Parsly: Procedureless Protocol Compiler

A Software Engineering Approach to
Simplify Protocol Development Process

Han Zheng
tim.zheng@hivechat.org

December 3, 2019

1 Introductions

1.1 The Cycle of Protocol Development

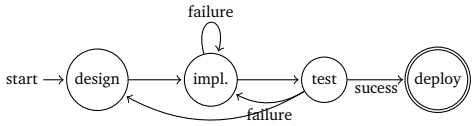

Figure 1: development cycle

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

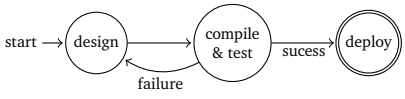

Figure 2: new development cycle

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* (see fig 3.1) 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. We will talk about this in section 3

Moreover, when it comes to proof driven development, 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 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. BSON adds the length 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: BSON standard v1.1 grammar

BSON uses 32-bit integer to indicate the length of the strings, binaries, as well as the whole document, and this is fixed by the specification (see fig 3.1). 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.

$$
\begin{aligned}
M &:= \phi\ \psi \\
\phi &:= \text{int8} \\
&\quad \mid \text{int16} \\
&\quad \mid \text{int32} \\
\psi &:= \text{element}\ \psi \\
&\quad \mid \text{""} \\
\text{int} &:= \text{int8} \\
&\quad \mid \text{int16} \\
&\quad \mid \text{int32} \\
&\quad \mid \text{int64} \\
&\quad \mid \text{int128} \\
\text{binary} &:= \phi\ (\text{byte*}) \\
\text{element} &:= \text{int} \\
&\quad \mid \text{binary}
\end{aligned}
$$

Figure 4: parsly serialization grammar

Parsly employs the same method for data serialization for performance, however, 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, int16 and int32, see fig 3.1 (notice that parsly does not serialize the name of any data types).

4 Analysis

5 Conclusion

```
message Person {
  required string name = 1;
  required int32 id = 2;
  optional string email = 3;

  enum PhoneType {
    MOBILE = 0;
    HOME = 1;
    WORK = 2;
  }

  message PhoneNumber {
    required string number = 1;
    optional PhoneType type = 2 [default =
        HOME];
  }

  repeated PhoneNumber phone = 4;
}
```

Figure 5: protocol described in proto3 language