TÉCNICO LISBOA

Dynamic Invocation

Tiago Miguel Santos Rodrigues

Thesis to obtain the Master of Science Degree in

Information Systems and Computer Engineering

Supervisor: Prof. António Paulo Teles de Menezes Correia Leitão

Examination Committee

Chairperson: Prof. António Manuel Ferreira Rito da Silva
Supervisor: Prof. António Paulo Teles de Menezes Correia Leitão
Member of the Committee: Prof. Alexandre Paulo Lourenço Francisco

October 2019

Declaration

I declare that this document is an original work of my own authorship and that it fulfills all the requirements of the Code of Conduct and Good Practices of the Universidade de Lisboa.

Acknowledgments

I would like to thank my parents for their friendship, encouragement and caring over all these years, for always being there for me and without whom this project would not be possible. I would also like to thank my grandparents for their support throughout of these years.

I would also like to acknowledge my dissertation supervisor Prof. António Leitão for his insight, support and sharing of knowledge that has made this thesis possible.

Last but not least, to all my friends and colleagues that helped me grow as a person and were always there for me during the good and bad times in my life.

To each and every one of you – Thank you.

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 Model (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 IDLs to make RPIs.

Keywords

Remote Procedure Invocation; Interface Description Language; Productivity; Metaprogramming; Dynamic Invocation; Templates.

Resumo

Hoje em dia, há um grande interesse em invocar remotamente a functionalidade disponibilizada por aplicações de software (e.g., editores de texto, ferramentas CAD, ferramentas BIM, motores de jogos). Para este fim, é necessário aceder remotamente à funcionalidade disponibilizada pelas aplicações de software a partir das suas APIs. Desta forma, os plug-ins destas aplicações têm de implementar um mecanismo que possibilite a RPI. No entanto, as abordagens correntes, utilizando IDLs para estruturar mensagens de pedido e resposta, e especificar a interface remota de um servidor, afetam negativamente a produtividade durante o desenvolvimento dos plug-ins. Isto porque, quando é preciso adicionar novos procedimentos, a especificação desta informação tem de ser feita manualmente em ficheiros IDL. Por isso, nós introduzimos a DI, um mecanismo que, baseado nas capacidades de linguagens de programação, usa metadados para invocar remotamente os procedimentos implementados num plug-in, melhorando a produtividade. A DI utiliza técnicas de metaprogramação para abstrair os detalhes de uma RPI, por exemplo, a definição da estrutura das mensagens, a ordem de serialização, e a forma de invocação de um procedimento.

Como o esforço para implementar DI depende das capacidades de metaprogramação das linguages, nós vamos apresentar uma implementação de DI em C++, uma linguagem que possui estas capacidades através de métodos complexos. Vamos demonstrar como as características dos C++ templates ajudam a contornar esta dificuldade para no final obter uma solução que não afete a produtividade de forma tão acentuada como as outras abordagens.

Palavras Chave

Remote Procedure Invocation; Interface Description Language; Produtividade; Metaprogramação; Dynamic Invocation; Templates.

Contents

List of Figures

List of Tables

List of Algorithms

Listings

Acronyms

API	Application Programming Interface
BIM	Building Information Model
CAD	Computer-Aided Design
DI	Dynamic Invocation
GD	Generative Design
GUID	Global Unique Identifier
IDL	Interface Description Language
RPI	Remote Procedure Invocation

1

Introduction

Contents

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 Computer-Aided Design (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 Application Programming Interfaces (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. For example, Microsoft Visual Code, a source-code editor, offers a full-featured development environment through a remote development feature[1]. With this programmers can:

- Develop and Debug their code remotely without impacts on local machine's performance and configuration;
- Avoid hindering their development by continually sending the code to the remote machine that is supposed to execute it.

In the next section we provide a more precise example of the benefits of remotely accessing a software application plug-in's API.

1.1 Example: Generative Design

Generative Design (GD) consists in a process that allows the design of models through algorithms [3]. 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.

Most CAD software applications provide APIs that enable GD. However, different software applications use advanced programming languages for their APIs, making it harder for less experienced programmers, such as architects, to implement plug-ins for these software applications. Not only that but the scheme in which the plug-ins are set-up varies from software application to software application.

Figure 1.1 shows a clear example of this. On the left we have the scheme to implement a plug-in in Revit, in this case, the implementation of the plug-in must be inside an `Execute` method of a `IExternalCommand` C# subclass. While on the right, ArchiCAD scatters the code to set-up the plug-in across different C++ functions, each of them with their own purpose.

[1] https://code.visualstudio.com/docs/remote/remote-overview

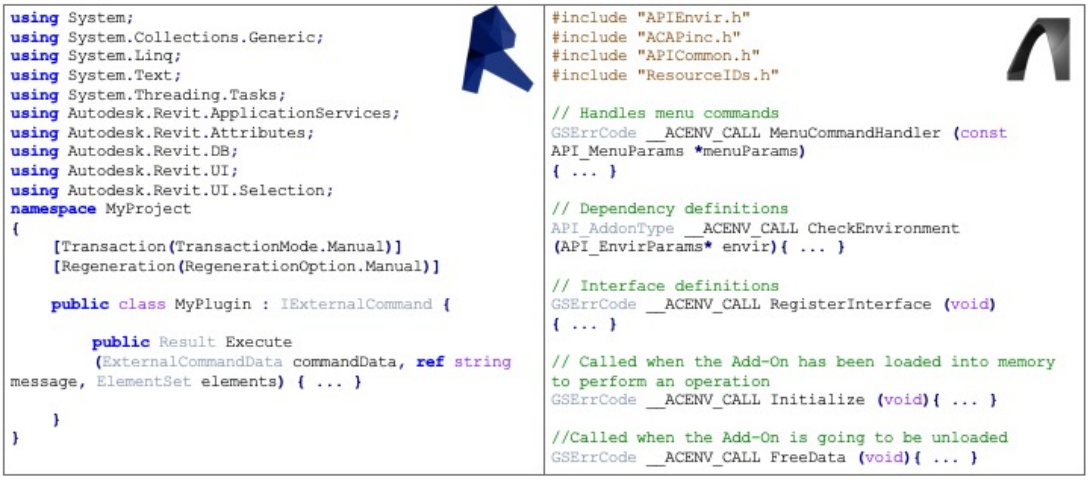

Figure 1.1: Revit scheme (left) vs. ArchiCAD scheme (right).

In [2], a plug-in for ArchiCAD, in C++, was developed with the intent to remotely provide different CAD and Building Information Model (BIM) operations for GD. This allowed architects to do GD through a client program using very simple directives provided by an abstraction component (see Listing 1.1 for an example). This component not only abstracted the ArchiCAD set-up scheme, and communication, but also complex details needed to implement a procedure using ArchiCAD's API (see Listing 1.2).

Listing 1.1: Example of client code to create a slab on ArchiCAD. Retrieved from [2].

```
slab(xy(0, 0), xy(5, 0), xy(5, 5), xy(0, 5))
```

Listing 1.2: Code to create a slab on an ArchiCAD plug-in. Retrieved from [2].

```
void createSimpleSlab(){
    API_Element element;
    API_ElementMemo memo;
    if (ACAPI_OpenUndoableSession("Session") != NoError){
        ErrorBeep("slab");
        return;
    }
    BNZeroMemory(&element, sizeof(API_Element));
    BNZeroMemory(&memo, sizeof(API_ElementMemo));
```

```
    element.header.typeID = API_SlabID;
    if (ACAPI_Element_GetDefaults(&element, &memo) != NoError) {
        ErrorBeep("ACAPI_Element_GetMemo");
    }
    element.header.floorInd = 0;
    element.slab.level = 0.0;
    element.slab.modelElemStructureType = API_CompositeStructure;
    element.slab.buildingMaterial = 2;
    element.slab.poly.nCoords = 4;
    element.slab.poly.nSubPolys = 1;
    element.slab.poly.nArcs = 0;
    memo.coords = reinterpret_cast<API_Coord**>
            (BMAllocateHandle((element.slab.poly.nCoords + 1)
                        * sizeof(API_Coord), ALLOCATE_CLEAR, 0));
    memo.pends = reinterpret_cast<Int32**>
            (BMAllocateHandle((element.slab.poly.nSubPolys + 1)
                        * sizeof(Int32), ALLOCATE_CLEAR, 0));
    memo.parcs = reinterpret_cast<API_PolyArc**>
            (BMAllocateHandle((element.slab.poly.nArcs)
                        * sizeof(API_PolyArc), ALLOCATE_CLEAR, 0));
    if (memo.coords == NULL || memo.pends == NULL || memo.parcs == NULL) {
        ErrorBeep("Not enough memory to create slab polygon data", APIERR_MEMFULL);
        ACAPI_DisposeElemMemoHdls(&memo);
    }
    (*memo.coords)[1].x = 0.0;
    (*memo.coords)[1].y = 0.0;
    (*memo.coords)[2].x = 5.0;
    (*memo.coords)[2].y = 0.0;
    (*memo.coords)[3].x = 5.0;
    (*memo.coords)[3].y = 5.0;
    (*memo.coords)[4].x = 0.0;
    (*memo.coords)[4].y = 5.0;
    (*memo.coords)[5] = (*memo.coords)[1];
    (*memo.pends)[1] = element.slab.poly.nCoords;
    if (ACAPI_Element_Create(&element, &memo) != NoError){
        ErrorBeep("ACAPI_Element_Create (slab)");
    }
    ACAPI_DisposeElemMemoHdls(&memo);
```

```
48      ACAPI_CloseUndoableSession();
49  }
```

In order to communicate with the client programs, software application's plug-ins need to implement a mechanism that allows its functionality to be remotely invoked.

With this work we aim to study different methodologies that can be used to develop Remote Procedure Invocation (RPI) mechanisms, i.e., a mechanism which can be attached to a plug-in to remotely expose the APIs procedures. In the next section we will explain the RPI mechanism and present some of the tools that enable it.

1.2 Remote Procedure Invocation

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. Although request/reply-based protocols are quite simple, the implementation of a RPI mechanism is more complex, since its difficulty lies more on the:

- Client-server dependability, i.e., client and server can use different technologies (e.g., programming languages), which most likely have different means of representing data, and their interfaces composition [4]. A form of consensus to represent and expose data is needed. For example, by using a serialization method or data format for the communication;
- Implementation of a process to make the server invoke its own procedures based on the client's request data. This also depends on the programming language used to implement the server, some languages might have features that ease this process while others make it more difficult. For example, some metaprogramming strategies, such as reflection, are able to interpret the request data (metadata) and have the language invoke its own functions/methods from it.

Looking to reduce the extra effort needed from programmers to solve these issues, many RPI frameworks, such as gRPC [5] and Apache Thrift [6], have been developed. Both gRPC[2] and Apache Thrift[3] already support a vast set of programming languages in different platforms, and abstract the most part of their process of remote invocation.

Most of these frameworks achieve this with the use of a language and platform neutral schemes in the form of a Interface Description Language (IDL). In this case, IDLs are used to define the structure of additional types and server's remote interface as services. Files written in these languages can be compiled to generate code for (1) definition of additional types specified by the programmer and their

[2] https://grpc.io/about/
[3] https://thrift.apache.org/docs/Languages

corresponding serialization methods, (2) client and server stubs responsible to dispatch the procedure's arguments, and (3) service handlers where the procedures are implemented or called.

Firstly, in order to make a RPI, 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 the service stub dispatches the procedure's arguments to the corresponding service handler where the procedure is invoked. In Figure 1.2, the overall architecture of these frameworks can be seen. Note that some RPI frameworks can use different server configurations, serialization methods and transport protocols. It is up to the user to specify which one to use.

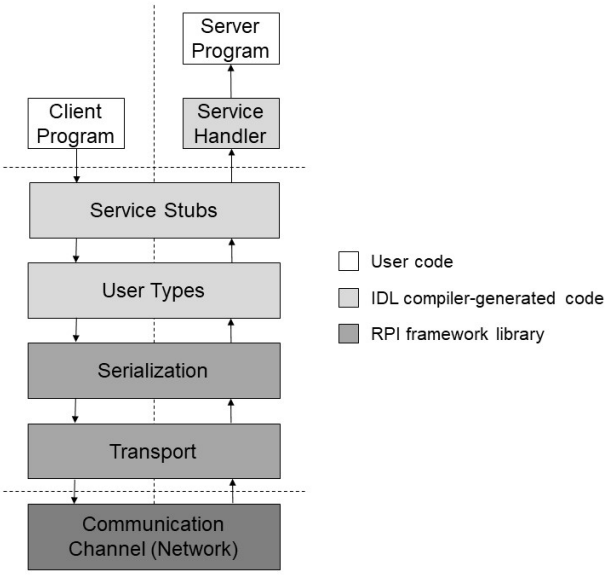

Figure 1.2: Overall architecture of an RPI framework. Inspired from [1].

Furthermore, to add new procedures to the application, the programmer follows a workflow which can be described in the following steps:

1. Create an IDL to define the server's remote API. See Listing 1.3, for an example of a gRPC IDL file;

2. Compile the IDL file to automatically generate the code for the stubs, serialization functions of user types (if specified), and service handler classes where the procedures will be implemented;

3. Add the respective generated files to the client and server projects. Most tools allow the specification of destiny folders at compilation;

4. Write the client's code which uses stubs generated, during the IDL compilation, to communicate with the server, see Listing 1.4;
5. Write/Call the implementation of the procedure in the service handler class generated after compilation, see Listing 1.5;
6. Register the new service when setting up the server, see Listing 1.6;
7. Compile and execute client and server code, respectively.

Listing 1.3: Example of an IDL file for gRPC for a Calculator service.

```
service Calculator {
  rpc Add (AddRequest) returns (AddReply) {}
}

message AddRequest {
  int32 a = 1;
  double b = 2;
}

message AddReply {
  double r = 1;
}
```

Listing 1.4: Example of the client's side, in Python, required code to invoke the stub of the Calculator service in gRPC (line 5).

```
1 #Client side code
2 def run():
3   channel = grpc.insecure_channel('localhost:50051')
4   stub = calculator_pb2_grpc.CalculatorStub(channel)
5   response = stub.Add(calculator_pb2.AddRequest(2, 3.0))
6   print("Greeter client received: " + response.r)
```

Listing 1.5: Example of the generated and implemented procedure for the Calculator service, in C++. Only line 5 is written by the programmer, the rest is automatically generated.

```
1 //Procedure code generated and implemented (lines 5-7)
```

```
2 class CalculatorServiceImpl final : public Calculator::Service {
3   Status Add(ServerContext* context, const AddRequest* request,
4       HelloReply* reply) override {
5     reply->set_message(request->a() + request->b());
6     return Status::OK;
7   }
8 };
```

Listing 1.6: Example of the server side code to register the service (line 5), in C++, using gRPC.

```
1 int main(){
2     CalculatorServiceImpl service;
3     ServerBuilder builder;
4     builder.AddListeningPort("0.0.0.0:50051", grpc::InsecureServerCredentials());
5     builder.RegisterService(&service);
6     std::unique_ptr<Server> server(builder.BuildAndStart());
7     return 0;
8 }
```

While the code generation simplifies the implementation of remote invocation, the writing of the IDL file and handlers can make the addition of new procedures tedious and monotonous, hindering productivity during development. For example, if there was a need to add a new procedure to the previous example, the process described would have to be repeated regardless of how simple the service was.

Alternatively, instead of using a RPI framework, programmers can use solely IDL-based methods to serialize and deserialize messages. One example of such methods is Protocol Buffers [7] which is the default serialization method used by gRPC. 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 1.3. Note that with this solution the programmer has no abstractions for the transport protocols, having to use one to send and receive the serialized data over a communication channel.

In [2], a plug-in for ArchiCAD was implemented based on this approach, the implementation uses C++ function pointers to invoke the different procedures that the plug-in provides, and Protocol Buffers to serialize and deserialize the messages sent between the client application and the plug-in. However, as we will see in the next chapter, Chapter 2, the usage of an IDL only for the serialization of structured data does not simplify the process of adding new procedures.

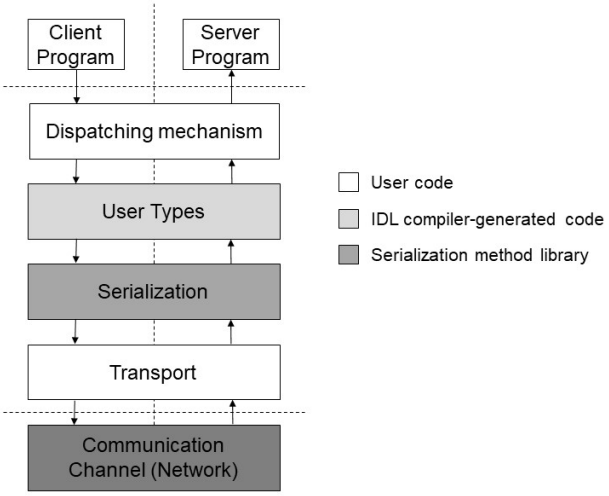

Figure 1.3: Overall architecture of an RPI mechanism with a serialization method. Inspired from [1].

1.2.1 Remote Procedure Invocation in software application plug-ins

The focus of this thesis is to find a solution to decrease the effort required to make a procedure remotely available on a software application plug-in (server). The reason why we will focus on this aspect is because some software applications impose restrictions on the plug-in's characteristics which make them not benefit as much from the abstractions of RPI frameworks or from the efficiency of some serialization methods.

For example, if during the execution of a software application plug-in the user interface is blocked, and the software application 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 software application will always be blocked and multi-threading will not work properly.

In the same manner, it does not matter if a serialization method is extremely efficient if most of the time of the remote invocation is spent by the software application executing a procedure. For instance, a BIM application might take 40 seconds to create a building model, however, from those 40 seconds, less than a second is spent with communication 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 implementation of the functions provided by the API, we decided that it would be more appealing if we could find a way to save time during development. This could be done by implementing a mechanism which would not require the programmer to waste seconds, or even

minutes, specifying the parameters of a RPI in an IDL file.

Thus, in the next section, we will introduce Dynamic Invocation (DI), a mechanism that enables RPI without requiring the user to explicitly define the structure of messages sent between client and server.

1.3 Dynamic Invocation

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 RPI approaches because its use should not require the specification of services and/or messages on IDLs. 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 [8], 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.

We will study two possible alternatives based on metaprogramming, which, using metadata, automatically define a deserialization order of procedure's parameters and dispatch them to the desired function without the need to define request and response messages. The two alternatives are based on: Reflection and C++ Templates.

The first alternative 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 [8] and is composed by two activities [9]:

- 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 [8].

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 programatically assemble bits of code which will produce a certain effect at runtime [10]. In other words, the idea is to have the compiler do the work of assembling the code that deserializes arguments, and that dispatches them to a procedure.

We will explain and propose a solution that takes advantage of C++ templates by using compile-time generated linked typelists whose elements types are deduced from the procedure's parameters and forwarded to the procedure. Because the compiler does most of the work, the programmer does not have to waste any time defining the structure of request and response messages.

1.4 Objectives

This thesis aims to solve the problems associated with IDL-based mechanisms which delay the addition of new procedures to software application plug-ins. Therefore, the concept of DI is introduced as a mechanism that makes RPI possible through the use of many different programming languages capabilities. The result of this work will be an implementation of DI in C++ which takes significantly less time to add new remote procedures. The solution will be evaluated and compared with other approaches which use the RPI framework Thrift and serialization method of Protocol Buffers to enable RPI.

1.5 Organization of the Document

This thesis is organized as follows: Chapter 1 introduces the motivation, state of art, and DI. In Chapter 2 we will study two approaches for RPI which use a RPI framework and a serialization method with an IDL. We will also propose a few implementations for DI. Chapter 3 gives a detailed view on the implementation and decisions taken for our solution which are then discussed and compared to other approaches in Chapter 4. Chapter 5 concludes the report along with the future work.

2

Related Work

Contents

In the previous chapter we already showed how some RPI frameworks, such as gRPC, affected the development of applications especially, when adding new functionality in the form of remote procedures. In this chapter we are going to analyse where different approaches to the mechanisms previously presented stand when developing a plug-in that allows RPI.

Firstly, we will demonstrate how one can add new remote procedures using the RPI framework Apache Thrift. Secondly, we will analyse a software application plug-in already developed for ArchiCAD in C++, which uses the serialization and deserialization mechanisms of Protocol Buffers through an IDL: we will explain its implementation (i.e., how procedures are invoked and how parameters are dispatched to the procedures) and how the addition of newer functionality is done. Lastly we discuss possible implementations of DI depending on metaprogramming capabilities and analyse how difficult it is to add new procedures for each language, emphasizing on the C++ case which will be later addressed on this thesis.

2.1 RPI with an IDL - Apache Thrift

Apache Thrift is a RPI framework, which, just as gRPC, utilizes an IDL to specify a remote interface of a server application [6]. In the IDL, the methods of a remote interface are declared as services, and, after the IDL compilation, this services generate stubs for client and server applications. An example of a Thrift IDL file is shown in Listing 2.1. Note that unlike gRPC (see Listing 1.4), Thrift does not require the programmer to define specific types for request and response messages.

Listing 2.1: Example of an `add` remote method which returns the result as the sum of a 32-bit integer and a double.

```
service Calculator {
    double add(1:i32 a, 2:double b)
}
```

The client service stub is used by the client application to invoke the remote procedures of the service, as exemplified in Listing 2.2. The stub is responsible to serialize the parameters passed in the remote invocation (line 8), and send them to the server following the chosen transport protocol. It is important to notice that the implementation is not dependent on serialization and transport protocols. The programmer is free to choose which protocols to use (lines 1-3).

Listing 2.2: Using the client stubs of Thrift for a remote invocation of `add`, in C++.

```
1 auto socket = make_shared<TSocket>("localhost", 9090);
2 auto transport = make_shared<TBufferedTransport>(socket);
```

```
3  auto protocol = make_shared<TBinaryProtocol>(transport);
4
5  CalculatorClient client(protocol);
6  try{
7    transport->open();
8    client.add(2, 3.0);
9    transport->close();
10 } catch (TException& tx) {
11   cout << "ERROR: " << tx.what() << endl;
12 }
```

When the invocation request is received the server stub is used to deserialize the values of the parameters sent by the client and dispatch them to an handler, which is also generated automatically for each service. However the programmer has to write the code to be executed in the corresponding method of the handler, as visible in Listing 2.3 (line 8).

Listing 2.3: Handler generated for the Calculator service, in C++. The programmer has to write the implementation of the procedure on the corresponding method, line 6-9.

```
1  class CalculatorHandler : virtual public CalculatorIf {
2   public:
3    CalculatorHandler() {
4      // Your initialization goes here
5    }
6    double add(const int a, const double b) {
7      // Your implementation goes here
8      return a + b;
9    }
10 };
```

Additionally, to make the service handler remotely available it is necessary to add it to the server during its initialization, as exemplified in Listing 2.4. The protocols specified by the client in Listing 2.2 (lines 1-3) need to be compatible to the ones specified on the server in Listing 2.4 (lines 3-5). Thrift also allows the programmer to use different server configurations. In this example, a `TSimpleServer` instance was used which creates a simple single-threaded blocking server. However, if the programmer used an instance of `TThreadedServer`, a multi-threaded server with a thread per connection and blocking I/O would be used.

Listing 2.4: Adding the Calculator handler (line 2) to a instance of `TSimpleServer`, in C++.

```
1 TSimpleServer server(
2   std::make_shared<CalculatorProcessor>(std::make_shared<CalculatorHandler>()),
3   std::make_shared<TServerSocket>(9090),
4   std::make_shared<TBufferedTransportFactory>(),
5   std::make_shared<TBinaryProtocolFactory>());
6
7   server.serve();
8 };
```

Although these abstractions, which allow the programmer to choose different types of complex protocols and server configurations, can be one big advantage of using RPI frameworks, most of the times, as we have mentioned in the previous chapter, software applications restrict the protocols and configurations which can be used. In these cases, there is no real benefit from abstracting a complex configuration, such as, multi-threaded non-blocking servers, if the result is essentially the same as using a single-threaded blocking server.

As a consequence, the programmer sacrifices the development time of the plug-in and does not get much benefit out of it. 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 (Listing 2.1); compile the IDL; implement the method of an handler (Listing 2.3); and, if the method is implemented in a new service/handler, add the handler to the server (Listing 2.4). 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, the programmer will always waste a few seconds writing procedure specifications until he gains some experience.

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.

2.2 RPI with an IDL - Protocol Buffers

Instead of using a RPI framework, one can implement his own communication mechanism, and use a serialization method to ensure that client and server applications communicate with each other. Note that there are serialization methods, such as Protocol Buffers, which rely on the specification of data through an IDL [7]. Although, instead of specifying services and their methods like RPI frameworks, Protocol Buffers specify a structure to request and response messages for each remote invocation.

In [2], a plug-in for a software application (ArchiCAD) was implemented in C++. The solution used Protocol Buffers as the method for serializing data between client and server applications. Since Protocol Buffers only provide serialization functions for the types defined in an IDL file, in the next subsection, we will see how the author of this solution implemented a form of identifying which procedure should be invoked, based on the client request, and how to dispatch the arguments in the request message to the procedure.

2.2.1 Implementation

In this solution, to identify procedures, the author opted to do a mapping of function identifiers to function pointers. It is important to notice that this is necessary for C++ since it does not possess any forms of reflection, thus, the only way to refer to a function pointer through an identifier, is to implement a map. For the identifiers, the names of the procedures/functions were used, as `std::string`, and for the function pointers, the choice was to implement all procedures as `void(*)(void)` function pointer types.

Whenever the identifier of the function is obtained, after the deserialization of data, the corresponding pointer is invoked (see Listing 2.5, line 7), by finding the correct iterator of the map with the function identifier (line 2).

Listing 2.5: Code that showcases how the functions of the ArchiCAD plug-in with Protocol Buffers were invoked. Retrieved from [2].

```
1 void callScript(const std::string& pFunction){
2     scriptMap::const_iterator iter = m.find(pFunction);
3     if (iter == m.end()){
4         quit();
5     }
6     else{
7         (*iter->second)();
8     }
9 }
```

One might wonder how can every function be mapped into `void(*)(void)` when every procedure should receive a different type and number of arguments, and also return other types besides `void`. However, in this case, the author managed to do this by reading and writing the data directly from the socket in the procedure's implementation.

By taking a closer look at Listing 2.6, one can see that, in fact, in line 7, there is a call to read data from the socket and populate the fields of the message (`circleMsg`) (request message defined in the Protocol Buffers IDL), which are then used in lines 12-14. Also, in line 28, there is a function call to write the Global Unique Identifier (GUID) of the newly created element in the socket stream back to the client.

Listing 2.6: Implementation of a procedure to create a circle in an ArchiCAD plug-in with Protocol Buffers. Retrieved from [2].

```
1  void createCircle(){
2      API_ElementMemo memo;
3      API_Element     circleElement;
4      GSErrCode       err;
5      circlemessage   circleMsg;
6
7      readDelimitedFrom(getClientSocket(), &circleMsg);
8
9      BNZeroMemory(&memo, sizeof(API_ElementMemo));
10     BNZeroMemory(&circleElement, sizeof(API_Element));
11
12     circleElement.circle.origC.x = circleMsg.p0x();
13     circleElement.circle.origC.y = circleMsg.p0y();
14     circleElement.circle.r = circleMsg.radius();
15
16     circleElement.circle.ratio = 1.0;
17     circleElement.header.typeID = API_CircleID;
18     circleElement.header.layer = 1;
19     circleElement.header.floorInd = 0;
20
21     err = ACAPI_Element_Create(&circleElement, &memo);
22     if (err != NoError){
23         //createCircleNoCom();
24         ErrorBeep("ACAPI_Element_Create (circle)", err);
25     }
26     sendElementID(getClientSocket(), circleElement);
```

```
27
28      ACAPI_DisposeElemMemoHdls(&memo);
29  }
```

Although this solution simplifies the process to invoke a function in C++, it suffers from a tight coupling problem between the code necessary to do an RPI and the procedures implemented in the plug-in. The `createCircle` function cannot be compiled without a client socket. Ideally, the procedures should be able to be compiled, executed and work properly independently of the sockets. This way not only the procedures are independent but it also opens the possibility to do unit testing on them.

While the procedures should not depend on the RPI mechanism, the RPI needs to include the procedure's declarations in its scope, otherwise it would not be able to invoke and map them.

It is important to note that these implementation problems are related to the overall architecture of the solution. Therefore, the fact that Protocol Buffers were used as a serialization method has no influence in these issues. In the next section we will explain how to add a new procedure with this solution and study the impact that Protocol Buffers have in this process.

2.2.2 Addition of new procedures

For the sake of simplicity, an example will be used to demonstrate how the addition of a new procedure is done in this solution. If one desired to add a new procedure `add` which would receive an integer and a double, as arguments, and return the sum of both as a double, the following steps should be taken:

1. Add to the Protocol Buffers IDL file the structure of the request and response message, as illustrated in Listing 2.7;
2. Compile the IDL file to generate the classes for each message type and their corresponding serialization and deserialization methods;
3. Implement the `add` procedure on the plug-in as presented in Listing 2.8.
4. Add the new implemented `add` function to the mapping of function pointers so it can be invoked with the mechanism described in Listing 2.5;
5. Compile and execute the code.

Listing 2.7: Code that should be added to the Protocol Buffers IDL to implement a two argument `add` procedure.

```
1 message AddRequest {
2   required int32 a = 1;
```

```
  required double b = 2;
}

message AddResponse {
  required double result = 1;
}
```

Listing 2.8: Implementation of a remote `add` procedure that sums an integer and a double and returns its result.

```
void add(){
    AddRequest  addRequest;
    AddResponse addResponse;

    readDelimitedFrom(getClientSocket(), &addRequest);

    addResponse.set_result(addRequest.a() + addRequest.b());

    writeDelimitedTo(getClientSocket(), addResponse);
}
```

Aside from the absence of writing code on an handler this entire process is analogous to the one of RPI frameworks, showed in Section 1.2 and in Section 2.1. In the end, for this solution, there is no evident benefit from using serialization methods with IDLs and using an RPI framework when adding new procedures. None of these approaches is ideal to our problem since plug-ins are always evolving and gaining more functionality depending on the needs of their users. Therefore, IDL-based approaches should not compose an optimal solution for our problem.

One simple way to solve this problem, in this example, would be to use a serialization method which does not use an IDL to generate code for the serialization and deserialization of data. There are a vast number of serialization methods that do not adopt the specification of messages on a neutral language representation, such as IDLs. If we assume that there is an implementation, in the form of a library, for a serialization method which provides the following methods to serialize and deserialize data:

- `readInt` - Receives the client socket as argument and returns an integer value read from the socket stream;
- `readDouble` - Receives the client socket as argument and returns a double value read from the socket stream;
- `writeDouble` - Receives the client socket and double value as argument and writes the value to

the socket stream;

Using these directives we could change the implementation of `add` in Listing 2.8 to Listing 2.9.

Listing 2.9: Changed implementation of the `add` procedure which does not use IDL.

```
1 void add(){
2     int a,b;
3
4     a = readInt(getClientSocket());
5     b = readDouble(getClientSocket());
6
7     writeDouble(getClientSocket(), a + b);
8 }
```

Although there is no more need for an IDL, a new problem surged with this implementation. The programmer imposes the order in which arguments are read from the socket stream. Previously this job was done automatically in the code generated by the IDL. Despite not wasting time with the writing of a IDL file, this quick fix is much more error prone. If the programmer reads data in the wrong order, errors will occur. These type of errors can pass undetected, resulting in strange behaviour of the program. They are also difficult to debug, especially if the serialization method is binary (i.e., the data is hard to read). In the Protocol Buffers case such errors would be detected when compiling the IDL file.

This is also one of the reasons why we introduced DI in this work. The objective is to infer an order of deserialization of the arguments from the procedure's parameter types, without having the programmer specify it manually, either on an IDL or in his code. However, this inference is difficult to achieve. While some programming languages which have reflection mechanisms can achieve this with introspection by checking the types of the function's arguments, some languages, such as C++, do not possess such means. Therefore we have to resort to more advanced techniques of the language, such as C++ templates, to achieve the same benefits as other programming languages with reflection. This way procedures can be implemented as normal C++ functions, as exemplified in Listing 2.10, without any dependencies to the RPI mechanism code. In the next section we will see how this problem is solved by different programming languages.

Listing 2.10: Ideal implementation of the `add` procedure that would be remotely available.

```
1 double add(int a, double b){
2     return a + b;
3 }
```

2.3 Dynamic Invocation Examples

The complexity of implementing DI greatly varies from programming language to programming language. Some programming languages, such as, Python, Julia and Ruby, often named as scripting languages, abstract most of their implementation details at runtime (e.g., type-checking and memory management), making them much more dynamic and less verbose. As a consequence of that, they usually provide powerful operations which can greatly simplify the process of DI.

However, although it could be advantageous to opt only for scripting languages to implement DI mechanisms, not every software application provide APIs for scripting languages. Some of them opt for more static and verbose languages which usually require a compilation step, before execution, to improve runtime performance through optimizations done by the compiler. Because of that, these languages are often preferred to implement software applications over scripting languages, thus, it is more easy to provide the API in the language which the software application was implemented on. Nonetheless, compiled languages do not support the same powerful operations of scripting languages. Therefore, the implementation of DI is way more challenging.

In the next sub sections we will explain different DI mechanisms which can be implemented depending on the programming language used.

2.3.1 Using eval functions - Intercession

Access to the language evaluator is one of the most powerful operations possessed by scripting languages. To allow the evaluation of code at runtime, the language provides eval functions that can take a representation of the programming language's expressions as arguments and evaluate them.

A function call is an expression, which means that if an eval function receives a representation of code that corresponds to a function call it should be able to invoke it. To enable DI we can implement a very simple remote expression interpreter which evaluates expressions at runtime. For a Julia's example, see Listing 2.11.

Note that, in line 5, we read a line from the socket stream, as a string, and parse it using the `Meta.parse`, passing the resulting abstract syntax tree to the eval function in line 6. This happens because Julia's `eval`[1] does not evaluate code as a string, but as type that represents Julia's code, in this case an abstract syntax tree. This is the result of Julia possessing a property of programming languages which allows them to represent their own programs through data, called homoiconicity. Homoiconicity eases the implementation of reflection mechanisms on programming languages, allowing powerful operations to be used in a programming language, such as eval functions.

Even though some languages might not possess homoiconicity, they can still provide eval functions,

[1] https://docs.julialang.org/en/v1/base/base/#Base.MainInclude.eval

for example, Python has an `eval`[2] function that receives expressions as arguments in the form of a string.

Listing 2.11: Example of a simple remote interpreter in Julia that evaluates strings as Julia expressions.

```julia
function remote_interpreter()
    server = listen(8080)
    conn = accept(server)
    while true
        line = readline(conn)
        result = eval(Meta.parse(line))
        write(conn, result)
    end
    close(conn)
end
```

Alternatively, the remote interpreter does not need to receive a valid expression to invoke a function, it can receive an arbitrary message in a different format. As long as the message has an identifier to identify the function (e.g., name) and its arguments for the function call, the interpreter can decode/deserialize both function name and arguments, and manually build a function call expression for eval, as exemplified in Listing 2.12. This example is very specific to Julia. For Python the solution would probably involve the concatenation of strings since its eval function receives a string as argument.

Listing 2.12: Example of a manual parser which builds an expression of a function call for eval. It assumes that the function name and arguments were already deserialized.

```julia
function invoke(function_name, args)
    eval(Expr(:call, Symbol(function_name), args...))
end
```

Using a remote interpreter to remotely invoke a plug-in's procedures would not require any alterations in the code when adding new procedures. As long as the module where the procedures are implemented is available on the eval's scope (because there are ways in some programming languages to restrict the scope of available functions and variables for security purposes) the function will always be called.

However, since not every programming language can implement this sort of behaviour we need to find alternatives for other languages, such as, Java, C# and C++. For them, it does not matter if the data that is sent in the message corresponds to their code or not, because, on their own, they are not able to evaluate it. Nonetheless the data is still useful since typically to invoke a function we need an

[2] https://docs.python.org/3/library/functions.html#eval

identifier (usually the name of the function) and the value of its arguments (if necessary). The identifier is essentially metadata to help the language identify the method which needs to be invoked.

Java and C# are able to use this metadata to obtain a reified entity, called a metaobject, from which they invoke the desired function by passing its arguments [8]. Metaclasses and metaobjects are a manifestation of introspection, a reflection activity, that allow the programmer to observe the structure of the entities of the programming language. In the next subsection we will explain how one can achieve DI using introspection to invoke a procedure.

2.3.2 Using metaobjects and wrappers - Introspection

As a metaprogramming strategy, one of the main reasons to use introspection is to write generic programs, i.e., programs that implement a certain behaviour for different types [11]. Reflection is able to do this with metaclasses. Metaclasses abstract the identical operations for similar entities, for example, the invocation of methods through different instances (metaobjects) of the same metaclass.

To achieve DI, we can use the method's metaobjects to invoke the procedure's function, and the deserialization and serialization functions. This process can be described in the following manner (Algorithm 2.1):

1. Firstly, in line 2, a vector is initialized to store the procedure's arguments. In line 3 the socket stream is retrieved to read and write, both the procedure's arguments and return values, respectively. Then, in line 4, a metaobject for the method that implements the procedure is obtained through the method's name;

2. Secondly, using the metaobject of this method, the types of the arguments are retrieved and for each one of them a metaobject that corresponds to a deserialization function (for that type) is obtained (line 6). After that, the value of the argument is read from the stream, by invoking the deserialization function (through its metaobject), and pushed to the vector of arguments (line 7);

3. Thirdly, the metaobject of the procedure's method, obtained in the first step, is invoked with the vector of arguments populated in the previous step, storing the result in a variable (line 8);

4. Lastly, unless the procedure has no return value, the metaboject class of the serialization method is retrieved (lines 9-11), and it is invoked (through its metaobject) with the stream and result of the procedure (line 11).

There are two assumptions made in this solution that should be taken into account:

- Both serialization and deserialization functions need to have names correlated with the types they serialize/deserialize, i.e., one should know the name of serialization and deserialization functions

Algorithm 2.1: DI with Metaclasses Strategy

```
1  begin
2      args ⟵ vectorInit()
3      stream ⟵ getSocketStream()
4      reifiedProcedure ⟵ getMethodMetaclass(procedureName)
5      foreach type ∈ reifiedProcedure.getArgumentTypes() do
6          reifiedRead ⟵ getMethodMetaclass("read" + type.getName())
7          args.push(reifiedRead.invoke(stream))
8      result ⟵ reifiedProcedure.invoke(args)
9      returnType ⟵ reifiedProcedure.getReturnType()
10     if returnType.getName()! = "void" then
11         reifiedWrite ⟵ getMethodMetaclass("write" + returnType.getName())
12         reifiedWrite.invoke(stream, result)
```

based, only, on the type name. Using the previous solution as an example, if one wanted to read a type `Person` a concatenation of the string `"read"` and `"write"`, with `"Person"` would get the name of the deserialization (`readPerson`) and serialization (`writePerson`) functions, respectively;

- The programming languages must be able to use heterogeneous containers for the method invocation (line 8), i.e., there must be a way to invoke the method with a data structure whose objects are of different type. Java[3] and C#[4] solve this problem by declaring an array of `Objects`, a base class of all classes, and having the `invoke` method receiving an array of this type.

Note that in the first assumption the correlation does not need to be necessarily on the name of the functions, it can also be on return types (for deserialization) or arguments (for serialization). For instance, if the programming language allows overloading (i.e., allow different methods to have the same name but different signatures) there might be different `read` functions with different return types. As long as the lookup is done for the read which returns the desired type everything should work fine. Types and method names correspond to metadata so, as long as a language can introspect its metadata anything can be used to establish this relation. Java annotations are another alternative which can be used for this case.

Additionally, regarding the second assumption, there are other programming languages, aside from Java and C#, which are able to invoke a method similarly to the example Algorithm 2.1 (line 8), such as Go[5], see Listing 2.13.

Listing 2.13: Example of how Go invokes methods by name, using reflection.

[3] https://docs.oracle.com/javase/7/docs/api/java/lang/Object.html
[4] https://docs.microsoft.com/en-us/dotnet/api/system.object?view=netframework-4.8
[5] https://golang.org/pkg/reflect/#Value.Call

```
reflect.ValueOf(&result).MethodByName(procedureName).Call(args)
```

However, there are also cases where the programming language either does not have metaclasses or its metaclasses do not support an `invoke` operation. In these cases, instead of using metaobjects one can implement a wrapper that gets a reference to a function, and calls it by forwarding the arguments, see Listing 2.14 for an example in Python. The $*$[6], in line 3, also known as asterisk notation, basically tells that the list `args` will be unpacked into a sequence of items before the function call is done.

Listing 2.14: Example of a Python wrapper that receives a string with the name of a procedure and a list with its arguments and returns the value of the invocation of that procedure.

```
1 def wrapper(procedureName, args):
2     func = globals().get(procedureName)
3     return func(*args)
```

In the end, just like what happened with eval functions these approaches do not require any form of tuning when a new procedure is added to their scope. This is because the programming languages which use these approaches already have access to enough built-in metadata, in the form of metaclasses or mappings, avoiding additional specifications in the user's code.

In the next subsection we will see a few features of C++ templates that help the implementation of a DI mechanism.

2.3.3 Using C++ templates

Although C++ does not possess sufficient capabilities to implement the previous approaches, it still has some workarounds. For instance, even though C++ cannot automatically retrieve a function reference as shown in Listing 2.14 (line 2), the programmer can implement a mapping which maps the function name to the corresponding function pointer. This would be enough to solve the problem for C++, however, as seen in Algorithm 2.1 the procedure's function is not the only function invoked by name. Serialization and deserialization functions would also have to be invoked, thus, they would require a mapping too. C++ templates provide mechanisms to call serialization and deserialization functions based on the type we want to serialize or deserialize, without any mappings.

Templates allow the creation of generic programs through generic types, as exemplified in Listing 2.15. A generic type is an abstract type which can take the form of any type at compile time. A generic type only exists before compile time. After compilation, the compiler generates the code for a specific type (substituting the generic type), through a process called instantiation [10]. Instantiation is only triggered when explicitly using a function or class, as visible in Listing 2.15 (line 8). Note that

[6] https://docs.python.org/3/reference/expressions.html#expression-lists

generic class templates and function templates are not classes or functions, they are only patterns used by the compiler for instantiation.

Listing 2.15: Definition of generic class template `Foo` with generic type `T` with example of the code needed to trigger the instantiation of `Foo<int>` class.

```
template<typename T>
class Foo {
public:
  T bar;
};

int main(){
  Foo<int> f;
  f.bar = 500;
}
```

Generic types allow the programmer to define patterns and similar behaviours regardless of typing. There are a lot of features of C++ templates that benefit from generic types. We will now introduce a few of them and later in Chapter 3 we will show how we used them to implement DI.

Function Template Argument Deduction is a property possessed by function templates which allow the programmer to omit template arguments when calling a template function (see Listing 2.16, line 2). Before C++17, classes could not infer the types of their templates through the invocation of their constructors (line 4), therefore, the alternative was to use functions to construct a templated object (line 2). However, since C++17 class template argument deduction is possible through the constructors of a class removing the need to implement auxiliary construct functions (line 6)[7]. This feature is particularly useful when passing function pointer types as arguments. Especially for functions which have a lot of parameters, as there is no need to specify them.

Listing 2.16: Example of a function template argument deduction and class template argument deduction before and after C++17.

```
//Function template argument deduction
auto p1 = std::make_pair(85, 'U');
//No class template argument deduction (Before C++17)
auto p2 = std::pair<int, char>(80, 'P');
//Class template argument deduction (After C++17)
auto p3 = std::pair(83, 'S');
```

[7] https://en.cppreference.com/w/cpp/language/class_template_argument_deduction

Partial/Explicit Template Specializations allow the specification for a family of types (partial) or for a specific type (explicit) [10]. Partial specializations specialize their template parameters. In Listing 2.17 (line 5-6), type parameter `T` was specialized to `T*` for the class definition. In this case, this means that if a pointer type is passed as template argument (line 13), the implementation of `Foo<T*>` will be used. For explicit specializations the template parameters needs to be empty (line 8) and the specific type must be on the parameter list when defining the class (line 9). Compilers prioritize explicit specializations over primary class templates or partial specializations (line 14).

Listing 2.17: Example of partial and explicit template specialization.

```
1  //Primary class template
2  template<typename T>
3  class Foo { ... };
4  //Partial specialization for pointers
5  template<typename T>
6  class Foo<T*> { ... };
7  //Explicit specialization for int
8  template<>
9  class Foo<int> { ... };
10
11 int main(){
12     Foo<float> f1;        //uses Foo<T>
13     Foo<int*> f2;         //uses Foo<T*>
14     Foo<int> f3;          //uses Foo<int>
15 ...
16 }
```

As showcased, partial specializations are useful to implement different behaviour for a specific family of types such as pointers or function pointers. And explicit specializations are useful to define base cases when implementing recursion for template metafunctions.

Variadic Templates have been in C++ since C++11. They can accept a variable number of arguments of any type, allowing the use of templates in places where an arbitrary number of arguments needs to be passed [10]. They are useful for the implementation of wrapper functions, as illustrated in Listing 2.18. Note that the use of `"..."` (line 2-3) tells the compiler that `list` (parameter pack) needs to be expanded. Variadic templates are also important to define template classes for heterogeneous containers, such as, linked typelists and tuples.

Listing 2.18: Example of a wrapper function in C++ that takes a function pointer and a list of arguments through a variadic template and invokes the function with the arguments.

```
1 template <class F, class... Args>
2 auto wrapper(F f, Args... list) {
3   return f(list...);
4 }
```

Linked typelists and tuples are used to store different types of data. However the type of the elements of the list must be specified during a declaration because the compiler has to know which typelist should be instantiated (see Listing 2.19, line 1). Also, the indexing of this lists cannot be done at runtime, otherwise the compiler would not know the type of the indexed element at compile time, resulting in a compile time error. This can be solved with the capabilities of C++ templates. C++ allows the indexing of tuples through the use of a `get` template function. This function uses non-type template parameters (unsigned integer parameters) to index a tuple (lines 3-4). This is possible because the compiler is able to expand code which corresponds to the indexing of the tuple from the non-type template parameter.

Listing 2.19: Example of a declaration of a typelist and use of `get` to index a tuple.

```
1 Typelist<int,float,int> list;
2
3 auto t = std::make_tuple(1, "Boop", 3.14);
4 std::cout << "(" << std::get<0>(t) << ", " << std::get<1>(t)
5               << ", " << std::get<2>(t) << ")\n";
6 //Prints (1, Boop, 3.14)
```

2.4 Summary

In this chapter, we studied how two different approaches based on a RPI framework (Apache Thrift) and a serialization method (Protocol Buffers) influence the addition of new procedures. We saw that both of these approaches require the writing of a procedure's specification in an IDL which is a lengthy process as it cannot be automated and requires the programmer to manually specify the procedure's information. Additionally, the approach which uses the Apache Thrift RPI framework requires the programmer to also write additional code in an handler which is generated through the compilation of the IDL. In order to circumvent the issues of these IDL-based approaches we presented DI. In Table 2.1 a table with the tasks required by each mentioned approach to enable RPI is presented.

We saw several methodologies which implement DI. We studied approaches which used reflection, using intercession, through eval functions, and introspection, through metaclasses and mappings, to

enable DI. Also, a few features of C++ templates were explained as they will be used in the next chapter to implement an approach with C++ templates since this language does not possess built-in means for reflection. Several examples in different programming languages were given for each approach. In Table 2.2 we present a summary of which languages can use a specific mechanism.

Table 2.1: Table that shows what each studied approach should do after implementing a new procedure to enable RPI of that procedure.

	Write Handler	Write IDL	Add Mapping of New Procedure
Apache Thrift	✓	✓	-
Protocol Buffers	-	✓	✓
Templates	-	-	✓
Remote Interpreter	-	-	-
Metaclasses	-	-	-
Wrappers	-	-	-

Table 2.2: Table which summarizes all the possible DI approaches for each language mentioned in this chapter.

	Julia	Python	Ruby	Go	Java	C#	C++
Remote Interpreter w/ eval (Intercession)	✓	✓	✓	-	-	-	-
Reflection w/ Metaclasses (Introspection)	-	-	-	✓	✓	✓	-
Reflection w/ Wrappers (Introspection)	✓	✓	✓	-	-	-	-
Templates	-	-	-	-	-	-	✓

3

Dynamic Invocation in C++

Contents

In the previous chapter we saw different alternatives to achieve DI. We have also shown that C++, although not possessing reflecting capabilities, has a few features that provide acceptable workarounds for this problem. In this chapter we will demonstrate how we used these features to implement DI in C++. We will divide the explanation of the implementation of our solution in three sections:

- Firstly, we will see how we managed to represent different procedures with different number of arguments and return types in our code;
- Secondly, we will explain how the different procedure's arguments, passed in a stream, are read and stored for an invocation;
- Thirdly, we will show in what way the arguments are passed to the procedure for its invocation and by what method the result is written back in the stream.

Then, in another section we will explain how the addition of new procedures is done, and finally, in the last section, we will showcase an example of how the different entities of our solution interact in the procedure's invocation process.

3.1 Representing different procedures

Primarily, to enable RPI, it is necessary to establish and implement a mechanism of communication with client applications through a communication channel. As a consequence of that, we defined a class, `RpiService`, which implements the communication details (e.g., setting up sockets and data streams), and server configurations (e.g., blocking and single-threaded) of our solution. To start this service the method `Run` of the class should be called with the corresponding port as an argument, as illustrated in Figure 3.1.

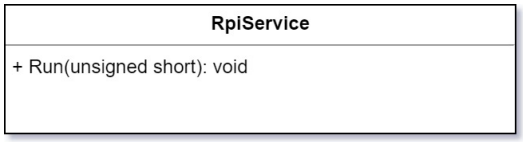

Figure 3.1: `RpiService` class with the `Run` method in a class diagram.

Additionally, the `RpiService` class also stores the handlers which will be used to invoke different procedures. In our implementation each procedure is implemented as a C++ non-member function (functions which do not belong to a specific object). However, storing different types of function pointers in the same container is not a feasible solution. Remember how in [2] all procedures were implemented as functions of the same type (`void(*)(void)`) to avoid this problem. However, since we do not want to impose such restrictions, we encapsulate function pointers in a C++ generic class template (`Handler`),

which we will call handlers, and have these handlers implement a common interface, as illustrated in Figure 3.2. This common interface is composed by two different methods which divide 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;

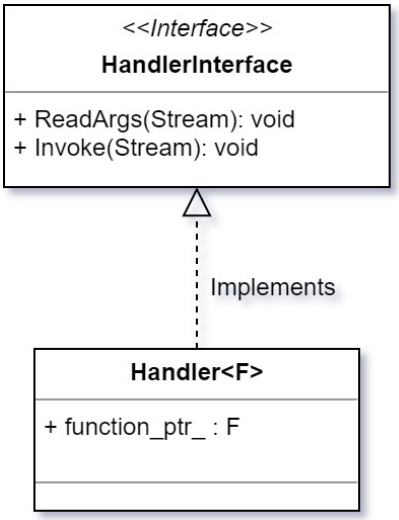

Figure 3.2: `Handler` generic class templates implement `HandlerInterface` and encapsulate a procedure function pointer (`function_ptr_`).

Even though the classes instantiated from the generic class template, `Handler`, may seem similar (only differ on type of function pointer, `F`), they have no relation whatsoever. The compiler uses name mangling techniques during instantiation, and ensures there is no relation between different handlers. For instance, an instance of `Handler<int(*)(int)>` is different from an instance of `Handler<float(*)(float)>`. This is where the presence of a common interface helps. By having the handlers implement a common interface, we have a type safe method to:

- Store different handlers in the `RpiService` containers, `handlers_map_` (a map of procedures names to `Handler` instances) and `available_handlers_` (a dynamic vector with the handlers of procedures which can be invoked), as is illustrated in Figure 3.3. The idea is to have the client request for a procedure in the first RPI, and push an handler from the map to the vector, returning the index of the handler in the vector to the client. Then, to invoke the procedure the client is only required to send the index with the procedure arguments, transferring less data in the channel. Also, the invocation is much faster because the server only has to index a vector instead of searching for an handler in a map. This process will be explained in more detail in Section 3.5;

- Call the handlers methods through their common interface in a container, as exemplified in Listing 3.1 without having to specify any information about the type of the handler.

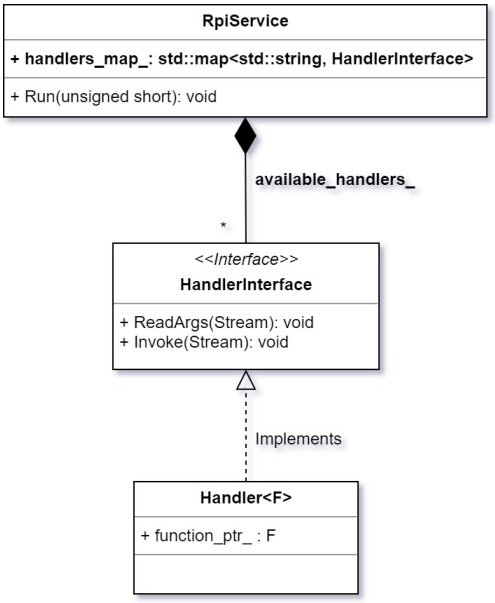

Figure 3.3: `RpiService` stores different instances of handlers (`Handler`) as interfaces (`HandlerInterface`) in a map and vector, `handlers_map_` and `available_handlers_`, respectively.

Listing 3.1: Using an instance of `Handler` as a `HandlerInterface` in a container to make a RPI.

```
available_handlers_[i]->ReadArgs(socket_stream);
available_handlers_[i]->Invoke(socket_stream);
```

This technique, which allows the abstraction of a specific type through its interface, is called type erasure and relies on dynamic polymorphism properties of the C++ programming language to perform single dispatch, i.e., resolve which implementation of a method should be called according to the type of the instance calling it. Note also that, in C++, it is required to declare the methods of the interface as `virtual` to enable single dispatch, as presented in Listing 3.2.

Listing 3.2: `HandlerInterface` methods declared as virtual.

```
class HandlerInterface {
public:
  virtual void ReadArgs(Stream &stream) = 0;
  virtual void Invoke(Stream &stream) = 0;
};
```

Single dispatch ensures that the correct method of an `Handler` instance is called, and since each instance encapsulates a specific function pointer, its invocation can be easily achieved. This encapsulation is achieved by using C++ templates partial specialization techniques to implement a generic behaviour for all types of functions as visible in Listing 3.3:

- First, in lines 1-2, we define the primary class template with just one template parameter which corresponds to type of the function pointer being encapsulated;
- However, having the type of the function pointer alone is not enough because we cannot extract the return type and arguments types from it. Therefore, in lines 4-5 we do a partial specialization of the `Handler` generic class template for function pointer types, separating the return type, `R`, from the arguments types, `Args`. These types can be automatically inferred, if a C++17 version is used, when a `Handler` class is instantiated by calling the constructor in line 8 with a function pointer as argument. For C++14 and C++11, we need to implement a handler factory function to automatically infer these types. We will see an example of such functions later in this chapter;
- The parameter pack `Args` with the types of the function pointer arguments, was also used to declare an attribute (line 13). This attribute corresponds to a heterogeneous linked typelist which will be used to read and store different arguments from a data stream.

Listing 3.3: Partial Specialization of `Handler` to pointer types.

```
1  template<typename F>
2  class Handler : public HandlerInterface {};
3
4  template<typename R, typename... Args>
5  class Handler<R(*)(Args...)> : public HandlerInterface {
6  public:
7    using F = R(*)(Args...);
8    Handler(F f) : function_ptr_(f) {}
9    void ReadArgs(Stream &stream) { ... }
10   void Invoke(Stream &stream) { ... }
11 private:
12   F function_ptr_;
13   ArgumentList<Args...> function_arguments_;
14 };
```

When the function has no return value, i.e., `R` is `void` the `Invoke` method dos not need to write a value to a stream, thus, its implementation should be different. For this specific case we used another

partial specialization as seen in Listing 3.4. Later in this chapter we will see the difference between each implementation of the `Invoke` method.

Listing 3.4: Partial Specialization of `Handler` for functions with no return value.

```
1  template<typename... Args>
2  class Handler<void(*)(Args...)> : public HandlerInterface {
3  public:
4    using F = void(*)(Args...);
5    Handler(F f) : function_ptr_(f) {}
6    void ReadArgs(Stream &stream) { ... }
7    void Invoke(Stream &stream) { ... }
8  private:
9    F function_ptr_;
10   ArgumentList<Args...> function_arguments_;
11 };
```

In the next section we will explain how our solution uses the linked typelist in the `ReadArgs` method to read and store arguments from a data stream. Then, in another section, we will show how the `Invoke` method invokes the procedure by forwarding these arguments to the function pointer, writing its return back in the data stream.

3.2 Reading and storing arguments from a stream

As we have seen back in Chapter 2, establishing a generic behaviour to read/deserialize arguments' values, while storing them in a container can be a difficult process in some programming languages, especially when there is not enough built-in metadata available and no heterogeneous containers. As C++ does not possess any form of metadata to retrieve the types of a function arguments, in this solution, we solved this issues by implementing a linked typelist. By iterating the list we continuously read and store the values of the different arguments. The types used to define a list are used to invoke the correct deserialization function. Thus, because the elements of the list will have different types we had to define it with the help of C++ templates and implemented a generic class template, as visible in Listing 3.5:

- The primary class template (lines 1-2) is defined using a variadic template, `Types`, to allow different type combinations, such as, `ArgumentList<int>`, `ArgumentList<float, int, double>`, and even `ArgumentList<>`;
- Then, in lines 4-5, we do a partial specialization separating the type of the fist element of the list, `Head`, from the types of the rest of the elements of the list, `Tail`, allowing us to implement the

standard behaviour of a linked list (lines 6-17). Notice that we implemented an additional `Read` method (lines 8-11) which stores a value read from the stream in the list's head (line 9), and recursively calls the `Read` method on the list's tail (line 10). It is by calling this method in the first argument of the list that the values are recursively read into the list from the data stream;

- Lastly, we had to make an explicit class specialization for an empty list (lines 19-22) to define a stop case for the `Read` recursion, in these case the list has neither head or tail and its `Read` method does nothing.

Listing 3.5: Implementation of a linked typelist as a `ArgumentList` generic class template.

```
1  template<typename... Types>
2  class ArgumentList {};
3
4  template<typename Head, typename... Tail>
5  class ArgumentList<Head, Tail...> {
6  public:
7    ArgumentList() {}
8    void Read(Stream& stream) {
9      head_ = stream.Read<Head>();
10     tail_.Read(stream);
11   }
12   Head GetHead() { return head_; }
13   ArgumentList<Tail...> GetTail() { return tail_; }
14 private:
15   Head head_;
16   ArgumentList<Tail...> tail_;
17 };
18
19 template<>
20 class ArgumentList<> {
21 public:
22   void Read(Stream&) {}
23 };
```

Using this implementation, we can store in each handler of our solution a typelist, `function_argumen-ts_` (see Figure 3.4), and call its `Read` method in the `ReadArgs` method to read and populate the elements of the list, as visible in Listing 3.6, which will be forwarded for the invocation of the procedure in `Invoke`.

Listing 3.6: Implementation of the `ReadArgs` method of an `Handler` class.

```
1 void ReadArgs(Stream &stream) {
2     function_arguments_.Read(stream);
3 }
```

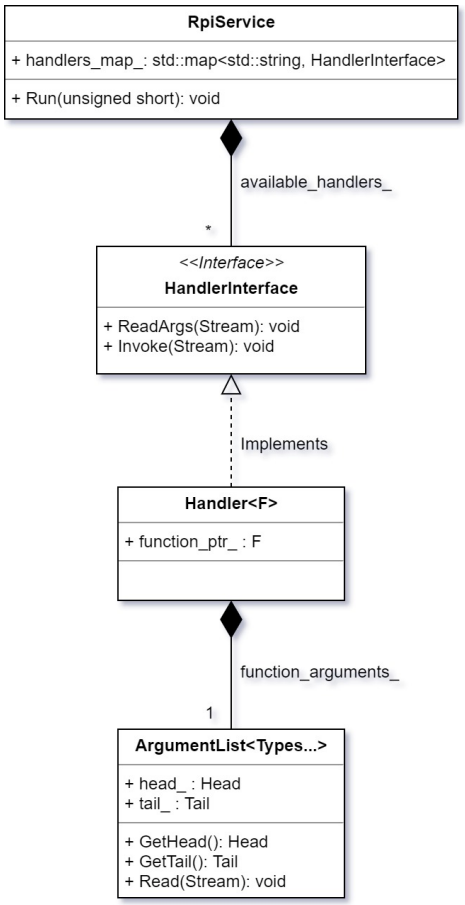

Figure 3.4: Handlers store a linked typelist (`ArgumentList`) as an attribute.

3.3 Invoking the procedure

After the values of the arguments are read, by calling `ReadArgs`, it is necessary to forward them to a function pointer, and write the returned value in a data stream. This is done in the `Invoke` method of an `Handler` class, as seen in Listing 3.7. The arguments of a list are forwarded to a function pointer by calling a wrapper function, and the result of the invocation is stored in a `ret` variable (line 2). Then, the result previously obtained is written in the stream (line 3). Note that for the partial specialization of handlers with functions that have no return, the `Invoke` method does not require the return to be stored and written as shown in Listing 3.8.

Listing 3.7: Implementation of the `Invoke` method of an `Handler` class.

```
void Invoke(Stream &stream) {
    auto ret = WrapperInvoke(function_ptr_, function_arguments_);
    stream.Write(return_);
}
```

Listing 3.8: Implementation of the `Invoke` method of an `Handler` class with no return (see Listing 3.4).

```
void Invoke(Stream &stream) {
    WrapperInvoke(function_ptr_, function_arguments_);
}
```

Since the arguments are stored in a typelist (`ArgumentList`) and not in a template parameter pack the implementation of the function `WrapperInvoke` is a bit more complex than the one we have shown in Listing 2.18. In the specific case where the list is empty the problem is simply solved with overloading, as shown in Listing 3.9 (line 6-9). However, when arguments need to be forwarded, the complexity increases (lines 1-4). In this instance, we use another wrapper, `WrapperDetail`, and not only forward the function pointer and the list, but also a sequence of integers with numbers to index the list. For example, if the list has `N` arguments an `std::index_sequence<0,1,...,N-1>` is generated (line 3).

Listing 3.9: Implementation of our wrapper to forward a list with arguments and an empty list.

```
1 template <typename F, typename... Args>
2 auto WrapperInvoke(F&& f, ArgumentList<Args...> list) {
3   return WrapperDetail(f, list, std::index_sequence_for<Args...>());
4 }
5
6 template <typename F>
7 auto WrapperInvoke(F&& f, ArgumentList<>) {
8   return f();
9 }
```

The reason behind using this sequence is tied with the fact that the compiler can automatically expand this parameter packs just as we have seen in Listing 2.18 (line 3) to index the list. However, for this to happen we have to resort to template metaprogramming, and implement a metafunction which indexes the `ArgumentList` from a template parameter. The idea is to have the compiler expand successive metafunction calls with the index sequence as template arguments.

As a consequence of that, we implemented a small metafunction in a generic class template, `Argume-`

ntGet, which uses recursion to retrieve a value of an element of that list, as visible in Listing 3.10. The ArgumentGet class template has a non-type template parameter N (line 1), which is where indexes will be passed to retrieve an element of the list. To retrieve an element, while the non-type template parameter is greater than 0 (N>0) the static member function Apply is called extracting the tail of the list and decrementing the value of N (line 5). When N=0, the head of the list is the indexed element, and the explicit template specialization of ArgumentGet, lines 9-10, is used to retrieve the head of the list (line 13).

Listing 3.10: Metaprogram ArgumentGet to index a list using a non type template parameter.

```
1  template<std::size_t N>
2  struct ArgumentGet {
3      template<typename Head, typename... Tail>
4      static auto Apply(ArgumentList<Head, Tail...> t) {
5              return ArgumentGet<N - 1>::Apply(t.GetTail());
6      }
7  };
8
9  template<>
10 struct ArgumentGet<0> {
11     template<typename Head, typename... Tail>
12      static auto Apply(ArgumentList<Head, Tail...> t) {
13          return t.GetHead();
14     }
15 };
```

With this, it is possible to index an ArgumentList and expand successive ArgumentGet calls, as shown in Listing 3.11 (line 3), successfully invoking the function pointer of a procedure. A result of the expansion done by the compiler for a function that has N number of arguments can be seen in Listing 3.12.

As previously shown, in Listing 3.7 the result of the invocation is written in a data stream. Or, if the function has no return, the Invoke method ends. In the next section we will see how these mechanisms facilitate the addition of a new procedure to our solution and how all these templates are instantiated when the new procedure is added.

Listing 3.11: Using the ArgumentGet metafunction and index sequence to expand all elements' values of a list.

```
1  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)...);
}
```

Listing 3.12: Result of the successive expansion of `ArgumentGet` calls, done by the compiler, for a function with `N` number of arguments.

```
return f(ArgumentGet<0>::Apply(list),
            ArgumentGet<1>::Apply(list),
                ArgumentGet<2>::Apply(list),
                    ...
                        ArgumentGet<N - 1>::Apply(list));
```

3.4 Addition of new procedures

We now exemplify the steps needed to add new procedures to our solution with the intent to make them remotely available. Consider, for example, that the programmer decided to add a new procedure `add` which returned the sum of an integer and a double as a double value the following steps should be taken:

1. Implement the `add` procedure in the plug-in, similar to the one in Listing 2.10. The implementation of the procedure should not have any dependencies to our solution. There is no need to specify details about data streams or sockets in this implementation;

2. Add a mapping of the function that implements the procedure, as shown in Listing 3.13. The addition of the new procedure to the map is simple, requiring only 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;

3. Compile and execute the code.

Listing 3.13: Example of how to add a new procedure to our solution.

```
class MyService : public RpiService{
  MyService::MyService() {
    ADD_FUNCTION(foo);
```

```
4       ADD_FUNCTION(bar);
5       ...
6       ADD_FUNCTION(add);
7     }
8   };
```

This macro basically consists of calling a method of the `RpiService` class, `InsertMap`, whitch takes a `std::string` and pointer to a `Handler` as arguments and insert this pair in the map, `remote_functions_map_`, (see macro definition and expansion in Listing 3.14). This `Handler` is pushed to a vector if an invocation request is received, `available_handlers_`, and invoked as shown in Listing 3.1.

Listing 3.14: Definition of the macro that adds a pair of `std::string` and `Handler` to the map, and the result of its expansion for `ADD_FUNCTION(add)`.

```
1 #define ADD_FUNCTION(function) \
2   InsertMap(#function, MakeHandler(&##function##));\
3 //ADD_FUNCTION(add) expands to: InsertMap("add", MakeHandler(&add));
```

Notice that even in the macro expansion it is not necessary to specify the template argument (function pointer type) of the `Handler` being created. As it was said in Section 3.1, this necessity can be avoided by using a factory function for the handlers' class and have the types being deduced with function template argument deduction. Our factory function is the function `MakeHandler`, which is used in our macro, as see in Listing 3.14. Again, for C++17, these types of functions are not necessary because the template arguments can be deduced by calling the constructor of a generic template class. Although template argument deduction can help us eliminate the need to specify argument types, the compiler cannot use it to deduce overloaded functions. In this case, the functions have the same name, which means that the compiler has no means to know which function is being passed just from their name.

Listing 3.15: Function that creates an `Handler` from a specific function pointer

```
1 template<typename F>
2 std::shared_ptr<Handler<F>> MakeHandler(F&& f) {
3   return std::make_shared<Handler<F>>(f);
4 }
```

3.4.1 Template Instantiations

It is the call of the `Handler` constructor in `MakeHandler` (line 3) that triggers the instantiation of this class by the compiler, at compile time. The compiler uses the partial specialization presented in Listing 3.3 to

instantiate the handler for the `add` procedure, visible in Listing 3.16.

Listing 3.16: Instantiation of a `Handler` for a `double(*)(int, double)` function.

```
class Handler<double(*)(int, double)> : public HandlerInterface {
public:
  using F = double(*)(int, double);
  Handler(F f) : function_ptr_(f) {}
  void ReadArgs(Stream &stream) {
    function_arguments_.Read(stream);
  }
  void Invoke(Stream &stream) {
    auto return_ = WrapperInvoke(function_ptr_, function_arguments_);
    stream.Write<double>(return_);
  }
private:
  F function_ptr_;
  ArgumentList<int, double> function_arguments_;
};
```

The instantiation of the `Handler<double(*)(int, double)>` class causes an instantiation of `ArgumentList` class template and `WrapperInvoke` function template, because virtual member functions (`ReadArgs` and `Invoke`) are used by the service to perform RPI (see Listing 3.1). Since the list is used in both member functions, and the wrapper is used in `Invoke`, both templates also need to be instantiated.

By instantiating the `ArgumentList<int, double>` class (see Listing 3.17, line 1-13), it is also necessary to instantiate a class for the second element of the list (lines 15-27), and for the empty list (lines 29-32).

Listing 3.17: Instantiations triggered by using an `ArgumentList<int, double>`.

```
1 class ArgumentList<int, double> {
2 public:
3   ArgumentList() {}
4   void Read(Strean& stream) {
5     head_ = stream.Read<int>();
6     tail_.Read(stream);
7   }
8   int GetHead() { return head_; }
9   ArgumentList<double> GetTail() { return tail_; }
```

```
10 private:
11   int head_;
12   ArgumentList<double> tail_;
13 };
14
15 class ArgumentList<double> {
16 public:
17   ArgumentList() {}
18   void Read(Strean& stream) {
19     head_ = stream.Read<double>();
20     tail_.Read(stream);
21   }
22   double GetHead() { return head_; }
23   ArgumentList<> GetTail() { return tail_; }
24 private:
25   double head_;
26   ArgumentList<> tail_;
27 };
28
29 class ArgumentList<> {
30 public:
31   void Read(Stream&) {}
32 };
```

Because the `add` procedure has two arguments, a non-empty `ArgumentList` is passed to the `WrapperInvoke` template function, therefore only the template function that receives non-empty lists is instantiated (see Listing 3.9, lines 1-4). The resulting instantiation, seen in Listing 3.18, calls another template function, `WrapperDetail` (line 2), meaning that this template function also needs to be instantiated (lines 5-8).

Listing 3.18: Instantiation of both `WrapperInvoke` and `WrapperDetail` template functions.

```
1 double WrapperInvoke(double(*)(int, double) f, ArgumentList<int, double> list) {
2   return WrapperDetail(f, list, std::index_sequence_for<int, double>());
3 }
4
5 double WrapperDetail(double(*)(int, double) f, ArgumentList<int, double> list,
6                        std::index_sequence<0, 1>){
7   return f(ArgumentGet<0>::Apply(list), ArgumentGet<1>::Apply(list));
```

```
8 }
```

Finally, to index the list, the `WrapperDetail` template function uses the static method `Apply` of the generic class template `ArgumentGet`. Therefore, the methods and the class have to be instantiated, as demonstrated in Listing 3.19. Notice how the methods instantiated from `ArgumentGet<1>::Apply(list)` return `double` which is the type of the second element of the list and how the method instantiated from `ArgumentGet<0>::Apply(list)` return an `int`.

As the `ArgumentList<int, double>` and `ArgumentList<double>` (lines 3 and 10, and 14), are already instantiated the addition of the `add` procedure does not require the compiler to do any more instantiations. Remember that all these instantiations were done automatically just by adding a macro call in Listing 3.13. The only thing which was specified was the function name, neither the return or arguments' types had to be specified for this.

Listing 3.19: Metaprogram `ArgumentGet` to index a list using a non type template parameter.

```
1  struct ArgumentGet<1> {
2    //Instantiated from ArgumentGet<1>::Apply(list)
3    static double Apply(ArgumentList<int, double> t) {
4      return ArgumentGet<0>::Apply(t.GetTail());
5    }
6  };
7
8  struct ArgumentGet<0> {
9    //Instantiated from ArgumentGet<0>::Apply(list)
10   static int Apply(ArgumentList<int, double> t) {
11     return t.GetHead();
12   }
13   //Instantiated from ArgumentGet<1>::Apply(list)
14   static double Apply(ArgumentList<double> t) {
15     return t.GetHead();
16   }
17 };
```

3.5 Interaction

In this section we explain an example of an interaction of our entities through a sequence diagram. In Figure 3.5, we use as example the invocation of function `add(int, double)` which returns the sum of

an integer and a double as a double. The process can be described in the following steps:

1. When the service starts, the function `add` is added to the map of names to handlers, in `handlers_map_`;

2. Starting the communication, the client sends a message to the service with an id of 0 and string "add". This means that the client wants the server to make the procedure named "add" available for future invocations. The server pushes the corresponding `Handler` on the map to a dynamic vector, `available_handlers_`, and returns an id of the `add` procedure back to the client;

3. Knowing the id of the operation the client sends it on a message along with the arguments to pass to the function;

4. The service reads the id of the message, and indexes the vector getting the correct instance of `Handler`. The `ReadArgs` method is called with the data stream containing the arguments, and then, the `Handler` calls the `Read` method of its `ArgumentList`, reading and storing the arguments on the list;

5. Finally, the `Invoke` method is called, forwarding the arguments in the `ArgumentList` to the function pointer, `function_ptr_`, of `add`, storing and writing the result, 4, in a data stream.

The use of a dynamic vector is not entirely necessary. We could simply use a map for the process of DI. In this case, the step 2 would not be necessary and instead of sending an id, in 3, the client could send the key of the handler in the map, which is "add" for this example. However, we opted for this approach since, aside from the first call where the handler has to be pushed from the map to the vector, there are no additional overheads to find the handler in the map (as the handler is already on the vector and the id of the operation corresponds to its index). Note that, in this solution, an `Handler` is generated at compile time through instantiation, which means that there are no overheads installing an handler. However, in some cases (code generation, for example) where handlers may be installed at runtime, this approach enhances performance as it only installs an handler one time (first invocation). Also, as we will see further in this thesis, other client applications used this type of protocol so we made our implementation compatible with these applications.

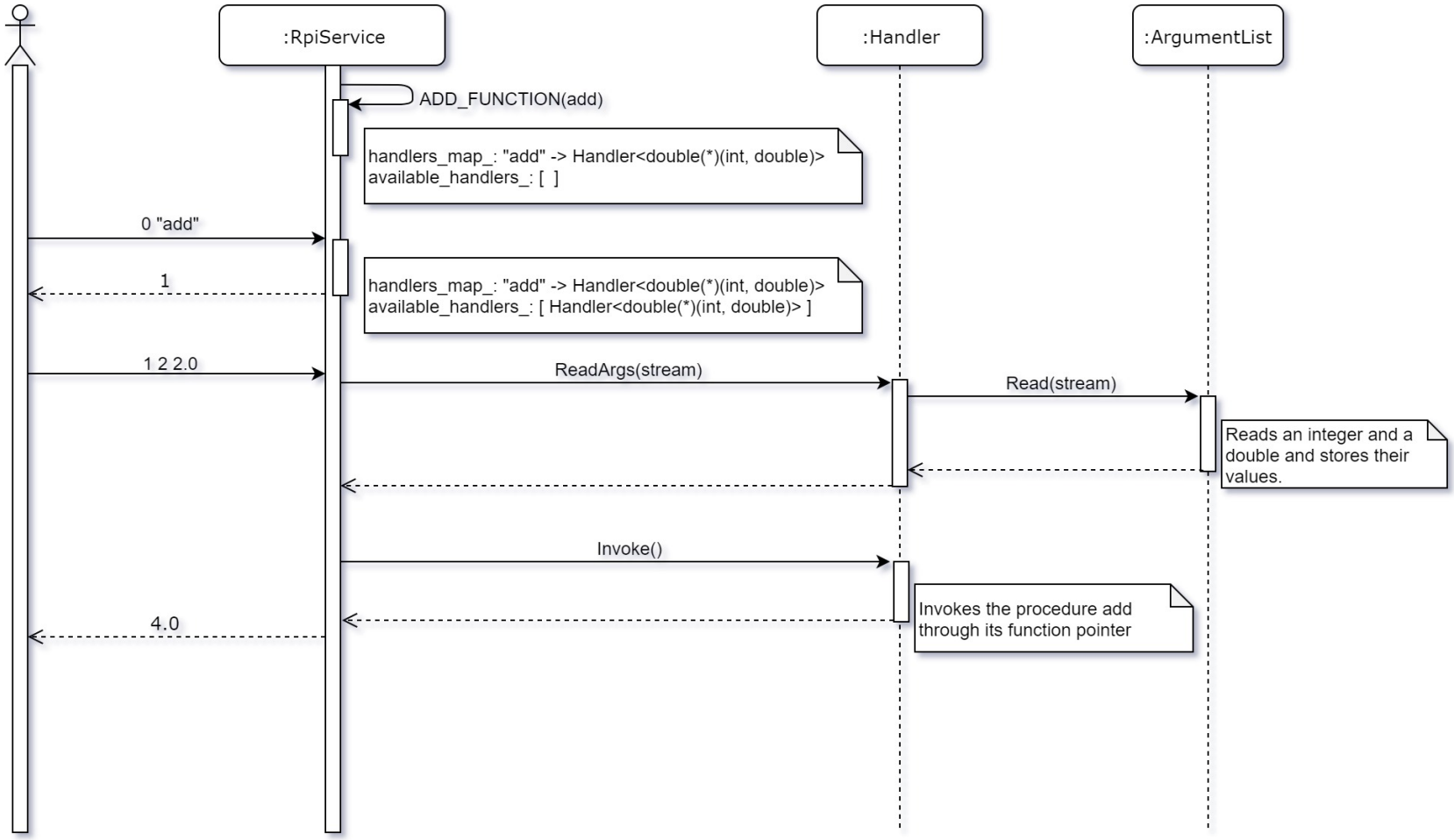

Figure 3.5: A sequence diagram that shows how to invoke an `add` function.

3.6 Summary

In this chapter we explained the implementation of our solution and how it achieves DI. We went over the details on how we managed to represent different procedures in handlers using a generic class template, `Handler`. As the procedures are implemented in C++ non-member functions each handler encapsulated a function pointer for that function.

However, it is not easy to store different handlers instantiated from different class templates since the classes that result from the instantiation are not related, despite their name being, apparently, the same. Therefore, we used type erasure techniques and had each handler implement a common interface `HandlerInterface` which implemented two methods: `ReadArgs`, a method which is called to read arguments from a data stream, and `Invoke`, a method which forwards these arguments to the encapsulated function pointer and writes the result of the invocation in a data stream. Through this common interface we were able to store and use the handlers by calling its interface methods, performing a RPI, in a `RpiService` class.

To perform an invocation, the service class invokes the `ReadArgs` method on an handler. The handler uses a heterogeneous linked typelist (a list with elements of different types) and performs consecutive reads on each element of the list. With this, the values of the arguments are retrieved and stored in the list.

Then, the service invokes the `Invoke` method of the handler which uses wrappers to forward the arguments in the list to the function pointer with the intent of invoking the function. We implemented a small metafunction with C++ template metaprogramming capabilities to index the arguments of the function stored in the list, and took advantage of the expansion done by compilers on variadic templates to perform the indexing in all elements of the list. This allowed the entire list to be indexed and forward all of its elements to the function pointer. The result of the invocation is stored and written in a data stream ending the RPI process.

After that, we showed how to add a new procedure to our solution by adding a macro call in a service class constructor. This macro only takes the name of the function as argument, creates an handler and stores the handler in a map. The creation of the handler triggers the instantiation of several class and function templates to achieve the DI of the procedure.

Finally, we demonstrated how the entities of our solution interacted to perform a remote invocation. In the next section we will evaluate our solution and compare it against other alternatives which we have already studied in Chapter 2.

4

Evaluation

Contents

In this chapter we evaluate our solution in terms of: productivity, as the time spent adding new procedures to a software plug-in which needs to be remotely invoked; runtime performance, analysis on the total time spent by our solution to do a remote invocation (Round Trip Time), and the actual time spent on the execution of the procedure in the plug-in.

We will compare the productivity of our solution to possible alternatives which use Apache Thrift, an RPI framework, and a similar solution to the one presented in Section 2.2 with Protocol Buffers.

The chapter begins by explaining the environment in which these solutions were evaluated. After that, we will show the results of the evaluation and explain the conclusions drawn from the experiments. Finally, we will show an example of an use case where our solution can be used to perform GD.

4.1 Evaluation environment

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 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. More on this in later in this chapter.

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 [2] 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 in ArchiCAD, only the main thread can call functions of the API, which means that having the RPI mechanism running in another thread would not solve this issue.

4.2 Productivity when adding new procedures

In this section we will evaluate and compare the three approaches. For that reason, we counted the time needed for each approach to add a new procedure. Every approach requires different tasks, and we only counted the time of tasks which could not be automated. For example, when compiling an IDL file, additional code is generated. Although this code has to be included in the project, this task can be automated either by specifying the source folder during compilation, or by having a script moving the files to their correct destination. Therefore, we did not consider this task when measuring the time spent on adding a new procedure.

However there are a few important tasks, seen in Table 2.1, which cannot be automated and, consequently, we wanted to measure their time for our evaluation. They are:

- Addition of a mapping to identify a procedure. As C++ does not possess means of reflection for introspection, some approaches, such as Protocol Buffers and DI, require the programmer to manually add a pair that maps an identifier to a procedure. On the other hand, Thrift does this job automatically, since the stubs generated know how to dispatch the client's requests to the correct procedure handlers;

- Writing of the procedure's interface in an IDL. Thrift and Protocol Buffers require the specification of service's interface or message's types through an IDL. While Thrift generates code for stubs, handlers and additional data types, Protocol Buffers only generate serialization methods for each message type. The dispatching of these messages is solved with the map of procedures;

- Implementation of handlers. Thrift generates handler classes with a method where the procedure functions should be called. After the compilation of the IDL file, the method's implementation is empty, meaning that the programmer has to write the desired functionality in them. Additionally, the programmer also has to copy this class into his code.

It is important to note that these experiments were done 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 priorly 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 approach to add a new procedure `createCircle`. This operation draws a simple circle of radius, `r`, in a point, `(x,y)`, and returns the identifier of the circle. The result of the first measurements can be seen in Figure 4.1.

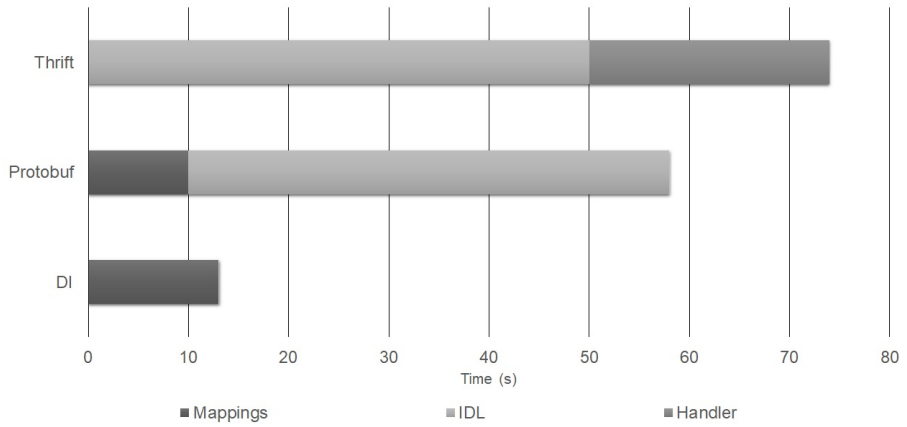

Figure 4.1: The time needed to make an `int createCircle(double x, double y, double r)` procedure remotely available by the three alternatives which used Thrift, Protocol Buffers, and DI, for each task.

By looking at the previous chart it is already evident that our solution takes significantly less time than the other approaches. The writing of the procedure's data in an IDL is, by far, the task which takes the most time. However, as these measurements depend on a lot of variables the results might differ from experiment to experiment. This means that the results obtained in Figure 4.1 might not accurately demonstrate small differences in each RPI approach. Therefore, looking for more accurate results, we decided to do more experiments.

A total of thirty experiments, ten for each approach, were realized. The average times for each task are shown in Figure 4.2, and a depiction, in groups, of the measurements obtained for each experiment can be seen in Figure 4.3.

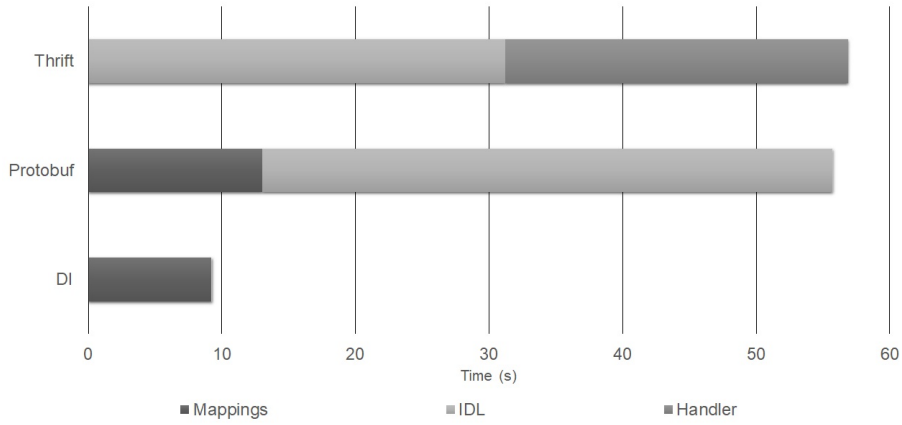

Figure 4.2: The average time needed to make an `int createCircle(double x, double y, double r)` procedure remotely available by the three alternatives which used Thrift, Protocol Buffers, and DI, for each task.

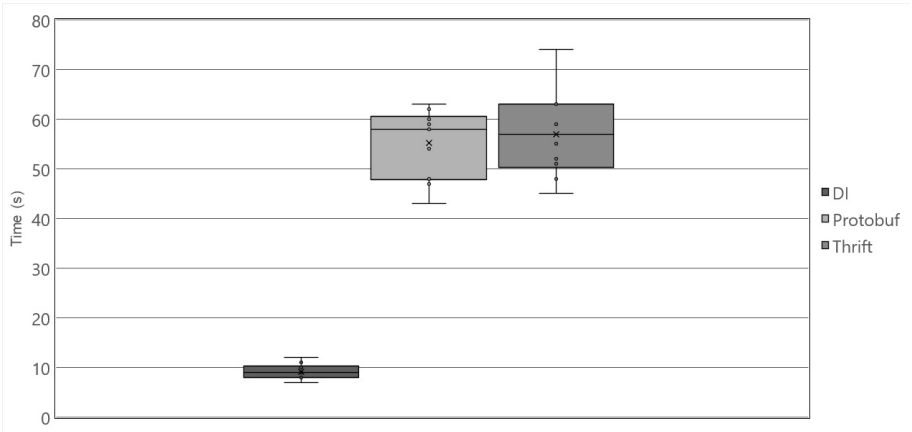

Figure 4.3: Distribution of the thirty experiments in groups (quartiles) through a whiskers plot for each alternative.

The results in Figure 4.2 show that the DI alternative, on average, is still 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 4.3 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.

This higher deviation from the average value (represented as a cross in Figure 4.3), 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 (see Listing 4.1), are relatively simple and require only a simple line of code to be added. As seen in Figure 4.4, as expected, the deviation and

length of the upper and lower whiskers for mappings is smaller than other tasks which take more time.

Listing 4.1: Examples of the instructions necessary to add a procedure `createCircle` to a map, on the DI and Protocol Buffers approaches.

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

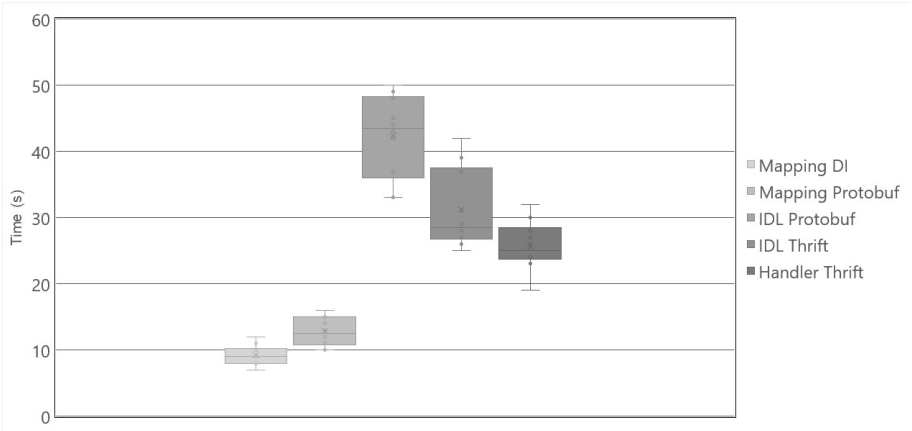

Figure 4.4: Distribution of the thirty experiments in groups (quartiles) through a whiskers plot for each task.

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. Listing 4.2 and Listing 4.3 showcase the IDL files used for this evaluation. 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 of additional time are enough to double the time it takes to add a procedure for IDL-based alternatives, such as, Protocol Buffers and Thrift.

Listing 4.2: IDL file used to generate Protocol Buffers serialization methods for `Circle` and `Id` class.

```
message Circle {
    required double x = 1;
    required double y = 2;
    required double r = 3;
}

message Id {
    required int32 id = 1;
}
```

Listing 4.3: IDL file used to generate Thrift stubs and handlers for `createCircle` function.

```
service GenerativeDesign {
    i32 Circle(1:double x, 2:double y, 3:double r)
}
```

Protocol Buffers IDL requires the programmer to separately specify a message type for request and response messages, as visible in Listing 4.2. As a consequence of that, Protocol Buffers IDL files takes, on average, more time to write (see Figure 4.4) than Thrift, which only requires the programmer to specify the return and parameter types of a remote procedure as a method of a service, as seen in Listing 4.3. However, instead of using mappings like Protocol Buffers, Thrift relies on the specification in the IDL to automatically generate stubs that dispatch the procedure's arguments to an handler, as the one shown Listing 4.4. After the compilation of an IDL this handler class methods come without any implementation. The programmer is the one responsible to write on the handler's method, the code that he wants to be executed (lines 10-11).

Although this process of writing an handler is simple, just 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 opening a file and add a mapping (see Listing 4.1). For that reason, the total time taken by Protocol Buffers and Thrift tends to even out as shown in Figure 4.2 and Figure 4.4.

Listing 4.4: Handler generated by compiling the Thrift IDL shown in Listing 4.3.

```
class GenerativeDesignHandler : virtual public GenerativeDesignIf {
 public:
  GenerativeDesignHandler() {
    // Your initialization goes here
  }
```

```
6
7    int32_t Circle(const double x, const double y, const double r) {
8      // Programmer's code
9      return createCircle(x, y, r);
10   }
11
12 };
```

One interesting evaluation which can be done to the approaches is analysing how dependent this measurements are on the complexity (number and types of the arguments) of the procedure we want to invoke. For example, we studied how long would it take if one tried to make a procedure that creates walls and returns the ids of each wall, as visible in Listing 4.5, remotely available. Note that in this case not only did we use a procedure with different C++ primitive types, but it also uses vectors of a custom type `XY`, `p0` and `p1` (line 4). This custom type is a C++ structure that defines a two dimensional point from two `doubles`, `x` and `y`.

Listing 4.5: Declaration of a procedure which creates walls depending on the size of `p0` and `p1`, which contain the beginning and end points of a each wall.

```
1 std::vector<int> createWalls(std::string type, short material, double thickness,
2                      short home_story, short stories_to_top,
3                      double top_offset, double bottom_offset,
4                      std::vector<XY> p0, std::vector<XY> p1,
5                      double angle, std::string reference_line);
```

We did five experiments for each alternative and obtained the results shown in Figure 4.5. While the solution which used DI maintained the same values, the solutions that used IDLs severely increased the time spent in comparison to their previous results, going from almost a minute, to almost three.

The reason behind the increase of time is due to the fact that all the parameters of the procedure will have to be specified in the IDL, resulting on much bigger files. Listing 4.6 shows the amount of information that needs to be written on an IDL file just to generate serialization methods for the `createWalls` procedure using Protocol Buffers. In lines 1-4 the structure `xy` is defined, and in lines 6-18 and 20-22 the parameters and return of `createWalls`, i.e., request and response messages are specified, respectively.

Listing 4.6: IDL file used to generate Protocol Buffers serialization methods for `Walls` and `Ids` class.

```
1 message xy {
2     required double x = 1;
3     required double y = 2;
```

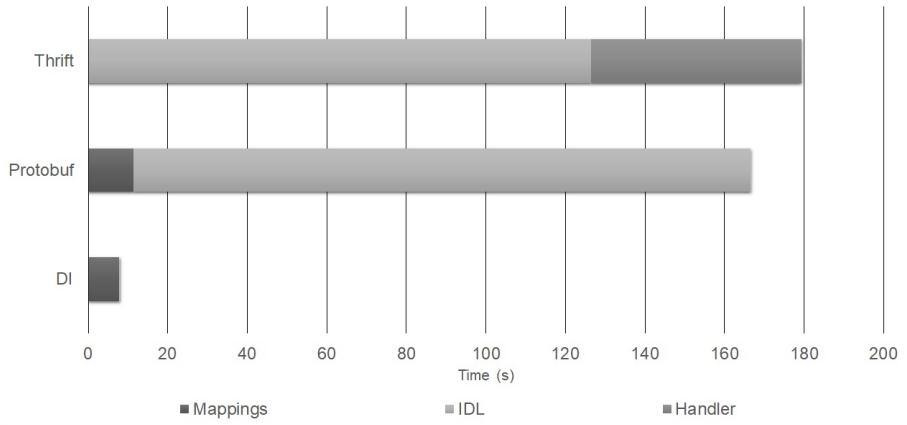

Figure 4.5: The average time needed to make a `createWalls` operation remotely available using the three alternatives Thrift, Protocol Buffers, and DI.

```
4  }
5
6  message Walls {
7      required string type = 1;
8      required int32 material = 2;
9      required double thickness = 3;
10     required int32 home_story = 4;
11     required int32 stories_to_top = 5;
12     required double bottom_offset = 6;
13     required double top_offset = 7;
14     repeated xy p0 = 8;
15     repeated xy p1 = 9;
16     required double angle = 10;
17     required string reference_line = 11;
18 }
19
20 message Ids{
21     repeated int32 id = 1;
22 }
```

The same thing happens with Thrift which also requires all parameters of the procedure to be specified (see Listing 4.7, lines 7-18). With Thrift, the implementation of the handler is also a bit delayed since more arguments need to be written by the programmer to make the function call (see Listing 4.8, lines 16-21).

Listing 4.7: IDL file used to generate Thrift stubs and handlers for `createWalls` function.

```
struct xy {
    1: optional double x;
    2: optional double y;
}

service GenerativeDesign {
    list<i32> Walls(1:string type,
                    2:i16 material,
                    3:double thickness,
                    4:i16 home_story,
                    5:i16 stories_to_top
                    6:double bottom_offset
                    7:double top_offset
                    8:list<xy> p0,
                    9:list<xy> p1,
                    10:double angle,
                    11:string reference_line)
}
```

Listing 4.8: Handler generated by compiling the Thrift IDL shown in Listing 4.7.

```
class GenerativeDesignHandler : virtual public GenerativeDesignIf {
 public:
  GenerativeDesignHandler() {
    // Your initialization goes here
  }

  void Walls(std::vector<int32_t> & _return, const std::string& type,
                const int16_t material, const double thickness,
                const int16_t home_story, const int16_t stories_to_top,
                const double bottom_offset, const double top_offset,
                const std::vector<xy> & p0, const std::vector<xy> & p1,
                const double angle, const std::string& reference_line) {
    // Your implementation goes here
    printf("Walls\n");
    //Programmer's code
    _return = createWalls(type, material, thickness,
```

```
                                home_story, stories_to_top,
                                bottom_offset, top_offset,
                                p0, p1,
                                angle, reference_line)
    }

};
```

The problem with writing IDLs 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, as visible in Figure 4.6, 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.

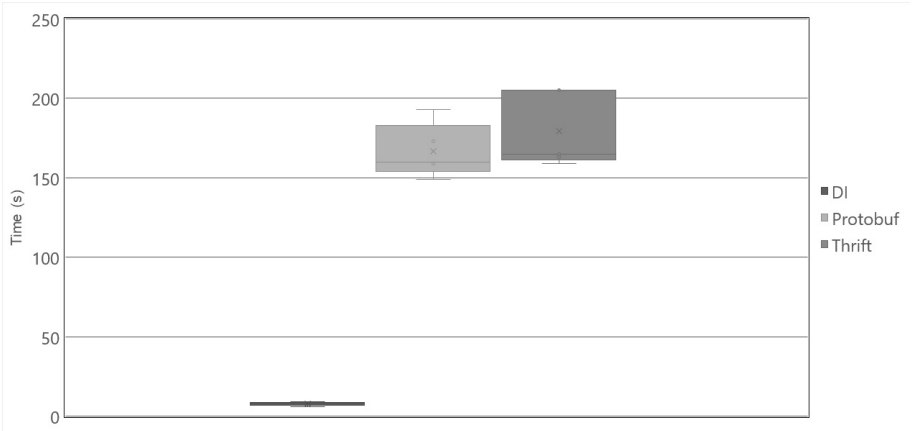

Figure 4.6: Distribution of the fifteen experiments in groups (quartiles), for the `createWalls` procedure, through a whiskers plot for each alternative.

In our solution, based on DI, it is not necessary to specify the parameters of the function, as well as their types, as visible in Listing 4.9. 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, as seen in Figure 4.5, and the deviation from

the average value is also minimal, as illustrated in Figure 4.6.

Listing 4.9: Code necessary to make a `createWalls` operation remotely available using our solution based on DI.

```
1 ADD_FUNCTION(createWalls);
```

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 create a procedure to implement one of these elements it will have to specify most of this attributes as parameters. Overall, a plug-in used for GD will have dozens of these operations.

4.3 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 (see Listing 1.2), 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 4.1 we show the results of the analysis performed on the time spent in communication and procedure execution of a `createWalls` operation (see Listing 4.5). 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.

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 accentuaded the difference is, even though, for every single wall four more values (wall's coordinates)

Table 4.1: The difference between the time during the procedure execution and the time spent in communication for different numbers of walls.

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

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 because 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 IDLs, 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. The only requirements are that these protocols should not use any IDL (to not hinder productivity), and the programmer has to implement a `Read` and `Write` function template that overloads functions based on template parameter to be used in Listing 3.5 (line 9) for read, and Listing 3.7 (line 3) for write.

For an example, as seen in Listing 4.10, notice that `Write` does not need to use `Type` because C++ template overload works with function parameters, but not on return types (which is why we encapsulate `Type` in a struct for `Read`).

Listing 4.10: Using C++ function template overloading to implement `Read` and `Write` functions which overload functions based on template parameters.

```
template<typename T>
struct Type {};

//Primary Read function template
template<typename T>
T Read(Type<T>) {}

template<>
int Read(Type<int>) {
    ... //TODO
}

//Template function that uses Type to choose implementation of Read(Type<T>)
template<typename T>
T Read() {
  return Read(Type<T>{});
}

//Primary Write function template
template<typename T>
void Write(T value) {}

template<>
void Write(int value) {
  ... //TODO
}
```

4.4 Use case: Generative Design with Khepri

Our solution can be used with a programming environment which explores GD: Khepri. Khepri is implemented in Julia and provides a set of abstractions which can be used by simple Julia programs to perform remote invocations on BIM and CAD software applications' plug-ins, masquerading details of the communication between user programs and a software application plug-in.

In this case we used our solution in an ArchiCAD plug-in, and Khepri to communicate with our solution, allowing a simple user program to remotely invoke the functionality implemented in an ArchiCAD plug-in. Figure 4.7 shows a diagram of the components of this use case.

User programs consist on the use of a set of directives provided by Khepri which allow GD with CAD

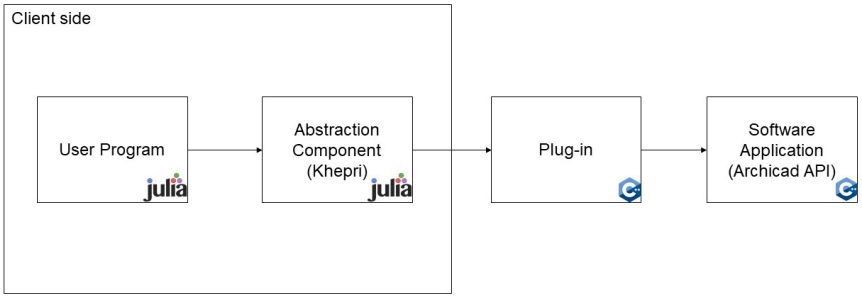

Figure 4.7: Diagram that illustrates the overall architecture of this use case.

and BIM operations, as exemplified in Listing 4.11. Initially, the program has to declare the back-end (line 2), i.e., the software application which will be used to produce the modelling results. Then, using Khepri's geometrical abstractions, such as, the cartesian coordinate system, the program can remotely invoke the procedures implemented in the plug-in (line 4-8).

Listing 4.11: Using Khepri to invoke CAD procedures in ArchiCAD.

```
1 using Khepri
2 backend(archicad)
3
4 circle(xy(0, 0), 3)
5 spline(xy(-2,2), xy(2,0), xy(-2,-2))
6 spline(xy(1, 2), xy(3,4), xy(4,-1), xy(6,5))
7 line(xy(0.0), xy(0,3))
```

As shown in Listing 4.11, aside from the specification of the back-end, the program does not have to specify any details of the communication with the plug-in. Khepri is responsible for this process. The protocol used by Khepri to communicate with software application's plug-ins is similar to the one presented in Figure 3.5. For example, in line 5, the spline call needs to send a request to the plug-in to make the spline operation available, returning its id. After receiving the id, the request to a spline invocation is done. Since Khepri already knows the id of the `spline` procedure, in line 6 it can immediately request an invocation. Notice how this process is abstracted and both calls to `spline` are similar.

One of the reasons which made us opt for implementing this protocol in our solution was because Khepri used it, and the idea was to use our solution with Khepri from the start. Note that, as we have

said back in Section 3.5 our solution for DI in C++ does not depend on the protocol used.

The communication done with Khepri is done over a TCP/IP socket and the data is serialized in a binary format compatible to the one used by C#'s `BinaryReader`[1] and `BinaryWriter`[2] classes. The reason for that is due to the fact that Khepri can be used for GD with other software applications, such as Revit. Revit is a BIM software application which implements GD procedures in a C# plug-in, thus, the use of a C# binary reader and writer.

Because Khepri already used this binary format, for the communication with the Revit's plug-in, we decided to also implement a compatible binary protocol in C++ using the method described in Listing 4.10. This made the set-up with Khepri a lot easier as these details were reused, avoiding major changes to be done on the Khepri's side.

Furthermore, our solution can also be used to add other back-ends (software applications), which implement their plug-ins in C++, to Khepri. For example, there is a work in development which intends to enable GD for Unreal Engine. In this instance, just like what was done with ArchiCAD, our solution will be used to link Khepri with the procedures implemented in Unreal Engine, allowing their remote invocation.

4.5 Summary

In this chapter we evaluated and compared our solution with two alternatives: one which used an RPI framework, Thrift, and another one which relied on serialization of messages with Protocol Buffers, and C++ function pointers to invoke a plug-in's procedures. All studied approaches were implemented in a C++ plug-in for ArchiCAD BIM software application along with a few procedures. The plug-ins launch single-threaded blocking servers which communicate with their respective C++ clients, also implemented in C++, over a TCP/IP socket.

Some experiments were made to measure the time taken to make a procedure remotely available. There, it was verified that our solution takes significantly less time (around 10 seconds) when compared to other alternatives which use IDLs to specify the data necessary for a remote invocation (more than 50 seconds). We have seen that the different measurements in these alternatives tend to deviate more from their average values, as a consequence of their lengthy tasks (writing in IDL files) and possibly because of the association between the programmer's experience with the IDL syntax. We also showcased that for more complex procedures (more arguments), in IDL-based approaches these values are increased as more data and information needs to be specified for each procedure. On the other hand, our solution, based on DI, is not affected by such circumstances as it only requires the programmer to specify in a macro call the name of the function where the procedure is implemented.

We also made a performance analysis where we compared the time spent on the execution of a

[1] https://docs.microsoft.com/en-us/dotnet/api/system.io.binaryreader?view=netframework-4.8
[2] https://docs.microsoft.com/en-us/dotnet/api/system.io.binarywriter?view=netframework-4.8

procedure and the time spent with communication. We saw that the time spend with communication was negligible (3% of total) when compared to the time spent executing the procedure, which renders the performance improvements of more advanced serialization methods which use data compression as almost insignificant. Because there might be software applications which do not take that much time executing a procedure, and our solution does not restrict the programmer to a specific serialization method, we showcased an implementation to incorporate different serialization methods into our solution.

In the end, we provided an example where our solution can be used. Along with Khepri, a programming environment which allows GD, our solution can enable the remote invocation of several CAD and BIM operations implemented in an ArchiCAD plug-in.

5

Conclusion

Contents

The interest of remotely invoking software applications plug-ins' procedures has been increasing recently. Many software applications, such as, CAD and BIM tools, take plenty of benefits from exposing their functionality. However, to expose the functionality, plug-ins must be implemented, and include a mechanism that allows the remote invocation of its procedures, RPI.

There are many RPI frameworks which can be used to this end. By using these frameworks most of the RPI process is abstracted, including the serialization and deserialization of arguments and their dispatch to handlers. These tools rely on the specification of the plug-in's remote interface as services in an IDL file, to generate the code for the stubs (who dispatch arguments) and service handlers, which implement the procedures. However, when using these frameworks, we have seen that the process of adding new remote procedures is lengthy, requiring the programmer to edit IDL files, and, consequently, change the service handlers.

Alternatively, instead of using RPI frameworks, one can implement an RPI mechanism. In this case, because of the differences in the characteristics of client and server applications (particularly, programming languages), it is necessary to use a mechanism which ensures both applications understand the exchanged data. Data serialization methods can be used to solve this problem. However, some serialization methods also require the programmer to specify the structure of the data in an IDL file. Although the compilation of these files only generates code for the serialization and deserialization of data, which means that the programmer does not have to change service handlers, the process of adding new features is still tedious and monotonous, and barely has any improvements when compared to RPI frameworks.

Nonetheless, even by using a serialization method which does not require an IDL, to enable RPI, the programmer needs to implement a mechanism that is able to invoke and dispatch arguments to a function or method, which has the implementation of a procedure. In this work, we presented DI as the mechanism that performs these invocations without specifying the data of the arguments which are forwarded to a function, relieving the tasks which the programmer has to perform to make a procedure remotely available.

By receiving a function or method identifier, i.e., metadata (e.g., function name), and values of the arguments for the invocation, DI should be able to invoke a procedure's implementation using metaprogramming techniques. In Chapter 2, we explained a few mechanisms which, based on reflection, could invoke functions as a normal function call expression, and invoke methods using metaobjects or other built-in metadata mappings to function references.

We also presented a few features of C++ templates which were used to implement our solution. In Chapter 3 we explained our solution, which mainly focuses on using the properties of generic types in C++ template programming to define a generic behaviour for the invocation of C++ non-member functions (functions where the plug-in's procedures are implemented). These generic types are materialised

by the compiler at compile time, by a process called instantiation, generating the code necessary for an invocation. We demonstrated how a simple macro call can trigger the instantiation of all this code simplifying the whole process of making procedures remotely available.

Finally, in Chapter 4 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 avaialbe as 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.

Our solution also contributed to the development of Khepri, a programming environment that enables GD. It allowed the addition of ArchiCAD to the set of software applications which Khepri can perform invocations on.

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.

5.1 Future Work

There are several paths available for the development of future work. First, the current solution could be improved to allow the invocation of C++ member functions, i.e., functions that are associated to an object. That is because these types of functions are invoked on instances of a class, meaning that there must be a mechanism to create, manage, and delete, these instances. Note that static member functions do not suffer from this issues as they are not associated with an instance, but with a class.

Another improvement, is the implementation of a mechanism to remotely invoke overloaded functions (functions with the same name but different signatures). Currently, the solution does not support the invocation of overloaded functions because the responsibility of deducing the types of the arguments

and return of a function is done by the compiler through the function identifier/name. Therefore, it is impossible for the compiler to deduce the correct function to invoke without specifying arguments or return types.

Our solution implements DI in C++ following a imperative programming paradigm. It uses features of procedural programming by scattering some functionality across different non-member functions (e.g., wrappers and remote procedures), and also utilizes object oriented features, such as polymorphism, for the implementation of generic behaviour. Although in this work we talked briefly about how DI could be implemented in other programming languages, it would be interesting to do a generic study on how the implementation of DI could be done in different programming paradigms, using different programming language features.

Another line of work, which was already partially explored in this document, would be to integrate our solution with different client applications and software application plug-ins to enable RPI of a software application's API. In this work, we used our solution alongside Khepri to enable GD for ArchiCAD. The same could be done for different software application, which implement plug-ins in C++, for example, Unreal Engine.

Alternatively to this work, some software applications might benefit from faster RPIs or from the abstraction of the RPI implementation. In these cases it would be worth to study and compare: (1) different communication protocols, and serialization methods (with and without an IDL), to decrease the time spent in an RPI, and (2) different RPI frameworks which abstract the RPI implementation with the code generation of IDLs. Note that there are also RPI frameworks which do not require the specification of the remote interface in an IDL. For example, XML-RPC, requires this specification to be done in the code itself.

Finally, outside of the spectre of software applications, it would be interesting to see how DI implementations could fare against the usual RPI frameworks (e.g., gRPC and Thrift) in microservice architectures, i.e., architectures where applications are decomposed in several simple, small, and remote services. Microservice architectures benefit from the use of IDLs because each service may be written in a different programming language and the code generation saves the time of specifying the data on each service's code, increasing development productivity. Therefore, it would be interesting to compare the productivity during development of a DI approach and an RPI framework used in microservices, to assess if the DI approach would still have the edge.

Bibliography

[1] R. Abernethy, *Programmer's Guide to Apache Thrift*. Manning Publications, 2019.

[2] G. Barreto, "Generative Design for Building Information Modeling," M.S. Thesis, Instituto Superior Técnico, 2016.

[3] K. Terzidis, *Expressive Form: A conceptual approach to computational design*. Routledge, 2004.

[4] N. Dragoni, S. Giallorenzo, A. L. Lafuente, M. Mazzara, F. Montesi, R. Mustafin, and L. Safina, "Microservices: yesterday, today, and tomorrow," in *Present and ulterior software engineering*. Springer, 2017, pp. 195–216.

[5] "gRPC," Accessed on: Sep. 5, 2019. [Online]. Available: https://grpc.io/

[6] M. Slee, A. Agarwal, and M. Kwiatkowski, "Thrift: Scalable cross-language services implementation," *Facebook White Paper*, vol. 5, no. 8, 2007.

[7] "Google Protocol Buffers," Accessed on: Sep. 15, 2019. [Online]. Available: https://developers.google.com/protocol-buffers/

[8] P. Maes, "Concepts and experiments in computational reflection," in *ACM Sigplan Notices*, vol. 22, no. 12. ACM, 1987, pp. 147–155.

[9] P. K. McKinley, S. M. Sadjadi, E. P. Kasten, and B. H. Cheng, "Composing adaptive software," *Computer*, no. 7, pp. 56–64, 2004.

[10] D. Vandevoorde and N. M. Josuttis, *C++ Templates: The Complete Guide (2nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., 2018.

[11] D. R. Musser and A. A. Stepanov, "Generic programming," in *International Symposium on Symbolic and Algebraic Computation*. Springer, 1988, pp. 13–25.