Dynamic Invocation
Tiago Rodrigues
tiagomsrodrigues@tecnico.ulisboa.pt
Instituto Superior Técnico, Lisboa, Portugal
October 2019
Abstract
Nowadays, there is considerable interest in remotely invoking the functionality provided by software applications (e.g., source code editors, Computer-Aided Design (CAD) tools, Building Information Modeling (BIM) tools, game engines). To this end, it is necessary to remotely access the functionality provided by the software application’s Application Programming Interface (API). As such, software application plug-ins need to implement a mechanism which enables Remote Procedure Invocation (RPI). However, most RPI approaches, using Interface Description Languages (IDLs) as a neutral language to structure request and response messages, and specify server remote interfaces, negatively impact the productivity of plug-ins during development. That is because when new procedures are added, this information needs to be manually written in an IDL file. Therefore, we introduce Dynamic Invocation (DI) as the mechanism that could, based on high-level programming language capabilities, use metadata to remotely invoke a plug-in’s procedures (API), reducing the overheads of previous approaches, thus, increasing productivity. DI utilizes mainly metaprogramming techniques to abstract most details of a RPI, such as, definition of message’s structure, order of serialization and deserialization of procedure parameters, and the invocation of the function that implements the procedure.
As the effort to implement DI in a programming languages heavily depends on its metaprogramming capabilities, we will present an implementation of DI in C++, a language which possesses complex means for metaprogramming. We will showcase how one can use the properties of C++ templates to bypass this difficulty, and compare the productivity to approaches which rely on IDL to make RPI.
Keywords: Remote Procedure Invocation, Productivity, Metaprogramming, Dynamic Invocation, Templates.

1. Introduction
Throughout the years, with the increasing development of software applications, computers have proven to be of great aid to automate difficult and tedious tasks. Although software applications solve a lot of problems, sometimes, they might not be enough for a given context. For instance, although CAD tools offer features to deal with the most common use cases of architectural design, an architect might require some additional features for his area of expertise.

Because of this, most software applications provide APIs which are used by programmers to implement new functionality that might be useful to some users. The set of newly implemented procedures extends the interface provided by the API and are loaded by software applications as plug-ins.

In some cases it is advantageous to have a software application running in a different machine, and remotely invoking its procedures. An example of this is the use of CAD and BIM software applications to enable Generative Design (GD). GD consists in a process that allows the design of models through algorithms [7]. It is particularly useful in architecture as it allows architects to easily adjust their models’ parameters in programs, according to their project constraints and requirements.

In [3], to explore GD, a plug-in for ArchiCAD (BIM software application) was implemented. This plug-in communicated with a programming environment called Rosetta. Rosetta provided a set of directives which could be used by architects to generate their models through algorithms in client programs. By having these programs remotely invoking the functionality implemented in the plug-in, with Rosetta, architects could write programs in a user-friendly programming language using simple directives which abstracted all the details of the communication and implementation of the plug-in.

However, in order to communicate with client programs, plug-ins need to implement a mechanism that allows its functionality to be remotely invoked, enabling RPI. RPI is a form
of inter-process communication which relies on request/reply-based protocols to ensure clients access a server's API. The client requests the server for a certain procedure, available on its API, and the server replies with the result.

There are many approaches which can be used to enable RPI. The most popular rely on the specification of a remote interface or message formats in an alternate neutral language called IDL. IDLs can be compiled to generate code for different programming languages. Also, these approaches provide useful abstractions to simplify the implementation of an RPI mechanism.

However, there is an inherent problem with these approaches. During development, as new procedures are added, the IDL files need to be continuously changed, which ultimately, ends up hindering the productivity during the development of a plug-in without much benefits.

For example, unlike applications with a microservice architecture, i.e., applications divided in several small remote servers possibly written in different programming languages, the development of plug-ins does not benefit as much from a neutral language to represent an interface or message formats. For microservice architectures, IDLs allow code to be generated for different microservices, effectively reducing the effort to implement each microservice. On the other hand, the plug-in’s implementation is usually done in a single service/program, which means that the neutral language of IDLs is just adding another layer of complexity and not saving much time.

In this paper we introduce DI, a mechanism that enables RPI without the overheads of IDL-based approaches. DI consists on a mechanism implemented on the server side which uses features of high-level programming languages to allow the remote invocation of procedures by a client. It is different from the previous approaches because its use should not require the specification of remote interfaces or message formats in an IDL. Thus, little to no time is spent adding new procedures to a plug-in.

Metaprogramming, the ability of programs to treat other programs as their own data [4], facilitates the implementation of DI through the use of its different strategies. For instance, if a programming language possesses any means to evaluate its own expressions (use its own expressions as data, i.e., reflection) in the form of strings, as long as the client sends valid expressions (as a string), function calls, through the stream, the only thing the server has to do is: read the string, evaluate it, and send the result back, functioning as a remote interpreter.

Although the solution previously explained is a mechanism that could be easily implemented in most scripting languages, software application plug-ins do not use these types of languages as frequently to justify discarding other groups of programming languages. For some types of programming languages, such as statically-typed and compiled languages (e.g., Java, C#, C++), this dynamism is not achievable as there are no built-in functions in the language that implement code evaluation.

There are different alternatives based on metaprogramming, which, using metadata, can perform an RPI. One of them is based on reflection. Reflection is a key strategy of metaprogramming. It is described as a systems’ ability to reason about, and possibly alter, its own behaviour [4] and is composed by two activities [5]:

- Introspection - allows a system to observe its own behaviour;
- Intercession - allows a system to act upon the observations done with introspection and modify its own behaviour.

Note that the previous solution based on code evaluation is also a form of reflection through intercession.

In order to possess mechanisms that allow reflection, a programming language must define entities that represent itself within itself. This ability, which is necessary to reflective systems, is defined as reification or self-representation [4].

The idea of using reflection, for languages such as Java and C#, consists in obtaining reified entities of the message deserialization methods, invoke them (to get the values procedure arguments), and forward the results to the invocation of the desired procedure, which is also invoked using its own reified entity obtained with introspection.

There are languages, however, that do not possess means for reflection, for example, C++. Nonetheless, it is still possible to abstract the previous problems in C++. Templates provide a powerful and complex feature that enables metaprogramming. With templates the compiler can programmatically assemble bits of code which produce a certain effect at runtime [8]. In other words, the idea is to have the compiler do the work of assembling the code that performs an RPI.

The result of this work will be an implementation of DI in C++ which takes significantly less time to add new procedures. The solution will be evaluated and compared with other approaches which use different RPI mechanisms to enable RPI.

This paper is organized as follows: Section 1 introduces the motivation, state of art, and DI. In Section 2 we will study two approaches for RPI.
which use an IDL for the specification of remote interface and message formats. Section 3 gives an overview on the implementation of our solution, which is then compared to other approaches in Section 4. Section 5 concludes the report along with the future work.

2. Background

There are many approaches which can be taken to perform reliable and efficient RPI across applications and software applications plug-ins. Some of these approaches enable RPI through abstraction, i.e., they provide libraries, in different programming languages, which can be used by the programmer to abstract details of a RPI, such as, transport protocols, serialization methods, and server configurations (e.g., multi-threaded, non-blocking servers). Additionally, these approaches are also equipped with code-generation engines that transform a simple interface and data definition language, IDL, into RPI libraries for both client and server applications. Since, in this case, all of the RPI process is abstracted, these approaches are usually referred to as RPI frameworks. An example of a common RPI framework widely used is Apache Thrift [6].

To make an RPI with Apache Thrift, first, a client program uses a service stub to call a procedure. Then, the parameters of the procedure are serialized with a serialization protocol. If there were any user-defined types in the IDL they are also automatically serialized. After the data is serialized, following a transport protocol, the data is sent to the server side over a communication channel. The server deserializes the data to the corresponding primitive or user-defined types and uses the service stub to dispatch the procedure’s arguments to the corresponding service handler where the procedure is invoked. In Figure 1, the overall architecture of these frameworks can be seen.

If a programmer wants to add a new procedure to the server (plug-in), the following steps should be taken:

1. Write in an IDL the procedure’s interface;
2. Compile the IDL file to automatically generate the code for the stubs, serialization functions of user-defined types (if specified), and service handler classes where the procedures will be implemented;
3. Write/Call the implementation of the procedure in the service handler class generated after compilation.

Notice that some of these steps sacrifice a lot of time during development. Every time a new procedure is added to the server it is required to: write the procedure as a method’s service in an IDL; compile the IDL; and, implement the method of an handler. Although the time for some of these tasks is barely significant, for example, the time it takes to compile an IDL is around 1 or 2 seconds, assuming that the programmer will have to open a terminal which is already set-up for the compilation, there are other tasks, such as the writing of the IDL, which will always take some seconds or even minutes to accomplish. In addition to that, the programmer needs also to understand the syntax of the IDL. While the Thrift’s IDL syntax is relatively easy to understand, as exemplified in Listing 1, the programmer will always waste a few seconds writing procedure specifications until he gains some experience.

Listing 1: Example of the specification of a remote service in Thrift.

```
(service Calculator

  double add(1:i32 a, 2:double b)

)`
```

There are also many factors which might influence the use of these RPI frameworks, for example, these frameworks require the programmer to install and configure their libraries in his machine. For inexperienced programmers this might be an obstacle as most of the times they end up following a set of guidelines while not being fully aware of what they are doing, increasing the chances of failing the installation.

Alternatively, instead of using an RPI framework, programmers can use solely IDL-based methods to serialize and deserialize messages. One example of such methods is Protocol Buffers [1]. With this approach the parameters and return values of the procedures are specified in an IDL as request and response messages. Because the IDL only generates serialization functions for the messages, there are no stubs to dispatch these messages to handlers (which are not generated). This allows programmers to implement their own dispatching mechanism, avoiding the necessity of continuously writing handlers, as presented in Figure 2.
that with this solution the programmer has no ab-
stractions for the transport protocols, having to use
one to send and receive the serialized data over a
communication channel.

Figure 2: Overall architecture of an RPI mechanism with a se-
rialization method. Inspired from [2].

In [3], a plug-in for ArchiCAD was implemented
based on this approach, the implementation uses
C++ function pointers to invoke the different pro-
cedures that the plug-in provides, and Protocol
Buffers to serialize and deserialize the messages
sent between the client application and the plug-
in. The different procedures were implemented
as `void(*)(void)` function pointer types and were
mapped from their respective names.

In this solution, to invoke a procedure the client
sends the name of the procedure which he wants
to invoke and the corresponding function pointer
types is retrieved, and invoked. The parameters,
and return values of the procedure, are deserial-
lized, and serialized, respectively, inside the imple-
mentation of the procedure, which is why all pro-
cedures could be implemented as `void(*)(void)`.

While this simplifies the overall implementation and
dispatching of arguments to the procedure, it cre-
ates a dependency issue because the procedure
needs to know how to read and write data to a
socket stream, making the implementation of the
procedures dependent on the RPI mechanism.

Not only that, but because IDL-based serial-
ization method was used, this solution also suf-
fered from productivity problems since every newly
added procedures requires the programmer to:

1. Add the new procedure to the mapping of
function pointers so it can be invoked;
2. Add to the Protocol Buffers IDL file the struc-
ture of the request and response message;
3. Compile the IDL file to generate the classes for
each message type and their corresponding
serialization and deserialization methods.

Although it is no longer necessary to write or
call the procedure in an handler class similar to
RPI frameworks, and the addition of the mapping
is fast, this approach still relied on writing an IDL
file, which hinders the development of the plug-in
when adding new procedures.

The focus of this work is to find a solution to
decrease the effort required by a programmer to
make a procedure remotely available on a software
application plug-in. One of the reasons why we
keep focus on this aspect is because some soft-
ware applications impose restrictions on the plug-
in’s characteristics which make them not benefit as
much from the abstractions of RPI frameworks or
efficiency of some serialization methods.

For example, if the software application blocks
the user interface of the software application and
does not allow the plug-in to be multi-threaded,
it does not matter if a single-threaded blocking
server is used instead of a multi-threaded non-
blocking server. There is no point on having a RPI
framework abstract the implementation of a multi-
threaded non-blocking server because the soft-
ware application will always be blocked and multi-
threading will not work.

In the same manner, it does not matter if a se-
rialization method is extremely efficient if most of
the time of the remote invocation is spent by the
software application executing a procedure. For in-
stance, a BIM application might take 40 seconds
to create a building model, however, from those 40
seconds, less than a second is spent with commu-
ication and serialization. In this case, the effect
that an efficient serialization method will have in the
runtime performance is almost insignificant.

Since the programmer has no control over the
time spent by the software application executing a
procedure as he cannot change the implementa-
tion of the functions provided by the developer's
API, we decided that it would be more appealing
if we could find a way to save time during devel-
opment. This could be done by implementing a
mechanism which would not require the program-
mer to waste seconds, or even minutes, specifying
the parameters of a RPI in an IDL file.

In the next section, we will briefly explain our
solution, which implements DI with metaprogram-
ing techniques, and showcase the work neces-
sary to add a procedure to our RPI mechanism.

The solution was implemented in C++, a language
which does not possess reflective capabilities. We
used metaprogramming features of C++ templates
to have the C++ compiler generate the code nec-
essary for an RPI at the same time as the plug-in is
compiled, essentially removing the need to specify
and compile an IDL. It is important to note that for
this to be possible the serialization method chosen
should not use IDLs.
### 3. Methodology

Besides the fact that we did not want to use any IDL-based serialization mechanism in our solution, we also did not want to impose restrictions in the types of function pointers which could be remotely invoked. Therefore, to allow different types of function pointers, we encapsulated each of them in a generic class template `Handler` using partial template specialization as exemplified in Listing 2.

**Listing 2:** Partial Specialization of `Handler` to pointer types.

```cpp
template<typename F>
class Handler : public HandlerInterface {};

template<typename R, typename... Args>
class Handler<R(*)(Args...)>
    : public HandlerInterface {
    using F = R(*)(Args...);
    Handler(F f) : function_ptr_(f) {}
    ...
};
```

Since after the compiler instantiation (i.e., when the compiler generates a class for a specific template type) generic class templates have no relations, we made the `Handler` template inherit an `HandlerInterface` common interface, to allow different `Handler` classes to be used and stored through this interface.

This common interface is composed by two different methods which implement the RPI process in two different consecutive steps:

- **ReadArgs(Stream)** - method that reads arguments from a data stream and stores their values;
- **Invoke(Stream)** - method that forwards the arguments stored in `ReadArgs` to a function pointer, and writes the obtained result to a data stream;

In order to invoke these methods a `RpiService` class stores different handlers and performs invocations on them by invoking both methods consecutively.

To invoke a procedure, the service calls the `ReadArgs` method of a handler, a linked typelist is recursively iterated, reading the arguments of a procedure according to the type of the current head of the list, as illustrated in Listing 3, in the `Read` method. Notice, in the second class template, how template partial specialization was used to separate the type of the head from the tail, and how explicit template specialization was used to define the base case to stop the recursion when reading from the stream.

**Listing 3:** Implementation of a linked typelist as a `ArgumentList` generic class template.

```cpp
template<typename... Types>
class ArgumentList {};

template<typename Head, typename... Tail>
class ArgumentList<Head, Tail...> {
public:
    ArgumentList() {}
    void Read(Stream& stream) {
        head_ = stream.Read<Head>);
        tail_.Read(stream);
    }
    ...
};
```

After the invocation of `ReadArgs` an handler has a typelist populated with the arguments. The service class invokes the `Invoke` method of that handler instance to dispatch the arguments of the list to the function pointer stored in the handler. The arguments are forwarded by using `ArgumentGet` to index each element of the list. The indexes are provided by an `index_sequence` which consists of a sequence of integer values from 0 to N-1 where N is the size of the list.

**Listing 4:** Using the `ArgumentGet` metafunction and `index_sequence` to expand all elements values of a list.

```cpp
template<typename F,
    typename... Args, 
    std::size_t... Is>
auto WrapperDetail(F&& f, ArgumentList<Args...> list, std::index_sequence<Is...>)
{
    return f(ArgumentGet<Is>::Apply(list)...);
}
```

The result of this invocation is written to a data stream which is sent to a client application.

In order to add a new procedure to our solution the only thing that the programmer has to do is to add a mapping of the function that implements the procedure, as shown in Listing 5. This only requires the call of a macro named `ADD_FUNCTION` in the `RpiService` class constructor, or in a subclass of `RpiService`. Notice that all procedures are added in the same class constructor, i.e., we do not have to create a new class for each procedure.

**Listing 5:** Example of how to add a new procedures to our solution.

```cpp
class MyService : public RpiService {
    MyService::MyService() {
        ADD_FUNCTION(foo);
        ADD_FUNCTION(bar);
        ...
    }
};
```

This macro call essentially transforms the function pointer in an `Handler` to be stored in the service class. The compiler detects the creation of this `Handler` and triggers the instantiation process
at compile time, generating all the necessary code to invoke the procedure. Notice that we only specify the name of the function to create an Handler class. This means that, although the types of the arguments and return are automatically deduced by the compiler, the solution does not allow the invocation of overloaded functions (as it is not possible to deduce an overloaded function from the name alone).

4. Results & discussion
In this section we will talk about the mechanisms and tools used to evaluate our solution. We chose ArchiCAD as the software application in which our solution would be evaluated. The reason behind this choice is the fact that the developer API of ArchiCAD is implemented in C++, the same language that is the target of our solution. Also, we wanted to pair our solution with a software application which could be used for GD.

We used the development kit for ArchiCAD 21, which uses the Visual C++ 14.0 compiler of Visual Studio 2015. This is compatible with our solution because the C++ template features used are available since C++11.

Furthermore, to test several examples of RPI we implemented a few CAD and BIM operations in an ArchiCAD plug-in. We used three different RPI approaches to invoke the procedures of the plug-in, them being:

- A RPI framework, Apache Thrift, which relies on the compilation of an IDL file to generate stubs and handlers for the procedures, and serialization functions for new types specified in the IDL. This mechanism communicates with a client application, implemented in C++, through a normal TCP/IP socket channel which serializes and deserializes data in a binary format (Thrift’s binary protocol);

- An implementation of a solution identical to the one developed in [] which uses Protocol Buffers as a serialization method and function pointers for the invocation of procedures. Recall that this solution relies on the specification of message types in an IDL which are sent between a client and server, in a binary format. The client application used to communicate with this solution is also written in C++ and uses TCP/IP sockets as transport protocol;

- An implementation of our solution, which performs DI on the procedures implemented in the plug-in. The communication with a client, implemented in C++, is also done via TCP/IP sockets. In this case we used a custom made binary protocol for the serialization of data over the channel. Later on the chapter we will explain the reason of this choice.

These three mechanisms are provided through blocking single-threaded servers. We opted for this configuration because ArchiCAD blocks all interaction with the software application’s interface when executing a plug-in. Therefore, even if the server was non-blocking the software application would still be blocked. Moreover, only the main thread can call functions of the developer API, which means that having the RPI mechanism running in another thread would not solve this issue.

4.1. Productivity analysis
The experiments done in this section were executed by a programmer (the author of this document) which was already familiarized with the tools. For each experiment, to avoid wasting additional time, all necessary files were already opened prior to the start of the experiment. The one exception being the file with the handlers, generated in the Thrift approach, which can only be opened after the compilation of the IDL file. Note that this is the only IDL-generated file that needs to be opened for editing, the rest of the files generated with Thrift and Protocol Buffers do not need to be open. Additionally, we also had a command prompt already set on the right directory path to compile the IDL file after its writing.

The idea behind these decisions was to have a minimal impact of other variables, such as inexperience, and time spent in minimal tasks (e.g., opening/switching files and setting directory paths), in the measurements of each approach.

We measured the time spent by each task, of the three approaches, to add a new procedure createCircle. Table 1 showcases the tasks measured for each approach. The createCircle operation draws a simple circle of radius, \( r \), in a point, \((x, y)\), and returns the identifier of the circle.

In order to get more accurate results, a total of thirty experiments, ten for each approach, were realized. The average times for each task are shown in Figure 3, and a depiction, in groups, of the measurements obtained for each experiment can be seen in Figure 4.

The results in Figure 3 show that the DI alternative, on average, is quicker than the other approaches by a 40 seconds difference (about five times faster on average). Not only that, but the difference between the highest registered measure and the lowest registered measure is much smaller when compared to Protocol Buffers and Thrift. This is depicted in Figure 4 by observing, for each box, the length that goes from the highest point of the upper whisker to the lowest point of the bottom whisker.
Table 1: Table that shows the tasks necessary for each approach.

<table>
<thead>
<tr>
<th></th>
<th>Write Handler</th>
<th>Write IDL</th>
<th>Add Mapping of New Procedure</th>
</tr>
</thead>
<tbody>
<tr>
<td>Apache Thrift</td>
<td>✓</td>
<td>✓</td>
<td>-</td>
</tr>
<tr>
<td>Protocol Buffers</td>
<td>-</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Templates</td>
<td>-</td>
<td>-</td>
<td>✓</td>
</tr>
</tbody>
</table>

![Figure 3: The average time needed to make an int createCircle(double x, double y, double r) procedure remotely available by the three alternatives which used Thrift, Protocol Buffers, and DI, for each task.](image)

![Figure 4: Distribution of the thirty experiments in groups (quartiles) through a whiskers plot for each alternative.](image)

This higher deviation from the average value (represented as a cross in Figure 4), is probably a consequence of the fact that most of these tasks need to be done manually, i.e., they cannot be automated. Every task, such as, the addition of a mapping, writing of an IDL, and implementation of an handler method, need to be manually done by the programmer. Therefore, it is expected that the more time is spent on these tasks the more accentuated the deviation will be.

For example, the tasks which consist on adding a map, as exemplified in Listing 6, are relatively simple and require only a simple line of code to be added. As seen in Figure 5, as expected, the deviation and length of the upper and lower whiskers for mappings is smaller than other tasks which take more time.

Listing 6: Examples of the instructions necessary to add a procedure createCircle to a map, on the DI and Protocol Buffers approaches.

```c
// Mapping DI
ADD_FUNCTION(createCircle);
```

```c
// Mapping Protobuf
remote_methods_.push_back(&createCircle);
```

![Figure 5: Distribution of the thirty experiments in groups (quartiles) through a whiskers plot for each task.](image)

In contrast with mappings, the writing of an IDL file requires more effort and time since not only the programmer has to write it, which already takes more time than adding maps, but he also has to spend time understanding and learning the IDL syntax. The inexperience of the programmer with a tool can also be an obstacle and delay this task. It is important to note that during the evaluation of the approaches, which use IDLs, we did not take into account the time necessary to learn the tool. One could expect the results to be worse if a programmer’s inexperience with these tools was high.

For example, the indecisiveness about a syntax detail, and consultation of documentation, are simple steps which might cause, in some cases, significant overheads, taking seconds, or even minutes of the programmer’s time. Additionally, since there is a lot more writing in IDL files, these alternatives are much more prone to typing errors, which when passed as undetected can also impact productivity. Although these overheads might not seem significant, a few seconds and minutes are enough to double the time it takes to add a procedure for IDL-based alternatives, such as, Protocol Buffers and Thrift.

The writing of handlers, in Thrift, suffer from a similar problem, although to a lesser degree, as they require only the writing of a simple function call to the procedure’s function. The additional work of opening the file, and copying the handler class to the project code is slower than adding a mapping, for example.
In addition to these experiments, we did five more for each alternative to check how dependent these measurements are on the complexity (number of arguments) of the procedure in Listing 1:wallprocedure. We want to check the difference between the times obtained by the three approaches, knowing the writing of the IDLs will be severely impacted as more information will have to be written in the IDL file. This is proved with the obtained results shown in Figure 6.

Listing 7: Declaration of a procedure which creates walls depending on the size of \( p_0 \) and \( p_1 \), which store the beginning and end points of each wall.

```c++
std::vector<int> createWalls(
    std::string type, short material,
    double thickness,
    short home_story, short stories_to_top,
    double top_offset, double bottom_offset,
    std::vector<XY> p0, std::vector<XY> p1,
    double angle, std::string reference_line);
```

While the solution which used DI maintained the same values, the solutions that used IDLs, as expected, increased the time spent in comparison to their previous results, going from almost a minute, to almost three.

![Figure 6: The average time needed took to make a createWalls operation remotely available using the three alternatives Thrift, Protocol Buffers, and DI.](image)

The problem with writing IDL files does not solely lie on the writing of IDLs, the fact that the function has many arguments makes it hard to write a specification for it in one go. The programmer is often forced to waste a few seconds and look at the function declaration to check which parameters were not yet declared or the type of a specific parameter. This was not evident in our fifteen experiments, see Figure 7, since after one or two experiments most of the parameters and types were already memorized. The deviation increased a bit but we believe this was caused by the increase in time and not so much by the previously mentioned problem.

It is important to note that a programmer using these alternatives will not write the same procedure repeatedly like what we did for the experiments. If we did one or two experiments for different users the results might have showed some significant differences. This is also one of the reasons why we decided to do five experiments instead of ten for this case, because this would probably make the average value go lower.

![Figure 7: Distribution of the fifteen experiments in groups (quartiles), for the createWalls procedure, through a whiskers plot for each alternative.](image)

In our solution, based on DI, we have already seen that it is not necessary to specify the parameters of the function, as well as their types, Listing 6. The only thing that our solution is dependent on, is the name of the function. Therefore it does not matter if the function has three or eleven arguments since the types are deduced by the compiler as we have seen in the previous chapter. The time spent adding a procedure is almost the same as with the createCircle operation, Figure 6, and the deviation from the average value is also minimal, as is illustrated in Figure 7.

If a plug-in were to implement, for example, ten procedures with many arguments (ten or more), the time expended with our solution to make them remotely available would probably be around 100 seconds (10 seconds for each procedure, according to our results). However, if we used an approach which relies on an IDL, based on our results, it would take somewhere between 20-30 minutes to add ten procedures (around 2-3 minutes to each procedure).

It is important to note that this is not an unrealistic example, for instance, most BIM software applications define their elements, such as, walls, doors, stairs, and slabs, with many attributes, such as, material, type, angles, height, story, and many more. This means that if one wants to implement a procedure to create one of these elements it will have to specify most of this attributes. Overall a plug-in used for GD will have dozens of these operations.

4.2. Runtime performance analysis

Before evaluating the runtime performance of our solution, it is important to recall that the time spent on the remote invocation of a procedure is dependent on two variables: the time which is spent with the client-server communication, and the time spent by the software applications executing the procedure’s code.

Notice that the programmer has no control over the second variable. For example, in our solution
we are using ArchiCAD’s API to implement procedures on a plug-in. When implementing a procedure it is necessary to invoke several functions of that API, however, we have no control over the operations performed by these functions. For that reason, the only thing which can be controlled by us when performing a RPI is the time spent on communication.

In Table 2 we show the results of the analysis performed on the time spent in communication and procedure execution of a createWalls operation visible in Listing 7. Note that the total is the interval that goes from the start of the serialization of the request message, on the client, to end of the deserialization of the response message, also on the client, after the invocation (Round Trip Time). The procedure execution is the time spent executing the createWalls function.

Table 2: The difference between the time during the procedure execution and the time spent in communication for different numbers of walls.

<table>
<thead>
<tr>
<th>#Walls</th>
<th>Execution (ms)</th>
<th>Communication (ms)</th>
<th>Total (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>30.813 (97.17%)</td>
<td>0.899 (2.83%)</td>
<td>31.712</td>
</tr>
<tr>
<td>2</td>
<td>57.273 (98.42%)</td>
<td>0.922 (1.58%)</td>
<td>58.195</td>
</tr>
<tr>
<td>3</td>
<td>86.785 (98.91%)</td>
<td>0.954 (1.09%)</td>
<td>87.739</td>
</tr>
<tr>
<td>4</td>
<td>129.605 (99.14%)</td>
<td>1.124 (0.86%)</td>
<td>130.729</td>
</tr>
<tr>
<td>8</td>
<td>250.524 (99.51%)</td>
<td>1.243 (0.49%)</td>
<td>251.767</td>
</tr>
<tr>
<td>16</td>
<td>515.440 (99.73%)</td>
<td>1.394 (0.27%)</td>
<td>516.834</td>
</tr>
</tbody>
</table>

The results show that most of the time is wasted on the execution of the procedure and only a small percentage is spent in the communication. Moreover, the more complex the procedure is, the more accentuated the difference is, even though, for every single wall four more values (wall's coordinates) will have to be serialized and deserialized. For instance, for 16 walls, 64 double values (32 coordinates) will have to be transferred along with 9 more arguments. However, the impact of serializing 60 more doubles is only around 0.5 milliseconds (difference between the communication time for 16 and 1 wall). On the other hand, creating/ drawing 16 walls makes the execution of the procedure 485 milliseconds slower than the creation of just one wall (difference between the execution time of 16 walls and 1 wall).

As a BIM software application, most of the procedures implemented on ArchiCAD will also have similar results. That is these procedures generate different objects, such as, doors, beams, stairs, slabs, and many more.

Realistically, when using these types of software applications which take considerable amounts of time to execute a procedure, it is not worth to use efficient mechanisms of data serialization with data compression based on IDL, such as Protocol Buffers, and sacrifice productivity during development because of a few tenths of milliseconds. It is much more appealing to gain a few dozens of minutes during development.

It is important to note that our solution does not depend on a specific serialization method. If a software application benefits from rapid serialization and deserialization of data, a faster protocol can be used alongside our solution to enable fast serialization and better productivity during development.

5. Conclusions

In this paper we talked about how some approaches, used to enable RPI, negatively impact the development of plug-ins for software applications by relying on IDLs to specify remote interfaces and message formats, hindering the productivity when adding new procedures to a plug-in. To try and solve this issue we introduced DI, a mechanism that enables RPI through the use of metaprogramming techniques. We presented an implementation of DI in C++ which uses techniques of C++ templates, and showcased the work necessary to add new procedures during development.

We evaluated our solution in terms of: productiveness when adding new procedures to the RPI mechanism; and performance. Our solution was compared with two other approaches, one which used function pointers and Protocol Buffers as serialization method, and another which used the Apache Thrift RPI framework. Both of these approaches relied on the specification of data in an IDL file. The results showed that our solution saves a lot of the programmer’s time during development and it is not affected by a procedure’s arity (i.e., number of parameters of a function). For a procedure with eleven parameters, our solution required, on average, 10 seconds to add the procedure. On the other hand, other approaches which used an IDL took almost 3 minutes to make the same procedure remotely available, proving our point that the writing of more information in an IDL file severely hinders the productivity during development.

In terms of performance we saw that in some cases the software application spends most of the time executing the code of a procedure, i.e., executing the calls made to the server application’s API, and that the time wasted with the communication is barely significant. This means that it is not relevant to improve RPI performance since we have no control over the software application’s API and, thus, the performance gains might be insignificant in the overall RPI time.

In this solution, we valued productiveness when adding new remote procedures since some of the current approaches, based on IDLs, severely impact the development. And, the evaluation done in
this work showed that this goal was attained.

References


