Parsly: Procedureless Protocol Compiler

A Software Engineering Approach to
Simplify Protocol Development Process

Han Zheng
tim.zheng@hivechat.org

December 4, 2019

1 Introductions

1.1 The Cycle of Protocol Development

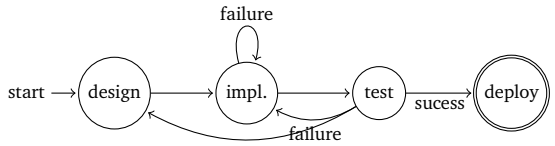

Figure 1.1: development cycle

To better illustrate the process of protocol development, we divide it into four phases in a cycle: design, implement, test and deploy (see fig 1.1). The devloper first need to design the message types and message fields of the protocol, and then implement the protocol with a programing language. Then the test phase comes, where the developer verifies that the implementation is correct. If the implementation is correct, the protocol is deployed, and the development can move on. Otherwise, the developer needs to go back and find out whether the problem lies in the design or the implementation. The implement phase usually turns out to be time consuming, as it involves noises like debugging and solving problems in the lower level protocols, like sticky packet of TCP.

1.2 The Goal

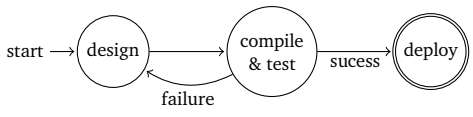

Figure 1.2: new development cycle

Protocol compiler is a tool that helps automate the process of network protocol implementation. It compiles a network protocol discriptions in a domain specific language (DSL), which declares message types and message fields, to a target programing language, like C++, Java and Python. This has saved programmer considerable amount of time by helping them to skip the implementation step, where programmers are prone to problems like nullpointer crash, sticky packet, serialization, and so on. Therefore, protocol compiler simplifies the development cycle into three simple phases (see fig 1.2)

2 Limitation of Current Methods

There are several protocol compilers that exists. The most commonly used one is Google's *Protocol Buffers* (also known as *protobuf*). Protobuf takes in a protocol declaration in their DSL called *proto3* (fig 3.5) and generates the parsing logic in a popular collection of programing languages. During the runtime, *protobuf* uses BSON, the binary representation of JavaScript Object Notation (JSON), to serialize and deserialise the messages. This has given programmers the convenience to handle the data by storing the message in JSON format.

However, there are several drawbacks in *protobuf*. These drawbakcs mainly lies in the aspect of learning curve, performance, and correcness proving with formal verification.

First of all, although *protobuf* has a well-designed DSL that concisely describes the protocol, it is one more language for the programer to learn and one more place to make mistakes. The programers need to read the documentations and do some practices in order to get started, which makes it harder to be used in fast-paced agile developent theme.

Speaking of performance, although *protobuf* did a great job on space optimization by employing BSON for data serialization, BSON is not flexible enough for the users to tunne for optimization. Besides, *protobuf* serializes message field type names into the message, which can be redundant therefore unnecessary. We will talk about this in section 3.1

Moreover, when it comes to proof driven deveolpment, where, in order to prove the correctness of the implementation of the protocol, the developer needs to get the formal specification of the DSL. However, the specificaiton is not officially available, and even if it will be available in the future, it means more work for the developer, as there is one more layer of language to reason about.

3 Parsly

Therefore we introduce parsly, a protocol compiler that compiles protocol descriptions in JSON to C++ parsing logics. Parsley implements a serialization method that is similar to the BSON format but gives user more freedom of space optimizations.

3.1 Design

Parsly is designed to be light-weight and portable, and works on UNIX operating systems as well as Windows. It consists of two parts, a C++ library called *libagio* and a protocol compiler called *pop*. Parsly implements this library (*libagio*), which simplifies parsing logic for the compiler to easily work with. Taking the advantage of the modern C++ template, it is able to do many compile-time optimization for the serialization algorithms. As for the compiler, it is implemented in Python to achieve a high portability and lower cost of maintainance.

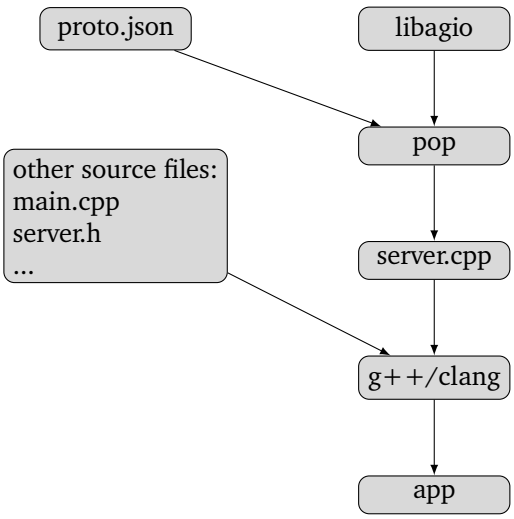

Figure 3.1: build process

To compile a network application, the user needs to go through three steps, see fig 3.1. First, design and write the protocol in JSON, with the attributes and formats provided by parsly. A working example is shown in fig 3.4. Then the user runs *pop* by supplying the JSON protocol description and the target C++ output source file. *Pop* will compile the protocol description into C++ code written with the *libagio*'s API, and inject the code into the parsing function of the source file. The user is then able to run the C++ compiler to build the application.

3.2 Performance Analysis

According to the BSON's documentation, BSON is designed to be efficient in space, but in some cases it uses even more space than JSON because it adds the length indicator of strings and subobjects which makes traversal faster.

```
document :=int32 e_list "\x00"
  e_list :=element e_list
          | ""
 element :="\x01" e_name double
          | "\x02" e_name string
          | "\x03" e_name document
          | "\x04" e_name document
          | "\x05" e_name binary
          | ...
  string :=int32 (byte*) "\x00"
  binary :=int32 subtype (byte*)
 subtype :="\x00"
          | "\x01"
          | "\x02"
          | ...
```

Figure 3.2: BSON standard v1.1 grammar

BSON uses 32-bit integer (int32) to indicate the length of the strings, binaries, as well as the document, and this is fixed by the specification, see fig 3.2. Therefore, no matter how long the string, binary or the document is, the length of the bytes that describes the data is fixed. This has left no space for user to optimize for space. For a message containing massive short strings, BSON will not perform well at all.

Parsly employs the same method for data serialization for performance with the difference that the length indicator size can be controlled by the user in the JSON description. This is achieved by extending the indicator ϕ to multiple sized data types like int8,

$$
\begin{aligned}
M &:= \phi \text{ e\_list} \\
\phi &:= \text{int8} \\
&\mid \text{int16} \\
&\mid \text{int32} \\
\text{e\_list} &:= \text{element e\_list} \\
&\mid \text{""} \\
\text{element} &:= \text{unscoped} \\
&\mid \text{scoped} \\
\text{unscoped} &:= \text{int8} \\
&\mid \text{int16} \\
&\mid \text{int32} \\
&\mid \text{int64} \\
&\mid \text{int128} \\
\text{scoped} &:= \phi \text{ (byte*)}
\end{aligned}
$$

Figure 3.3: parsly serialization grammar

int16 and int32, see fig 3.3.

As might be easily noticed from the figure, parsly does not serialize the name of any data types. To save even more space, parsly ignores all the type names in serialization process. It only judges a message field to be of two types: scoped or unscoped. The scoped field is a section of field guarded by a length indicator, while the unscoped field is of the fixed-sized basic types like int8, int16 and int 32. This not only saves the space to store the type names and the length indicators of the type names, but also makes the deserialization much faster because the mapping from the message field to the data type is already decided at the compile time by parsly.

Therefore, by tuning the size of length indicator of the scoped field and compiletime type erasure, parsly can theoretically achive a higher space efficiency and time efficiency than Google's *protobuf* (benchmark is not the subject of this paper).

4 Conclusion and Limitations

To automate the process of implementation of protocol, we developed parsly, the protocol compiler. Parsly is similar to the existing protocol compiler *protobuf*, but it has several advantages in terms of performance and usability. Parsly has a higher space efficiency, which is achieved by user tunable data length indicator and erasure of message field data types. More over, it replaced *protobuf*'s DSL for protocol description by JSON, which is more flexible in terms of formal reasoning in proof driven development. However, such a pre-compiler has the limit rage of target programing language, as it depends on a C++ library to provide a convenient parsign logic interface. This disadvantage can be overcomed by re-implementation on other programing languages that supports generics.

```
{
   "version" : [0, 1, 0],
   "protocol" : {
      "header" : 4,
      "flags" : ["Request", "Response"]
   },

   "fields" : {
      "Request" : [
         ["string", "query"],
         ["uint32", "page_number"],
         ["uint32", "result_per_page"],
      ],

      "Response" : [
         ...
      ],
   }
}
```

Figure 3.4: parsly protocol description in json

```
message Request {
  string query = 1;
  int32 page_number = 2;
  int32 result_per_page = 3;
}

message Response {
 ...
}
```

Figure 3.5: protobuf protocol description in proto3