of the sender as an integer, an integer count that specifies the length of the payload and last but not least a variable-length payload either as a text or as binary data. Modeling such protocol message types is definitely possible with object-oriented languages but protocol message types have several specific properties that needs further boilerplate code in object-oriented languages. The following list summarizes these:

1. Defining the length of the field is paramount since the fields in a single message will be parsed by calculating the boundaries. In general-purpose programming languages (GPLs), this aspect is handled in a lazier manner. We usually choose from byte, short integer, normal integer and long integer variable types based on our needs of precision. Some languages, like C, do not strictly define the byte sizes of these type, whereas others, like Java, do. Still in the latter case, the byte length of a specific field is not explicitly reflected in the code, that is, the programmer must be conscious that, for example, a long variable takes 64 bits. Protocol message type definitions warrant for a more precise notation that explicitly expresses field lengths.
2. GPLs do not allow easily accessing fields on a per bit basis. Protocols do need such feature so that they can keep the network footprint low and avoid wasting bandwidth. One-bit boolean fields grouped to one or more bytes as flags are frequent. In GPLs, there is no native type that maps to such fields. Although it is possible to manipulate particular bits of a larger integer by using bitwise operators or by helper accessor methods, it requires more coding and makes the code less readable. A protocol message type definition language certainly needs to be able to handle data bit by bit.
3. Protocol messages often use counter fields that describe how many of a particular entry is found in the variable-length payload part of the message. These counters make it possible to properly parse the variable-length part of the message. In protocol

definitions, it would be practical to directly associate counters to the corresponding list of entries.

4. After the protocol message type is modeled, the serialization and deserialization of messages must be implemented so that messages can be transmitted and received over the network. Some GPLs, like Java, offer a standardized way for serialization but it does not fit well serialization of protocol messages. Serialization of protocol messages has several specific characteristics:
   (a) Usually, it has to be strictly ordered based on the specification order of the fields.
   (b) Length of fields is strictly specified, possibly on a per-bit basis.
   (c) Counts of entries must be handled as well.
   (d) Some fields may be encoded in a specific manner, for example, the *Domain Name System (DNS)* [9] protocol specifies its own encoding for the requested domain names.

When implementing the serialization of protocol messages, these requirements must be properly addressed. The above reasons suggest introducing a DSL for protocol message types that takes into account the above criteria and allow for easy and fast modeling of protocol message types.

Kövesdán et. al published an intent catalog [10] that lists and describes the possible motivating factors behind creating DSLs and their main characteristics. *ProtoKit* uses the following intents:

1. *Specialized Tool:* GPLs are not able to properly express all of the features of network protocols.
2. *Modeling Tool:* from the textual description, a model is constructed. This is used later for code generation.
3. *Domain-Specific Formalism:* since human language is ambiguous, developers may decide to also include the *ProtoKit* description in specifications.
4. *Human-Friendly Notation:* the *ProtoKit* language is much more concise than, for example, a Java class definition, therefore it is easier to read and write for non-programmers.

3 The Metamodel

The metamodel that we used for modeling network protocols is depicted in Figure 1. The elements of the metamodel are the following:

- *DataType:* message type or complex data structure. The class that is used to encapsulate the whole message is not treated in any special way so there is no need for introducing other metamodel element. A complex data structure may as well be embedded into another one.
- *BinaryField, IntegerField, StringField, ListField, CountField, LengthField, BitField:* specific types of fields that are embedded into messages or complex data structures.

- *BitFieldComponent: BitField* is further divided into components, which occupy only specific bits of the member. We generate specific getters and setters for these to handle the appropriate bits transparently.
- *Formatter:* some fields are encoded in a specific way. This kind of encoding can be implemented with the support of formatters. For formatters, only a template is generated that has to be filled in by the developer of the application.
- *ProtocolModel:* aggregates the *DataType* and *Formatter* elements into a model. It can be used to traverse the model for code generation.
- *Field:* abstract type of fields used as messages or complex data structure members.

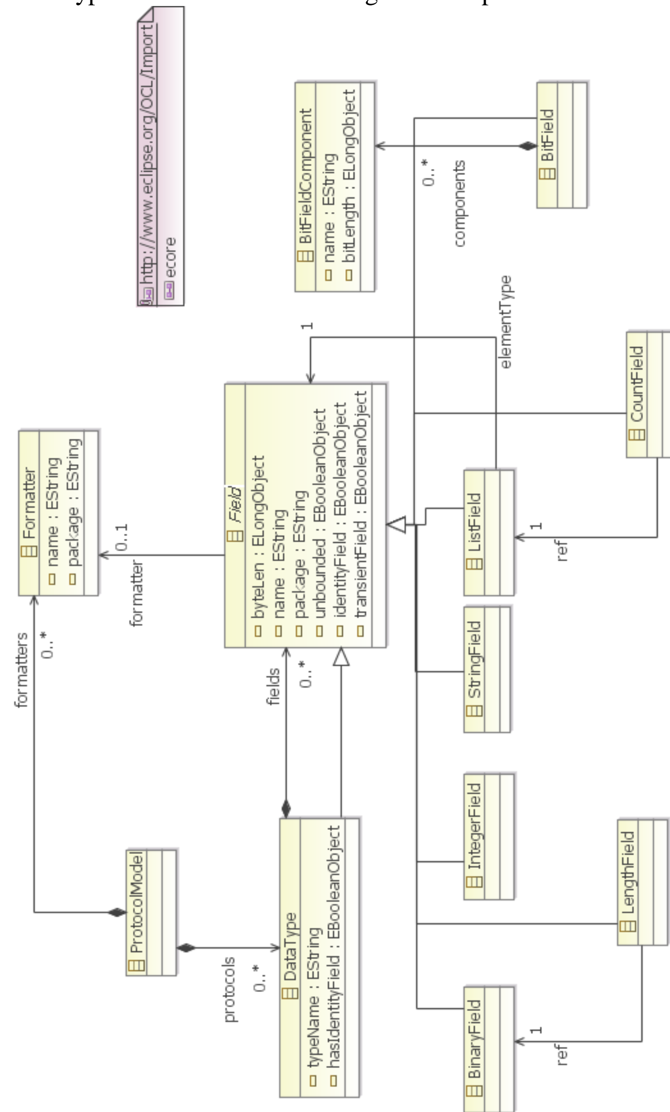

Fig. 1. The metamodel used for the *ProtoKit* language

4 The Concrete Syntax of the Language

In this section, we briefly describe the concrete syntax of *ProtoKit*. The syntax is somewhat similar to class diagrams. This is helpful for the developers since the nature of the models is also similar to class diagrams. A model starts with the *package* keyword and an identifier. These will define the package of the generated Java classes. After this, we can define protocol messages and embedded data types. The former starts with the *protocol* keyword and the latter uses the *datatype* keyword. The definition of messages and data types is given in curly braces. We specify fields by their name and type, separated by a colon. The type may have arguments that refer to the length of the field or to a referred field in case the field is a counter or a length field. After a field of the string type, we can also specify a formatter with its name. Components of *bitfields* are also defined in curly braces and these are always treated as integer types so only their length is specified in bits. The names of normal fields (that are not components of *bitfields*) can be preceded by the *transient* keyword and an asterisk. The former means that the field will not be serialized[2] and the latter marks the fields that should determine the identity of instances. This is used for generating *equals()* and *hashCode()* methods. The ANTLR grammar of the language is cited in the following code listing.

```
grammar NetworkProtocol;

start: packageDefinition? protocolDefinition+;

packageDefinition: 'package' name = ID;

protocolDefinition:  ('protocol'|'datatype') name = ID '{'
variableDefinition+ '}';

variableDefinition: trans='transient'? identityVar='*'? name =
ID ':' (intType|stringType|binaryType|embeddedType|bitfieldType
|listType|countType|lenType);

intType:   type = 'int' ('(' len = NUMBER ')')?;

stringType:  type = 'string' ('(' len = (NUMBER|'*') ')') for-
matterDefinition?;

binaryType:  type = 'binary' ('(' len = (NUMBER|'*') ')');

embeddedType:  type = ID;

bitfieldType:  type = 'bitfield' '{' bitfieldDefinition+ '}';

listType: type = 'list' '(' (listElement = ID) ')';
```

[2] It can be disputed whether this feature should be part of the modeling language since it is not related to the actual model but to the implementation. However, it allows for adding implementation-specific members to the generated classes without having to modify them. This does not make it necessary to deal with manual changes when regenerating classes.

```
countType: type = 'count' '(' len = NUMBER ',' countedList = ID
')';

lenType: type = 'length' '(' len = NUMBER ',' countedField = ID
')';

bitfieldDefinition:  name = ID ':' bitLength = NUMBER;

formatterDefinition: 'formatter' name = ID;

ID: [a-zA-Z]+;

NUMBER: [1-9] [0-9]*;

WS: [ \t\r\n]+ -> skip
```

5 Implementation Decisions

In this section, we describe step by step what implementation decisions we considered during the developing of *ProtoKit*. This gives a deeper insight into the development process and allows for understanding how our motivations and the requirements affected the architecture and the development process. By examining these points regarding a new DSL that is being developed, we can also reuse these experiences as a recipe.

1. *Implementing generic functionality in the runtime framework.* It is not a trivial decision at what extent the code should be factored out into a generic runtime framework and what should be generated. Developing generic code is generally more challenging and sometimes may have worse performance than a customized solution. For example, in Java environment, it is often done by using reflection, which has a performance hit. On the other hand, generic code is more reusable and helps to reduce the generated code. It is easier to generate custom code than having a generic framework but the latter is easier to test since it does not depend on the input model. The reusable framework can be covered by extensive unit tests, whereas the generated code is not trivial to test. It may be tested for the complete functionality but it is difficult to think of all of the possible corner cases. Because of these considerations, in *ProtoKit* we have implemented most of the functionality in a reusable runtime framework. The most important example of this is the generic serialization logic. The generated classes only encompass the serialization parameters. It would have been possible to generate the serialization logic and to make it part of the generated class but this would have resulted in a significantly more complex generator.
2. *Separating tree parser logic from code generator.* In theory, it is possible to process the input file in an event-driven approach with a single read since we can store arbitrary amount of details in the state of the parser that will be required in further phases of the processing. However, this technique has several drawbacks. Because of the event-driven nature, the code fragments that are generated are triggered by visiting a certain grammar rule. This does not facilitate generating non-consecutive code fragments from the same node. For example, if the generated code is a Java class, certain nodes can be mapped to Java variables. Java variables are usually declared

with *private* visibility and accompanying getter and setter methods are provided with *public* scope. Although it is not mandatory, the variable declarations are conventionally placed together at the beginning of the class definition and the getter and setter pairs come after all variable definitions. A single variable and its getter and setter methods are generated from the same node so they cannot be generated in the conventional order by a purely event-driven manner. This requires the parser to use its internal state to store some details. This makes the parser more complex and partly leads to building a semantic model in the memory. Secondly, the use of a pure event-driven approach does not allow for validating cross-references in the input file. To address these issues, we have separated the tree parser logic and the actual code generation. The only responsibility of the tree parser is building a semantic model. By using a semantic model, validation and code generation become much easier. Furthermore, the use of a semantic model better facilitates changes in the syntax or building another, possibly visual modeling language on top of the code generator.
3. *Decoupling code generation and formatting.* Although generated code does not necessarily need to be read by humans, readability is definitely a great advantage. For example, it may help debugging or facilitate the comprehension of how the *ProtoKit* tooling works. However, hardcoding formatting, like indentation level, line breaks etc. in the generator significantly deteriorates the readability of the generator itself. The readability of the generator is definitely more important than the readability of the generated code so this is not a viable trade-off. However, collecting the output in a buffer and using a code beautifier before flushing the code to the disk has proven to be a favorable solution. This solution makes the code formatter reusable. In *ProtoKit*, the code formatter provided by *Eclipse JDT* [11] has been used.
4. *Using template language for code generation.* When traditional programming languages are used for code generation, fragments of the generated text must be quoted and concatenated to the variables that are substituted. All of this is written to a buffer with method calls. These method calls, the quotation marks and the concatenations deteriorate the readability and make it hard to see what output will be actually generated. Template languages reverse the logic: everything that is written will be part of the output by default and only variable substitutions and branching need special markup. In Java environment, *Xtend* [12] is a good choice of a template language. It does not compile directly to bytecode but to Java source code so it can be easily used anywhere where Java is used. *ProtoKit* is written mostly in Java but the code generator classes are implemented in *Xtend.* There are some branching statements and substitutions but the readability of the generator is much better than it could be in pure Java. The separation of the code formatter logic and the use of *Xtend* have significantly improved the productivity during the development of *ProtoKit*.
5. *Using a metamodeling framework and decoupling the validator logic.* The validation logic was first coded into the tree parser but as we added more features to the language, it started to deteriorate the readability so we decided to factor it out. At the same time, we introduced an explicit metamodel, described with the *Eclipse Modeling Framework (EMF).* [13] Using a modeling framework allows for reusing its validation solutions and we wanted to benefit from this. However, associating line and

column numbers from the input text with validation errors is more challenging. Its implementation requires attaching extra information to the model.

6 Evaluating *ProtoKit*

To demonstrate that *ProtoKit* is in fact useful and really helps the development of applications that use network communication, it must be put into practice. For this purpose, we have implemented a caching DNS server that either answers queries from its local cache or forwards queries to the configured resolver. *ProtoKit* was in fact easy to use. We could model DNS messages easily. The generated classes and the generic serialization logic highly simplified the process. Describing the caching DNS server is beyond the scope of this paper but we have summarized some statistics about the generated and handwritten code in Table 1.

Table 1. Statistics in code lines

Language	Generated	Reused	Hand-written	TOTAL
ProtoKit	0	0	39	39
Java	463	328	289	1080
TOTAL	463	328	328	1119

In this case, only about 27% of the Java code had to be manually implemented. If we also take into account the textual model of the protocol, we get about 29% hand-written code, which seems to be a really good proportion.

The caching DNS server is a simple but realistic application so we believe it is a good case study to evaluate *ProtoKit*. More complex applications also contain more application logic that is independent of the networking code. The proportion will be worse in such cases but *ProtoKit* is only meant to facilitate network messaging.

Using SaaS cloud services is always more common in computing, especially in mobile applications. As described above, cloud messaging protocols are similar to the DNS protocol in complexity: they use a request-response communication model and do not encompass complex interactions. Since *ProtoKit* performed well in modeling DNS messages, it will also be very useful for implementing the messaging between clients and the cloud services. The Android platform uses Java as its main programming language – however the class library is slightly different – which means that the generator can also be used for Android applications with no or few modifications. Furthermore, Java is also a popular platform in backend development, so the messaging code can be shared between the client and the cloud server backend. Besides, *ProtoKit* is easy to extend to support other target languages.

7 Conclusion

In this paper we have reported about our progress in developing the *ProtoKit* language and tooling. We have explained our motivations and how we created a DSL to solve

these issues. We have also summarized the decisions that we met during the implementation and why we took these decisions. Finally, we have also reported on a simple but realistic application that we developed with the help of *ProtoKit*. In this case study *ProtoKit* in fact simplified the development. However, the main focus of this solution is developing cloud messaging. Cloud messages use simple interactions but require a message structure that is easy to work with and can be serialized efficiently. This is exactly what *ProtoKit* is meant to be used for. We hope that it will prove to be useful in practice on the long term and that our experiences will help other developers that implement cloud-enabled applications.

Acknowledgments. This work was partially supported by the European Union and the European Social Fund through project FuturICT.hu (grant no.: TAMOP-4.2.2.C-11/1/KONV-2012-0013) organized by VIKING Zrt. Balatonfüred. This work was partially supported by the Hungarian Government, managed by the National Development Agency, and financed by the Research and Technology Innovation Fund (grant no.: KMR_12-1-2012-0441).

References

1. The Object Management Group: CORBA 3.3 Specification, *http://www.omg.org/spec/CORBA/3.3/*
2. Grosso, W.: Java RMI, O'Reilly Media (2001)
3. World Wide Web Consortium: Simple Object Access Protocol (SOAP) Specification, *http://www.w3.org/TR/soap/*
4. Richardson, L., Ruby, S.: RESTful Web Services, O'Reilly Media (2007)
5. International Telecommunication Union: ITU-T Z.100 Standard. Specification and Description Language (SDL), *http://www.itu.int/ITU-T/studygroups/com10/languages/Z.100_1199.pdf*
6. Quaresma, J.: A Protocol Implementation Generator, Master Thesis, *http://nordsecmob.aalto.fi/en/publications/theses_2010/jose_quaresma.pdf*
7. Fowler, M.: Domain-Specific Languages, Addison-Wesley (2010)
8. Kelly, S., Tolvanen, J.: Domain-Specific Modeling: Enabling Full Code Generation, Wiley-IEEE Computer Society Press (2008)
9. The Internet Engineering Task Force: Domain Names – Implementation and Specification, Request for Comments 1035, *http://www.ietf.org/rfc/rfc1035.txt*
10. Kövesdán, G., Asztalos, M., Lengyel, L.: A classification of domain-specific language intents, International Journal of Modeling and Optimization, vol. 1, no. 4, pp. 67–73 (2014)
11. The Eclipse Project: Eclipse Java Development Tools, *http://www.eclipse.org/jdt/*
12. Bettini, L.: Implementing Domain-Specific Languages with Xtext and Xtend, Packt Publishing (2013)
13. Steinberg, D., Budinsky, F., Paternostro, M., Merks, E.: EMF: Eclipse Modeling Framework (2nd Edition), Addison-Wesley Professional (2008)