Interoperability Marshaller/Unmarshaller

Youngsu Son, Jiwon Kim, Donguk Kim, Jungmin Son
Samsung Electronics

arload.son, jiwon.ss.kim, dude.kim, jungmin4.son@samsung.com

ABSTRACT

CORBA, XML Web Service which has been a standard of traditional Distributed Object systems, is a solution using Broker (Object Request Broker, Enterprise Service Bus) for monitoring and reliability of system. However, the system is too heavy to satisfy various requirements such as reliability, availability, and heterogeneity. On the Contrary, traditional RPC (Remote Procedure Call) is lighter than the legacy distributed object solution, but its availability is limited to specific platform and it has language dependency. Recently, several papers have been published describing lightweight solutions supported in several influential systems and languages [Google Protocol Buffer] and [Facebook Thrift]. This paper introduces a hybrid pattern which takes advantages of the strengths of both the heavier, legacy approaches and lightweight approaches.

Categories and Subject Descriptors

D.2.11 [**Software Engineering**]: Software Architectures – *patterns.*

General Terms

Architecture, Design

Keywords

Marshaller/Unmarshaller, Serializer

1. INTRODUCTION

A traditional distributed object such as CORBA or Web Service can provide the best solution for both interoperability and system monitoring.

However, if all we want is interoperability, we do not want to run a heavy implementation with resource requirements that prohibit the implementation running as a 'think client' (such as in a browser).

Although, traditional socket programming can be customized or optimized to a specific environment, such as a thin client, We want a reusable framework to avoid the effort/cost (and potential for programmer error) associated with using a low level API to implement each protocol.

Therefore it is needed to use Marshaller / Unmarshaller that supports light and various languages and is able to deliver data easily.

2. EXAMPLE

In case of Office Automation System, it consists of various devices and systems such as security System, power management system, fire management system, and so on. Some of the systems are implemented as rich clients, but others can be thin clients.

Systems with enough resources to support a, can use a Distributed Object System such as CORBA(Common Object Request Broker Architecture), WCF (Windows Communication Foundation), and JBOSS.

Thin Client systems cannot meet the resource requirements of a Rich Client and, often, such systems implement interoperability using low-level programming for example, Socket.

The Marshaller/Unmarshaller uses streams to communicate conveniently and easily supporting both Thin Clients and Rich Clients.

3. CONTEXT

It is necessary to be able to communicate each other in heterogeneous environment, via an easy to use and lightweight framework.

4. PROBLEM

In Server and Client System of heterogeneous environment, there are a server based on Java and a thin client. To develop a new protocol to communicate in a heterogeneous environment costs too much in both communication and development resources.

To achieve Location Transparency, a distributed object such as CORBA, WCF, and JBoss may be used; however, for a distributed system which needs quite simple communications we want to enable use of a thin client.

Therefore, a system that works well in a heterogeneous environment with both heavier, legacy and lightweight languages/environments is desired.

5. FORCES

The following items should be regarded as forces:

- Lightness - The system should be lightweight for low performance system
- Usability - Marshaling/Unmarshaling should be able to used without being aware of Server/Client's protocol
- Simplicity - Code should be simple, approachable, and free of unnecessary dependencies.
- Transparency - the system should consider a heterogeneous environment.
- Consistency Niche, language-specific features belong in extensions, not the core library.

6. SOLUTION

This system (refers figure 3) supports various Native Language, Protocol and Transport for communication in Heterogeneous environment.

This pattern has largely divided to two parts.

Contract Converter - compiles Intermediate Language and sends and receives data derived from various languages.

Channel - has various data formats and transport layer for communication in Heterogeneous environment.

6.1 Structure

Contract Converter (refers figure 4) is quite similar to Modern Compiler's which is well known. CodeGenerator is largely divided into Interpreter Pattern and Bridge Pattern.

- LexicalTokenizer - extracts Tokens from IDL.
- SyntaxParser - groups Syntax by unit. Syntax means separating a sentence to physical unit.
- SemanticAnalyzer - extracts semantics based on Syntax which is divided into physical unit.
- Intermediate CodeGenerator - converts inserted IDL files to Intermediate Code (Intermediate Representation) format.
- TargetCodeGenerator – replace IntermediateCode with StringTemplate by correspondent to Native Language. StringTemplate is combined with Name, ID, and then the final Skeleton code is generated.

Channel (refers figure 5) transports messages consists of following.

- ProtocolFactory - is AbstractFactory of ConcreteProtocolFactory that creates protocol.Client code is able to get necessary protocol by Abstract Factory, even though it doesn't know the specific concrete factory.
- ConcreteProtocolFactory - is Concrete Factory creating Concrete Protocol that decides specific data format for Communication
- Protocol - is abstract class of ConcreteProtocol which decides data transport format. It has readMessage and writeMessage that is Template Method and controls entire flows of ConcreteProtocol such as how to work.
- ConcreteProtocol - is an object to support a variety of data format. One or more than two protocols' combinations can be used in communication, for example CompactProtocol and SecurityProtocol.
- TransportFactory - is Abstract Factory of ConcreteTransportFactory which creates transport layer. A protocol can use diverse transport.
- ConcreteTransportFactory - is Concrete Factory which creates ConcreateTransport deciding how to transport in detail.
- Transport - is an Interface which enables to access ConcreteTransport as Interface. As a result, communication method can be easily changed from Inter Procedure Call to Remote Procedure Call
- ConcreteTransport - is an object to support a variety of transport method which can be added and extended continuously.
- Processor - is a dispatcher which decides how to process data getting from server.

6.2 Dynamics

6.2.1 Contract Convertor

Scenario #1. Skeleton Code derived from IDL Compiler
(Scenario #1 refers figure 6)

1. Extracts tokens from IDL Scripts.
2. Parse tokens and compose Abstract Syntax Tree.
3. Edits semantics in each syntax.
4. Finds pattern template related to Semantic elements. There are function setter/getter for data storing and Method Skeleton for service to support and so on
5. Concrete Pattern Template which is optimized to Target Language is created.
6. Skeleton code which combines significant name tokens from syntax tree and pattern template derived from each target language is created.

6.2.2 Channel

Scenario #2. Data Transmission from Server to Client.
(Scenario #2 refers figure 7)

1. IDL Compiler puts data into each field in Skeleton.
For example, Setter such as Person.SetName() and Person.SetAge() can be used to fill with data.
2. Once data is inserted, it is converted to message format that is sent by protocol.
3. Appropriate transport is created by TransportFactory.
4. ProtocolFactory decides which protocol to use.
5. When messages created by Skeleton are delivered to Protocol, Protocol converts them to proper format by parsing , and then sends them through Transport.
6. Waits for messages coming, and then when data arrives through Transport, it is parsed to correspondent protocol and delivered with Skeleton format (Native Language)

Scenario #3. Return the result to Client after server gets a message to Client.
(Scenario #3 refers figure 8)

1. Server parses messages coming from Processor(Dispatcher) and gets information about what kind of Protocol is used from Header data.
2. Reads entire messages from getTransport() to know which Transport was used in Protocol.
3. Protocol splits data into each message and read data from each file.
4. Messages needs to be unmarshalling as skeleton.
5. Server Skeleton gets data from correspondent field as object, processes them , and return them.
6. To send the messages which returns Processor to client, it is necessary to reorganize the messages and deliver them through

Transport to client.

7. Implementation

Step 1: Decide the message exchange format considering Quality of Service (QoS).

Based on interoperability to support, you must decide whether to use a system specific format or an IDL to get rid of language dependency.

Using IDL provides interoperability and flexibility in a multi-language environment, while degrading performance.

Also, you will need to have many discussions and tests regarding User Defined Data Type for each different supported language.

You may decide to make trade-offs between the high-cost of legacy approaches and the lower performance of an IDL. It is possible to use system-specific functionality to get better performance than an IDL without incurring the full cost of implementing CORBA and Web Services.

In case of Google, there are following reasons why they use their own structured data format.

- protocol buffers are 3 to 10 times smaller than xml
- protocol buffers are 20 to 100 times faster than xml
- protocol buffers are less ambiguous than xml.
- protocol buffers generate data access classes that are easier to use programmatically

Design data format and protocol which is quite suitable for the system environment to be used or use.

Step 2 : Test usability with objects' interface extracted in real target system to be used , before integrate objects.

Usability of the framework is as important as meeting the interoperability requirements. Designers maintain a variety of language and platform preferences they use to design systems. So, we want to design a platform that supports the designers' desired environment. Therefore usability is necessary.

Before implementing the class, write the code supporting main scenario first, and then get feedback from users. After that, define object models based on the code sample given from feedback. For example, a framework designer familiar with C/C++ made a file reading scenario like below.

```
static void Main(string[] args)
{
   StreamReader sr = File.OpenText("MyFile.txt");
   string s = sr.ReadLine();

   while (s != null)
   {
         s = sr.ReadLine();
           Console.WriteLine(s);
   }
}
```

List1. File Reading Scenario

The designer thought that the code above was natural. However, other designers familiar with other language gave feedback with more intuitive code sample.

```
static void Main(string[] args)
{
   foreach (string s in File.ReadAllLines("MyFiles.text"))
  {
            Console.WriteLine(s);
   }
}
```

List2. File Reading Scenario as feedback

This is just the power of feedback. Framework designers sometimes make mistakes by building a framework within a familiar culture of familiar custom or language.

When you design actual framework API, if you verify the framework scenario before implementation and get feedback, you can get refined scenario naturally.

Finally, you can design a good API by putting these together and constructing actual object model with that.

Step 3: Develop IDL Compiler

Developers can easily develop IDL compiler with Tools, for example yacc, and so on. To make compiler becomes easier than last decades.
By utilizing Lex that is an automation tool for tokenizing and yacc for syntax analyzing, it has become so easy to make a simple grammar compiler with small amount of knowledge about them.

Even though the IDL you are planning on supports complex functionality, it is not quite difficult to make it as well if you study Semantic in depth little bit more.
When you design IDL, you will think about simplicity and complex and diverse functionality.
Usually, simplicity is better for interoperability. Thrift and Protocol Buffer are good examples of such simplicity.

Custom IDL complier requires maintenance features to reuse it. Because it is not good way to make a new compiler when IDL is changed, you need to implement and manage version control of IDL. Just building changed IDL and dependent component is better to support new functionality. When function or target language's extension is implemented, Bridge Pattern which extends expression range of generated code can be used.

Step 4 : Consider extensibility of messages sending and receiving.

The information which is sent to marshaling/unmarshaling method can be added/ modified according to the application scenario or node's feature.
Various nodes are added in the system and thus, the exchange information is designed for extension. For that reason, it should consider using the Composite Message[1]or the Parameter Object pattern[2].

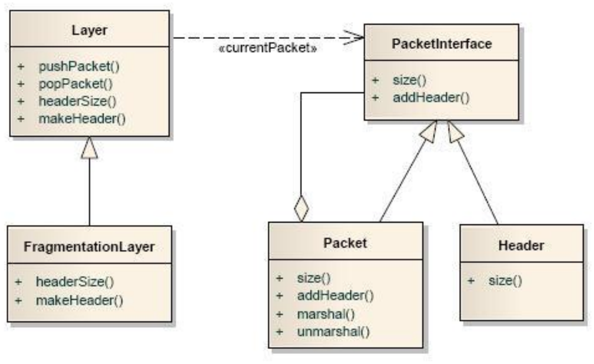

Figure 1 Composite Message Pattern

8. Known Use

- Google Protocol Buffers

Protocol Buffers are a way of encoding structured data in an efficient yet extensible format. Google uses Protocol Buffers for almost all of its internal RPC protocols and file formats.

- Facebook Thrift

Thrift is a software framework for scalable cross-language services development. It combines a software stack with a code generation engine to build services that work efficiently and seamlessly between C++, Java, Python, PHP, Ruby, Erlang, Perl, Haskell, C#, Cocoa, Smalltalk, and OCaml.

- Hadoop Avro

Hadoop's data serialization system that provides dynamic integration with scripting languages. Avro supports serialization among C, C++ and Java based system.

9. Case Study

Apps in Google consist of many heterogeneous apps such as GTalk, GMail, GoogleCalendar, Google Docs, and so on.

In order to support those heterogeneous system as one Cloud Service,
Google internally uses Interoperabilty Marshaller/Unmarshaller (Google Protocol Buffers) as their communication machanism.

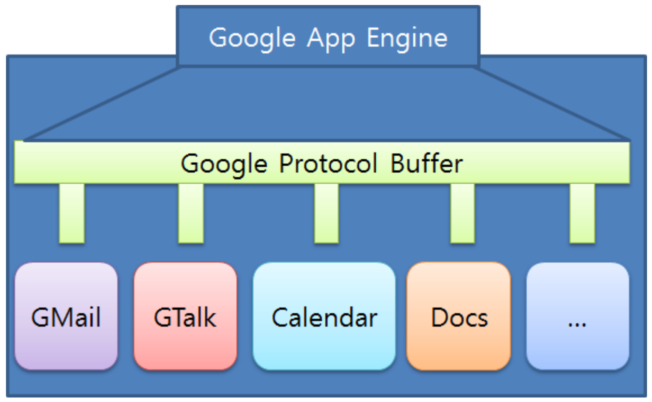

Figure 2 Google Apps Engine

10. Resulting Context

The advantages of this pattern include:

- It is possible to get efficient communication channels with transport layer protocol and data format for relatively thin client with low costs and resource consumption of data build in heterogeneous environment.
- Between heterogeneous client and server, there are no protocol limitations and reduced data format, serialization, and additional work about transport layer.
- In case of needs about various services, channels and data based on environment are selectively able to be offered.
- It is quite suitable to dynamic server-client configuration since there is no dependency between them by environment oriented and separated Transport, Protocol, and Service configuration.
- It is easy to add new Transport , Protocol or service which means easy maintenance.
- It is easy to reuse by separation of Transport, Protocol, and service layers of Contractor and Channel.
- It is possible to monitor each type, such as Protocol, Format, Transport Layer, in heterogeneous Transport.

Possible disadvantages are:

- It takes slightly time for marshalling/unmarshalling of various languages' data packets.
- It might cause subtle calculation overhead in a system architecture that doesn't need various language data.
- It might cause to make code dirty because of vast data type and environment based serializer.
- There is a tradeoff which means that it is quite enough for thin client communication but might cause speed limitation because of packet, format, calculation complexity of transport.
- It can cause only overhead in the environment which supports various services and client-server have same.

11. Related Pattern

Composite Message [1]

This pattern is used for marshaling/un-marshaling data, extending and adding messages you want to transfer while passing through layers.

And enable to exchange without influence to others by separation of Transport, Protocol, and Service. It is useful in case of dynamic channel configuration.

Pipe & Filter [6]

This pattern is used when adding or filtering messages you want to transmit flexibly according to the circumstance, used internally in the aforementioned Composite Message.

Broker [6]

This pattern removes direct dependency (location information, platform restrictions, etc.) between server and client.

And it is useful for coordination/Monitoring of messages between Client and Server and possible to register and deregister various services dynamically by Broker.

ACKNOWLEDGMENTS

We thank our shepherd Richard Lai for his valuable comments that contributed to improve this paper. The EVA (Pattern Evangelist Group), James Chang provided useful improvements and corrections.

REFERENCES

[1] Aamond Sane, Roy Campbell, “Composite Messages: A Structural Pattern for Communication between Components”, OOPSLA’ 95Workshop on Design Patterns for Concurrent, Distributed, and Parallel Object-Oriented Systems,1995.

[2] Martin Fowler, Kent Beck, John Brant, William Opdyke, Don Roberts, “Refactoring : Improving the Design of Existing Code”, Addison-Wesley Professional , 1999

[3] Robert S. Hanmer, “WatchDog”, Patterns for Fault Tolerant Software, WILEY, 2007

[4] James C. Hu, Douglas Schmidt, "JAWS: A Framework for High-Performance Web Servers", Domain-Specific Application Frameworks: Frameworks Experience By Industry, John Wiley & Sons, October, 1999.

[5] Michael Hicks, Jonathan T. Moore, Scott Nettles, “Dynamic Software Updating”, ACM Transactions on Programming Languages and Systems (TOPLAS) ,Volume 27 , Issue 6

[6] Frank Buschmann , Regine Meunier , Hans Rohnert , Peter Sommerlad , Michael Stal, "Pattern-Oriented Software Architecture Volume 1: A System of Patterns", WILEY, 1996

[7] Douglas C. Schmidt, Michael Stal, Hans Rohert, and Frank Buschmann, "Pattern-Oriented Software Architecture Volume 2: Patterns for Concurrent and Networked Objects", WILEY, 2000.

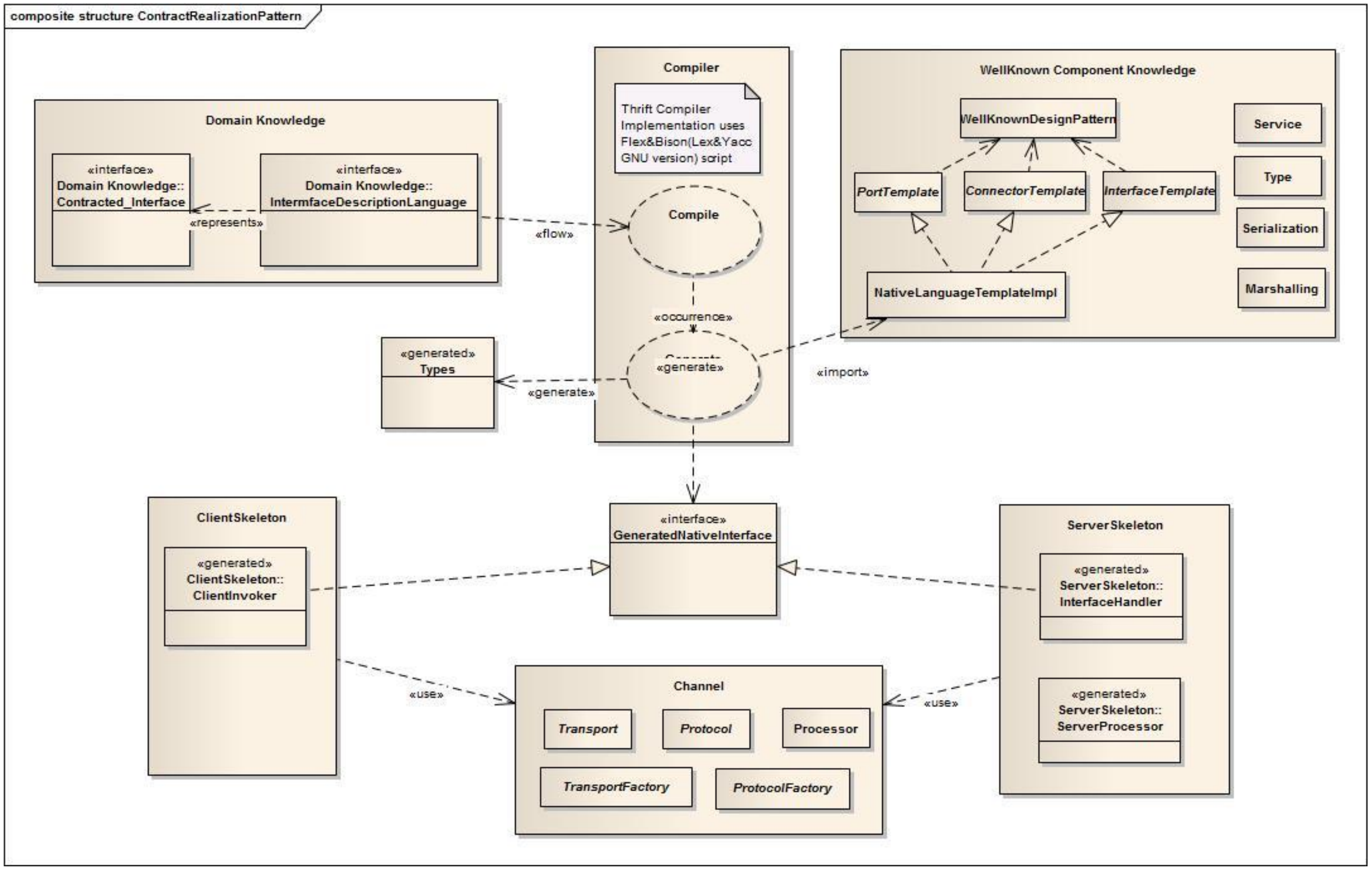
composite structure ContractRealizationPattern
Domain Knowledge
«interface» Domain Knowledge:: Contracted_Interface
«interface» Domain Knowledge:: IntermfaceDescriptionLanguage
«represents»
«flow»
Compiler
Thrift Compiler Implementation uses Flex&Bison(Lex&Yacc GNU version) script
Compile
«occurrence»
«generate»
«generate»
«generated» Types
«import»
WellKnown Component Knowledge
WellKnownDesignPattern
PortTemplate
ConnectorTemplate
InterfaceTemplate
NativeLanguageTemplateImpl
Service
Type
Serialization
Marshalling
ClientSkeleton
«generated» ClientSkeleton:: ClientInvoker
«interface» GeneratedNativeInterface
ServerSkeleton
«generated» ServerSkeleton:: InterfaceHandler
«generated» ServerSkeleton:: ServerProcessor
«use»
«use»
Channel
Transport
Protocol
Processor
TransportFactory
ProtocolFactory

Figure 3 Interoperability Marshaller/Unmarshaller Architecture

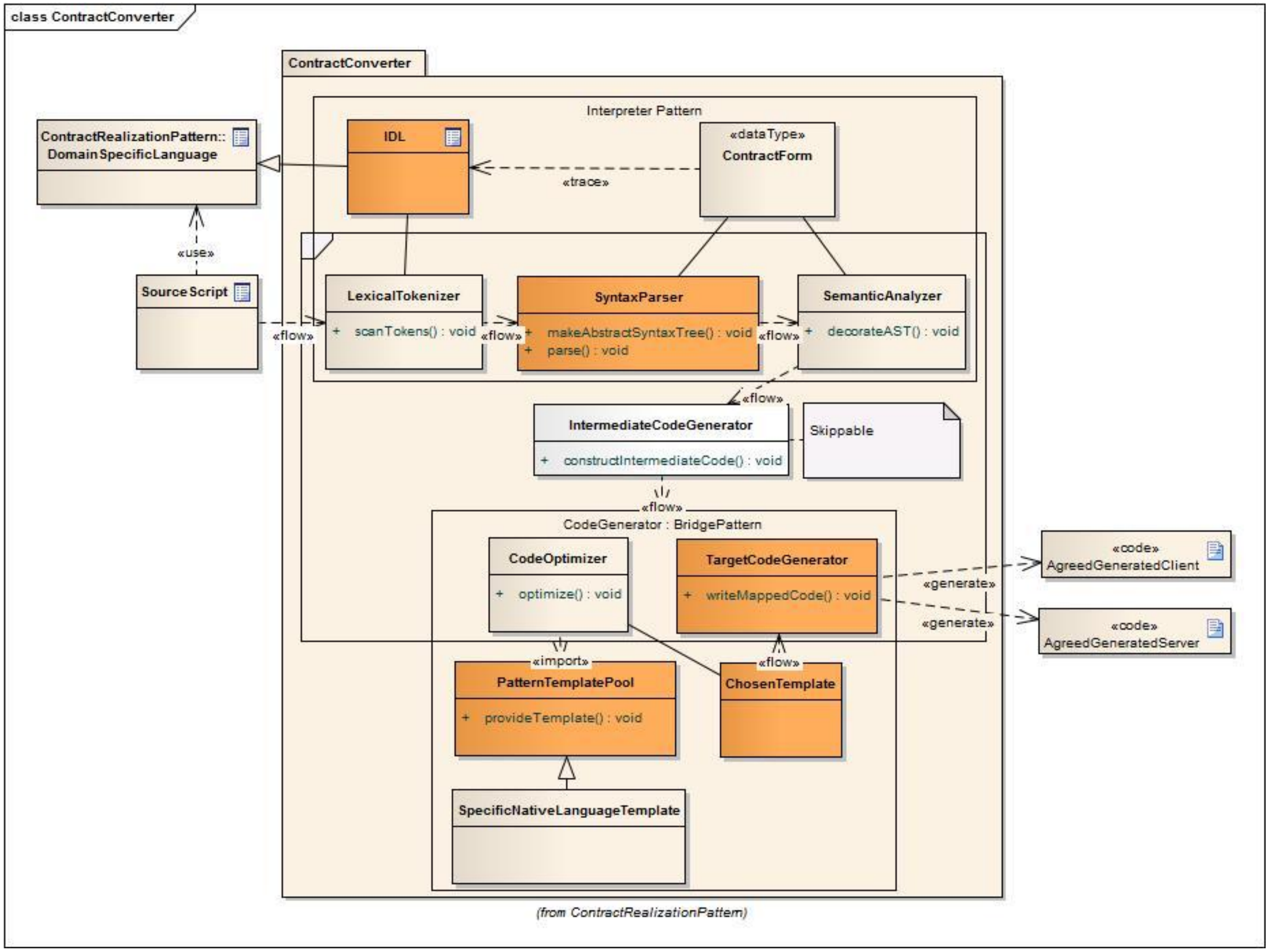

Figure 4 Contract Converter

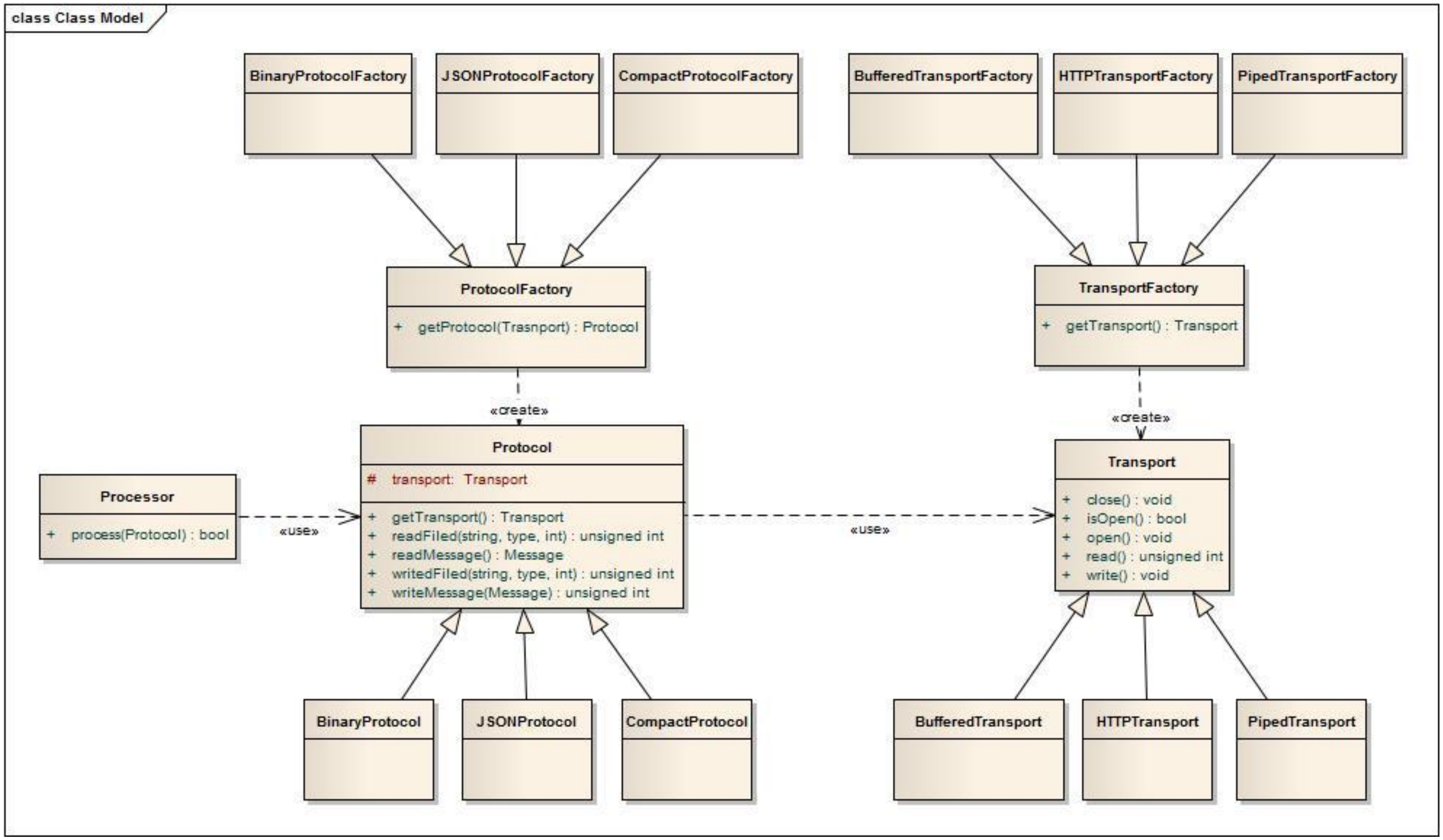

Figure 5 Channel

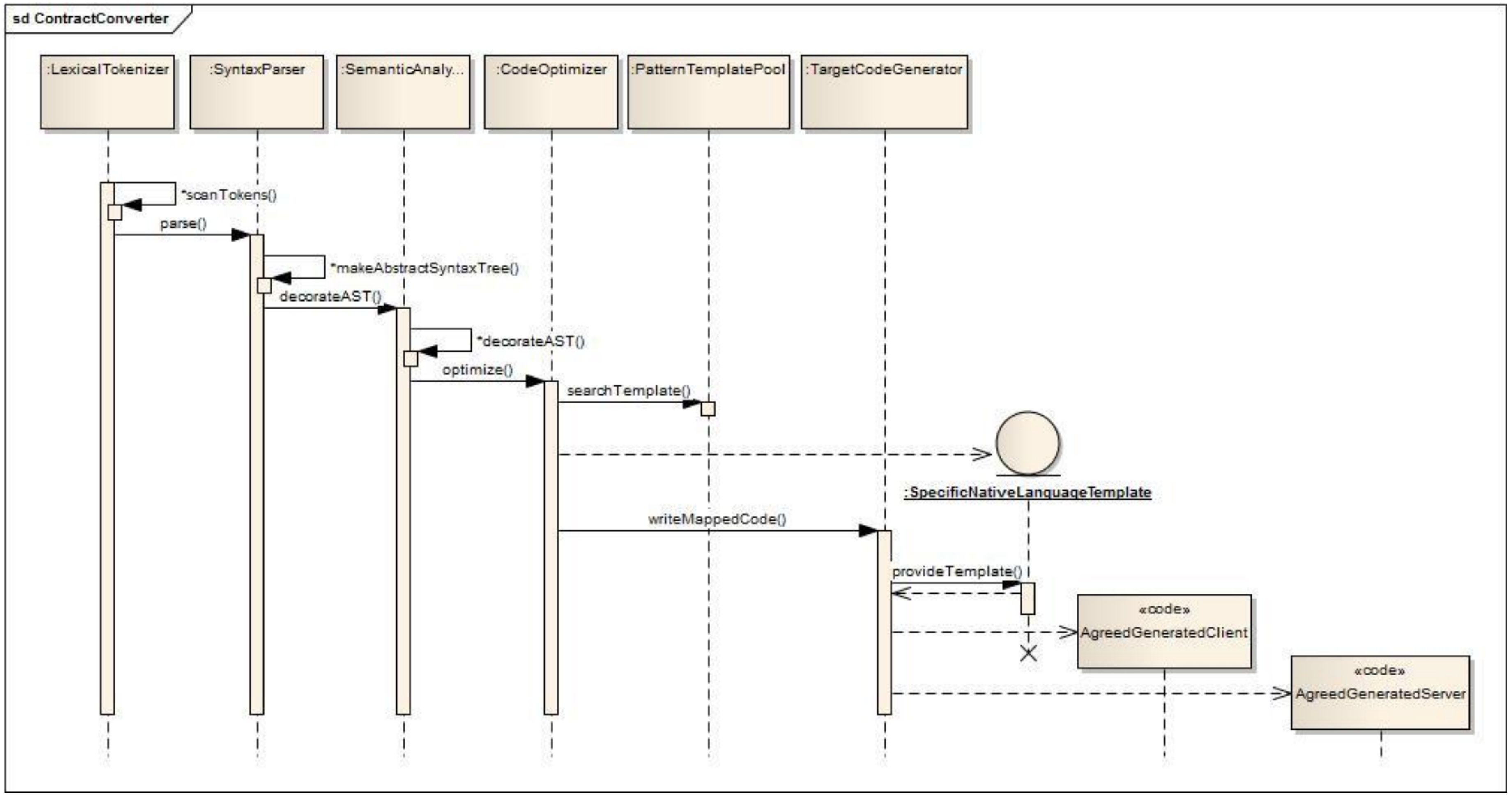

Figure 6 Contract Converter Sequence

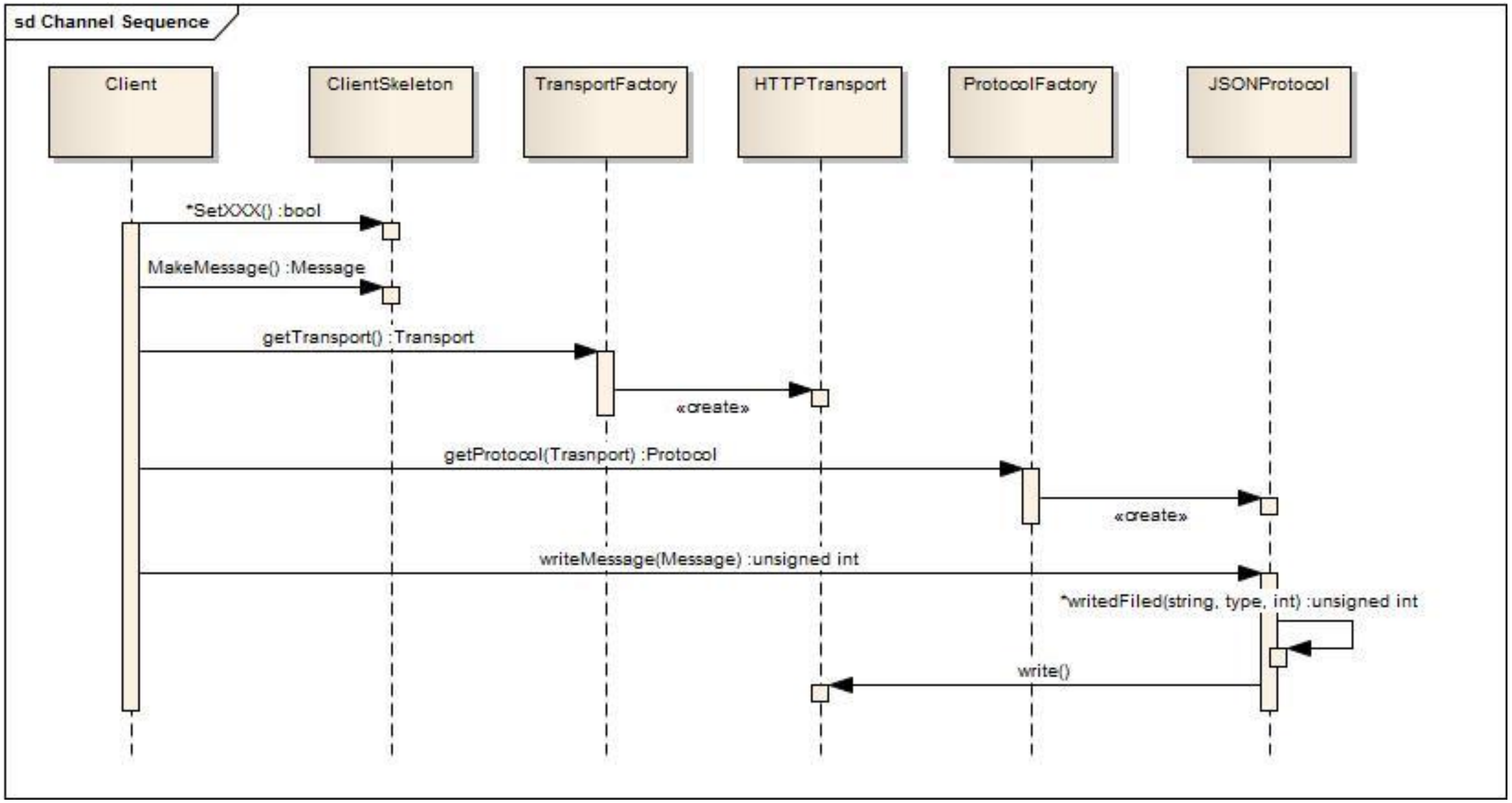

Figure 7 Channel Sequence

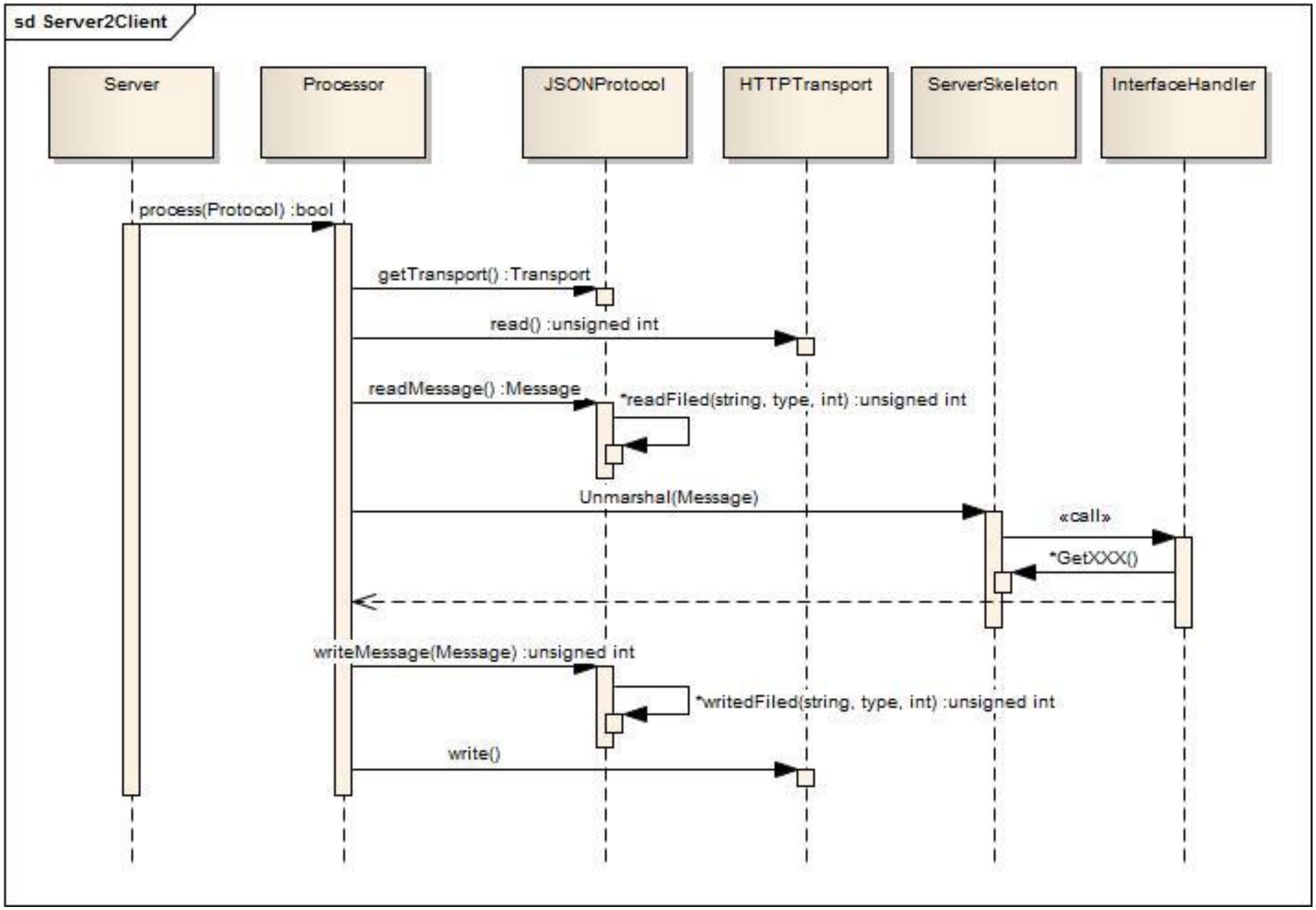

Figure 8 Server Handler Call Sequence