Ultra-Lightweight RPC Protocol Framework with Variadic Templates in C++11

Bartosz Zieliński*, Piotr Kruszyński, Maciej Sysak, Ścibór Sobieski, Pawel Maślanka
Department of Computer Science, Faculty of Physics and Applied Informatics, University of Łódź,
ul. Pomorska 149/153, 90-236 Łódź, Poland.

* Corresponding author. Tel.: +48 606 645 429; email: bzielinski@uni.lodz.pl
Manuscript submitted July 08, 2017; accepted October 12, 2017.
doi: 10.17706/jsw.12.11.858-873

Abstract: We describe a framework for very lightweight custom remote procedure call protocol. This framework uses variadic templates and other language features introduced in C++11 to furnish effective, cache friendly and hence energy efficient implementation. Our approach provides also type safety and modularity of RPC handlers. Such a solution is useful both in industrial and home automation applications with possibly severe limitations on processing power and network bandwidth (like using GSM in remote areas). Here, the latter constraint may apply to systems implemented on both ends of spectrum: low end embedded systems as well as high end multicore server systems, but under heavy load.

Key words: C++, meta programing, RPC protocol, variadic templates.

1. Introduction

An important part of distributed system design is the choice of inter-part communication protocol and communication abstractions. A popular choice for the latter is the remote procedure call abstraction. For the former, enterprise systems often choose SOAP or REST-full protocols based on HTTP, or systems based on CORBA. An advantage these systems have is the high level of abstraction and the presence of well-supported standard libraries for many languages. On the other hand, those systems are very heavy, both in terms of network bandwidth (e.g., the overhead of HTTP protocol in REST or relatively large XML files in SOAP) and processing power (e.g., the necessity of parsing HTTP headers and XML content or including heavy libraries in CORBA). This is not a large problem when running enterprise applications, e.g., on the java application server, where the actual business tasks are heavyweight themselves, and the overhead of the protocol is negligible in comparison with the application framework itself. On the other hand there are applications where such an overhead would be unacceptable. As an example, consider an industrial control system (see Fig. 1) which manages a group of independent devices connected using various communication technologies and protocols such as CAN [2] and Ethernet. Devices labelled as UC1, UC2, … etc., are connected using CAN bus (Controller Area Network) which allows to send control messages with a relatively small transmission speed (approx. 150kbit/s { 1Mb/s, depending on the distance). Examples include temperature sensors or control devices such as light switches or other devices requiring high availability, like control and

indicating devices, or solenoid valve controls. Devices such as IP cameras and phones, (UE1, UE2,… in Fig. 1) are connected using Ethernet, as they require much larger bandwidth than the modules from the previous category. Finally, there are wireless devices (labelled by UW1, UW2,… etc.), exchanging only control messages with the rest of the system, similarly to the modules from the first category. Because they communicate with radio waves, they can be used whenever cables are not viable (remote controls, difficult to reach places, portable bar-code scanners, mobile phones, pagers, etc.).

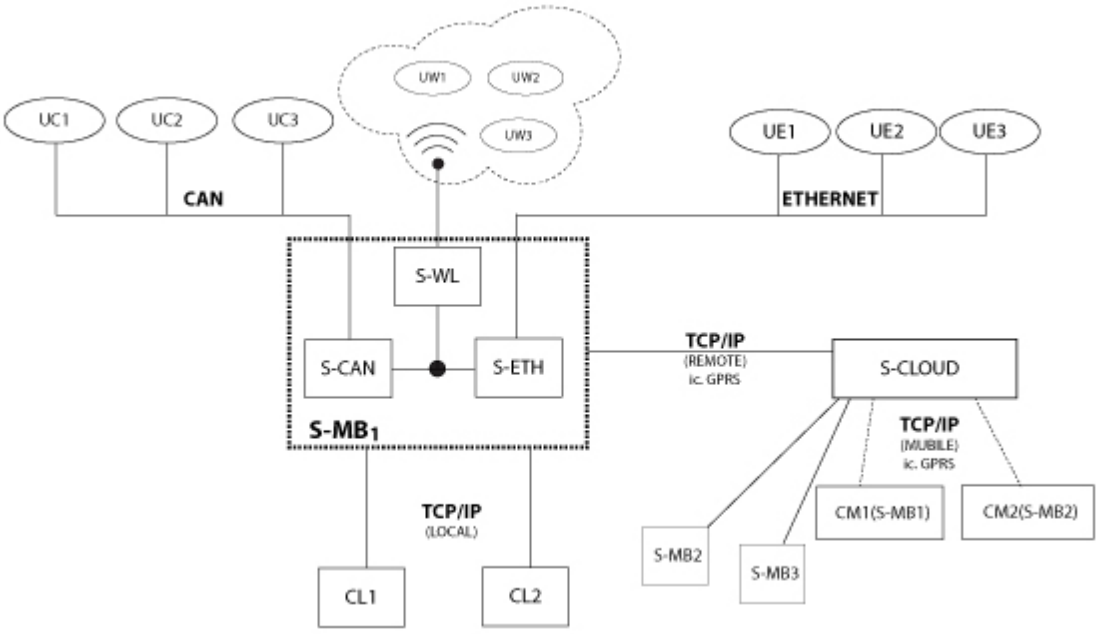

Fig. 1. Schema of an industrial control system.

We call the appliances of these three categories the end-point controllers (EPC). For the system to view them in the unified way, regardless of the transmission medium and communication protocol used, one uses multibus servers (S-MB1, S-MB2,… in Fig. 1). These are specialized embedded devices, able to link with all the EPC types, as well as to communicate with the rest of the system through network interfaces with TCP/IP stack. Apart from message translation, they provide services related to, e.g., authorization and verification of access to hardware resources, or emulation of some functionality not implemented directly by EPC's, e.g., temperature sensor and solenoid valve can be presented to the rest of the system as a single thermostat device by the multibus server which implements the thermostat logic.

The protocol we designed handles communication of clients (CL1, CL2,…, and also CM1(S-MB1), CM2(S-MB1),… in Fig. 1) such as wall touch panels, WWW applications, mobile and PC apps, as well as software connecting the system with other industrial systems. Clients may exchange messages either in the local or remote mode. Local mode means direct connection of client application with multibus server. Because of low availability of IPv4 addresses it is usually affected within the local area network (LAN). Connections in the alternative, remote mode, are routed through one of cloud servers, labelled by S-CLOUD in Fig. 1. The cloud server has a public IP address and is characterized by high availability.

Taking into account the above limitations we decided to implement the simple binary RPC protocol directly over the TCP/IP stack. The RPC packet contains the procedure identifier together with the

arguments, all in the binary format. Such a representation is very compact and light on network resources, but using binary representations of numbers forces the programmer to take into account differences in endianness and memory alignment requirements between different processor architectures. Text based protocols for network communication are popular because they avoid the aforementioned problems. Also, with a plethora of tools for generating parsers, while interpreting the textual format is much more computationally intensive than interpreting the binary format, the effort of the programmer is usually smaller, and less error prone. The framework presented in this paper aims to simplify the implementation of the protocol handling in the application program. Our framework requires the programmer only to write the RPC handlers and to link them with the appropriate remote procedure identifiers and argument types. The framework takes care of the mapping of RPC arguments to the arguments of the handler in the type safe way, taking into account memory alignment and endianness. Because we use C++ templates to implement the framework, the type checking is done at the compilation stage, which allows catching errors early. The use of C++ templates has other advantages, such as the lack of reliance on external tools and the possibility of code optimization many of the binary conversions are inlined at the compile time and can even be precomputed at the compile time.

An interesting aspect of our framework is that it makes prominent use of some of the new features introduced in C++11. It is well known that the previous version of C++ templates was already Turing complete ([7], see also [13] for a template implementation of a two stack compile time pushdown automaton), hence, in principle, we could have implemented the framework with equivalent functionality using the C++11 predecessor, it is however inconceivable that it could have been so natural and programmer friendly without some of the newly introduced features. Particularly useful for our project are variadic templates, which in many cases save the programmer the effort of working with handcrafted compile time lists [4] and dealing with their error prone and unnatural syntax. More importantly for our framework, they allow to define easily the templates of functions with variable argument lists, with compile time verification of the signature. This is one of the key techniques used here, pivotal for the easiness of use of our API.

1.1. Comparison with Previous Work

The need for more compact and lightweight support for data serialization and RPC calls than the extremely versatile but also extremely heavy solutions like SOAP, CORBA or even REST, which still drags with it all of the relative complexity of parsing HTTP protocol is well recognized. The two well-known and mature solutions, which could have been considered for the applications our framework is meant for, are Google's Protocol Buffers [3] and Apache Thrift [1], the latter with Facebook origins. Both projects are not tied to any specific language, but require instead to use a domain specific language which can be compiled into protocol support code in a dozen or so languages. Of the two projects the Google's one has a somewhat humbler scope, being concerned only with data serialization and deserialization, whereas Apache Thrift is a full-blown RPC framework with a DSL which allows not only to specify data but also services. Both frameworks support compact binary message formats (in case of Thrift this is but one choice among many, which include e.g., using JSON for the purpose). Unlike these two projects, our framework does not make use of a separate language and tools (it is just a C++ library) and thus makes integration with an existing C++ toolchain trivial. The fact that our framework is tied to C++ would be a problem in a highly heterogeneous environment utilizing multiple languages (such as the ones in Google and Facebook), but for applications we have in mind C/C++ is

about the only viable choice anyway. Using C++ directly allows us to exploit unique metaprogramming facilities of C++ in order to let the compiler generate efficient code by making use of compile time computations or creating opportunities for function call inlining. Note that this feature is relatively unimportant for typical applications of Protocol Buffers and Apache Thrift, where the cost of handling the communication protocol itself is negligible in comparison of the cost of doing the actual action associated with the remote procedure call, but it is crucial in applications where the RPC asks for a value of a device register.

The concept of metaprogramming is very powerful if a bit underused. Nevertheless, there exist well established language independent metaprogramming environments such as ASF+SDF (see e.g., [19], [8]) or its successor Rascal [14], [15]. In particular, Rascal was used in the implementation of Derric [18] | the language for specifying binary data | as well as digital forensics tool [18] based on specifications written in Derric. ASF+SDF and Rascal offer much more advanced metaprogramming capabilities than C++ templates, but at the cost of integration problems of two kinds: integrating metaprogramming environment with C++ toolset and defining appropriate grammar of C++ in the metaprogramming environment.

The advanced metaprogramming capabilities of C++ templates are well known. The book [4] is an excellent source of techniques for implementing fairly advanced compile time data structures (including compile time type lists) and algorithms using previous version of C++ templates. A plethora of literature exists (see, e.g., below) describing uses of C++ templates to metaprogramming tasks and efficient code generation. Applications of new template features of C++11 also already appeared in the literature, despite the brevity of time that passed from their introduction. Variadic templates, and some other new features, like lambda expressions and initializer lists are greatly appreciated by the high performance community, as they allow, among other things, to use natural syntax for multidimensional arrays, as

well as, together with the older template features, to increase performance, e.g., by unrolling the loops computing inner products (see eg. [5], [6], [17], [11]). The new C++ features were also vital in providing the actor semantics for C++ [9] with an internal DSL [12]. Variadic templates make also user api for calling reflected methods in [10] as natural as the native one. Finally, the book [16] contains some chapters devoted to C++ templates (including new features of C++ 11) and template metaprogramming in the context of microcontroller applications.

2. Custom RPC Protozcol

As explained above, most of the components of the described industrial system use the simple RPC protocol, in which the structure of the single message can be specified using the following pseudo-structure in C/C++:

```
truct sample_frame {
        uint32_t frame_len;
        uint32_t module_id;
        uint32_t procedure_id;
        /* args[] */
}; /* __packed */
```

The first three fields of the above structure are mandatory and form the header of the message. In addition, the structure may be extended with an arbitrary amount of optional fields, denoted above as

args[]. The binary representation of this structure in the program's runtime memory depends on the processor architecture, but the protocol must specify the unique representation for the actual frames sent through the network. Our protocol requires the numerical fields to use the little-endian representation without any alignment rules (and thus without any padding).

The first mandatory field - frame_length - contains the size in bytes of the structure, that is, both the header and optional fields. Because no padding is added, frame_length equals the sum of sizes of all fields contained in the frame.

The second field - module_id - identifies the module supplying the called function. Typically, a module is the type of the device attached to one of the busses described above. Multiple instances of the same device type are distinguished by one of the optional RPC arguments. Some of the virtual devices, and other functionalities, are also viewed as modules.

The last header field (procedure_id) contains the identifier of the particular procedure called within the given module. Each procedure can have different (perhaps empty) list of formal arguments, hence they are specified in the optional part of the frame, directly after the procedure_id field, and their interpretation and layout are procedure dependent.

Note that our protocol makes no direct provisions for return values from the remote procedure calls. If such are needed, they can be simulated by the designated return procedure call from the module to the calling client, with the returned value carried in one of the arguments. This approach simplifies the protocol and implies that the procedure calls are inherently asynchronous, which allows to avoid the costs of synchronous waits for return values.

Due to the clean and compact structure of messages sent, our RPC protocol is particularly well suited for systems utilizing networks with small bandwidth, where the charges from the network operator are proportional to the amount of exchanged data. In particular, the depicted industrial system uses GSM network to facilitate communication between some of the components in poorly urbanized areas, where only GPRS technology is available. Frequently, the connections are permanent (24 hours, 7 days a week), therefore the absence of excess content in the message frame significantly reduces costs of maintaining those connections.

Irrespective of the protocol handling (the highly performant implementation of which is presented in the subsequent section), parsing and creating protocol messages, by design, require as little processing as possible. This is particularly important for embedded applications as the low processor utilization translates into low energy consumption.

Observe that the format of the remote call facilitates both the modularity of the protocol (the procedure identifiers are required to be unique only within a module) as well as a limited kind of object orientation – the calls can be viewed as messages sent to the particular device instance identified by one of the arguments. The module corresponds to the class. Clearly this improves the versatility of the system and simplifies extending the system with new functionalities.

The presented protocol can be readily implemented at different transport layers e.g., it can use either UDP (messages are small and easily fit the single datagram) or TCP based streams. We choose the second alternative to take advantage of the session mechanism and message ordering guarantees provided by the TCP protocol. Furthermore, we assume that the protocols of the lower level take care of error detection and correction, as well as provide the transmission encryption, and therefore we do not implement these functionalities in the RPC protocol. The only validation of received frames, the protocol is capable of, is the comparison of expected frame size with the actual one. Expected frame

size is easy to compute, as it is determined by the types of formal arguments of the procedure identified by procedure_id and module_id. Validation of argument values can be performed only within the procedure call handler – the incoming data may convert correctly from the network to the host format but still be invalid from the point of view of system logic.

Another important issue is the default behaviour of the system upon receiving the procedure call with the unknown (procedure_id,module_id) pair. From among many strategies of dealing with this situation (e.g., marking the transmission as faulty, returning the error with the generic return procedure, etc.) we choose to ignore the unknown frames by skipping the frame_length bytes in the stream. Thus we ensure a degree of compatibility between components using different versions of the RPC protocol | such a components are compatible within the minimal common functionality they implement.

3. Template Framework for RPC Protocol

In this section we provide a detailed description of basic components of the RPC frame handling library. The library is implemented in fully standard compliant C++11, utilizing newly introduced variadic templates. In our exposition we employ a bottom-up approach, starting with the lowest level layers and then gradually moving up in the abstraction level.

3.1. Frame Converters

The messages are sent through the network in the form of compact binary data structures. In order to correctly interpret the values, this design choice necessitates the conversions of numbers between network and host formats.

The conversions in both directions are facilitated by two function templates: net2app and app2net (not shown) instantiated with a type to be converted. Function app2net transforms values from host to network format, net2app performs transformation in the opposite direction. Both function templates use static_assert to check if the template was instantiated with the supported type. This technique allows to produce, in case of unsupported instantiation, an error message more user friendly and informational than the standard compiler error messages produced by the incorrect template use. The function templates are wrappers for the respective static member functions of the value_converter template classes, listed in Figure 2, which do the actual job of converting values, and which have the same argument lists and return values as net2app and app2net functions, which are therefore not listed here, in order to save place.

In net2app and app2net the argument named value refers to the value in the native format, and the argument named buf is a pointer to the place in frame data where the value in the network format is supposed to start. The functions assume that frame data stretches at most buf_size bytes from the buf pointer.

Template function app2net places at location pointed by buf at most buf_size bytes of network format representation of value and returns the number of bytes actually placed at buf or zero if conversion was unsuccessful, possibly because the network representation of value required more than buf_size bytes.

Template function net2app stores in the variable passed as the first argument during the function call the value represented in network format in the initial bytes of frame data pointed by buf (but no more then buf_size bytes will be used). The function returns the number of bytes of the network representation of value or zero if the conversion was unsuccessful, perhaps because the buffer

contains only fragment of the value | this can easily happen if the data was received incompletely from the network.

As remarked above, the net2app and app2net function templates are wrappers for the respective static member functions of the value_converter template classes which do the actual job of converting values. For each supported type, the corresponding converter is the appropriate specialization of the primary value_converter template (see Figure 2). The primary template implements simply the default "type not supported" behaviour – the is_supported static field, which is used in the static_assert calls, is set to false and the static member functions net2app and app2net both return 0, which, as explained above, denotes the erroneous situation.

```
template <typename converted_type>
struct value_converter {
    static const bool is_supported = false;
    static size_t app2net(void *, const size_t,
            const converted_type &) { return (0); }
    static size_t net2app(converted_type &,
            const void *, size_t) { return (0); }
};
template <> struct value_converter<uint8_t> {
    static const bool is_supported = true;
    static size_t app2net(void *buf,
            const size_t buf_size, const uint8_t &value)
            { /* [...] */ }
    static size_t net2app(uint8_t &value,
            const void *buf, size_t buf_size)
    { /* [...] */ }
};
/* [...] */
template <> struct value_converter<char64_t> {
    static const bool is_supported = true;
    static size_t app2net(void *buf,
            const size_t buf_size, const char64_t &value)
    { /* [...] */ }
    static size_t net2app(char64_t &value,
            const void *buf, size_t buf_size)
    { /* [...] */ }
};
/* [...] */
```

Fig. 2. Helper class template and its specializations used to statically assert the support.

Fig. 2 presents also two example specializations providing support for uint8_t and char64_t types, the latter of which is defined using the following typedef: typedef char char64_t[64];

Introducing the support for new types requires just adding a new specialization of value_converter template, without any need to modify existing code. This technique works equally well for the primitive types as for C-arrays or even complex types with some limitations which will be elaborated in what follows. Also, template mechanism allows to instantiate only those specializations which are actually used, and, when appropriate, to inline the calls of conversion functions. In particular, when the host format is identical with the network format, especially in case of primitive types, the converter use may not incur any additional cost, as the appropriate specialization will simply expand into the call of the memcpy() copying the content from and into the buffer.

3.2. Frame Creation

In order to make a procedure call to a remote host it is necessary first to create an appropriate message, the structure of which was explained earlier in the paper. This task is facilitated by function template prepare_frame (see Figure 3), instantiated by the sequence of types of remote procedure call arguments. The first two function arguments (module_id and procedure_id) define the values of the corresponding fields in the frame header. The next two arguments define, respectively the address and size of the buffer in which the frame data should be stored. The remaining arguments correspond to the remote procedure call parameters. In case of success, the function template returns the size in bytes of the frame data stored in the buffer, otherwise prepare_frame returns zero.

```
template <typename... procedure_args>
size_t prepare_frame(uint32_t module_id, uint32_t procedure_id, void *buf,
        size_t buf_len, const procedure_args &... args) {
        if (buf_len < get_frame_size <procedure_args...>()) return (0);
        const uint32_t frame_len = (uint32_t)get_frame_size<procedure_args...>();
        if (prepare_frame_helper<
                sizeof...(procedure_args) + 3, 0, uint32_t,
                uint32_t, uint32_t, procedure_args...>::do_it(buf, frame_len, module_id, procedure_id, args...))
          return (get_frame_size<procedure_args...>());
        return (0);
}
```

Fig. 3. Frame creator template.

First, the template checks if the buffer provides sufficient space to store the constructed frame. The frame size is obtained by summing the constant header size with the sizes of all provided RPC parameters (see Figure 4). This approach assumes that the parameter data size does not change during conversion. The sum of sizes of all frame fields can be computed using function template get_frame_size(). The template, instantiated by the list of types of parameters of remote procedure call, returns the size in bytes of RPC frame data (including frame header). It uses function template get_total_size_of_elems() which computes the total size of parameters using the (compile-time) recursive algorithm (Fig. 4).

After performing validation, prepare_frame() function template (Fig. 3) calls static member function do_it() of the class template prepare_frame_helper shown on Fig. 5. The function converts all passed arguments, except the first one, to the network format and places them consecutively in the buffer pointed by the first argument, taking into account the offset passed in the template parameters. Note that the list of values to be converted passed to the do_it function starts with the header data, hence after the execution of do_it buffer buf contains the correctly constructed frame data.

The prepare_frame_helper template is instantiated with the number of remote procedure call fields (that is, header and parameters), the buffer offset and the list of types of RPC frame fields. Function template prepare_frame uses sizeof... variadic template operator to compute the number of RPC parameters, and it adds 3 to it in order to account for three header fields of type uint32_t prepended to the list of types passed to prepare_frame_helper. The offset passed by prepare_frame is 0 so that the frame data is placed at the beginning of the buffer. Turning to the arguments of the call to prepare_frame_helper::do_it() made inside prepare_frame include, in order, the pointer to the buffer to place the frame data in, the frame size as well as the module and procedure ids which constitutes the frame header data, and then the list of values of remote procedure call parameters. Assuming that the

data before and after conversion has the same size, prepare_frame returns the value computed by get_frame_size() to be interpreted as the size of the frame data stored in the buffer. In case of error this template returns 0.

```
template <size_t elems_left, typename head, typename... tail>
struct total_size_of_elems_helper {
        static const size_t size = sizeof (head) + total_size_of_elems_helper<elems_left - 1, tail...>::size;
};

template <typename head, typename... tail>
struct total_size_of_elems_helper<0, head, tail...> {
        static const size_t size = 0;
};

template <typename... elems>
size_t get_total_size_of_elems(void) {
        return (total_size_of_elems_helper <sizeof...(elems), elems..., void>::size);
}

template <typename... function_args>
size_t get_frame_size(void) {
        return (get_total_size_of_elems <function_args...>() + 3 * sizeof (uint32_t));
}
```

Fig. 4. Computing frame size.

```
template <size_t elems_left, size_t buf_offset, typename head, typename... tail>
struct prepare_frame_helper {
        static bool do_it(void *buf, const head &harg, const tail &... targs) {
                void *buf_addr = ((char *)buf) + buf_offset;
                if (app2net<head>(buf_addr, sizeof (head), harg) != sizeof (head))
                        return (false);
                return (prepare_frame_helper<elems_left - 1, buf_offset + sizeof (head),tail...>::do_it(buf,
                   targs...));
        }
};

template <size_t buf_offset, typename head, typename... tail>
struct prepare_frame_helper<1, buf_offset, head, tail...> {
        static bool do_it(void *buf, const head &harg, const tail &... /* targs */) {
                void *buf_addr = ((char *)buf) + buf_offset;
                if (app2net<head>(buf_addr, sizeof (head), harg) != sizeof (head))
                        return (false);
                return (true);
        }
};
```

Fig. 5. Frame building.

Actual frame building is done by the prepare_frame_helper template and its partial specialization presented in Fig. 5. The frame building algorithm uses the standard compile time type list recursion in which the list of types is matched against single head type and the remainder of the type list (tail). Each instantiation of do_it() static member function of prepare_frame_helper class template places the appropriate conversion of the value harg of the head field in the buffer with offset passed as one of the template instantiation parameters and then verifies the correctness of execution of this operation. In case of failure the do_it() simply returns false. What happens next depends on the value of the elems_left template parameter. The primary template of prepare_frame_helper::do_it() calls static

member function do_it() of prepare_frame_helper instantiated with template parameter elems_left reduced by one, buf_offset increased by the size of just placed head field, and the list of field types reduced to tail – the argument list of do_it() is shortened accordingly. The termination of this recursion is assured by the partial specialization of this template for elems_left equal to one, where do_it() simply returns true at this point instead of performing a recursive call. Note that we had to define do_it() as a static member function because partial specialization of non-member functions is illegal in C++.

```
bool parse_and_invoke(const void *buf, size_t buf_len, size_t &consumed_buf_len) {
        uint32_t frame_length, module_id, procedure_id;
        if (buf_len < sizeof (uint32_t) * 3)
                return (false);
        /* Parsing header fields with net2app. */
        if (net2app<uint32_t>(frame_length, buf, sizeof (uint32_t)) != sizeof (uint32_t))
                return (false);
        /* Similarly for the remaining fields. */
        /* [...] */
        if (frame_length > MAX_FRAME_LENGTH || buf_len < frame_length)
                return (false);
        auto rh_it = registered_handlers.find(mkkey(module_id, procedure_id));
        if (rh_it == registered_handlers.end())
                return (false);
        if (rh_it->second->parse_args_and_invoke(((const char *)buf) +3 * sizeof (uint32_t),
          buf_len - 3 * sizeof (uint32_t)) == false)
                return (false);
        consumed_buf_len = (size_t)frame_length;
        return (true);
}
```

Fig. 6. RPC implementation dispatcher.

3.3. Frame Parsing

The previous section described RPC frame creation. The current section is devoted to RPC frame processing carried out inside the function parse_and_invoke() (see Fig. 6). Parsing of the RPC message begins with examining the frame header, and, based on the header content, choosing parser for the remote procedure call's parameters. The parser extracts and converts the arguments and then passes them to the appropriate function which implements the actual action associated with the called procedure.

The first argument of parse_and_invoke() is the pointer to the buffer holding the received RPC message, the length of which is passed in the second argument. The last argument, passed by reference, serves to return the amount of bytes actually consumed by a successful parse_and_invoke() call. The function returns the boolean which indicates whether the execution was successful. After verifying that the passed buffer is sufficiently large to hold its header, the header fields (frame_len, module_id and procedure_id) get extracted and converted into the host format. If the frame size is equal to or less than both the maximum frame size for the system and the message size (the failure of the latter check indicates that the RPC call's transfer through the network was incomplete) the combination of procedure_id and module_id is passed (as uint64_t value) to the find method of registered_handlers collection (see Fig. 7). The method returns the appropriate call handler as a pointer to the parser_base class which supplies the interface for the method parse_and_invoke(). Its task is to retrieve the values

of all the RPC arguments from the processed message frame and pass them to the function implementing the actual action of the remote procedure. A general realization of this functionality is presented in Fig. 7. The template generic_function_frame_args_parser provides the implementation of parse_and_invoke() method. The method first checks if the supplied frame fragment which is supposed to hold arguments does have at least the expected size. The actual parsing of the arguments is performed by the static do_it() method template defined inside template struct parse_args2 (see Figure 8), which itself is defined inside template struct parse_args. The rationale behind this somewhat convoluted definition is that it allows us to use two independent parameter packs: of types of arguments already parsed and those yet to be parsed. The internal template is instantiated with the list of types of arguments not yet parsed. The external is parametrized with the amount of arguments still waiting to be extracted from the message frame, the type of function to be called when all arguments have been processed and finally the parameter pack of types of arguments successfully retrieved. The template function do_it is recursive (as a template, the instantiations are not). Each recursive call computes a single argument value by converting (with an appropriate value_converter) the binary data at the beginning of the buffer passed as the first argument. The computed value is placed on the function call stack inside the variable arg. Then if there are still some unprocessed arguments the method calls its other instantiation (hence the call is not, strictly speaking, recursive; as the compiler is expected to inline all the calls of do_it anyway, this point is moot) with elems_left reduced by 1, the buf pointer increased by bytes_used and the parsed value added to the end of args pack. If all the arguments are parsed, i.e., when elems_left equals zero, an alternative implementation of do_it is called, placed inside a separate specialization of parse_args template, which invokes the implementation of the remote procedure, passing it the parsed arguments. The invocation is preceded by the sanity check which verifies if all the declared bytes of message were processed. If not, then do_it simply returns false which indicates error, instead of invoking the call-back.

```
class parser_base {
public:
        virtual bool parse_and_invoke(const void *buf, size_t buf_len) = 0;
        virtual ~parser_base(void) { }
};

template <typename callback_type, typename... function_args>
class generic_function_frame_args_parser : public parser_base {
        callback_type callback;
public:
        bool parse_and_invoke(const void *buf, size_t buf_len) {
                if (buf_len != get_total_size_of_elems<function_args...>())
                        return (false);
                return (parse_args<sizeof...(function_args), callback_type>::template
                  parse_args2<function_args..., void>::do_it(buf, buf_len, callback));
        }
        generic_function_frame_args_parser(callback_type &&_callback): callback(std::move(_callback)) { }
};

typedef std::unordered_map<uint64_t /* module_id | procedure_id */, parser_base *>
  registered_handlers_map;
registered_handlers_map registered_handlers;
```

Fig. 7. RPC dispatcher metadata.

```
template <size_t elems_left, typename callback_type, typename... args_parsed>
struct parse_args {
        template <typename head, typename... tail>
        struct parse_args2 {
                static bool do_it(const void *buf, size_t buf_len, callback_type &callback, args_parsed &... args) {
                        head arg;
                        size_t bytes_used = net2app<head>(arg, buf, buf_len);
                        if (bytes_used != sizeof (head))
                                return (false);
                        void *new_buf = ((const char *)buf) + bytes_used;
                        size_t new_buf_len = buf_len - bytes_used;
                        return (parse_args<elems_left - 1, callback_type, args_parsed...,
                          head>::template parse_args2<tail...>::do_it(new_buf, new_buf_len, callback,
                            args..., arg));
                }
        };
};

 template <typename callback_type, typename... args_parsed>
 struct parse_args<0, callback_type, args_parsed...> {
          template <typename head, typename... tail>
          struct parse_args2{
                  static bool do_it(const void * /* buf */, size_t buf_len, callback_type &callback,
                    args_parsed &... args) {
                          if (buf_len != 0)
                                  return (false);
                          callback(args...);
                          return (true);
                  }
          };
 };
```

Fig. 8. Recursive argument parser.

4. Load Tests and Comparison with Dynamic Implementation

The depicted implementation of RPC framework was tested and compared, with respect to the execution time, with alternative, dynamic representation not utilizing template metaprogramming. Both implementations were written with the following assumptions and constraints in mind:

- Ensuring type safety of arguments – we use value converters during creation and parsing of message frames for types corresponding to respective formal parameters of remote procedures.
- Versatility of the system allowing easy extension with the support for new remote procedures.

The alternative dynamic implementation we use for the sake of comparison was in fact the prototype of solution depicted in this paper, and it served as a fragment of an industrial project. It was only slightly simplified for the tests.

We ran the separate tests of message creation and message parsing mechanisms. For each test the 200MB buffer was created in RAM, which was then filled with message frames with a number of parameters depending on test run. Note that it follows that the number of frames used depends on the number of parameters, as frames with less arguments are smaller and more of them can fit into the (constant) 200MB of buffer memory. The range of parameter numbers tested was between 0 and 50, with denser steps in the vicinity of zero. For each parameter number the appropriate test (either

measuring the buffer filling time or time of parsing all the messages in the buffer and calling the appropriate remote procedure handlers) was repeated twenty five times for each implementation. The only task of the remote procedure handlers used in the second kind of test was to read the values of passed parameters. In this way we ensure that the times measured are dominated by the handling of the protocol, and not by the business logic of the procedure call itself.

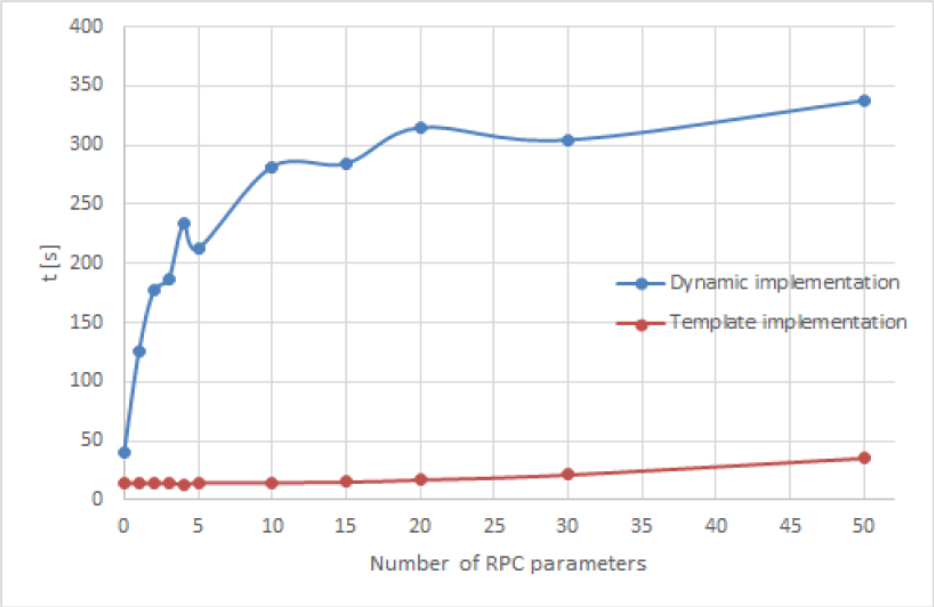

Fig. 9. Filling buffer with frames testing results.

In order to ensure the greatest possible comparability of tests, both implementations were written in C++, compiled with the same, gcc compiler version 4.7.2 with identical optimization flags. Tests were performed on the specially prepared testing platform based on Raspberry Pi Model B rev 2 (CPU 700MHz ARM 1176JZF-S, 512MB RAM) with Rasbian Debian Wheezy Linux distribution, kernel version 3.10.25.

The chart in Figure 9 shows comparison of the average time (in seconds) which took the respective implementations to create 200MB message frames plotted against the number of parameters. The measured times from which averages were computed were very consistent and so the standard deviations, also shown on the chart, are too small to be visible. The plot allows one to conclude that in the case of template based implementation the time increases with the increase in the number of parameters so slowly that it is close to being constant within the tested range. On the other hand, in case of dynamic implementation the time first increases very quickly with the number of parameters, then at about 6 parameters the plot starts to flatten out, becoming almost as slowly increasing as the plot for template based implementation in case of more than 30 parameters. Notwithstanding the flattening of plot for the dynamic implementation, in the entirety of the examined parameter range one sees the pronounced superiority of the template based solution.

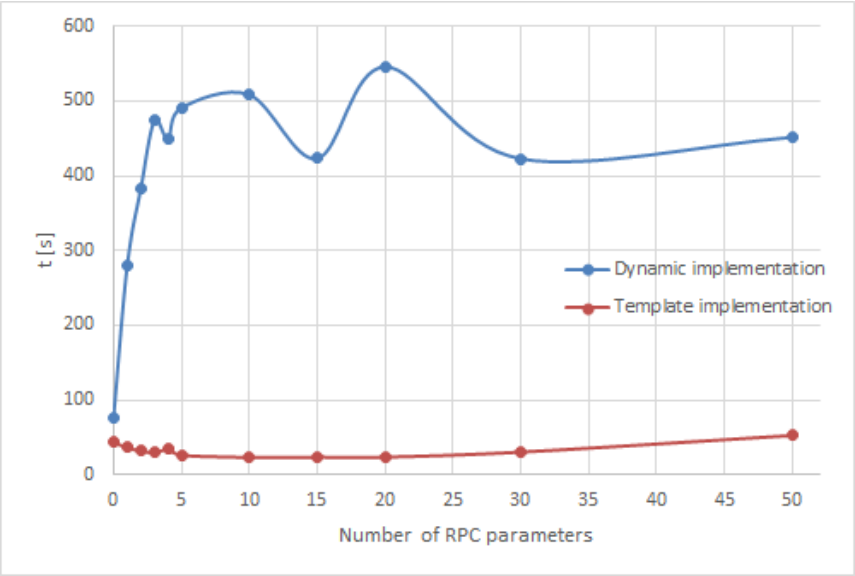

Fig. 10. Reading buffer and calling procedures testing results

The chart in Fig. 10 shows the average time needed to parse 200MB of RPC message frames by both tested implementations, again plotted against the number of RPC parameters. Again, the results in different test runs are very consistent, which leads to standard deviations being to small to be visibly marked on the plot. For the template based solution, the time is nearly constant throughout all the tested range of RPC parameter numbers. On the other hand, similarly as in the case of frame creation, the time first rises quickly, and then stabilizes for larger number of parameters with a slight upward trend. Also for the case of frame parsing, the chart shows consistent advantage of the template solution over the dynamic one for all tested numbers of RPC parameters. Summing it up, the tests established that the template based implementation of our RPC protocol is definitely more efficient than the dynamic prototype, both in frame creation and frame parsing. Note that most interesting from a practical point of view are the test results for the number of RPC parameters between 0 and 10, and it is within this range that the time increases most quickly for the dynamic implementation. The tests prove the high efficiency and low overhead of template based implementation, which is thus well suited for the environments with limited computing resources as well as with strict energy use requirements.

5. Conclusion

We have created the ultra lightweight RPC framework for industrial applications. The framework utilizes crucially the advanced metaprogramming capabilities of C++, including some of the newly introduced features such as variadic templates. The tests demonstrated that for the applications we have in mind, where the handling of the call (argument serializing and deserializing, checking correctness, etc.) dominates the actual intended functionality, the use of those techniques was justified as it reduces the execution time by a factor of ten.

To our best knowledge this is the first ultra lightweight RPC C++ framework, unique also because it makes no use of any external tools. We also think that our paper demonstrated some interesting uses of the new features introduced in C++ language.

The implementation of a mechanism for creating and parsing frames does not engage the memory allocator because all values used (such as a buffer for the message's frame or RPC parameters) are

stored on the stack. As a consequence, the depicted implementation is free from memory allocation overhead. This efficiency improvement is especially important in the concurrent environment, particularly the server one, where memory allocators implicitly use synchronization. In addition, favouring stack over heap allocations decreases probability of cache misses as a stack, unlike heap, is always implemented as a continuous memory region.

The usage of C++ templates provides a plenty of opportunities to the compiler for function inlining, which is particularly beneficial in case of relatively simple functions such as value converters between network and host formats. In particular, when the host format is identical with the network format the use of value converters reduces to copying values from/to message frame buffer without any additional overhead associated with the conversion function call.

An important advantage of our solution is its complete type-safety. The RPC parameters are converted to fundamental C++ types so that they can be passed as arguments to functions, methods or lambda expressions serving as remote procedure call handles, in a way which is statically verified by the compiler. This approach allows to avoid runtime error caused by programmer's mistakes.

Currently, the implementation assumes that value converters do not change the size of data. This limitation resulted from the absence of need for such ability in the present applications of our framework, rather than from some fundamental technical obstacles.

The architecture of our system facilitates adding new procedures and type converters in an easy and safe way which does not require modifying of existing code base. Instead, the programmer simply adds new template specializations or registers new handlers.

Finally our implementation is fully compliant with C++ standard, and as such is highly cross platform, allowing easy porting to new embedded environments, on the condition that the platform supports C++11 compiler.

References

[1] Apache Thrift. Retrieved from http://thrift.apache.org/

[2] CAN protocol. Retrieved from http://www.can-cia.de/index.php?id=161

[3] Protocol Buffers. Retrieved from https://developers.google.com/protocol-buffers/

[4] Alexandrescu, A. (2001). *Modern C++ Design: Generic Programming and Design Patterns Applied*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[5] Andres, B., Koethe, U., Kroeger, T., & Hamprecht, F. A. (2010). *Runtime-Flexible Multi-Dimensional Arrays and Views for C++ 98 and C++ 0x*. arXiv preprint arXiv:1008.2909.

[6] Aragón, A. M. (2014). A C++ 11 implementation of arbitrary-rank tensors for high performance computing. *Computer Physics Communications*.

[7] Bohme, M., & Manthey, B. (2003). The computational power of compiling C++. *Bulletin of the European Association for Theoretical Computer Science*, *81*, 264 – 270.

[8] Brand, M., Deursen, A., Klint, P., Klusener, S., & Meulen, E. (1996). Industrial applications of ASF+SDF. *Lecture Notes in Computer Science*.

[9] Charousset, D., & Schmidt, T. C. (2013). Libcppa - designing an actor semantic for C++11.

[10] Bayser, M. D., & Cerqueira, R. (2012). A system for runtime type introspection in C++. *Proceedings of the 16th Brazilian Conference on Programming Languages* (pp. 102-116).

[11] Dickens, M., & Laneman, J. N. (2012). On the use of an algebraic language interface for waveform definition. *Analog Integrated Circuits and Signal Processing*, *73(2)*, 613-625.

[12] Fowler, M. (2010). *Domain-Specific Languages*. Addison-Wesley Signature Series (Fowler). Pearson

Education.
[13] Gutterman, Z., Yavneh, I., & Gil, J. Y. (2004). Symbolic pre-computation for numerical applications. Master's thesis, System and Software Development Laboratory.
[14] Klint, P., Hills, M., Bos, J. V. D., Storm, T. V. D., & Vinju, J. (2011). Rascal: From algebraic specification to meta-programming. *Proceedings Second International Work-shop on Algebraic Methods in Model-based Software Engineering* (pp. 15-32).
[15] Klint, P., Storm, T. V. D., & Vinju, J. (2011). *EASY meta-programming with rascal. Proceedings of the 3rd International Summer School Conference on Generative and Transformational Techniques in Software Engineering III*.
[16] Kormanyos, C. (2013). *Real-Time C++: Eficient Object-Oriented and Template Microcontroller Programming*..
[17] Szugyi, Z., Torok, M., Pataki, N., & Kozsik, T. (2012). High-level multicore programming with C++ 11. *Computer Science and Information Systems*.
[18] Bos, J. V. D., & Storm, T. V. D. (2011). Bringing domain-specific languages to digital forensics. *Proceedings of the 33rd International Conference on Software Engineering* (pp. 671-680).
[19] Brand, M. G. J. V. D., Deursen, A. V., Heering, J., Jong, H. A. D., Jonge, M. D., Kuipers, T., Klint, P., Moonen, L., Olivier, P. A., Scheerder, J., Vinju, J. J., Visser, E., & Visser, J. (2001). The ASF+SDF meta-environment: A component-based language development environment. *Proceedings of the 10th International Conference on Compiler Construction*.

Bartosz Zieliński received his master degrees in physics (2001) and mathematics (2002) from University of Lodz, Poland. He received his PhD in mathematics from University of Wales Swansea, UK in 2005.

He works as an adjunct professor in the Department of Computer Science, Faculty of Physics and Applied Informatics, University of Lodz since 2005. He also worked for several years in the Polish Academy of Sciences. He is an author of publications in mathematics and computer science. In 2014 he was awarded the Prize of Chancellors of Lodz Universities for Young Scientists. In 2005 he received Clack Wilson prize.