Politecnico di Milano - Department of Electronics, Information and Bioengineering

POLITECNICO
MILANO 1863

BarbequeRTRM
Google Protocol Buffer based communication

[Coding Project]

Student Fabio Codiglioni
ID 919897

Course Advanced Operating Systems
Academic Year 2018-2019

Advisor Giuseppe Massari
Professor William Fornaciari

September 2, 2019

Contents

1 Introduction

1.1 Problem statement

Refactoring of the BarbequeRTRM ↔ RTLib communication infrastructure, basing it on *Google Protocol Buffer*. Possibly, make a performance comparison with the current FIFO-based implementation.

1.2 Summary of the work

- Definition of a `proto` file holding all the *Protocol Buffer* declarations.
- Declaration of the additional configuration options in a `KConfig` file.
- Refactorization of the pre-existing code and CMake files to accomodate such additional configuration.
- Separation of the pre-existing implementation and the future one into separate directories.
- Implementation of the RTLib side of the infrastructure.
- Implementation of the Barbeque side of the infrastructure.

2 Design and implementation

Before describing all the implementation choices, it's worth noting that *Protocol Buffers* are just a mechanism aimed to ease the process of serializing structured data. For this reasons, the actual communication channel – i.e., the underlying FIFO – was left untouched.

2.1 Configuration

On of the first step was to add the configuration option – accessible via `make menuconfig` – that allows to choose between the old *FIFO-only* implementation and the new *Protobuf-and-FIFO* one. Then the original code underwent a reorganization, so that the files of the two implementations were separated into different directories. This was followed by a tweak in the CMake files, in order to include the correct directories in the build process.

2.2 Messages declaration

The pre-existing code featured a series of native C-style `structs` used to represent each type of message and some `enums` to restrict the domain of some fields. The adopted approach was to translate all the message-type `structs` into a single protobuf `message`, in order to limit the number of types declared[1]. This was suggested also by the specific serialization technique of the adopted version 3 of the framework: scalar fields are only explicitly serialized if their value is different from the default one, so the unset fields are not actually transmitted. Separate protobuf `messages` were used to translate the message header and the other message-specific `structs`, e.g., the constraints. The original C-style `enums` were left untouched, and the corresponding fields were translated as unsigned integers.

Since some response types carry additional fields compared to the *standard* response type, an additional `enum` field was added to the protobuf header in order to correctly translate these responses back and forth between `structs` and protobuf classes.

2.3 Implementation

In order to speedup implementation, a *helper* class composed of only static member functions was used, called `PBMessageFactory`. This class was originally conceived to hold all the conversion code from struct to protobuf. This idea has later been dropped because, contrary to expectations, it was generating a lot of repeated code. The only member functions left in the class were the ones related to the header conversion, as the header is present in almost everything that goes through the wire. The actual conversion code was implemented in place, where the pre-existing code was simply handling `structs`.

The message-type structs described in Section 2.2 were not directly transmitted on the wire: a wrapper C-like struct was used to hold the message type, the size of the wrapper struct and of the wrapped one. This approach was left also in the new implementation, as the receiving side must know how much bytes are to be read from the channel. The only difference in the new implementation is that the payload is now represented by an array of `unsigned char` instead of a `struct`, in order to accomodate the serialized protobuf data.

[1] Note that the type of a message is still clearly identifiable by one of the header fields.

Among all the message-type `structs`, there were two which had some nested repeated fields: `EXC_SET` and `BBQ_SYNCP_PRECHANGE`. However, they were treated differently because of how the upper level code behaved. The constraints sent with an `EXC_SET` message were embedded in the general protobuf `message` as a repeated field, while each `SYSTEM` sent with a `BBQ_SYNCP_PRECHANGE` message was sent as an individual message. This is due to the fact that `EXC_SET` is generated by the application side, where each message type has its own sending function, whereas `BBQ_SYNCP_PRECHANGE` is generated by the *barbeque* side, where there is a single sending function, agnostic of the message type, that gets called for each `struct` to be transmitted.

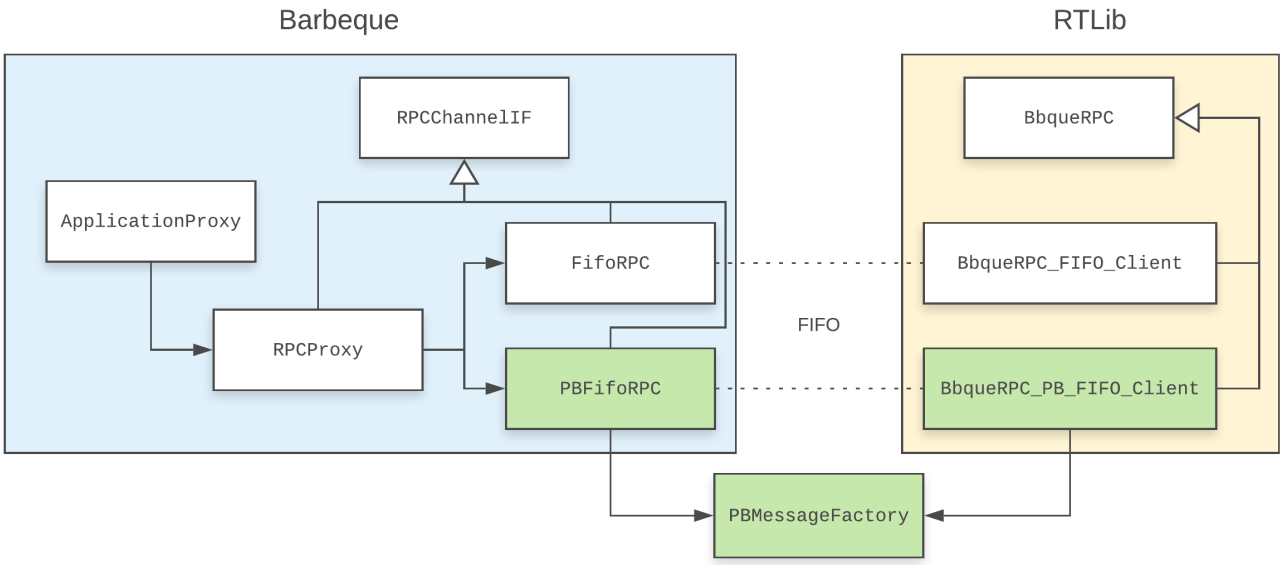

Figure 1: Class diagram. The classes in green are the new ones.

3 Experimental evaluation

The performance evaluation was carried out with the standard UNIX tool `time`. Table 1 contains the experimental values and their average. The slightly higher value of the protobuf average could be purely coincidental or could be due to protobuf data-filling functions requiring more instructions.

FIFO	**Protobuf**
85 ms	78 ms
83 ms	82 ms
80 ms	82 ms
90 ms	85 ms
75 ms	86 ms
72 ms	90 ms
81 ms	90 ms
79 ms	81 ms
77 ms	82 ms
74 ms	76 ms
80 ms	**82 ms**

Table 1: Collection of execution times. The last row contains the average.

4 Conclusions and Future Works

The current implementation suffers from poorly optimized memory usage: the array used to hold the serialized protobuf data is part of the wrapper `struct`, and its size is fixed at compile-time. Since such wrapper `structs` are always declared as local variables, the array is allocated on the stack. The size of the array, 1 KiB, is not a problem for a common x86 machine – the target platform of this project – but it could be for an embedded system. For this reason, a smarter approach would be to separate the array from the wrapper struct and to allocate the array on the heap, sizing it so that it exactly fits the serialized data. The actual transmission would be then broken into two parts: first the wrapper `struct`, then the array.