Imperial College London

Individual Project Interim Report

Native Calls

Author:
Mohamed Eltuhamy

Supervisor:
Dr. Cristian Cadar

Co-Supervisor:
Petr Hosek

Department of Computing

January 2014

Contents

Chapter 1

Introduction

Over the past decades, the web has quickly evolved from being a simple online catalogue of information to becoming a massive distributed platform for web applications that are used by millions of people. Developers have used JavaScript to write web applications that run on the browser, but JavaScript has some limitations.

One of the problems of JavaScript is performance. JavaScript is a single threaded language with lack of support for concurrency. Although web browser vendors such as Google and Mozilla are continuously improving JavaScript run time performance, it is still a slow interpreted language, especially compared to compiled languages such as C++ (need reference). Many attempts have been made to increase performance of web applications. One of the first solutions was browser plugins that run in the browser, such as Flash or Java Applets. However, these have often created browser bugs and loop-holes that can be used maliciously to compromise security.

Native Client [1] (NaCl) is a technology from Google that allows running binary code in a sandboxed environment in the Chrome browser. This technology allows web developers to write and use computation-heavy programs that run inside a web application, whilst maintaining the security levels we expect when visiting web applications.

The native code is typically written in C++, though other languages can be supported. The code is compiled and the binary application is sandboxed by verifying the code to ensure no system-calls are made. This is done by compiling the source code using the gcc[1] based NaCl compiler. This generates a NaCl module that can be embedded into the web page. Because no system calls can be made, the only way an application can communicate with the operating system (for example, to play audio) is through the web

[1]The GNU Compiler Collection (gcc) is an open-source compiler that supports C, C++, and other languages [2]

browser, which supports several APIs in JavaScript that are secure to use and also cross-platform. This means that the fast-performing C++ application needs to communicate with the JavaScript web application.

```
// Send a message to the NaCl module
function sendHello () {
  if (HelloTutorialModule) {
    // Module has loaded, send it a message using postMessage
    HelloTutorialModule.postMessage('hello');
  } else {
    // Module still not loaded!
    console.error("The module still hasn't loaded");
  }
}

// Handle a message from the NaCl module
function handleMessage(message_event) {
  console.log("NACL: " + message_event.data);
}
```

Listing 1.1: JavaScript code

```
// Handle a message coming from JavaScript
virtual void HandleMessage(const pp::Var& var_message) {
  // Send a message to JavaScript
  PostMessage(var_message);
}
```

Listing 1.2: C++ Code

The way Native Client modules can communicate with the JavaScript web application (and vice versa) is through simple message passing. The JavaScript web application sends a message in the form of a JavaScript string to the NaCl module. The NaCl module handles message events by receiving this string as a parameter passed into the `HandleMessage` function. For example, Listing 1.1 shows a simplified example of how JavaScript sends a message to the NaCl module, and Listing 1.2 shows how the native module handles the message and sends the same message back to the JavaScript application. This allows for straight forward, asynchronous communication between the native code and the web application. Modern web browsers support message passing using the `postMessage` API. This was designed to allow web applications to communicate with one or more web workers [2].

However, message passing puts more burden on the developer to write the required communication code between the NaCl module and the application. For example, consider

[2]Web workers[3] are scripts that run in the background of a web page, independent of the web page itself. It is a way of carrying out computations while not blocking the main page's execution. Although they allow concurrency, they are relatively heavy and are not intended to be spawned in large numbers. Typically a web application would have one web worker to carry out computations, and the main page to do most of the view logic (such as click listening, etc.)

a C++ program that performs some heavy computations and has functions that take several parameters of different types. To make the functionality accessible from the web application, the developer would need to write a lot of code in the `HandleMessage` function. A message format would need to be specified to distinguish which function is being called. Then the parameters of the function call would need to be identified, extracted, and converted into C++ types in order that the parameters are passed into the C++ function. Then a similar procedure would need to be done if the function would return anything back to the web application.

The purpose of this project is to allow developers to easily invoke NaCl modules by creating a Remote Procedure Call (RPC) framework on top of the existing message passing mechanism. To achieve this, the developer will simply write an Interface Definition Language (IDL) file which specifies the functions that are to be made accessible from JavaScript. The IDL file will be parsed in order to automatically generate JavaScript and C++ method stubs that implement the required communication code using message passing. This is similar to how RPC is implemented in other frameworks, such as ONC RPC or CORBA (need references).

The main contributions of this project is to create a tool that parses IDL files and generates JavaScript and C++ method stubs, a message format that will be used in communication, and support libraries in JavaScript and C++ that will use message passing to do the actual communication. This will allow functions in the Native Client module to be called directly from the JavaScript application. We will evaluate how much this will help developers by seeing how many lines can be saved, in different program contexts. We will also analyse the speed and efficiency of using RPC over hand-written message passing.

Chapter 2

Background

2.1 Native Client

Native Client (NaCl) can be thought of as a new type of plugin for the Google Chrome browser that allows binary programs to run natively in the web browser. It can be used as a 'back end' for a normal web application written in JavaScript, since the binary program will run much faster. A NaCl module can be written in any language, including Assembly, so long as the binary is checked and verified to be safe by the NaCl sandbox [1]. However, NaCl provides a Software Development Kit (SDK) that includes a compiler based on gcc that allows developers to compile C and C++ programs into binary that will work directly with the sandbox without further modifications. Thus, writing NaCl-compatible C++ programs is just as easy as writing normal C++ programs. The difference is that the sandboxes disallow unwanted side-effects, including system-calls. Since many applications might want to have these side effects, Native Client provides a set of cross-platform API functions that achieve the same outcomes, but by communicating with the browser directly. To avoid calling NaCl syscalls directly, an independant runtime (IRT) is provided, along with two different C libraries (newlib and glibc) on top of which the Pepper Plugin API (PPAPI or 'Pepper') is exposed. It can be used to do file IO, play audio, and render graphics. The PPAPI also includes the `PostMessage` functionality, which allows the NaCl module to communicate with the JavaScript application.

2.1.1 NaCl Modules and the Pepper API

A native client application consists of the following [4]:

HTML/JavaScript Application This is where the user interface of the application will be defined, and the JavaScript here could also perform computations. The HTML file will include the NaCl module by using an embed tag, e.g.

```
<embed src="myModule.nmf" type="application/x-nacl" />
```

Pepper API Allows the NaCl module communicate with the web browser and use its features. Provides `PostMessage` to allow message passing to the JavaScript application.

Native Client Module The binary application, which performs heavy computation at native speeds.

2.1.2 Communicating with JavaScript using postMessage

The HTML5 `postMessage` API was designed to allow web workers to communicate with the main page's JavaScript execution thread. The JavaScript object is copied to the web worker by value. If the object has cycles, they are maintained as long as the cycles exist in the same object. This is known as the structured clone algorithm, and is part of the HTML5 draft specification [5].

In a similar way, `postMessage` allows message passing to and from NaCl modules. However, sending objects with cycles will cause an error. NaCl allows sending and receiving primitive JavaScript objects (`Numbers`, `Strings`, `Booleans`, `null`) as well as dictionaries (key-value `Objects`), arrays, and `ArrayBuffers`. ArrayBuffers are a new type of JavaScript object based on Typed Arrays [6] that allows the storing of binary data.

Another key difference is that message types need to be converted into the correct type on the receiving end. For example, sending a JavaScript `Object` should translate into a dictionary type. The JavaScript types are dynamic in nature. A JavaScript `Number` object could be an integer, a float, a double, 'infinity', exponential, and so on. Sending C++ data to JavaScript is simple since it is converting from a more specific type to a less specific type (e.g. from `int` in C++ to `Number` in JavaScript). But converting from a JavaScript type to a C++ type requires more thought. The PPAPI provides several functions to determine the JavaScript type (e.g. `bool is_double()`). It also allows us to extract and cast the data into our required type (e.g. `double AsDouble()`). From there, we can use the standard C++ type to perform the required computations.

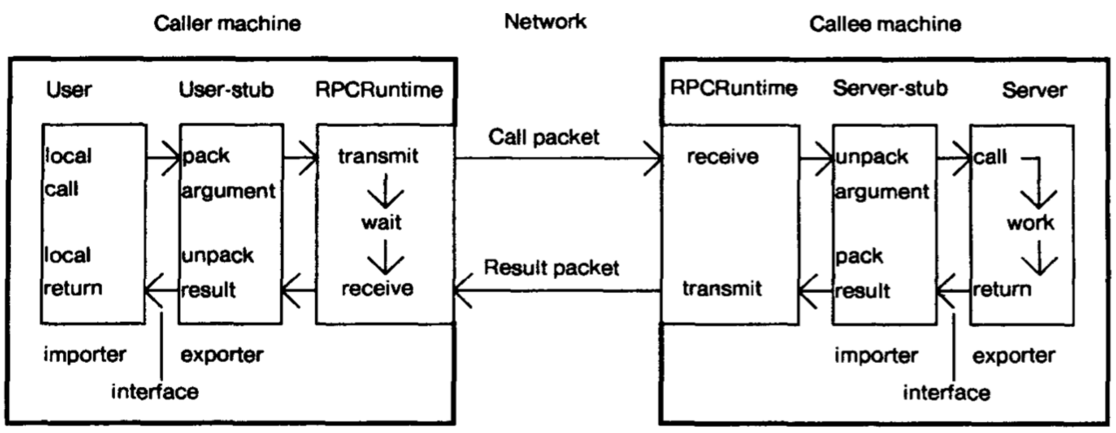

FIGURE 2.1: The basic components of an RPC framework, from [7]

2.2 Remote Procedural Call (RPC)

RPC is used to uniformly call a procedure that is on a different machine, or on the same machine but on different processes. RPC is implemented on top of a transmission protocol and should work regardless of the communication method being used. For example, we could use TCP/IP for network communications, or any Inter-Process Communication (IPC) method if the caller and callee are on the same machine but in different processes. Normally, RPC implementations would consist of the following steps, as shown in Figure 2.1.

1. The caller code is written normally, and so is the server code, but the stubs are automatically generated using interface definition files.
2. When the remote call is made, it calls the user stub which packs the parameters and function call information into a packet.
3. The packet gets transferred to its destination (either across the network as in Figure 2.1, or across the processes on the same machine using IPC). This is done through the RPCRuntime, which is a library that works on both ends (caller and callee) to handle communication details.
4. The packet is received at the callee end by the RPCRuntime. It is then passed on to the server stub.
5. The arguments and function call information are unpacked and a normal call is made to the actual procedure.
6. When the procedure returns, it is passed back to the server stub where it is packed and transmitted back to the caller, which unpacks it and uses the result.

2.2.1 The role of the RPCRuntime

The RPCRuntime is responsible for carrying out the actual communication of the RPC call information between the caller and the callee. It exists both in the caller and callee endpoints. When the caller makes a RPC call, the information is sent from the RPCRuntime sitting in the caller side, and is received by the RPCRuntime in the callee side. When the callee returns, the return data is sent from the callee's RPCRuntime to the caller's RPCRuntime.

In order to keep the context of a remote call, the RPCRuntime also sends some extra meta data along with the arguments. This meta data includes:

1. A call identifier. This is used for two reasons:
   (a) To check if the call has already been made (i.e. to ensure no duplicate calls)
   (b) To match the return value of the callee with the correct caller.
2. The name of the procedure the caller is calling.
3. The arguments (parameters) we wish to pass to the remote procedure.

The RPCRuntime on the caller side maintains a store of call identifiers that are currently in progress. When the remote functions return, they send the same call identifier along with the return value. That call identifier is then removed from the caller's store to indicate that the remote call has completed. The call identifier is also used to implement the call semantics. Call semantics could be *at least once*, where the RPC system will keep trying to call the remote procedure if the transport fails, and/or *at most once*, where the system will ensure that the function is not called more than once (which is needed for nonidempotent functions).

2.3 RPC Implementations

2.3.1 Open Network Computing (ONC) RPC

ONC is a suite of software originally developed and released in 1985 by Sun Microsystems[8]. It provides a RPC system along with an External Data Representation (XDR) format used alongside it. The ONC RPC system implements some tools and libraries that make it easy for developers to specify and use remote functions. These are:

1. **RPCGen Compiler:** As mentioned earlier, the role of the user and server stubs is to pack and unpack arguments and results of function calls. To pack the arguments, the stub looks at the argument types and matches them with the number of arguments and their types of the server (callee) function definition. Thus, the stubs need to be written with knowledge of the interface of the actual procedures that will be called. We can define these interfaces in an abstract way, so that we could generate these stubs automatically even if the languages used in the different endpoints are different. In ONC RPC and many other systems, this abstract representation is in the form of an Interface Definition Language (IDL) file. When we pass the IDL file into the RPCGen compiler, it automatically generates the stubs we need to perform remote procedure calls.

2. **XDR Routines:** These convert the types of the parameters and return values to and from the external data representation. XDR routines exist for many C types, and the system allows you to write your own XDR routines for complex types.

3. **RPC API library:** This is an implementation that fulfils the role of the RPCRuntime described in 2.2.1. It provides a set of API functions that set up the lower level communication details, binding, and so on.

Remote procedures in ONC RPC are identified by a program number, a version number, and a procedure number. There also exists a port mapper that map the program number to a port, so that several programs can run on the same remote machine.

2.3.1.1 XDR files

In ONC RPC, the XDR format is used to define RPC definitions. For example, the RPC definition in listing 2.1 defines an interface for a simple function that takes in a character string and returns a structure containing two fields. As discussed in 2.3.1, we can see the program number is `80000` and the procedure number of the `generate_keypair` function is `1`.

```
/* File: keypairgen.x */
struct key_pair_t
{
  string  public_key<500>;
  string  private_key<500>;
};

program KEYPAIRGEN_PROGRAM
{
  version KEYPAIRGEN_VERSION
  {
    /* Produce a public/private key pair using a passphrase  */
    key_pair_t generate_keypair (string) = 1;
  } = 0;
} = 80000;
```

LISTING 2.1: An example RPC definition for a key-pair generator function

We can use the RPCGen compiler to then create client and server stubs. Passing the definition file `keypairgen.x` (shown in listing 2.1) into `rpcgen` will produce the following files:

- **keypairgen.h** The header file, which would be included in both client and server code. Includes the actual C definition of the result_t structure we defined in the XDR.
- **keypairgen_clnt.c** The client stub, which packs the parameters and uses the RPC API library to execute the actual remote procedure call.
- **keypairgen_svc.c** The server stub, which uses the RPC API to set up a listener for RPC calls. RPC calls are received, parameters are unpacked, and the actual function implementation (of `generate_keypair`) is called.
- **keypairgen_xdr.c** Defines methods for packing more complex structures, such as the `key_pair_t` structure we defined.

Now we need to write the actual implementation of the RPC procedure we wish to call remotely, namely `generate_keypair`. This will include the generated header file and follow the specification we defined, as shown in listing 2.2.

```
#include "keypairgen.h"

key_pair_t *
generate_keypair_0_svc(char **argp, struct svc_req *rqstp)
{
  static key_pair_t  result;
  // TODO: Actual implementation
  return(&result);
}
```

LISTING 2.2: An example server-side implementation of the procedure defined in 2.1

Finally, we call the remote procedure from the client, which includes the same header file and simply calls `generate_keypair_0`, passing in the string parameter.

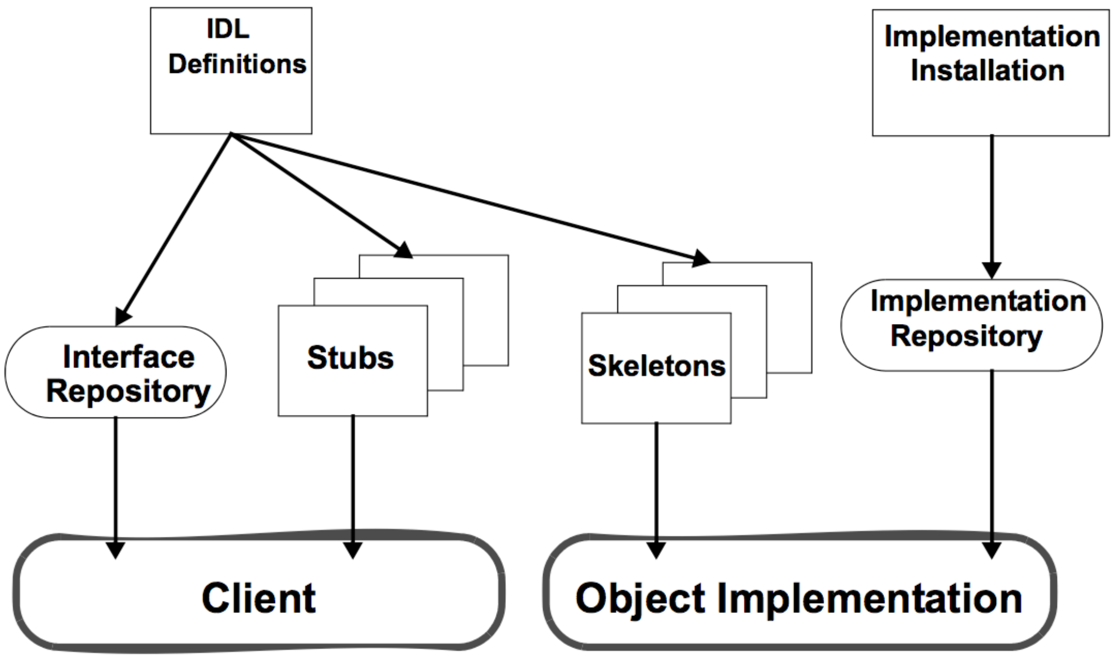

FIGURE 2.2: Interface and Implementation Repositories in CORBA, from [9]

2.3.2 Common Object Request Broker Architecture (CORBA)

CORBA is a RPC implementation introduced in 1991 by the Object Management Group (OMG) to address some issues with existing RPC implementations and provide more features for object oriented programming.

Remote method calls on objects revolve around the use of the Object Request Broker (ORB)[9]. A client invokes a remote object by sending a request through the ORB. The ORB locates the object on the server, and handles the communication of the remote call from the client to that object, including parameter and result packing. The client is statically aware of the objects it could invoke through the use of IDL (known as OMG IDL) stubs. These specify everything about the remote object except the implementation itself. This includes the names of classes, method interfaces, and fields. The OMG IDL is independent of any language, and bindings exist for several languages.

Remote objects could also be invoked dynamically at runtime, as CORBA supports dynamic binding. This works by adding the interface definitions to an Interface Repository service. The implementation of the remote object is not aware how it was remotely invoked as shown in Figure 2.2.

TODO: Write more about CORBA using the CORBA paper. [10]

2.3.3 JSON-RPC and XML-RPC

XML-RPC is a simple RPC protocol which uses XML (Extended Mark-up Language) to define remote method calls and responses. It uses explicit data typing - the method name and parameters are hard-coded in the message itself. Messages are typically transported to remote servers over HTTP[1]. Many implementations of XML-RPC exist in several different languages.

For example, we could represent the RPC function call we defined before (Listing 2.1) as the XML-RPC function call shown in Listing 2.3.

```
<methodCall>
  <methodName>
    generate_keypair
  </methodName>
  <params>
    <param><value><string> myPassPhraseHere </string></value></param>
  </params>
</methodCall>
```

Listing 2.3: An example XML-RPC call

The XML-RPC implementation could give a better interface for the XML calls. For example, run-time reflection APIs could be used to dynamically translate procedure calls into XML-RPC requests. The response from the server would also be in XML form. For example, the response for the request shown in Listing 2.3 would be as shown in Listing 2.4.

```
<methodResponse>
  <params>
    <param>
        <value>
          <struct>
            <member>
              <name>public_key</name>
              <value><string>qo96IJJfiPYWy3q3p5nvbNME87jG</string></value>
            </member>
            <member>
              <name>private_key</name>
              <value><string>IIEpAIBAAKCAQEA4eLvDruo9CswdW</string></value>
            </member>
          </struct>
        </value>
    </param>
  </params>
</methodResponse>
```

Listing 2.4: An example XML-RPC response

[1]Hyper-text transport protocol (HTTP) is the most common transfer protocol used by clients and servers to transfer data on the web

XML-RPC supports simple types like integer, double, string, boolean values. It also supports some complex types like arrays, and associative arrays (`struct`). An example of this is in Listing 2.4, where our structure has two keys with their respective values. Binary data can be Base64[2] encoded and sent in a `<base64 />` tag. Because of the fixed language, XML-RPC is naturally cross-language compatible, as it is up to the two ends (client and server) to implement and use their own XML parsers and converters. Several libraries in different languages exist that do this.

XML-RPC and JSON-RPC are very similar. JavaScript Object Notation (JSON) is a simple and light-weight human-readable message format. Its advantages over XML is that it is a lot lighter and simpler. However, XML can be extended to support complex user-defined types using XML-Schemas, which is not possible directly using JSON. JSON-RPC is also a protocol and message format, for which different implementations exist. For example, we can easily represent the RPC call and response shown in Listings 2.3 and 2.4 using the JSON-RPC protocol format as shown in Listing 2.5.

```
// JSON-RPC request:
{
  "jsonrpc": "2.0",
  "method": "generate_keypair",
  "params": ["myPassPhraseHere"],
  "id": 1
}


// JSON-RPC response:
{
  "jsonrpc": "2.0",
  "result": {
    "public_key": "qo96IJJfiPYWy3q3p5nvbNME87jG",
    "private_key": "IIEpAIBAAKCAQEA4eLvDruo9CswdW"
  },
  "id": 1
}
```

Listing 2.5: An example JSON-RPC request and response

Both XML-RPC and JSON-RPC have well-defined protocols [11][12], and are implemented in many different languages.

[2]Base64 is an encoding scheme that represents binary data as an ASCII string

2.3.4 WebIDL

WebIDL is a specification [13] for an interface definition language that can be used by web browsers. It is used in several projects, including Google Chrome's Blink project[3].

The WebIDL syntax is similar to ONC RPC's XDR syntax (see section 2.3.1.1). Listing 2.6 shows the same interface as Listing 2.1, but this time using WebIDL.

```
dictionary keypair {
  DOMString public_key;
  DOMString private_key;
};

interface KEYPAIRGEN {
  keypair generate_keypair(DOMString passphrase);
};
```

Listing 2.6: An example interface definition written in WebIDL

Just like in ONC RPC, the WebIDL files can be parsed and used to generate stub methods for the client and server. Because they are language independent, the client and server files that are generated could be in different languages.

Open source parsers exist for WebIDL, and a standard-compliant one is provided in the Chromium project[4].

There are also open source C++ bindings for WebIDL, such as esidl[5]. Similarly, bindings for JavaScript also exist[6].

2.4 Data representation

When designing RPC systems, the data representation of the messages being transferred, including how the parameters are marshalled, needs to be defined. This is because the client and sever might have different architectures that affect how data is represented. There are two types of data representation: implicit typing and explicit typing.

Implicit typing refers to representations which do not encode the names or the types of the parameters when marshalling them. Only the values of the parameters are sent. It is up to the sender and receiver to ensure that the types being sent/received are correct,

[3]http://www.chromium.org/blink/webidl

[4]`https://code.google.com/p/chromium/codesearch#chromium/src/tools/idl_parser/idl_parser.py`

[5]esidl is a library provided with the es Operating System project, which is an experimental operating system whose API is written in WebIDL. The WebIDL compiler can be obtained from: https://github.com/esrille/esidl and documentation can be found here: http://code.google.com/p/es-operating-system/wiki/esidl

[6]https://github.com/extensibleweb/webidl.js

and this is normally done statically through the IDL files which specify how the message will be structured.

Explicit typing refers to when the parameter names and types are encoded with the message during marshalling. This increases the size of the messages, but simplifies the process of de-marshalling the parameters.

This section gives an overview of some of the different message formats that can be used with RPC.

XML and JSON

XML and JSON are widely used data representation formats that are supported by many languages. They are supported by default in all web browsers, which include XML and JSON parsers. XML and JSON - based RPC implementations exist, and we discuss them in Section 2.3.3.

Although XML and JSON are both intended to be human-readable, JSON is often more readable. JSON is also more compact, as it requires less syntax to represent complex structures, in contrast to XML which requires opening and closing tags. Here is an example of representing a phone book in XML and JSON.

```
[
        {
                "name" : "John Smith",
                "id"   : 1,
                "phonenumber": "+447813945734"
        },
        {
                "name" : "Jane Taylor",
                "id"   : 2,
                "phonenumber": "+442383045711"
        },
]
```

Listing 2.7: Representing a phone book using JSON

```
<numbers type="array">
    <entry>
        <field name="name">John Smith</field>
        <field name="id">1</field>
        <field name="phonenumber">+447813945734</field>
    </entry>
    <entry>
        <field name="name">Jane Taylor</field>
        <field name="id">2</field>
        <field name="phonenumber">+442383045711</field>
    </entry>
</numbers>
```

Listing 2.8: Representing a phone book using XML

The XML would also need to be parsed to make sense of the data. For example, a 'field' tag could be parsed and converted into a C++ data structure, but that would require us to understand the structure we're using. Sometimes the structure is well defined, like in the XML-RPC protocol (see Section 2.3.3). However, this strict parsing requirement is easier to error check, since if the XML was parsed successfully, we can be more confident that the data type is correct.

Protocol Buffers

Google Protocol Buffers are "a language-neutral, platform-neutral, extensible way of serializing structured data for use in communications protocols, data storage, and more", according to the developer guide[7]. They are used extensively within many Google products, including AppEngine[8].

Messages are defined in .proto files. Listing 2.9 shows an example, adapted from the Developer Overview.

```
message Person {
  required string name = 1;
  required int32 id = 2;
  optional string email = 3;

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

  message PhoneNumber {
    required string number = 1;
    optional PhoneType type = 2 [default = HOME];
  }

  repeated PhoneNumber phone = 4;
}
```

Listing 2.9: A .proto file

The .proto file is then parsed and compiled, to generate data access classes used to change the content of an instance of the representation. They also provide the methods required for serialization. These methods are shown in Listing 2.10. When serialised data is converted to raw binary.

[7]https://developers.google.com/protocol-buffers/docs/overview

[8]Google AppEngine is a Platform as a Service that allows developers to run their applications on the cloud

```
// Serialization
Person person;
person.set_name("John Doe");
person.set_id(1234);
person.set_email("jdoe@example.com");
fstream output("myfile", ios::out | ios::binary);
person.SerializeToOstream(&output);

// De-serialisation
fstream input("myfile", ios::in | ios::binary);
Person person;
person.ParseFromIstream(&input);
cout << "Name: " << person.name() << endl;
cout << "E-mail: " << person.email() << endl;
```

Listing 2.10: Manipulating and serialising a .proto-generated class

Many RPC implementations which use protocol buffers exist. Although Java, C++, and Python are the languages that are officially supported, people have created open source implementations for other languages, including JavaScript[9].

[9] https://github.com/dcodeIO/ProtoBuf.js

Chapter 3

Project & Evaluation Plan

This section will not be included in the final report. It includes what I think the key milestones of this project are, and how I should approach implementing them. Because most of this project is new stuff to me, I spent a long time researching it. Writing the background section helped me to understand how I can approach the project, but I do not know how long it will take me to do each part.

3.1 Project Key milestones

- Implement `NaClRPCGen`. This is the main deliverable of the project. It will generate JavaScript and C++ header files using an input IDL file. These will be the stubs. This will consist of:
  - Finding a suitable WebIDL parser, or making my own.
  - Use the parser to create NaCl C++ bindings, and create JavaScript stub headers.
  - Initially, use a simple message format. Then if I have time, use a more efficient format like protobuf.
- Implement the 'RPCRuntime' in both JavaScript and Native Client. This will be implemented on top of the message passing framework that already exists.
- Write an application that uses these tools.
- Complete the writeup to show how my RPC framework works, including the architecture and tools used. Give justification for each tool used, noting other alternatives I could have used.

3.2 Implementation status & plan

At the time of writing, none of these things have been implemented, as I spent time to research ideas and think of this plan. I plan to do a simple implementation this term that will include a parser and stub generator for very basic JavaScript and C++ types, e.g. just numbers. This exercise should help me understand how difficult it would be to do it for all types, and it would probably expose some areas I should think about in my architecture and implementation. I could come up with an incremental approach to implement the full product. This would ensure that even if the project is not completed by the deadline, I will at least have it working for *some* types of programs.

3.3 Evaluation Plan

I think evaluation my project will include two types of evaluation, one quantitative and one qualitative.

The quantitative part includes measuring how much overhead the RPC framework adds to the simple message passing approach. I will need to measure this for different types of applications: e.g. applications which need to continuously call RPC functions might behave differently to applications which call them every once in a while.

The qualitative evaluation includes seeing how much development time is saved when using RPC. This could be measured by the number of lines the developer needs to write to achieve the same thing with message passing and with RPC.

Bibliography

[1] Bennet Yee, David Sehr, Gregory Dardyk, J Bradley Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar. Native client: A sandbox for portable, untrusted x86 native code. In *Security and Privacy, 2009 30th IEEE Symposium on*, pages 79–93. IEEE, 2009.

[2] Richard M Stallman and Others. *Using the GNU Compiler Collection for GCC 4.8.2.* Free Software Foundation, 2013. http://gcc.gnu.org/onlinedocs/gcc-4.8.2/gcc/.

[3] Ian Hickson. *Workers.* World Wide Web Consortium, May 2012. http://www.w3.org/TR/2012/CR-workers-20120501/.

[4] Google Developers. *Native Client Technical Overview.* Google Developers, November 2013. https://developers.google.com/native-client/overview.

[5] W3C. *HTML5 Specification.* World Wide Web Consortium, August 2013. http://www.w3.org/html/wg/drafts/html/master/infrastructure.html.

[6] David Herman and Kenneth Russell. *Typed Array Specification.* Khronos, 2013. https://www.khronos.org/registry/typedarray/specs/latest/.

[7] Andrew D Birrell and Bruce Jay Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems (TOCS)*, 2(1):39–59, 1984.

[8] W Richard Stevens. *UNIX network programming*, volume 1. Addison-Wesley Professional, 2004.

[9] ISO/IEC 19500-1. *Common Object Request Broker Architecture (CORBA), Interfaces.* International Organization for Standardization, 2012.

[10] Steve Vinoski. Corba: Integrating diverse applications within distributed heterogeneous environments. *Communications Magazine, IEEE*, 35(2):46–55, 1997.

[11] JSON-RPC Working Group. *JSON-RPC 2.0 Specification*, 2010. http://www.jsonrpc.org/specification.

[12] Dave Winer. *XML-RPC Specification.* UserLand Software, 1999. http://xmlrpc.scripting.com/spec.html.

[13] Cameron McCormack. *Web IDL.* World Wide Web Consortium, 2012. http://www.w3.org/TR/WebIDL/.