Using Protocol Buffers for Resource-Constrained Distributed Embedded Systems

Wolfgang Schwitzer
Vlad Popa

TUM-I1120
November 11

Abstract

Protocol Buffers are a widely used, robust and efficient data interchange format contributed and maintained by Google. Specifications of custom messages, fields and enumerations are comfortably defined in the Proto language and then compiled to a large variety of target programming languages like C++, Java and Python. This makes Protocol Buffers an excellent choice when heterogeneous system platforms and programming environments have to communicate with each other. In this paper, we present our compiler Protobuf-Embedded-C that generates C as target language. This compiler generates lean and self-contained C-code for resource-constrained, distributed and embedded real-time systems. We implemented this new compiler from scratch, because an alternative Protocol Buffers compiler targeting at resource-constrained and real-time systems was not available at the moment of writing. We discuss the features and architecture of this compiler that accepts a subset of the Proto language and give an outline of possible future extensions. To show the practicability, we present a successful application of generated Protobuf-Embedded-C code for an E-Energy-Grid demonstrator, developed together with a major German power supplier during the national research joint-project SPES2020.

This document refers to Protobuf-Embedded-C version 1.0M1 (protoc-0.1.jar).

Contents

1 Introduction

This paper describes the application of Google's Protocol Buffers [1] as data interchange format for resource-constrained, distributed and embedded real-time systems. We discuss the development of a compiler [2] that accepts a subset of the features of the Proto language and generates lean and self-contained C-code. In this paper, we give an overview of the architecture and features of this compiler and describe the successful use of generated Protocol Buffers code within the research project SPES2020 [3] for an *E-Energy-Grid* demonstrator.

Protocol Buffers. Protocol Buffers [1] are an efficient, robust and comfortable to use serialization method originally developed by Google. Data structures like messages, fields and enumerations are defined in a ".proto" file written in the so-called *Proto language*. These data structure definitions are read by a compiler and translated to a target programming language. The source code generated by this compiler allows to read and write the defined data structures. A variety of target programming languages is supported by different compilers. Currently, the core Protocol Buffers implementation by Google supports the target programming languages C++, Java and Python.

Why another Generator for C? The code generated for C++, Java and Python depends on libraries that require several 100 kilobytes of memory and for Java and Python, an additional runtime environment or an interpreter is usually required on the target device. Other implementations for the C language with an extensive support of features do exist already (see [4], for example). For some embedded devices with only a view kilobytes of memory, the libraries used by these implementations might be too memory-consuming. If you have to program such a resource-constrained embedded controller in the C language and you only need to use a very reduced number of features of Protocol Buffers, then Protobuf-Embedded-C might be an adequate tool for you. This document presents our Protocol Buffers generator called Protobuf-Embedded-C, which is suitable for resource-constrained embedded applications written in the C programming language. The following requirements drive the development of Protobuf-Embedded-C:

1. Generated C-code runs on low-power and low-memory embedded controllers.

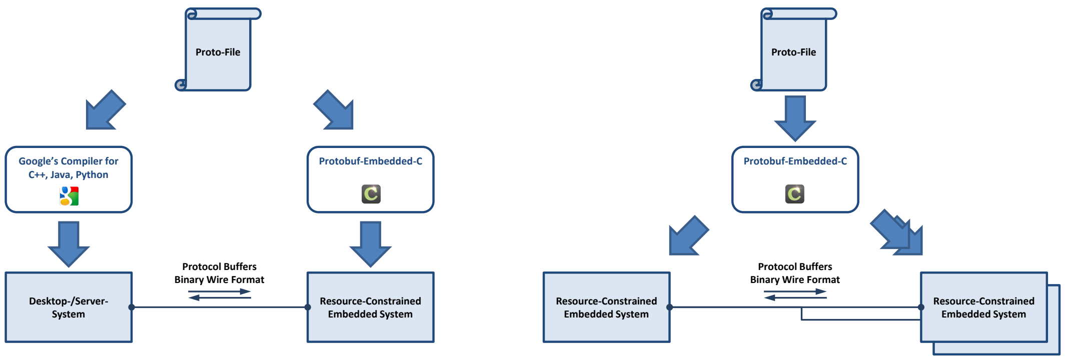
2. Generated C-code runs on real-time systems with static memory allocation.

3. Code is completely generated, so it is self-contained (no libraries on the target device).

4. API is easy to learn and (where possible) close to the concepts of the original Protocol Buffers implementations by Google.

Please note, that the requirements above imply a quite reduced functionality. For example, unbounded *repeated* fields and strings with unbounded length do not match well with requirement (2). However, we believe that the API provided by Protobuf-Embedded-C is still powerful enough to get most of the basic Protocol Buffers communication jobs done.

Typical usage scenarios of Protobuf-Embedded-C. Figure 1 illustrates two typical usage scenarios of Protobuf-Embedded-C. In the scenario depicted in figure 1(a), a general purpose desktop- or server-system communicates with resource constrained embedded systems, using Google's C++, Java and Python compiler on the desktop-/server-side and Protobuf-Embedded-C on the embedded side. In the scenario depicted in figure 1(b), a number of embedded systems communicate directly with each other, using Protobuf-Embedded-C.


Proto-File
Google's Compiler for C++, Java, Python
Protobuf-Embedded-C
Protocol Buffers Binary Wire Format
Desktop-/Server-System
Resource-Constrained Embedded System
Proto-File
Protobuf-Embedded-C
Protocol Buffers Binary Wire Format
Resource-Constrained Embedded System
Resource-Constrained Embedded System

(a) (b)

Figure 1: Typical usage scenarios of Protobuf-Embedded-C: (a) connect desktop- or server-systems with embedded systems; (b) connect several embedded systems directly to each other.

Outline of this paper. This paper is organized as follows. Section 2 presents a quick introduction for Protobuf-Embedded-C users. Section 3 describes the list of features supported so far. This section is followed by a presentation and overview of the internal architecture and implementation of the Protobuf-Embedded-C compiler in section 4. The automated testing procedure of these features and the quality assurance aspect of Protobuf-Embedded-C is explained in section 5. Afterwards, section 6 presents how Protobuf-Embedded-C has been successfully integrated in an E-Energy-Grid demonstrator as a case example. Planned and optional future extensions are sketched in section 7. Finally, this paper concludes with some remarks in section 8.

2 Quick Start Guide

This section is a quick start guide for using Protobuf-Embedded-C, which covers the following basic topics:

1. Downloading and using the compiler.
2. Understanding the generated API.
3. Using the generated API.

We follow a small running example adapted from the original "phone number example" on Google's Protocol Buffers site [1].

Download the compiler and compile a first example. Download the most recent version of the "protoc-<version>.jar"-file from the *Downloads* section of the Protobuf-Embedded-C project-site [2]. At the time of writing, this is "protoc-0.1.jar", which is ready to be executed on the command line as a *Java-archive*. Suppose there is a "phone.proto"-file that looks like this:

```
//File: phone.proto
//Optional Embedded C internal options:
//@max_repeated_length=100
//@max_string_length=32

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

message PhoneNumber {
      repeated int32 number = 1;
      required string person = 2;
      required PhoneType type = 3;
}
```

This example solely servers the purpose of demonstrating the key features of Protobuf-Embedded-C. A "real" application would use a much more elaborate data model, of course. To generate the C-code that allows for reading and writing "PhoneNumbers", the following command line needs to be executed with *Java (1.6+)*, while *protoc-0.1.jar* and *phone.proto* need to be in path:

```
java -jar protoc-0.1.jar phone.proto
```

Understanding the generated API. Now, the files *phone.c* and *phone.h* are generated next to *phone.proto*, where *phone.h* provides you with the API to deal with PhoneNumber messages:

```
1  /* Excerpt from generated file: phone.h */
2
3  #define MAX_REPEATED_LEN 100
4  #define MAX_STRING_LEN 32
5
6  enum PhoneType {
7      _MOBILE = 0,
8      _HOME = 1,
9      _WORK = 2
10 };
11
12 /* Maximum size of a serialized PhoneNumber-message, useful
13 for buffer allocation. */
14 #define MAX_PhoneNumber_SIZE 1240
15
16 /* Structure that holds deserialized PhoneNumber-message. */
17 struct PhoneNumber {
18     int _number_repeated_len;
19     signed long _number[MAX_REPEATED_LEN];
20     int _person_len;
21     char _person[MAX_STRING_LEN];
22     enum PhoneType _type;
23 };
24
25 int PhoneNumber_write_delimited_to(struct PhoneNumber
26                     *_PhoneNumber, void *_buffer, int offset);
27
28 int PhoneNumber_read_delimited_from(void *_buffer,
29                     struct PhoneNumber *_PhoneNumber,
30                     int offset);
```

In line 14 of phone.h the maximum size of the serialized message is defined. It is computed internally with the help of MAX_REPEATED_LEN and MAX_STRING_LEN which are defined in the annotations at the beginning (lines 3, 4) of the phone.proto file. For example, this size is useful for static allocation of buffers, which could contain the specified messages. In our example, a size of 1240 bytes to store a phone number seems to be quite large, so you can experiment with reducing the maximum length of repeated fields from 100 to a lower number in line 3 of phone.proto. Please note, that this maximum size, which is calculated for each message, is just the "worst-case" and the sizes of transmitted serialized messages are often significantly smaller, due to the Protocol Buffers packed binary format.

The following structure *PhoneNumber* holds the deserialized message. The variables *_number_repeated_len* and *_person_len* contain the number of elements in the array *_number* and *_person*. Hence, *_number_repeated_len* $\leq$ MAX_REPEATED_LEN and *_person_len* $\leq$ MAX_STRING_LEN must hold.

Using the generated API. For each message M two methods, *M_write_delimited_to* and *M_read_delimited_from*, are generated. In this case

- *PhoneNumber_write_delimited_to* writes a struct of type PhoneNumber into a buffer starting at a given offset and
- *PhoneNumber_read_delimited_from* reads a struct of type PhoneNumber from a buffer starting at a given offset.

The following code fragment shows an example of how these methods can be used:

```c
#include <stdio.h>
#include "phone.h"

//the maximum number of phone numbers that can be received
#define PNUMBER_SIZE 256

static struct PhoneNumber pnumber[PNUMBER_SIZE];
static int count = 0;

// read all PhoneNumber messages from _buffer_read
void read(void *_buffer_read) {
    int old_offset = 0, offset = 0;

    while (*((char*)_buffer_read + offset) != 0) {
        offset = PhoneNumber_read_delimited_from(
                    _buffer_read,
                    &pnumber[count++],
                    old_offset);
        old_offset = offset;
    }
}

// print all PhoneNumber messages
void print() {
    int i, j;

    for (i = 0; i < count; ++i) {
        printf("_number_repeated_len: %d\n",
                pnumber[i]._number_repeated_len);
        for (j = 0; j < pnumber[i]._number_repeated_len;
                                                    ++j) {
            printf("%ld", pnumber[i]._number[j]);
        }

        printf("\n_person_len: %d\n",
                pnumber[i]._person_len);
        for (j = 0; j < pnumber[i]._person_len; ++j) {
            printf("%c", pnumber[i]._person[j]);
        }

        printf("\nPhone Type: %d\n", pnumber[i]._type);
    }
}

// write all PhoneNumber messages to _buffer_write
void write(void *_buffer_write) {
    int i, old_offset = 0, offset = 0;

    for (i = 0; i < count; ++i) {
        offset = PhoneNumber_write_delimited_to(&pnumber[i],
                      _buffer_write, old_offset);
        old_offset = offset;
    }
}
```

3 Supported Protocol Buffers Features

Protobuf-Embedded-C supports several core features of the original Proto language. Some of the features are not supported, because they do not match with the design goal of statically bounded buffer allocation (unbounded repeated messages or string lengths, for example). Some other features like the support of all data types and optional fields are missing in the first version and are scheduled to be implemented in future releases.

Currently supported features of the Proto language. The so-called *Proto language* is used in ".proto" files to describe the structure of messages, exchanged in the Protocol Buffers binary format. For an extensive documentation of the Proto language, please visit Google's Protocol Buffers homepage [1]. The following paragraphs summarize the language features supported in Protobuf-Embedded-C version 1.0M1 and give a BNF-like [5] formalization of the syntax, where appropriate:

Enumerations: an enumeration is introduced by the keyword `enum` followed by a unique enumeration identifier and a list of enumeration elements. The identifier of an enumeration element as well as its integer index must be unique within the defining enumeration. A valid index is an integer number within a range of $[0 \ldots 127]$:

$$enumDecl ::= \texttt{enum}\ id\ \{\ enumElement * \ \}$$

$$enumElement ::= id\ \texttt{=}\ index\ \texttt{;}$$

Messages: a message is introduced by the keyword `message` followed by a unique message identifier and a list of message elements. The identifier of a message element as well as its integer tag must be unique within the defining message. Valid tags are integer numbers within a range of $[1 \ldots 4095]$:

$$messageDecl ::= \texttt{message}\ id\ \{\ messageElement * \ \}$$

$$messageElement ::= modifier\ (type \mid id)\ id\ \texttt{=}\ tag\ \texttt{;}$$

Modifiers: the modifiers `required` and `repeated` are supported at the moment. The modifier `optional` is scheduled for a future release:

$$modifier ::= \texttt{required} \mid \texttt{repeated}$$

Elementary types: the elementary types `float`, `int32`, `bool` and `string` are supported at the moment. More extensive support of data types is scheduled for future releases:

$$\textit{type} ::= \texttt{float} \mid \texttt{int32} \mid \texttt{bool} \mid \texttt{string}$$

Custom types: if a message element is defined using the identifier of a user defined type instead of an elementary type (*id* instead of *type*), solely references to enumerations are supported at the moment. Support of references to other messages or the containing message itself (recursive definition) are scheduled for future releases.

Additional embedded-related features. Two additional language features, which are related to resource-constrained embedded systems, are introduced by the Protobuf-Embedded-C compiler. The maximum length of repeated message elements and the maximum length of strings can be constrained globally by using the declarations

- `//@max_repeated_length` and
- `//@max_string_length`.

An example of how to use these declarations is found in the code block on page 6 in lines 3 and 4. Note, that both declarations are planned to be available locally for each field in future releases.

Some notes on currently unsupported features. Two major features of the original Protocol Buffers compiler are currently not supported. First, compilation and generation of services with remote procedure calls is not supported. Second, compilation of packages and name resolution using multiple ".proto" files is not supported. Though service oriented architectures for resource-constrained embedded applications would be an exciting feature to add, first investigations showed that this could be quite complex to implement with respect to Protobuf-Embedded-C's design goals. Further research is necessary, to find out whether a subset of the Protocol Buffers's service and remote procedure call functionality can effectively and efficiently be realized in self-contained C-code for resource-constrained embedded systems.

4 Compiler Architecture

Protobuf-Embedded-C's compiler generates C-code, nevertheless, it is mainly written in the Java programming language. Large parts of the compiler's code are automatically generated by using the parser generator ANTLR [6, 7]. The output of files in the target language C is driven by the template language StringTemplate [8]. Actually, the main file "Protoc.java", which starts and coordinates the different compiler stages, has only about 100 lines of code. This section explains the architecture of Protobuf-Embedded-C's compiler and is especially interesting from a developer's point of view.

The ANTLR parser generator. ANTLR is a popular parser generator, which is available and continuously maintained since many years [7]. ANTLR is well documented [6] and generates robust and efficient lexer- and parser-implementations. These lexers and parsers can be generated in several languages, including Java, which is the main programming language we use for Protobuf-Embedded-C. As a remark concerning language theory, ANTLR generates parsers that can recognize languages belonging to the $LL(*)$ class of grammars, which is sufficient for many practical languages and so it is for recognizing the Proto language.

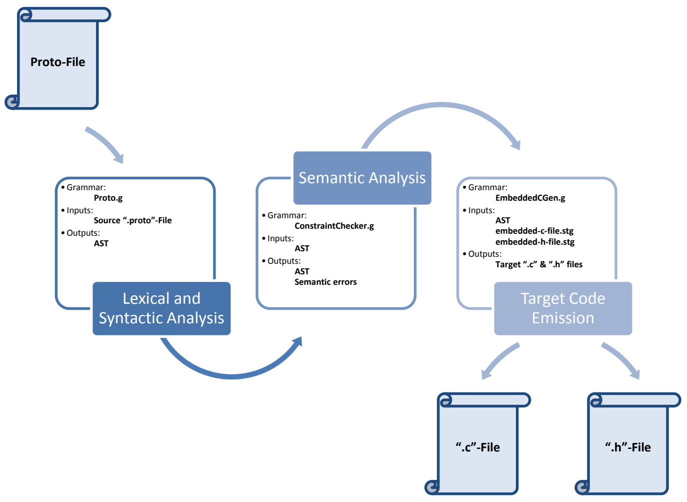


Figure 2: The compiler pipeline architecture and its stages.

Compiler pipeline architecture. A recent feature of ANTLR is its ability to construct modular staged compilers. In this approach, stages work directly on the initially parsed *abstract syntax tree* (AST), which is then recognized by so-called *tree grammars*. We decided to employ this modular staged compiler architecture for Protobuf-Embedded-C, using three separate stages for the different tasks *lexical and syntactic analysis*, *semantic analysis* and *target code emission*. This modular compiler pipeline architecture is shown in figure 2. Each stage of the compiler is specified in a dedicated ".g" grammar specification file. First, the abstract syntax tree is parsed by a combined lexical and syntactic analysis stage. Subsequently, the semantic analysis stage and the code emission stage are specified by *tree grammars* that directly operate on the abstract syntax tree.

Lexical and syntactic analysis. The lexical and syntactic analysis recognizes the Proto language, which is defined in the grammar file "Proto.g". Supported grammar rules of this language are explained in section 3. Input to the lexical and syntactic analysis is a ".proto"-file and the output is an abstract syntax tree of this file.

Semantic analysis. Semantic analysis is defined in the grammar file "ConstraintChecker.g". The semantic analysis checks namespace and integer range constraints. For example, the names of enumerations and messages must be unique within one ".proto"-file and tags of fields in messages must be within a range of $[1 \ldots 4095]$. Input to the semantic analysis is an abstract syntax tree and the outputs are semantic error messages if any errors are found.

Target code emission. Target code emission is defined in the grammar file "Embedded-CGen.g". In this last stage of the compiler, the ".c" and ".h" files are generated and written out. Input to this stage is an abstract syntax tree, which drives the execution of several templates, defined in the two template files "embedded-c-file.stg" and "embedded-h-file.stg". Outputs of this stage are the respective ".c" and ".h" implementation- and header-files, which contain the read- and write-code for messages specified in the original ".proto"-file.

5 Quality Assurance and Test Automation

As communication protocols are crucial for the stability of distributed applications, we believe that having a wide test case coverage by automated test generation and execution is crucial for the quality of the Protobuf-Embedded-C compiler. Therefore, we use automated "in-the-loop" testing, which covers an extensive set of equivalence class tests, robustness tests and a high amount of randomized tests.

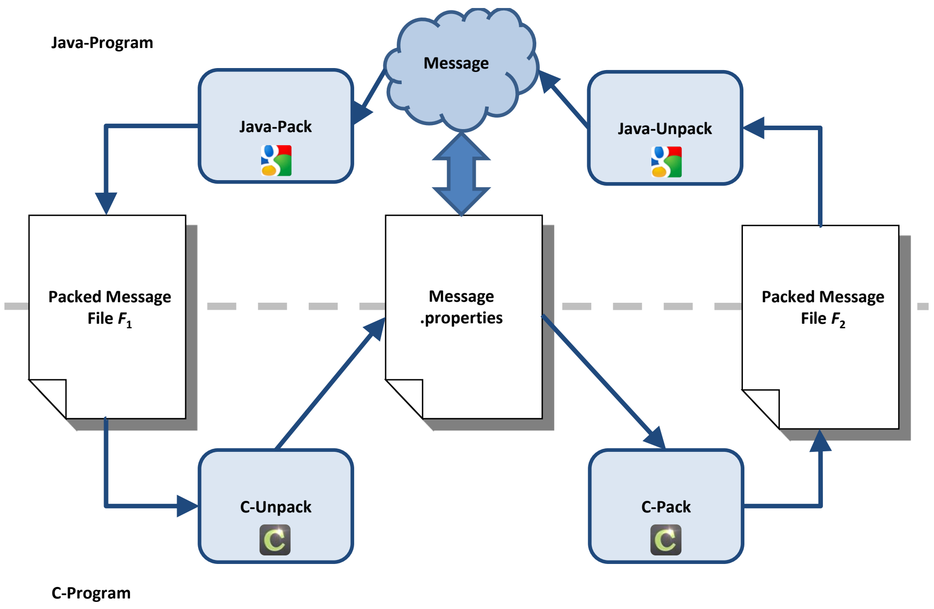


Figure 3: A message on its way along the testing loop.

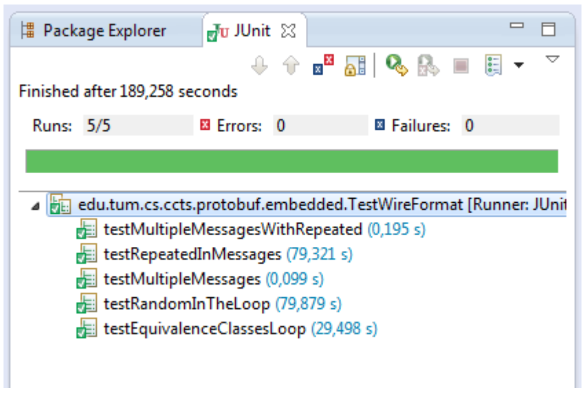
Test setup. In the testing phase we verify the correctness of results produced by Protobuf-Embedded-C with the help of Google's Java Protocol Buffers compiler as a reference implementation. At the beginning of a test case, a message is packed and written to a file F_1, using the generated code of the Java compiler. Subsequently, generated C-code of Protobuf-Embedded-C unpacks the message stored in file F_1 and writes down its interpreted contents in a "*.properties*"-file. In a following step, the C-code packs the message again and writes it to a file F_2. Finally, the contents of file F_2 are unpacked by the Java generated program. Now, the Java program is able to compare the original message with the unpacked message of F_2 and the message stored in the "*.properties*"-file. Figure 3 schematically shows the way of a message along this testing loop. In each such testing loop, two conditions are verified:

- The C-program sees the same message as the Java-program. This is tested with the help of the ".properties"-file.
- The Java-program finally receives the same message which has been initially sent.

A typical test run. A complete test run executes the following test cases:

- For each supported data type (int32, string, bool, float, enum) each possible combination is tested (equivalence class tests).
- All supported data types are tested again with 1000 random tests.
- The repeated feature of all supported data types is evaluated by 1000 tests of random messages.
- The testing of 100 messages without the "repeated" modifier packed in a single buffer.
- The testing of 100 messages with the "repeated" modifier packed in a single buffer.

The results of a typical test run on a Intel(R) Core(TM)2 Duo CPU with 2.4GHz can be seen in picture 4. A testing cycle with 100 messages in one buffer takes somewhere between $0.1s$ and $0.2s$. The equivalence class tests contain 360 individual testing loop executions, which are executed in a total of $\approx 30s$. Additional 1000 testing loops with random messages take a total of $\approx 79s$.



```
Package Explorer    JUnit

Finished after 189,258 seconds

Runs: 5/5      Errors: 0      Failures: 0

edu.tum.cs.ccts.protobuf.embedded.TestWireFormat [Runner: JUnit
    testMultipleMessagesWithRepeated (0,195 s)
    testRepeatedInMessages (79,321 s)
    testMultipleMessages (0,099 s)
    testRandomInTheLoop (79,879 s)
    testEquivalenceClassesLoop (29,498 s)
```

Figure 4: A typical test run is executing several thousands of single test cases and takes more than three minutes.

Conclusion. Quality with respect to functional correctness and robustness is an important aspect of a protocol compiler like Protobuf-Embedded-C. We plan to make integration tests with a wider range of embedded platforms. This means the generated C-code will be tested "in-the-loop" on different embedded controllers. For example, Protobuf-Embedded-C has been tested for an ARM Cortex M3 CPU in the "E-Energy-Grid Demonstrator" case example presented in section 6.

6 Case Example: The E-Energy-Grid Demonstrator

In this section we show a practical application of Protobuf-Embedded-C. We employ Protocol Buffers as wire format over TCP/IP in an E-Energy-Grid demonstrator which was build during the SPES2020 research project [3]. On the one hand, this demonstrator simulates a number of interactive clients that represent volatile energy producers and consumers. On the other hand, this demonstrator is connected with a so-called *E-Energy Server*. This server constantly collects data from the consumers and producers, minimizes the cost in a linear inequation system (based on collected data and business database information) and sends commands for the resulting optimal energy-mix to the clients. Each of the simulated clients is running on a C-programmed ARM Cortex M3 platform [9] having less than 40KB statically allocatable memory per device. In contrast, the server is running on a standard x86-based and Java-programmed PC with several GB of memory.

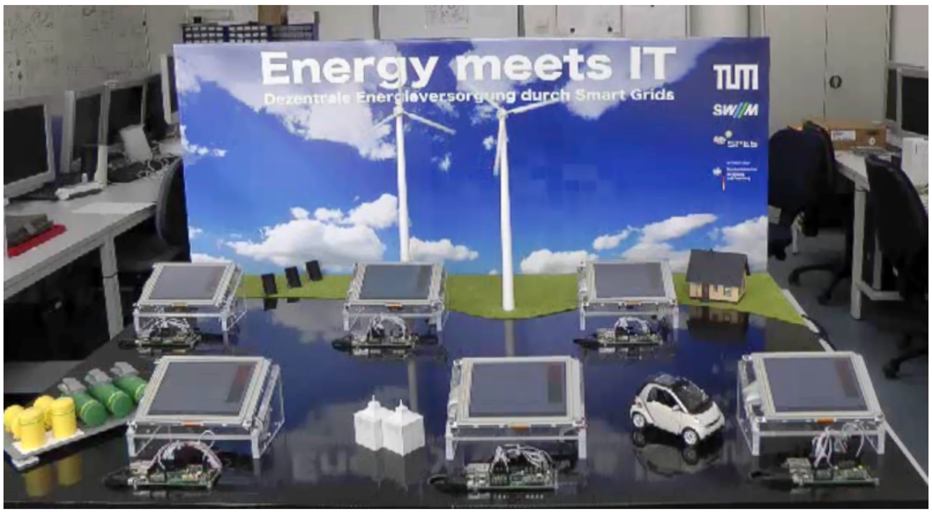


Figure 5: The SPES2020 E-Energy Grid Demonstrator.

SPES2020 and the E-Energy-Grid demonstrator. The national research joint-project SPES2020 "Software Platform Embedded Systems 2020" is dedicated to software and systems engineering for embedded systems from the industrial domains automation, automotive, avionics, energy and medical technology [3]. Within the energy domain, the SWM (a major German power supplier) together with the Technische Universität München developed an E-Energy-Grid control concept, based on a combination of distributed embedded software systems and a central business-logic server. In this concept, it is the server's responsibility to calculate a minimum cost energy-mix for the grid, so that the energy that is consumed and produced during the next 15 minutes remains in balance. To show the practicability of this concept, we built a demonstrator that simulates a small-scale E-Energy-Grid as depicted in figure 5. This demonstrator simulates two kinds of energy consumers (households and e-cars) and four kinds of distributed energy producers (photo-voltaics, wind-turbines, bio-gas plants and block thermal power stations). The consumers and producers constantly exchange monitoring and control messages with the server. We decided to use Protocol Buffers as communication protocol "on the wire" between these quite different classes of systems and programming environments. Note, that we deliberately did not use protocols like e.g. IEC 61870-5-104 (virtual power plant automation) here.

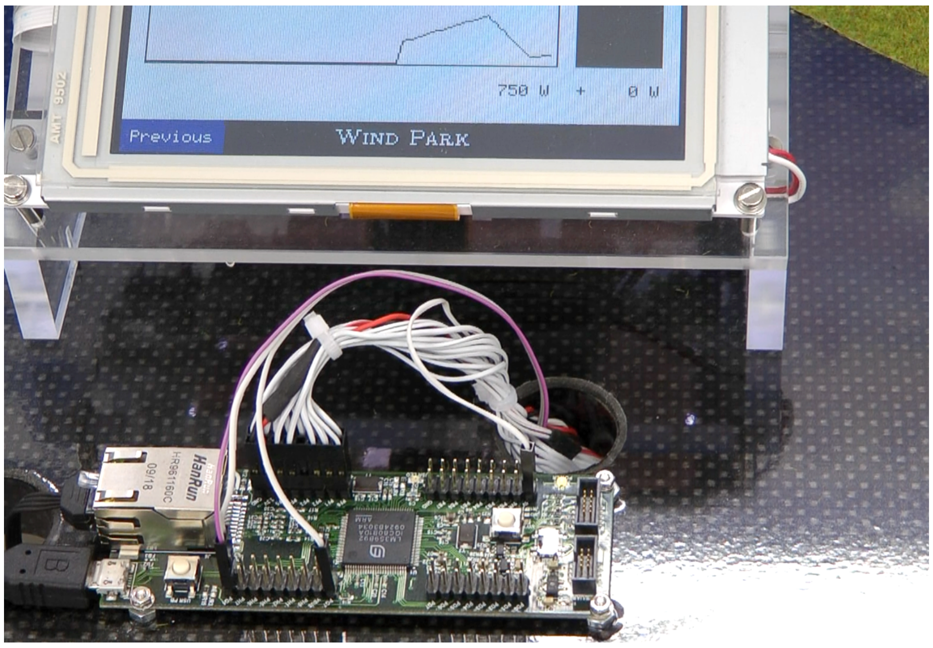


Figure 6: ARM Cortex M3-based embedded controller with touchscreen.

Distributed ARM Cortex M3 microcontrollers. The consumers and producers are simulated by ARM Cortex M3 microcontrollers, each connected to a small touchscreen for visualization and interactive user input as shown in figure 6. A standard ethernet network connection with a 100 MBit/s switch and TCP/IP transport protocol is used. With this system configuration, each controller has less than 40KB of memory available for application use. We use static memory allocation and static data structure sizes to guarantee a worst case execution time for the simulated consumer's or producer's local control loop. This is an example where it is important for Protobuf-Embedded-C to generate code that can be used in the context of static memory allocation and statically bounded buffer sizes.

Application of Protobuf-Embedded-C. Figure 7 shows the *topology* of the demonstrator's switched ethernet network. Note, that the depicted model in figure 7 is a rather coarse-grain

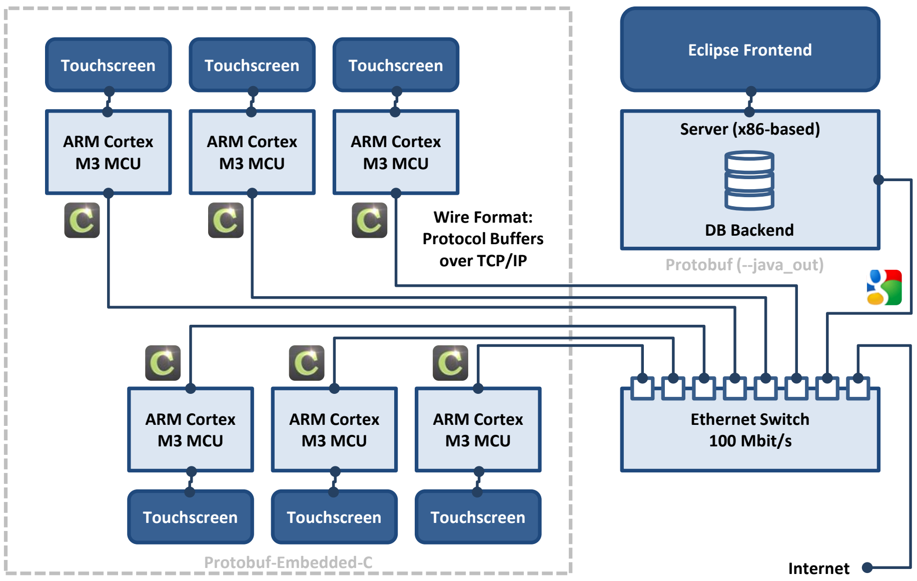


Figure 7: Switched ethernet network topology.

abstraction of the system's *technical architecture* (compare to [10]). Each of the six ARM Cortex M3 MCUs (inside the gray dashed box on the left hand side) is connected directly to a switch by a standard ethernet cable with RJ45 sockets. The x86-based server PC is connected to the switch as well. In this scenario, the same ".proto" file is re-used to

- generate C-code for the ARM controllers with Protobuf-Embedded-C and to
- generate Java-code for the server with Google's standard protoc compiler.

Whenever we needed to add fields to the transmitted messages during development of the demonstrator, this could conveniently be done by changing the ".proto"-file once and re-generating the protocol code for the different platforms subsequently.

Conclusion. We re-used the same ".proto"-file to generate C-code for the embedded controllers with Protobuf-Embedded-C and to generate Java-code for the server system with Google's standard protoc compiler. This allowed for flexible modifications of the protocol during the development phase. We were able to tune the sizes of messages by evaluating the resulting "MAX" message size constants in the generated code (see section 2). The generated C-code was compact and required only a few kilobytes of memory on the embedded client side. In the course of the project, we steadily increased the coverage of our automated test-suite for Protobuf-Embedded-C (see section 5). Finally, the E-Energy-Grid demonstrator could run for several hours, exchanging thousands of messages packed and unpacked by Protobuf-Embedded-C and Google's Java implementation, without observable errors caused by the generated protocols.

7 Future Work

At the moment of writing, Protobuf-Embedded-C is published as version 1.0, Milestone 1. This first version offers some basic features of the original Protocol Buffers implementation, yet still some important or convenient features are missing. Depending on feedback from users of Protobuf-Embedded-C, the most pressing feature extensions are summarized in this section.

Supporting more data types. More data types such as: *double*, *bytes*, *int32*, *uint32*, *uint64*, *fixed32* and *fixed64* will be supported beginning with 1.0M2. Note that depending on the embedded platform, floating point operations (especially on 64bit double fields) might be emulated and can be comparatively inefficient.

Supporting optional fields. The "*optional*" modifier is planned to be available beginning with 1.0M3. This feature also makes it necessary to implement the "default" values of optional fields.

Supporting a maximum repeated length and string length per field. In the current version of Protobuf-Embedded-C the "max_repeated_len" and "max_string_len" annotations globally define the maximum allowed length for all repeated fields and strings. We plan to make this more customizable on a per-field base, as repeated fields and strings may strongly vary in maximum length. This improvement, which is planned for 1.0M3, allows for refined tuning of maximum message sizes for static buffer allocation.

Platform specific testing procedure. The current testing procedure is presented in section 5. It is basically an integration test of Protobuf-Embedded-C with Google's Protocol Buffers Java implementation, both executed on a x86 platform. In future versions, the test automation of Protobuf-Embedded-C will include integration tests with a wider range of platforms. Because of the different architectures and tool-chains, the same generated code may produce various (and sometimes unexpected) outputs on different microcontrollers.

Providing an error handling API. Future versions should include an interrupt-safe error handling API to signal runtime errors during read- or write-operations.

Leaner compiler binary. The compiler binary "protoc-0.1.jar" has a size of about 1.8 MB, which can be reduced significantly in future versions by including only those parts of ANTLR, that are actually used by the compiler.

8 Concluding Remarks

We developed Protobuf-Embedded-C during the research project SPES2020 from scratch, because no suitable implementation for the C programming language that would run on our ARM Cortex M3 based systems was available. At the moment of writing, Protobuf-Embedded-C is available as the first milestone of version 1.0, which is just the first publication of source code under the Apache 2 license. Within a few weeks, Protobuf-Embedded-C 1.0M1 has been downloaded more than 50 times.

Design goals and trade-offs. On the one hand, it sounds exciting to use such a powerful communication infrastructure as Google Protocol Buffers on tiny embedded devices. On the other hand, the limitations of these tiny embedded devices (e.g. available memory, static memory management, real-time-constraints) make many features of Protocol Buffers at least challenging or even impossible to implement. It is ongoing research, how close Protobuf-Embedded-C can get to resource-constrained embedded services in a future version.

Practicability. Though many features of Google's original Protocol Buffers are not implemented yet, Protobuf-Embedded-C has proven its practicability for basic communication jobs on embedded controllers, which have only a few kilobytes of memory available. The application of Protobuf-Embedded-C in a heterogeneous platform environment is shown in the E-Energy-Grid demonstrator case example in section 6 of this document. In the meantime, embedded developers around the world begin to report the successful use of Protobuf-Embedded-C in their embedded applications.

Acknowledgement

The authors thank Martin Feilkas for developing and supporting the idea of "using a well-known, stable, easy to maintain and efficient" protocol on the wire in the SPES2020 demonstrator. The authors also thank Andreas Wandinger for his great support and feedback during the testing and debugging phase on the ARM Cortex M3 controllers. We also thank the embedded developers around the world, which have sent very constructive and positive feedback on Protobuf-Embedded-C to us since publication at Google code. Please note that this work was partially funded by the German Federal Ministry of Education and Research (BMBF), grant SPES2020, 01IS08045A.

Disclaimer

List of Figures

References

[1] Google, "protobuf – Protocol Buffers – Google's data interchange format – Google Project Hosting." http://code.google.com/p/protobuf/, 2011. [Online; accessed 29-September-2011].

[2] W. Schwitzer, V. Popa, and M. Feilkas, "protobuf-embedded-c – Protocol Buffers for Resource Constrained Embedded C Applications – Google Project Hosting." http://code.google.com/p/protobuf-embedded-c/, 2011. [Online; accessed 29-September-2011].

[3] Technische Universität München, "SPES2020 – Software Plattform Embedded Systems 2020." http://spes2020.informatik.tu-muenchen.de/, 2011. [Online; accessed 29-September-2011].

[4] Protobuf-C Authors, "protobuf-c – C bindings for Google's Protocol Buffers – Google Project Hosting." http://code.google.com/p/protobuf-c/, 2011. [Online; accessed 04-November-2011].

[5] Wikipedia, "Backus–Naur_Form – Wikipedia, the free encyclopedia." http://en.wikipedia.org/wiki/Backus%E2%80%93Naur_Form, 2011. [Online; accessed 16-November-2011].

[6] T. Parr, *The definitive ANTLR reference: building domain-specific languages*. Pragmatic Bookshelf Series, Pragmatic, 2007.

[7] T. Parr, "ANTLR Parser Generator." http://www.antlr.org/, 2011. [Online; accessed 29-September-2011].

[8] T. Parr, "StringTemplate Template Engine." http://www.stringtemplate.org/, 2011. [Online; accessed 29-September-2011].

[9] ARM, "Cortex-M3 Processor – ARM." http://www.arm.com/products/processors/cortex-m/cortex-m3.php, 2011. [Online; accessed 30-September-2011].

[10] J. Thyssen, D. Ratiu, W. Schwitzer, A. Harhurin, M. Feilkas, and E. Thaden, "A system for seamless abstraction layers for model-based development of embedded software," in *Software Engineering (Workshops)*, pp. 137–148, 2010.