Imperial College London

Individual Project Report

Native Calls

JavaScript - Native Client Remote Procedure Calls

Author:
Mohamed Eltuhamy

Supervisor:
Dr. Cristian Cadar

Co-Supervisor:
Petr Hosek

Department of Computing

June 2014

Abstract

Google Native Client provides a safe, portable way of embedding native plugins in the web browser. Native Client allows communication between the web page's JavaScript code and the Native Client module's C/C++ code. However, communication is through simple message passing. The project provides a remote procedure call (RPC) framework to allow calling C/C++ functions directly from JavaScript. A layered approach was taken to provide a straight forward RPC architecture.

A generator that uses a standardised Web Interface Definition Language (WebIDL) was implemented, and is used to produce JavaScript and C++ stubs that handle parameter marshalling.

The project is evaluated in terms of performance and amount of development effort saved. We found that the framework's performance impact was negligible for small amounts of data (up to 250 objects) being sent and received, however the impact started to increase linearly with the data size. We found that the generator saved a considerable amount of developer effort, saving 187 lines of code for a bullet physics simulation application, compared to previous implementation methods.

Acknowledgements

I would like to thank my supervisor, Dr. Cristian Cadar, for his advice, suggestions, and feedback regarding the project and report.

I would also like to thank my co-supervisor, Petr Hosek, for his ongoing enthusiasm, ideas, and engaging discussions about the project throughout the year.

Finally, I would like to thank my beloved family and friends for their ongoing support throughout my degree.

And say, "My Lord, increase me in knowledge."

Quran 20:114

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++. 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 potentially un-secure instructions or 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

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

(for example, to play audio) is through the web 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 sending and receiving messages from a Native Client module

```
// 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 showing the use of PostMessage and HandleMessage

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].

[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

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 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 traditional frameworks, such as ONC RPC (page 9) or CORBA (page 11).

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.

they allow concurrency, they are relatively heavyweight 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.)

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 languages, 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 as easy as writing normal C++ programs with the difference between them being that the sandboxes disallow unwanted side-effects and system calls. Since many applications might want to have this type of functionality, 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 independent 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 Portable Native Client

When Native Client was first released in 2011, it allowed operating system independent binary to run in a web application. However, it produced architecture-specific

applications using the same source code. These were called nexe modules. For example, it produced x86 64 bit as well as i386 binaries. However, for the developer, distributing different binaries for the same application was tedious, and architecture specific distributions go against the general trend of the truly independent web platform.

PNaCl was later introduced to solve the problem of lack of portability. Instead of producing architecture specific nexe executables, portable pexe modules are produced instead. These have a verified bitcode format. The PNaCl runtime, which runs as part of the browser, translates the bitcode into machine code. Because of their cross-platform nature, PNaCl (pexe) modules are allowed to run in Google Chrome without the user installing them, while NaCl (nexe) modules must be installed through the Chrome Web Store. However, NaCl modules allow inline assembly and different C standard library implementations, while PNaCl modules only support the newlib implementation and don't support architecture specific instructions.

2.1.2 NaCl Modules and the Pepper API

A Native Client application consists of the following [4]:

HTML/JavaScript Application: 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.

2.1.3 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 (`Number`, `String`, `Boolean`, `null`) as well as dictionaries (key-value `Object` types), 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.

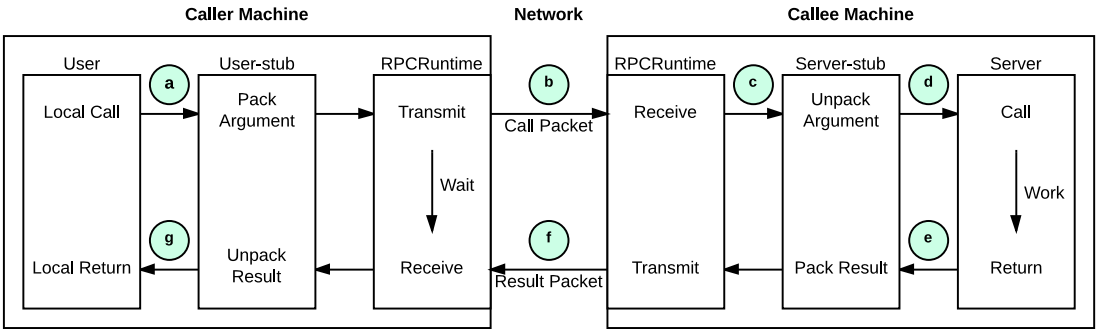


Figure 2.1: The basic components of an RPC framework, adapted from [7]

2.2 Remote Procedure Calls (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/can be automatically generated using interface definition files.
2. When the remote call is made, it calls the user stub **(a)** which packs the parameters and function call information into a packet.
3. The packet gets transferred **(b)** to its destination computer or process (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 (2.2.1), 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 callee's RPCRuntime. It is then passed on to the server stub **(c)**.
5. The arguments and function call information are unpacked and a normal function call **(d)** is made to the actual procedure.
6. When the procedure returns, the result is passed back to the server stub **(e)** where it is packed and transmitted back to the caller **(f)**, which unpacks it and uses the result **(g)**.

2.2.1 The Role of the RPC Runtime

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 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 (could be as a string or a pre-agreed ID between the caller and the callee) of the procedure the caller is calling.
3. The actual 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 function returns, the runtime sends 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 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, etc.

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.

```c
#include "keypairgen.h"

key_pair_t *
generate_keypair_0_svc(char **argp, struct svc_req *rqstp)
{
  static key_pair_t  result;
  // ... 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.

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. One of the main issues it addresses is the use and implementation of RPC on remote objects (instances of classes) to allow remote 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, by calling one of the IDL stubs or going through the dynamic invocation interface. 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 from the client as shown in Figure 2.2.

The ORB core allows the client to create and retrieve references to remote objects via the ORB interface. A client can create an object to get its reference, which is done by RPC calls to factory objects, which return the reference back to the client

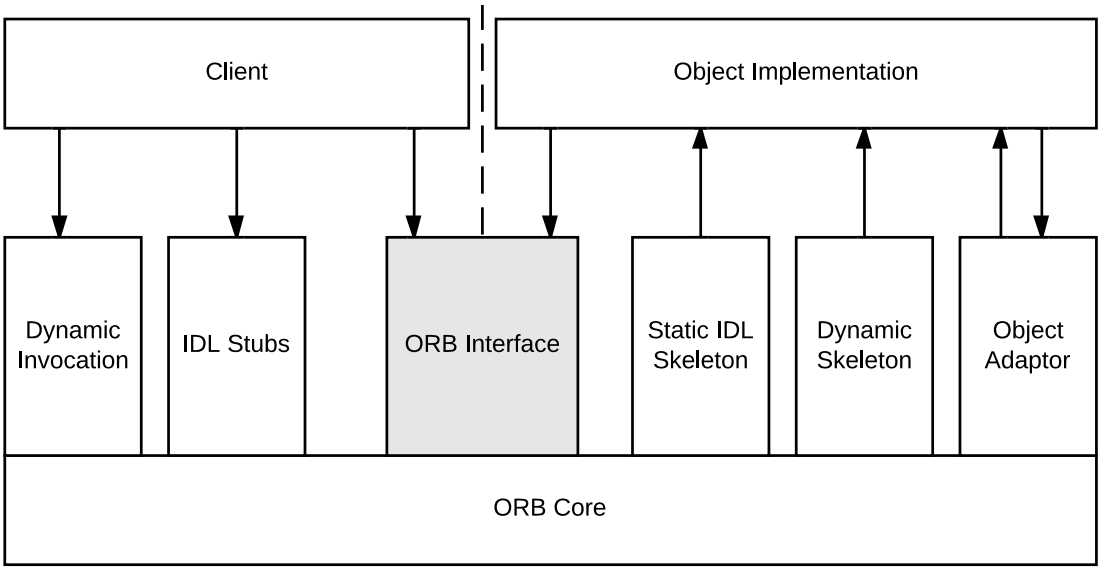


Figure 2.2: Interface and Implementation Repositories in CORBA, from [9]

after creating an instance. When the client has a reference, it can then use it to perform remote method invocations by dynamic invocations or through the IDL stubs. The client can also get information about the object by using its reference. This is done via the ORB's directory services, which maps object references to information about the object - including its fields, names and other properties. [10]

2.3.3 JSON-RPC and XML-RPC

XML-RPC is a simple RPC protocol which uses Extended Mark-up Language (XML) 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.

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

```
<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

XML-RPC supports simple types like integer, double, string, and 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.

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

XML-RPC and JSON-RPC are very similar. JavaScript Object Notation (JSON) is a simple and light-weight human-readable message format. Its advantage 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.

2.3.4 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 [14].

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 [15].

There are also open source C++ bindings for WebIDL, such as esidl[3]. Similarly, bindings for JavaScript also exist [17].

[3] 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 GitHub [16]

2.4 Data Representation and Transfer

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 server 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, 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 are 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 [18]. They are used extensively within many Google products, including AppEngine[4].

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

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.

```
// 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, developers have created open source implementations for other languages, including JavaScript [19].

2.5 Parsing and Generating Code

This section gives a brief overview of what parsers are and how they work.

In a nutshell, a parser takes a file containing some text written in some syntax, and produces an abstract syntax tree. An abstract syntax tree is simply a more structured representation of the text, and is constructed using the rules of the syntax and the semantics of the language.

We can use the abstract syntax tree to produce code in another language. For example, a compiler uses it to produce machine code. A *transpiler* uses it to produce source code in another language.

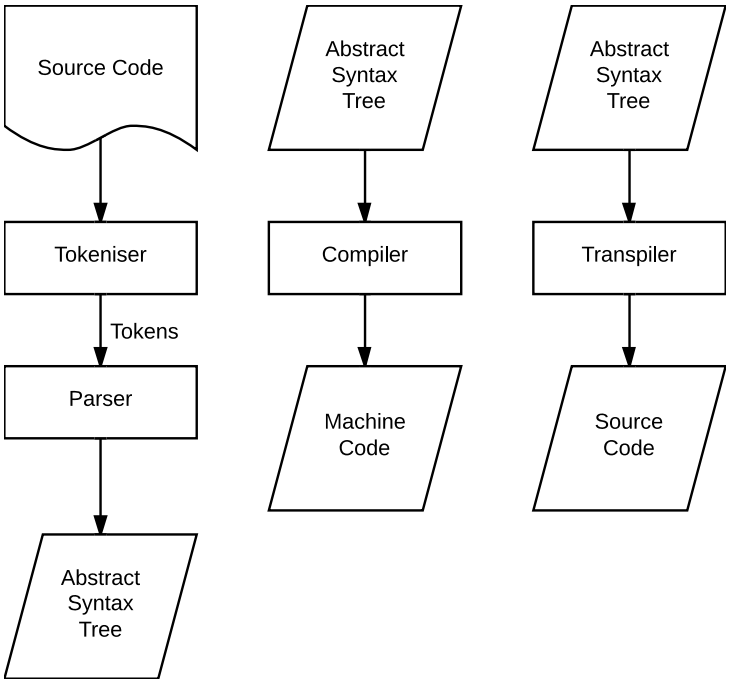


Figure 2.3: Flow charts showing the role of a parser, compiler and transpiler.

2.6 Native Client Development

Native Client applications are designed to be cross-platform. To provide cross-platform binaries, the Native Client SDK contains different tool chains - which include different compilers, linkers, assemblers, and other tools to build the application.

2.6.1 Toolchains

The tool chains provided by the SDK are:

- pnacl: This allows compiling C/C++ code into pnacl bitcode, as described in 2.1.1 (page 4). The pnacl toolchain uses the llvm compiler project, and the newlib and libc++ standard library implementations.
- nacl-gcc: This allows compiling C/C++ code into verified machine code. NaCl modules can use the newlib or glibc standard C library implementations and libc++ or libstdc++ C++ standard library implementations.

2.6.1.1 Simplified Building using `make`

The SDK also includes a Makefile called `common.mk` which is used by the included examples and demos to simplify the build process. This makes it easy to write an application for any of the toolchains, without worrying about compiler locations, include file paths, etc.

To specify the compiler, all that needs to be done is to specify the `TOOLCHAIN` environment variable. For example, running `make TOOLCHAIN=newlib` selects the nacl-gcc toolchain and newlib C standard library implementation.

There is also a `CONFIG` environment variable that can be set to specify the compiler's optimisation level. This can be `Release` or `Debug`.

To illustrate the usage of `common.mk`, Listing 2.11 shows the Makefile from the SDK's getting started example.

```
include $(NACL_SDK_ROOT)/tools/common.mk
TARGET = part2
LIBS = ppapi_cpp ppapi pthread

CFLAGS = -Wall
SOURCES = hello_tutorial.cc

# Build rules generated by macros from common.mk:

$(foreach src,$(SOURCES),$(eval $(call COMPILE_RULE,$(src),$(CFLAGS))))

ifeq ($(CONFIG),Release)
$(eval $(call LINK_RULE,$(TARGET)_unstripped,$(SOURCES),$(LIBS),$(DEPS)))
$(eval $(call STRIP_RULE,$(TARGET),$(TARGET)_unstripped))
else
$(eval $(call LINK_RULE,$(TARGET),$(SOURCES),$(LIBS),$(DEPS)))
endif

$(eval $(call NMF_RULE,$(TARGET),))
```

Listing 2.11: Using the common.mk makefile, as seen in the getting started example

Now, running `make TOOLCHAIN=newlib CONFIG=Debug` will compile and build the same sources for the newlib toolchain and Debug config. Running `make TOOLCHAIN=all` compiles and builds the same sources for pnacl, newlib and glibc.

2.6.1.2 Creating libraries

In C/C++, libraries can be created using the `ar` and `ranlib` tools. This creates a `.a` file which can be later linked with another program. Using the Native Client SDK, this is also supported, but the locations of these tools will depend on the tool chain used.

Thankfully, the common.mk file also provides Makefile macros that make this easy. Using the LIB macro will create the static libraries and also install the libraries into the relevant SDK location. This means it is easy to link in a library for a specific tool chain all using the `TOOLCHAIN` and `CONFIG` variables.

As for dynamic libraries (`.so` files), these are only supported using the glibc tool chain, but `common.mk` takes care of this for us too, by installing the `.so` file if the tool chain is glibc.

2.6.2 Porting existing libraries

Many open source C and C++ libraries have been ported to use the Native Client SDK and toolchains. The Native Client team have made a simple platform called naclports to simplify the porting of existing libraries.

Most of the time, porting an existing library is straight forward, as libraries often have generated makefiles, for example using a `./configure` script, which allows specifying the required compilers, linkers, etc. needed for building the library. naclports automatically fills in these details.

Any changes to default build process can be overridden in a script. Any changes to the code of the library can be specified in a patch file, which is applied before the library is built from the original sources.

For example, creating a NaCl port for libpng, an image processing library, is as simple as creating the `pkg_info` file shown in Listing 2.12 in naclports. naclports will then automatically download and install the library to the NaCl SDK. Afterwards, the libpng library used normally from any other C/C++ program.

```
NAME=libpng
VERSION=1.6.8
URL=http://download.sf.net/libpng/libpng-1.6.8.tar.gz
LICENSE=CUSTOM:LICENSE
DEPENDS=(zlib)
SHA1=a6d0be6facada6b4f26c24ffb23eaa2da8df9bd9
```

Listing 2.12: The libpng naclport `pkg_info` file

2.6.3 Using the Pepper Plugin API (PPAPI)

To interface with JavaScript, Native Client provides a C and C++ library that allows developers to easily control the browser. One important class that is used in all Native Client modules is the `pp::Instance` class, which is initialised when the embed tag is loaded in the HTML page. The class also has the `HandleMessage` and `PostMessage` functions, which implement message passing between the JavaScript and the C++ module. In JavaScript, JavaScript primitive types as well as reference types can be sent using postMessage. In C++, they are sent and received as `pp::Var` objects. Figure 2.4 shows how the `pp::Var` classes can be used.

Notice how C++ types can be extracted using the `pp::Var` class. For example, we can extract a C++ `double` from the `pp::Var` using the `pp::Var::AsDouble()` method. Arrays

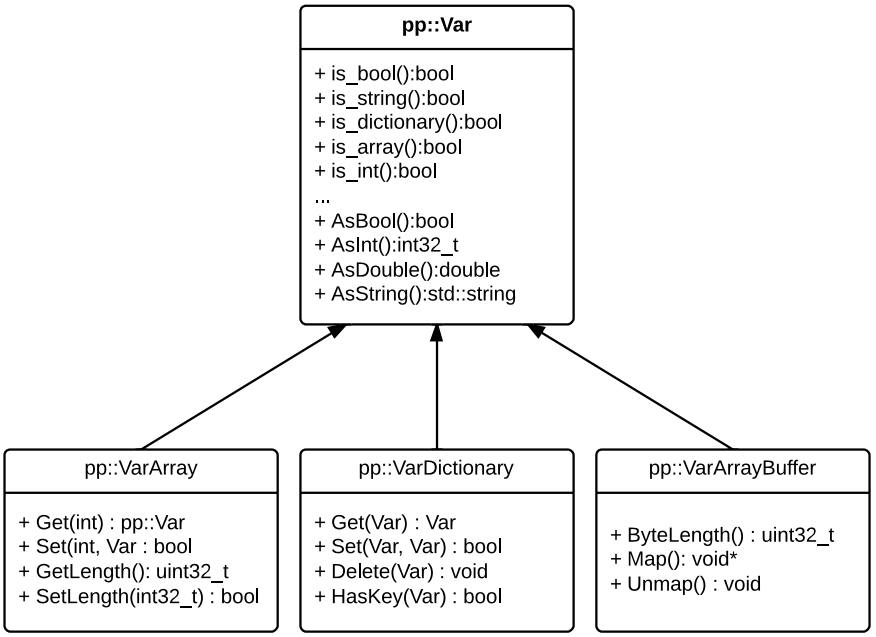


Figure 2.4: A simplified class diagram showing the `pp::Var` API

are sent and received as `pp::VarArray` objects. JavaScript objects (dictionaries) are sent and received as `pp::VarDictionary` objects. Binary data can be sent and received from and to the browser using the `pp::VarArrayBuffer` class. The `pp::VarArrayBuffer::map()` method returns a pointer to the sent binary data.

2.7 JavaScript Development

In this section, we show some common JavaScript patterns and code organisation techniques that are used throughout development.

2.7.1 JavaScript Modules

One way to achieve information hiding (like private data variables) is through the use of the *module pattern*. Essentially, everything is wrapped in a function which is immediately invoked. Listing 2.13 shows an example of this. Notice how the `private` variable can't be accessed from outside. We can use this pattern to define encapsulated classes as in Listing 2.14.

```
var SingletonObject = (function(){
  var private = 123;
  //... other private variables here
  return {
    foo: function(){
      console.log(private);
    }
    //... other exported public properties/methods here
  }
})();

SingletonObject.foo(); // output: 123
```

Listing 2.13: An example of using the module pattern

```
var MyClass = (function(){
  var private = 23;
  return function(){
    return {
      public: 12,
      privatePlusPublic: function(){
        return this.public + private;
      }
    }
  };
})();

var i = new MyClass();
console.log(i instanceof MyClass); // true
console.log(i.privatePlusPublic()); // 35
```

Listing 2.14: JavaScript 'classes' using the module pattern

There are a few of ways to organise modules into different files. We discuss the two most popular schemes, AMD and CommonJS.

2.7.2 Asynchronous Module Definition (AMD)

In AMD, modules are defined by specifying the dependencies and returning a single export from a factory function. The export could be a constant, a function, or any JavaScript object. Listing 2.15 defines a module that returns MyClass (which we saw in Listing 2.14), as well as another module which depends on MyClass.

```
// MyClass.js
define('MyClass', [], function(){
  var private = 23;
  return function(){
    return {
      public: 12,
      privatePlusPublic: function(){
        return this.public + private;
      }
    }
  };
});

// MainClass.js
define('MainClass', ['MyClass'], function(MyClass){
  var i = new MyClass();
  console.log(i.privatePlusPublic()); // 35
});
```

Listing 2.15: MyClass AMD Module

The advantage of using AMD is that we do not need to explicitly insert `<script />` tags in the HTML. It also makes it harder to set global variables, as they can only be set through the global `window` object, all variables declared are local only to the module. Since the files are asynchronously loaded, there is no need for a build process. Finally, this allows lazily loading scripts only when you need them. A popular implementation of AMD which works in the browser is RequireJS.

The disadvantage of using AMD is that most of the time, all dependencies are fetched anyway - so there's no need for them to be loaded asynchronously. Also, having many dependencies generates several HTTP requests, which can impact performance. Finally, writing the define function call at every file can often get tedious.

2.7.3 CommonJS Modules

CommonJS is another API that allows exporting modules from JavaScript files. Unlike AMD, it uses a straight forward, synchronous approach to module dependencies. Listing 2.16 shows the same MyClass module implemented using CommonJS.

```
// MyClass.js
var private = 23;
exports.MyClass = function(){
  return {
    public: 12,
    privatePlusPublic: function(){
      return this.public + private;
    }
  }
};

// MainClass.js
var MyClass = require("./MyClass.js").MyClass;
var i = new MyClass();
console.log(i.privatePlusPublic()); // 35
```

Listing 2.16: MyClass CommonJS Module

Notice how only the objects in the `exports` property are exported, so the `private` variable is still only accessible from the MyClass.js file.

The advantage of using CommonJS is that it has a much simpler, straight forward interface. The disadvantage is that it is only implemented natively on server-side projects. However, there is an open source library that allows CommonJS libraries to be used in the browser, called browserify. The tool 'builds' the module, by concatenating all the dependencies together with each module. In the end, one script is inserted into the HTML page. The advantage of this is that it is only one HTTP request to get all the JavaScript functionality. One issue is that now, instead of simply reloading the browser every time we need to test some JavaScript, we would need to build the JavaScript using browserify. Although this might have been an issue in the past, nowadays there exist good tooling that improve efficiency, for example, file watchers that build the JavaScript quickly every time a module is saved.

In the end, since the trade offs are comparable, we decide to take a AMD approach to modules for the browser JavaScript library since it is easier to test (i.e. no front end build steps). We take the CommonJS approach for the back end JavaScript generator, as it is the default module API in node.js.

Chapter 3

Related Work

3.1 Native Client Acceleration Modules (NaClAM)

In October 2012, John McCutchan at Google came up with the idea of using Native Client as a way to get native performance inside a normal JavaScript web application. He called it "Native Client Acceleration Modules (NaClAM)", with a slogan "*90% Web App. Native Performance Where You Need It*".

NaClAM is essentially a simple event-based RPC framework that allowed sending and receiving JavaScript objects as well as binary data. The RPC framework worked by using *event listeners* and *handlers* on both the JavaScript hand C++ ends.

On the JavaScript side, a library called NaClAM.js was provided, which allowed developers to attach listeners to a particular module using the `addEventListener(type,handler)` method. To send requests to the C++, the `dispatchEvent` method is used. On the C++ side, messages are handled inside one overridden method called `NaClAMModuleHandleMessage`. Here, checks are performed on the message received, and the appropriate method is called. Listing 3.1 shows an example of this.

```
void NaClAMModuleHandleMessage(const NaClAMMessage& message) {
  if (message.cmdString.compare("floatsum") == 0) {
    handleFloatSum(message);
  } else if (message.cmdString.compare("addfloatarrays") == 0) {
    handleAddFloats(message);
  } else {
    NaClAMPrintf("Got message I don't understand");
  }
}
```

Listing 3.1: NaClAM C++ message handler

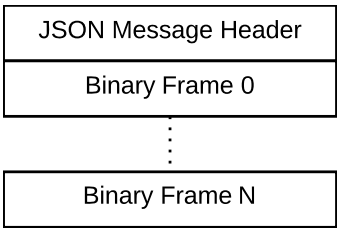


Figure 3.1: A NaClAM message including binary frames

3.1.1 Message Format

At the message passing level, NaClAM uses JavaScript/C++ strings to transport messages. These messages hold information about the message such as the command string (like "`floatsum`" in Listing 3.1). The strings are constructed using the jsoncpp library.

Crucially, however, they also tell the framework how many binary *frames* are expected to come after this message. Figure 3.1 shows an example of a message containing N frames. A frame is essentially a binary block of data sent by a separate call to `PostMessage`. The receiver collects all the frames before triggering the event handler.

3.1.2 Advantages and Disadvantages

NaClAM modules have the benefit of being simple and fast. There is a good distinction between the message information stored in the header and the message data stored as binary inside the frames. This allows developers to use the message header to implement event logic, while using the frames to transfer actual data. It also means that because the data is binary and almost no marshalling happens, the transfer speed is very fast, since binary data is *shared* between JavaScript and C++.

However, there are a few issues with using NaClAM modules. The first is a lack of overall, high-level structure. The developer has to be aware and understand exactly what the framework is doing behind the scenes to write their application, adding more burden on the developer especially since almost no documentation is provided. The second issue is how the message header types are implemented. Although the framework allows sending application data in binary frames, the header information is sent as JSON, and is manipulated by the jsoncpp library - so another library the developer needs to get used to. Importantly, this means the developer needs to unpack and pack the data they are sending in the header section by themselves by using the jsoncpp library. Another issue is how the framework does not use a callback

approach to asynchronous remote procedure calls. In other words, to 'return' a value from a C++ function back to JavaScript, a *different* event needs to be triggered from the C++, and handled by the JavaScript library. In other words, two different events need to be managed in both C++ and JavaScript for only one RPC call which returns data. If there are many functions like this, the developer needs to manage several different events, which is time consuming. Finally, although the framework has been demoed and gained a lot of popularity, it still seems to not be well tested, as no unit tests exist for either the C++ or JavaScript implementations have been written. This makes it feel like an experimental project, rather than a full, well supported framework.

Despite these issues, the Native Calls project was heavily influenced and inspired by the overall idea of the NaClAM project, especially its use cases and scenarios.

3.2 Node.js C++ Addons

Node.js is a JavaScript platform built on top of Chrome's V8 JavaScript engine that allows running JavaScript on server-side applications. Listing 3.2 shows an example of a node.js HTTP server.

```
var http = require('http');
http.createServer(function (req, res) {
  res.writeHead(200, {'Content-Type': 'text/plain'});
  res.end('Hello World\n');
}).listen(1337, '127.0.0.1');
console.log('Server running at http://127.0.0.1:1337/');
```

Listing 3.2: A simple node.js HTTP server

Although the full JavaScript implementation is available to use in Node.js, it is possible to extend node.js by implementing addons. Addons are implemented using C++, and therefore allow developers to use efficient C++ functionality inside node.js. In fact, the C++ API allows you to wrap a C++ object with a JavaScript one. Listing 3.3 shows an example of this.

```
class MyObject : public node::ObjectWrap {
 public:
  static void Init(v8::Handle<v8::Object> exports);

 private:
  explicit MyObject(double value = 0);
  ~MyObject();

  static v8::Handle<v8::Value> New(const v8::Arguments& args);
  static v8::Handle<v8::Value> PlusOne(const v8::Arguments& args);
  static v8::Persistent<v8::Function> constructor;
  double value_;
};
```

Listing 3.3: A node.js object wrapper

In the `Init` function, low level instructions that tell the JavaScript engine about the new object are given. In the `New` member function, an instance of the C++ object is 'wrapped' with the JavaScript object, using the node.js library. This means when we implement the `PlusOne` method, we can 'unwrap' the JavaScript object to get the C++ object instance, then perform the intended operation. Listing 3.4 shows how this works with the `PlusOne` method.

```
Handle<Value> MyObject::PlusOne(const Arguments& args) {
  HandleScope scope;

  MyObject* obj = ObjectWrap::Unwrap<MyObject>(args.This());
  obj->value_ += 1;

  return scope.Close(Number::New(obj->value_));
}
```

Listing 3.4: Implementing methods on wrapped objects

We can now create an instance of the object in JavaScript as though it was a native JavaScript object. Listing 3.5 shows an example of this.

```
var obj = new addon.MyObject(10);
console.log( obj.plusOne() ); // 11
console.log( obj.plusOne() ); // 12
```

Listing 3.5: Using the C++ object from JavaScript

3.2.1 Advantages and Disadvantages

The idea of simply extending JavaScript to use your own C++ methods is powerful. We saw how the class we created in C++ can be accessed directly from JavaScript in a native way. It also means we have full control over the data the function can accept, as the actual JavaScript object reference is passed into the C++. In other words, there is no parameter marshalling - everything is native.

The obvious issue we have with C++ addons to JavaScript is how can we use them in *browser* JavaScript? Well the answer is, we can't. However, we are able to set up a *local* node.js server which we can communicate with using the browser. This can be done over the websocket protocol, which allows full-duplex communication over a single TCP connection. However, now that messages need to be serialised, we would need to implement a RPC framework on top of web sockets.

3.2.2 Similar approaches in the browser

Although node.js addons do not actually solve our problem, the basic idea of them is that JavaScript is somehow *extended* to allow running C++ functionality. Conventional browser plugins such as NPAPI based plugins or ActiveX browser plugins have similar interfaces. Through the plugin framework, it is possible to directly access the DOM on the page where the plugin is embedded - a bit like how this is done in node.js, as described above.

Some frameworks such as FireBreath [20] have been created that allow cross platform plugins that support ActiveX, NPAPI, etc. A crucial difference for us, however, is that these plugin frameworks depend on direct access to the DOM of the page in the browser. When we remove this feature, these frameworks will not work. Native Client *only* allows access to the DOM through postMessage, and the data sent is *passed by value*, so the data is essentially copied across to the C++ module. What this means is that the RPC framework will need to handle all marshalling as well as transport of the messages between C++ and JavaScript.

3.3 Apache Thrift: Cross-language services

Apache Thrift is a framework that allows cross-language services development. Originally developed at Facebook, it was designed to provide reliable, efficient communication between languages and services. Many languages are supported,

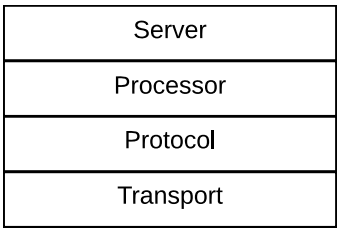


Figure 3.2: The layers of the Apache Thrift RPC stack

including C++, Java, and JavaScript. Thrift provides a cross-platform generator that can generate Thrift client and server pairs, where the client and server can be using different languages. Similar to other RPC frameworks, it uses its own IDL file format, Thrift IDL. The IDL file is used to generate code to support different languages.

Thrift's implementation is based around layers in the thrift stack (Figure 3.2). The transport layer is responsible for the transfer of messages. The protocol layer is an interface that defines how certain data structures should be mapped into a transferable format, such as JSON, XML, binary, etc. The process layer simply takes an input protocol, processes it using a handler, and writes to the output protocol. Finally the server sets up all the layers below it so that the system is functional as a whole.

In the following code listings, we give snippets showing how Apache Thrift is used to create a C++ server and JavaScript client. These were adapted from the original Thrift tutorial, which is available online [21].

```
enum Operation {
  ADD = 1,
  SUBTRACT = 2,
  MULTIPLY = 3,
  DIVIDE = 4
}

struct Work {
  1: i32 num1 = 0,
  2: i32 num2,
  3: Operation op,
  4: optional string comment,
}

service Calculator extends shared.SharedService {
   void ping(),
   i32 add(1:i32 num1, 2:i32 num2),
   i32 calculate(1:i32 logid, 2:Work w)
}
```

Listing 3.6: Thrift IDL File

```
class CalculatorHandler : public CalculatorIf {
```

```
 public:
  CalculatorHandler() {}

  int32_t add(const int32_t n1, const int32_t n2) {
    return n1 + n2;
  }

  int32_t calculate(const int32_t logid, const Work &work) {
    int32_t val;

    switch (work.op) {
    case Operation::ADD:
      val = work.num1 + work.num2;
      break;
    // ... other cases and implementation
    }
    return val;
  }

};

int main(int argc, char **argv) {
  // code here to set up processor, transport and protocol.

  TSimpleServer server(processor,
                       serverTransport,
                       transportFactory,
                       protocolFactory);


  server.serve();
  return 0;
}
```

Listing 3.7: A Thrift C++ Server

```
function calc() {
  var transport = new Thrift.Transport("/thrift/service/tutorial/");
  var protocol  = new Thrift.Protocol(transport);
  var client    = new CalculatorClient(protocol);

  var work = new Work()
  work.num1 = 1;
  work.num2 = 2;
  work.op = 1; //1==ADD

  var result = client.calculate(1, work);

  console.log(result); //3

}
```

Listing 3.8: JavaScript client for Thrift service

3.3.1 Advantages and Disadvantages

Apache Thrift is a large library which seems to be well supported and used by both industry giants and the open source community. Because the Thrift IDL file is used to generate both clients and servers in more than a dozen languages, it seems that it is generic enough to be used in any language context. Thrift's implementation seems to be well structured, showing a clear separation of concerns between each component.

However, there are a few issues with getting Thrift to work with Native Client. First, although it might be possible to implement a transport layer for thrift using postMessage in JavaScript, the C++ end (Native Client), writing a protocol for PPAPI might prove challenging. Moreover, the JSON protocol implementation uses strings instead of actual JSON objects, which will probably impact performance as it adds yet another marshalling step. For example, consider sending a JavaScript object from the web browser to a C++ function as a parameter. The marshalling will probably look something like this: we want to send a JavaScript Object, so Thrift for JavaScript will convert it into a JavaScript string in the protocol layer. When the JavaScript string is sent, it is marshalled by PPAPI as a `pp::Var`, which is then marshalled as a `std::string` using `pp::Var::AsString()`. Thrift for C++ will then take this string and de-marshal it in order to construct a C++ object. Finally, the C++ object is passed to the C++ concrete function. When we add on to this the fact that JavaScript strings are the slowest primitive types to transfer (according to our benchmarks in Section 5.3, on page 94), we can see that the performance impact might actually make a big difference in an application. To avoid this, several changes will need to be made on many of the layers discussed above, which might prove to be a challenging, non-trivial task.

Finally, although unofficial port for Apache Thrift has been made for Native Client [22], all of the communication code is still hand coded. The performance of using Thrift for Native Client is unclear, as there is no protocol implemented using PPAPI.

3.4 JSON-RPC Implementations

We briefly discussed the use of the JSON protocol RPC in background section 2.3.3 on page 12. Here, we discuss implementations of the protocol in JavaScript and C++.

3.4.1 JavaScript Implementations

In general, JavaScript implementations of the JSON RPC protocol are very simple, because JSON is consumed very naturally by the browser and JavaScript. Even if the JSON objects are sent and received as strings, implementations use the `JSON.stringify(JSObject)` to turn a JavaScript object into a string, and `JSON.parse(string)` to turn a string into a JavaScript object.

Although there are many implementations in JavaScript that are for several different domains and use cases, we discuss a simple one that uses `postMessage` for transferring the JSON messages between browser windows, called "PostMessage RPC (pmrpc)". pmrpc is an open source library available on GitHub [23] which aims to simplify cross-window communication by using postMessage. It shows the simplicity of remote procedure calls using JSON RPC. Listing 3.9 shows an example of using pmrpc to set up and call remote procedures between two windows.

```
// in the callee window, expose a procedure
pmrpc.register({
  publicProcedureName : "HelloPMRPC",
  procedure : function(printParam) { alert(printParam); }
});

// in the caller window, call the exposed procedure
pmrpc.call({
  destination : window.frames["ifr"],
  publicProcedureName : "HelloPMRPC",
  params : ["Hello World!"]
});
```

Listing 3.9: Using pmrpc to communicate between two windows

3.4.2 C++ Implementations

C++ does not have native support for JSON like JavaScript, but several open source libraries exist that can read and manipulate JSON strings. JsonCpp is one of the most popular JSON libraries for C++. We give a very brief overview of how it can be used to parse and write JSON objects. Listing 3.10 shows how JsonCpp is used.

```
Json::Value root;   // will contain the root value after parsing.
Json::Reader reader;
bool parsingSuccessful = reader.parse( config_doc, root );

if(!parsingSuccessful){ /* fails to parse */ }

// Get the value of the member of root named 'encoding',
// return 'UTF-8' if there is no such member.
std::string encoding = root.get("encoding", "UTF-8" ).asString();

// Get the value of the member of root named 'encoding',
// return a 'null' value if there is no such member.

const Json::Value plugins = root["plug-ins"];
// Iterates over the sequence elements.
for ( int index = 0; index < plugins.size(); ++index )
   loadPlugIn( plugins[index].asString() );

setIndentLength( root["indent"].get("length", 3).asInt() );
setIndentUseSpace( root["indent"].get("use_space", true).asBool() );

// don't need Json::Value constructor explicitly.
root["encoding"] = getCurrentEncoding();
root["indent"]["length"] = getCurrentIndentLength();
root["indent"]["use_space"] = getCurrentIndentUseSpace();

// jsoncpp to string
Json::StyledWriter writer;
std::string outputConfig = writer.write( root );
```

Listing 3.10: An example of using the JsonCpp library

Notice how the API for JsonCpp is very similar to `pp::Var` (see background section 2.6.3 on page 22). This is because `pp::Var` and `Json::Value` essentially do the same job - they are interfaces for JavaScript objects in C++. The crucial difference, however, is that `Json::Value` ends up being written to a string, while `pp::Var` objects are transferred as they are using PPAPI.

Now that we've looked at JsonCpp and how it is used, we look at an implementation of JSON RPC in C++ using JsonCpp called JsonRpc-Cpp [24]. Essentially, the framework will register methods using the `RpcMethod` class, which calls a function that accepts a `Json::Value` input and a `Json::Value` output passed by reference. Listing 3.11 shows an example of setting up a handler, adapted from the original documentation.

```
class MyClass
{
   public:
     void Init()
     {
       RpcMethod* method = new RpcMethod<MyClass>(
           *this, &MyClass::RemoteMethod, // sets up pointer to C++ method
           std::string("remote_method"),  // the json-rpc "method"
           std::string("Description"));   // optional description
       m_handler.AddMethod(method);
     }

     bool RemoteMethod(const Json::Value& msg, Json::Value& response)
     {
       // do stuff
     }

   private:
     Handler m_handler;
};
```

Listing 3.11: Using a JsonRpc-Cpp handler

3.4.3 Advantages and Disadvantages

Many JSON RPC implementations for several languages exist [25], including C++ and JavaScript as we have seen. The JSON-RPC protocol is very simple, and for most use cases, it is efficient and adequate. Although JSON RPC has a well defined protocol, it can be extended for specific implementations. Finally, JSON RPC does not need the messages to be sent and received as strings (although human-readability of the messages can be a bonus). We can use any other format to marshal and demarshal a JSON RPC message. Some of these formats are discussed in background section 2.4 on page 16.

JSON RPC JavaScript implementations are simple enough to be implemented and tested in JavaScript to work for any specific project. So although libraries in JavaScript exist, implementing it for JavaScript to work with a different project structure / architecture should be not too difficult. As for C++ implementations, many if not all use the JsonCpp project. None available are implemented for `pp::Var`. So to get JSON RPC to work with `pp::Var`, an implementation will need to be written from scratch to use `pp::Var`. This is also not too difficult, since as we've seen, `Json::Value` and `pp::Var` have similar APIs.

Chapter 4

Design and Implementation

This section describes the design and implementation of the RPC framework and generators.

4.1 RPC Framework

The structure of the RPC framework is based around the notion of layers. Each layer solves a particular task, in order to achieve the goal of getting from a JavaScript stub to a C++ function, and back. Figure 4.1 shows the overall structure and interactions of each layer.

The advantages of this approach is that each layer is independent of the other. For example, if we choose a different RPC schema (i.e. something other than JSON RPC), we could easily replace the JSON RPC layer. Or, if we choose to have the C++ function on the server instead of as a Native Client module, we can easily change the transport layer to use AJAX requests or Web Sockets.

The other advantage to this approach is that because the layers are independent and each layer has a simple interface, each layer can easily be tested. For example, to test the implementation of the run time layer, we can easily mock the JSON RPC layer, since we know its public interface.

In the end, we have four layers: the stub layer, runtime layer, JSON RPC layer and transport layer.

- Stub layer: This is responsible for parameter and result (de-)marshalling. Eventually, it calls methods in the runtime layer.

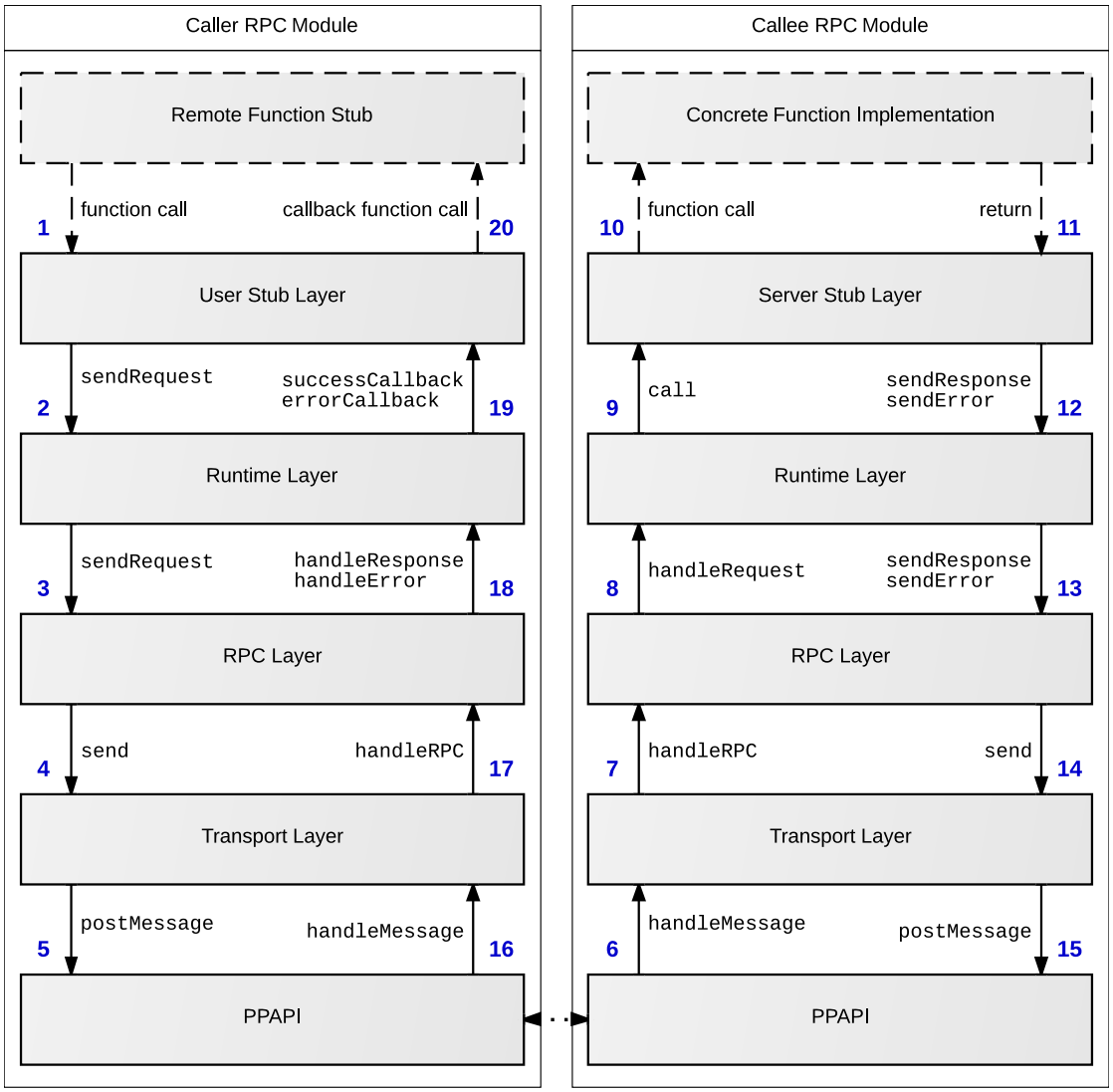


Figure 4.1: A layered approach to RPC. Numbering shows message flow.

- Runtime layer: This manages RPC requests and responses, matching responses with requests and calling the correct callbacks.
- RPC Layer: This implements an RPC protocol such as JSON RPC.
- Transport layer: This allows messages to be sent and received between the JavaScript and C++ runtimes.

Each layer is described in detail below.

4.1.1 Transport layer

The role of the transport layer is to implement the transportation of messages. Messages could be anything, JavaScript objects, strings, or even binary data. Moreover,

the receiver could be anything - a node.js server, or a Native Client module. Finally, the transport could use any transport mechanism - web sockets, HTTP/AJAX, WebRTC, postMessage, etc.

The important thing is that the transport must provide:

- An asynchronous API (should be non-blocking)
- The following public API:
  - a `send` function, that accepts a payload of any type.
  - a constructor which sets a message handler.
  - a message handler that must be invoked when a message is received.

The reason this approach was taken was to allow any possibility of executing remote procedure calls. It also allows the transport layer to be testable, since no concrete implementations of the layers above or below the transport layer need to be provided to test the functionality of the transport layer.

4.1.1.1 Implementing the Transport Layer in JavaScript

To implement the transport layer using a Native Client module, we first encapsulate the details of a Native Client module into its own class, called `NaClModule`. This class essentially does all the DOM manipulation for a module. To explain this, consider how a Native Client module is normally embedded in the page (as described in the background section 2.1.2 on page 5). The module is embedded onto the page using an `embed` tag. The `src` attribute points to the location of the NaCl Manifest - which tells the browser which (p)nacl executable to load. For example, `<embed src="myModule.nmf" type="application/x-nacl" />`. The `type` attribute tells the browser what MIME type the executable is. This could take values of either `x-nacl` for NaCl modules or `x-pnacl` for PNaCl modules.

All this detail is configured through the NaClModule constructor, which takes in an object for configuration. In other words, the same `embed` tag is created (but not actually placed on the page *yet*), using the following code:

```
var myModule = new NaClModule({
  src: 'rpc-module.nmf',
  name: 'rpc',
  id: "MyModule",
  type: 'application/x-pnacl'
});
```

Listing 4.1: Constructing a NaClModule Object

However, much of the details of this can be 'inferred' using the name of the module:

```
// creates an embed tag with the same attributes
var myModule = new NaClModule({"name": "MyModule"});
```

Listing 4.2: Construct a NaClModule using attribute inference

The attributes are inferred by using the NaClConfig global object, or a default config object if one does not exist. The id attribute is the same as the name attribute. The name and the type are inferred using the config object. Listing 4.3 shows an example of a config object and the corresponding embed attributes.

```
// the NaClConfig object is a global object.
// If required keys aren't found here, they are looked up in a
// default config object, defined inside the framework.
window.NaClConfig = {
  TOOLCHAIN: "pnacl",
  CONFIG: "Debug"
};
var myModule = new NaClModule({"name": "MyModule"});

/* produced embed tag:
<embed name="MyModule"                    //using name in constructor
       src="./pnacl/Debug/MyModule.nmf"   //using config and name
       id="MyModule"                      //using name
       type="application/x-pnacl" />      //using config
*/
```

Listing 4.3: Setting a NaClConfig object

Once a NaClModule is constructed, it can be loaded using the `load` method, which can take in an optional callback function as a parameter. The load method essentially inserts the `embed` element into the page. Event handlers can be registered with the module by using the `on` method. For example:

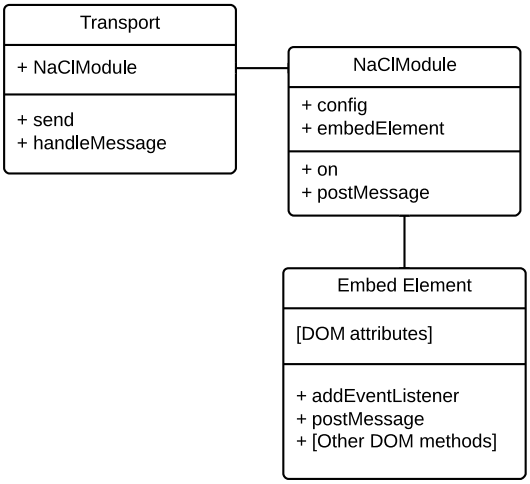


Figure 4.2: Class diagram showing the encapsulation of NaClModule and Embed element inside a Transport object in JavaScript

```
var myModule = new NaClModule({"name": "MyModule"});
myModule.on('load', function(){...});
myModule.on('message', function(){...});
myModule.on('crash', function(){...});
...
```

Listing 4.4: Registering different event handlers to a module

The NaClModule class therefore makes it easy to create and alter the HTML embed tag using only JavaScript.

Now, the transport layer encapsulate a NaClModule object, which is used as a low-level communication object in order to send and receive messages.

4.1.1.2 Implementing the Transport Layer in C++

Since the C++ module is a singleton, the structure of the transport in C++ is a lot simpler. Essentially, the transport is a class that extends the `pp::Instance` class provided by the PPAPI, which we use to send and receive messages using `pp::Instance::PostMessage` and `pp::Instance::HandleMessage`. These methods are overridden in order to link the transport with the layer above.

4.1.2 RPC layer

The RPC Layer is responsible for validating messages sent and received by the transport.

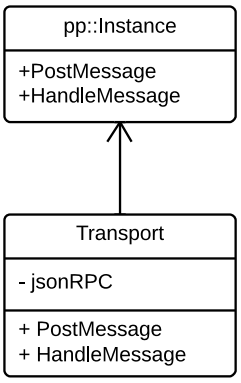


Figure 4.3: Class diagram showing inheritance of `pp::Instance` in C++

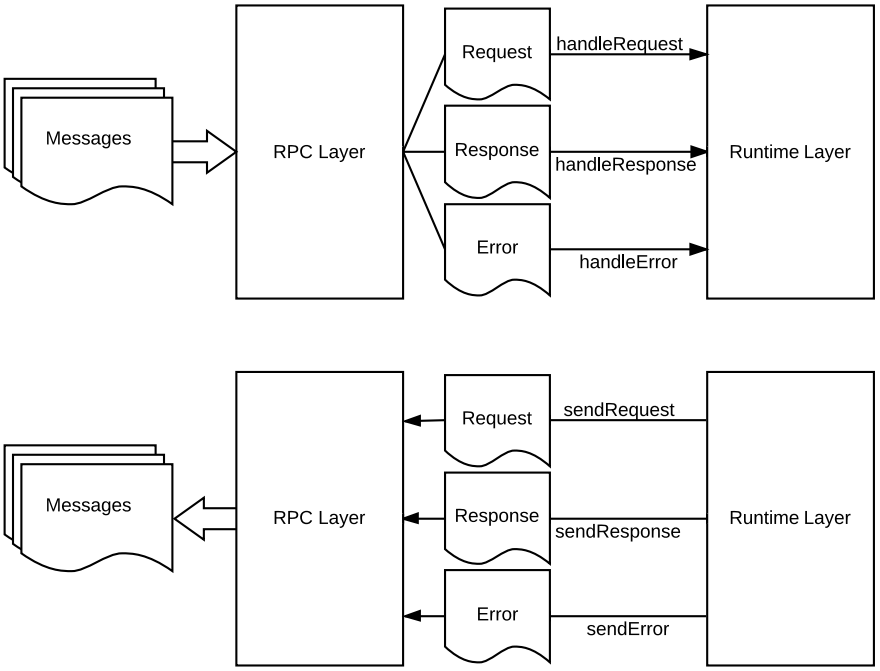


Figure 4.4: Flow diagram showing how messages are filtered by the RPC layer and forwarded to the runtime layer

Messages *received* by the transport could either be RPC *requests*, *responses*, or *errors*. If a message is one of these three, it should forward the message to the layer above (the *runtime*). An illustration of this is shown in Figure 4.4. If a message is not one of these three possibilities, the message should be ignored as it is not a RPC message.

The RPC layer can also provide RPC *send* functions, that allow messages to be sent to the layer below. It allows RPC requests, responses and errors to be sent.

Therefore, the RPC layer has the following API:

- a `handleMessage` function, which accepts a payload and is called by the Transport layer when a message is received. `handleMessage` should filter through the messages received to categorise them as requests, responses or errors. Depending on which type of message it is, the layer can call different methods of the layer above.
- a `sendRequest` function, which validates messages to be sent as requests, and forwards it down to the transport layer to be sent.
- a `sendResponse` function, which validates messages to be sent as responses, and forwards it down to the transport layer to be sent.
- a `sendError` function, which validates messages to be sent as errors, and forwards it down to the transport layer to be sent.

4.1.2.1 Choosing a protocol

To implement the RPC layer, we needed to choose a RPC protocol. We decided to go for the simplest protocol, which is JSON RPC. We discussed the advantages of using JSON RPC in the related work section, but there are also some benefits to do with using PPAPI to implement marshalling. Since we are using `pp::Var`, more or less all of the JavaScript types are marshalled for us using PPAPI when we use postMessage. One alternative would have been to implement a binary based protocol, like the Ice protocol, which uses Google Protocol Buffers. There are a couple of reasons why we did not take this approach. The first is that there is an extra marshalling step in both JavaScript and C++ as we are sending and receiving binary. This adds to the complexity of both ends. The other issue is that Protocol Buffers for JavaScript aren't officially supported by Google but were instead implemented by a member of the open source community on GitHub. Finally, it is unclear whether a performance gain or loss will be achieved when Protocol Buffers are used. If time permitted, it would have been an interesting experiment to test different protocol implementations and measure their performance characteristics. But for now, we decided to go for the simplest approach of using JSON RPC via PPAPI.

4.1.2.2 Implementing the JSON RPC Layer

To implement the API discussed above for JSON RPC, we first need to implement validators for messages. These determine what kind of message it is - request, response, error, or none. This is done in both the JavaScript and C++ implementations,

as both will have to use the protocol. Then, when a message is received, the layer simply uses these validators to check what kind of message it is. At the same time, it extracts the relevant information about the message. For example, if it is a request, it extracts the method name, parameters, etc.

To implement the JSON RPC protocol validators, we adhere closely to the specification. Detailed unit tests are written, including passing and failing cases. The validator is then written. This is done for both the C++ and JavaScript implementations.

As well as this, some helper functions were written to create valid RPC requests, responses, and errors.

On the C++ side, a RPCRequest object is created. The RPCRequest object encapsulates generic (i.e. not necessarily JSON RPC specific) information about a RPC call. This is passed by reference to the layers that need it, so that the validation and extraction only happens once.

4.1.3 RPC Runtime layer

The main job of the runtime layer is to coordinate RPC requests and responses. As described in the background section 2.2.1 (page 8), the runtime does this by keeping track of RPC requests, and matching the requests with the responses by the use of a call identifier.

The API the runtime provides is therefore as follows:

- send functions, that call the layer below.
  - `sendRequest = function(method, parameters, successCB, errorCB)` will give the request an ID, then keep track of that ID and the callback functions.
  - `sendResponse = function(id, result)` will construct a response message and send it to the layer below.
  - `sendError = function(id, errorCode, errorMessage, errorData)` will construct an error message and send it to the layer below.
- handler functions (`handleRequest`, `handleResponse`, `handleError`). The runtime will match the response's identifier with a previously sent request identifier. If a callback was provided, the callback will be called.

4.1.3.1 Implementing the runtime layer

The role of the runtime is different depending on the caller and the callee. Due to time constraints, the runtime has only been implemented for a JavaScript caller (client), and a C++ callee (server). However, the implementation is very similar, as it is only a language difference (in other words, the implementation in JavaScript will be the same as the implementation in C++ and vice versa).

We first consider the implementation of the caller's RPC runtime, implemented in JavaScript. To make a RPC request, the sendRequest function is called. The remote method name, parameters, success callback, and error callback are passed to this function. The runtime then constructs a RPC Request object, and gives the request a unique ID. The callbacks are added to a map of ids to callbacks. The listing below shows an example instance of a map, where two RPC requests have been made and are still waiting for responses.

```
{
  1: {
    success: function(){...},
    error: function(){...}
  },

  2: {
    success: function(){...},
    error: function(){...}
  },

  ...
}
```

Listing 4.5: An example of an id-callback map

Finally, the request object is sent. When the C++ server handles the request and sends back a response, the same ID is sent back. The JSON RPC Layer (described above) will handle the message and call one of the runtime's handlers. If it was a response, it will call the response handler. If it was an error, it will call the error handler. In either, the runtime will look for response's id inside the map. If the ID exists in the map, then the appropriate callback is called and then the key/value pair is removed from the map. Listing 4.6 shows how this is implemented for the case of a successful response.

```
RPCRuntime.prototype.handleCallback = function(rpcObject){
  // see if that id is even registered with us
  if(_.isUndefined(this.idCallbackMap[rpcObject.id])){
    logger.error("Received a callback response for invalid call");
    return false;
  }

  // get the success callback
  var successCallback = this.idCallbackMap[rpcObject.id].success;

  // call it
  if(_.isFunction(successCallback)){
    successCallback.call(null, rpcObject.result);
  }

  // delete it from the map
  this.idCallbackMap[rpcObject.id] = undefined;
  delete(this.idCallbackMap[rpcObject.id]);

  return true;
};
```

Listing 4.6: The handle callback method in JavaScript

For the callee's RPC runtime, written in C++, the implementation involves finding the required function and then calling it. Since functions aren't first class citizens in C++ (unlike in JavaScript), some thought needs to be put into how to dynamically call a function. The solution we took was to define stub classes for each RPC method. The stub classes are derived from the RPCServerStub class, which has an overrideable `virtual pp::Var call(const pp::VarArray* params, RPCError& error);` function. The `call` function takes in the parameters and returns the result as a `pp::Var`. In other words, all type marshalling and de-marshalling has to happen inside the `call` function.

When setting up the RPC Runtime object, we initialise it by adding all the functions we wish to expose. Listing 4.7 shows an example of adding RPC stubs to the runtime.

```
// define ServerStub_MyInterface_Foo...
// ...
// during set up...
pprpc::RPCRuntime* rpcRuntime = new pprpc::RPCRuntime(...);
rpcRuntime->AddServerStub("MyInterface::Foo", new ServerStub_MyInterface_Foo);
```

Listing 4.7: Adding RPC stubs to the runtime

The `AddServerStub` method works by simply maintaining a `std::map` of `std::string` method names to `RPCServerStub` objects. Notice how the stubs are 'flat' with respect to interfaces. What I mean here is how encapsulation is done using name mangling. This was done to simplify stubs, as in the end, interfaces are singletons which have a straightforward, static, API. In other words, since interfaces are stateless, we only represent the functions that an interface has. This greatly simplifies both the framework and code generation. However, implementing WebIDL interfaces in this way means we are limiting the implementation to only include functions (WebIDL operations), but WebIDL allows defining other interface members such as fields and constants. Since our main concern in this project was to use WebIDL to implement RPC functions we decided to stick to a simple approach of defining functions in C++, which is just through normal, namespaced function definitions in header files. The alternative, which could be a future extension, would be to implement interfaces as classes.

We hide the details of the ServerStub as this is discussed in the stub layer section (4.1.4, page 48).

Now that the stubs are registered with the runtime, when we receive a RPC request, the JSON RPC layer will call the runtime's `HandleRequest` method. This will look up the string name of the function being called in the map, in order to get the corresponding RPCServerStub. If it exists, the stub's `call` method is called.

4.1.4 Stub Layer

Finally, the stub layer is just a wrapper over the runtime layer's API, so that functions can be called 'natively' from within the language. The stub layer also performs parameter type checking and marshalling.

4.1.4.1 Implementing the stub layer

There is a distinction between the caller's stub layer (which we call the user stub) and the callee's stub layer (the server stub). The user stub has the job of parameter packing and calling the RPCRuntime's functions while the server stub is *called* by the RPC runtime, and *un*packs the parameter. This distinction can be seen in the RPC diagram shown in background section 2.2 on page 7.

Due to time constraints, RPC functionality has only been implemented one way, from JavaScript to C++. This means only the user stubs have been implemented on the

JavaScript side, and only the server stubs have been implemented on the C++ side. We give an overview of these implementations.

The user stub implementation using JavaScript is greatly simplified due to the fact that parameter packing only involves turning a function's parameters into an array, since the server stub is expecting `pp::Var` objects, which correspond naturally to JavaScript types. In other words, the PPAPI performs all the marshalling for us by converting JavaScript objects on the caller side into `pp::Var` objects on the callee side.

However, to make it more useful for the JavaScript developer, the RPC framework will optionally perform *type checking* on the caller side before calling the runtime. How JavaScript type checking is implemented is shown in the WebIDL bindings implementation section 4.2.6 on page 61, but the basic idea is the use of JSON Schemas which define the JavaScript types we are expecting.

What we want to achieve on the JavaScript side is the ability to access the defined modules, interfaces, and most importantly functions on the C++ side in the most natural way possible. We achieve this by dynamically constructing a JavaScript object that holds all this information. The object maps interface names to maps of function names to function stubs. To make this clearer, here is an example of such an object, which we shall call an RPC Module.

```
{
  "MyInterface": {
    "foo": function(){...},
    "bar": function(){...}
  },
  "SecondInterface": {
    "fun": function(){...}
  }
}
```

Listing 4.8: An example RPC Module in JavaScript

Now, to call `foo`, we call `MyInterface.foo()`. This gives a very natural way of using RPC functions in JavaScript, as that is exactly what they are - functions! We now finally look at how the actual stub function is implemented. This happens in JavaScript stub layer's `constructFunction` method. Essentially, it takes in a function definition as a JavaScript object, and returns a JavaScript function that is the stub. In that sense, the function definition is made *declarative*, a feature which makes the generator's job (discussed later) much simpler. Here is an example of a function definition:

```
{
  "name": "foo",
  "params": [{"$ref": "unsigned long"}],
  "returnType": {"$ref": "boolean"}
}
```

Listing 4.9: A function definition object

Then, the `constructFunction` method will take this object and use the information in it to produce a JavaScript function. The JavaScript function will do the type checking discussed earlier, but most importantly, it will call the RPC runtime to actually execute the RPC request. Here is a simplified example of a produced function:

```
function(){
  // get arguments dynamically
  var args = Array.prototype.slice.call(arguments, 0);

  // the expected number of parameters is known
  // if the user gives more params, then these are probably callbacks.

  // we extract the callbacks
  var userSuccessCallback, userErrorCallback, userParams;
  if (args.length > defnParamsLength) {
    userSuccessCallback = args[defnParamsLength];
    userErrorCallback = args[defnParamsLength + 1];
  }

  // extract the user given parameters
  // now we go through each parameter and assert it is valid type

  // before calling the runtime, we alter the callback
  var callback = function (d) {
    //type checking of the result
  };


  // finally call the runtime
  return runtime.sendRequest(
    interfacePrefix+defnName,
    userParams,
    callback,
    userErrorCallback
  );
}
```

Listing 4.10: A user stub function dynamically produced

To implement the C++ server stubs, we define a RPCServerStub class for each concrete function. For example, here is the `ServerStub_MyInterface_Foo` class used in Listing 4.7.

```
class ServerStub_StepScene : public RPCServerStub{
public:
        virtual pp::Var call(const pp::VarArray* params, RPCError& error){
                // unpack params
                // call concrete function
                // pack result, return
        }
};
```

Listing 4.11: Defining a server stub in C++

Notice how we have combined the role of the RPC server stub to include marshalling, de-marshalling and calling the actual function implementation. This was done to simplify the framework on the C++ side, and also to expose some of the implementation details to the C++ developer. This allows the C++ developer to tweak the framework in order to get higher performance if they need it, or to customise a specific case. There a few cases where this might be necessary. One case is when the developer wants to send and receive binary data but this is currently not supported, due to time constraints. In this case, the developer could go ahead and edit the generated code so that they can pass a `pp::Var` to their function. Another example where this might be useful is if the developer wants to make the callback asynchronous, by creating a thread, doing computation, and sending the result at a later time. Again, this is not supported at this stage, but the developer can simply edit the `call` function to allow this. In that sense, showing these implementation details gives power to the developer to allow them to do what they want, without forcing them to use a specific way of implementing their code.

Finally, in Listing 4.11, we purposely don't show how the parameter/result (de)marshalling happens. The implementation of this is shown in the WebIDL bindings implementation section 4.2.5 on page 58.

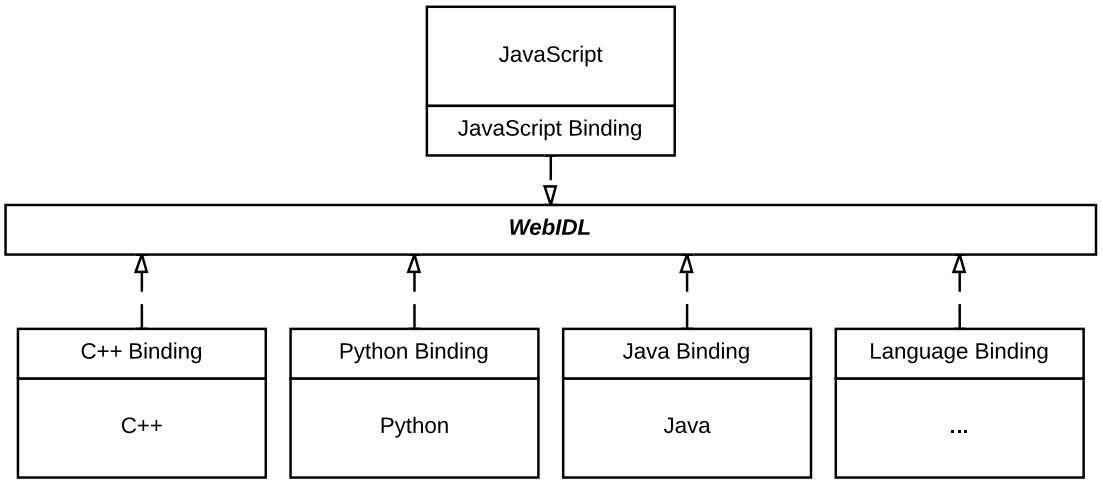


Figure 4.5: The WebIDL interface

4.2 WebIDL Bindings

In order to automatically generate stubs for JavaScript and C++ that allows communication between the two languages, an independent language, WebIDL, is used to define types and interfaces which will be used by both JavaScript and C++.

The reason why this is needed is because JavaScript and C++ have entirely different type systems, and since the communication is two-way, we cannot simply map a C++ type into a JavaScript type. Moreover, if the RPC framework were to be completely language independent, we would need a mapping between every language's type into a JavaScript type. Therefore, to generalise, WebIDL gives an intermediary type interface so that other languages can communicate with JavaScript. The WebIDL types and syntax is defined as a standard, and gives EcmaScript bindings. In other words, the conversion between WebIDL and JavaScript types is defined in the standard. It is then up to the developer of the other language to define a binding from that language to WebIDL. This is illustrated in Figure 4.5.

In this section, we mention the C++ WebIDL bindings used in the Native Calls project, and the design decisions behind them.

The implementation challenges involved in implementing these bindings are discussed at a later chapter.

4.2.1 Modules, Interfaces, and Functions

In Native Calls, we make a distinction between 'modules' and 'interfaces'. Essentially, a module contains several interfaces. And an interface contains several function

definitions.

When we define a module, we must define all the interfaces, type definitions, and dictionaries for it in the same generator call. The definitions could be in different IDL files.

In JavaScript, a module is represented as an object which has a property for each interface that module defines. Therefore, each interface has a property for each function that interface defines.

In C++, a module is represented as a class, which sets up the module. When setting up the module, each function interface is added. An IDL interface is represented by a C++ header file. The header file defines each function that is in the interface.

4.2.2 Number and String Types

WebIDL defines a number of numeric types and also provides the JavaScript bindings for each type. Table 4.1 shows the numeric types and their bindings in C++.

WebIDL Type	**Min int**	**Max int**	**C++ Type**
byte	-2^7	$2^7 - 1$	int8_t
octet	0	$2^8 - 1$	uint8_t
short	-2^{15}	$2^{15} - 1$	int16_t
unsigned short	0	$2^{16} - 1$	uint16_t
long	-2^{31}	$2^{31} - 1$	int32_t
unsigned long	0	$2^{32} - 1$	uint32_t
long long	-2^{63}	$2^{63} - 1$	int64_t
unsigned long long	0	$2^{64} - 1$	uint64_t
float			float
double			double

Table 4.1: The C++ WebIDL bindings for number types

It can be observed that the integer types are represented in C++ with the size information in it, even though C++ has equivalent type names for each of the WebIDL integer types. For example, C++ supports the `short` type, but we explicitly decide to represent `short` as `int16_t`. The reason why explicit size information is included in the type is because of different implementations of certain types. For example, depending on the C++ standard library implementation we use, a `long` can be represented in 32 bits or 64 bits. But because WebIDL explicitly defines the actual size of the integer types, to stick to the standard, we cannot tolerate this variation. For this reason, we use the explicit size types as shown above. This issue does not arise for `float` and `double` types as both C++ and JavaScript adhere to the IEEE 754 format.

Another interesting issue to note is that the bindings for large number types, such as `long long`, are represented in JavaScript by the *closest* numeric value. But because all JavaScript numbers are represented by 64 bit IEEE 754 ('double') types, the largest number that can be represented in JavaScript is actually $2^{53} - 1$. This means that often the conversion between the WebIDL type and the JavaScript binding is *lossy*, in the sense that it is not a one-to-one mapping. Although it would have been possible to overcome this issue by creating or using JavaScript 'BigNumber' library classes, I decided to adhere to the specification, using the lossy conversion. This was for a few reasons:

- Forcing the JavaScript user to use a number library is bad, as it adds more dependencies and is not conventional JavaScript e.g. the BigNumber library will have a different API to normal JavaScript numbers, and certain operations, such as addition, will not work properly.
- Using a different implementation, the RPC library could represent all data as *binary*. JavaScript supports binary data in the form of ArrayBuffers.
- It is fairly unlikely that the developer would want to send back such large numbers to the JavaScript, and since the developer is developing for the web platform, they should be aware of JavaScript's limitations - including numeric type support.

To represent string types, the `DOMString` WebIDL type is converted to the JavaScript `string` type, as defined in the standard. As for the C++ binding, the `std::string` class was chosen to represent DOMString. The alternative was to represent strings as character array buffers (`char[]`). I decided to use the `std::string` class for the following reasons:

- JavaScript uses unicode (utf8) for strings. The developer would need to do some encoding/decoding to handle unicode characters, which may not fit in a byte.
- Simplicity: The PPAPI supports an `AsString()` method on `pp::Var` objects, which extracts the string value as a `std::string` object.
- C++ developers use `std::string` when they can. `std::string` allows conversion to C strings using the `c_str()` method.

4.2.3 Dictionary Types

The WebIDL standard defines the binding of a WebIDL dictionary to be a JavaScript Object with the keys being the identifier names of each dictionary member, and values being of the member's type. For example, Listing 4.12 shows an example of a dictionary definition in WebIDL and the corresponding JavaScript object according to the specification.

```
// WebIDL
dictionary myObject {
  double id;
  DOMString name;
};

// Example JavaScript object
var myObj = {
  id: 31,
  name: "John Smith"
}
```

Listing 4.12: A WebIDL dictionary and its JavaScript binding

When a JavaScript object (and therefore a WebIDL dictionary) is sent to the NaCl module, it is represented in PPAPI as a `pp::VarDictionary` object. `pp::VarDictionary` allows extracting keys and values as `pp::Var`. See background section 2.6.3 on page 22 for more details.

We now consider how we can represent dictionaries in C++. The obvious approach is to represent a dictionary as a C `struct`. The fields of the struct will have corresponding names and types as defined in the dictionary. For example, the struct shown in Listing 4.13 corresponds to the dictionary shown earlier in Listing 4.12.

```
struct myObject {
  double id;
  std::string name;
}

// example use
struct myObject myObj;
myObj.id = 31;
myObj.name = "John Smith";
```

Listing 4.13: A C struct corresponding to the dictionary definition in Listing 4.12

The advantage of this is that the object passed to the C++ programmer will be a normal C++ struct. However, it will impact performance, since each field of

the struct will need to be individually converted. In fact, this makes marshalling dictionaries the slowest conversion, according to the benchmarks (see section 5.3, page 94).

However, other approaches are possible. One alternative is we could have simply passed the `pp::VarDictionary` object to the developer, without modifying it. The advantage of doing this is that it will simplify the C++ RPC library and therefore make it faster to send and receive complicated structures. However, there are a few problems with this approach:

- The C++ developer is now exposed to PPAPI. This adds a learning curve, as it is another library that the C++ developer would have to get used to in order to write their module.
- The C++ developer will need to do all the type marshalling by themselves. This renders the dictionary type definition that they wrote in WebIDL useless, and adds more burden on the developer.
- The use of `pp::VarDictionary` is actually an implementation detail of the RPC library. In other words, we simply use this as a way of transporting the data from JavaScript to C++. Perhaps someone could write another implementation that uses full binary transfer for example, using Protocol Buffers (see background section 2.4, page 16). In that case, passing the `pp::VarDictionary` to the developer would actually be more burden on the library, and probably impact performance.

Another approach is to represent a dictionary as a `std::map`. The advantage of this is that the map can be added to and deleted from dynamically and unlike structs, if a field is not specified, data is not allocated for it. The problem with `std::map` however is that the keys and values of the map have strict types. If the values have the same type, then a map will do fine; but what about if the values have different types, such as in the example in Listing 4.12? The only way around this is by using wrapper types. For example, using `pp::Var` again to represent the actual value, so the `std::map` will be from `std::string` keys to `pp::Var` values. But again, this means the developer will have to de-marshal the `pp::Var` to a standard library type, and this can get tedious when the value type is complex, for example, with multiple nested dictionaries.

In the end, we take the approach of individually, recursively de-marshalling the `pp::VarDictionary` into a struct type, as a trade off of simplicity and developer friendliness to performance.

4.2.4 Sequence Types

In WebIDL, there are two ways of specifying a collection of types: sequence types (`sequence<T>`) and array types (`T[]`). The difference, according to the specification, is that a sequence type is *passed by value* - meaning it is copied when passed into a function. Array types are passed by reference.

Since `postMessage` only transfers objects and values by value (i.e. structures are recursively copied), our RPC framework only supports sequence types. However, in JavaScript, they are represented in the same way (i.e. Array objects).

In C++, there are many ways of representing WebIDL sequence types, but we can assume that we have two options: using an array structure, or a standard library template class such as `std::vector`. We compare each approach below.

The advantage of using an array is that we do not need to use an extra library, and it might be faster for large arrays. The problem of using arrays is that anywhere we use the array, we will need to also pass its length. This can get tedious, especially if we have a function that accepts many parameters. To overcome this, it is possible to send the length of the array with the actual array by augmenting the array after a designated terminator element, such as a NULL or zero element. For example, to specify the array `[1,2,3,4]`, we send `[1,2,3,4,NULL,4]`. The `4` after the `NULL` element is the length of the array. The problem with this, however, is that we need some kind of encoding scheme to ensure that the terminator and length elements do not get counted as actual array elements. For some array types, an encoding might not exist. Moreover, processing will need to be done in order for the developer to get the length of an array, thus the developer would need to get used to another library that is not standard C++.

The advantage of using a vector is that they are dynamic and they encapsulate the length of the vector. This means they are easy to both use and marshal. The disadvantage is that it forces the user to use the `std::vector` library, especially in cases where the developer just wants an array.

In the end, I decided to go for the vector approach, for the following reasons:

- The performance is nearly the same, since we allocate the size of the vector before using it. Also, regardless of the approach taken, it will take O(n) time to marshal and demarshal the array, since it needs to be converted to/from a `pp::VarArray`.

- If the developer requires an array buffer, they can use the `std::vector::data()` method to get a pointer to the vector's internal buffer.
- Vectors are generally how C++ developers represent collections of items, so most of the time it is fine to use the `std::vector` library.

4.2.5 Implementation in C++

To implement WebIDL bindings in C++, we define how types are marshalled when sent and received from JavaScript. The IDL file will define *what* types are expected, whilst the C++ implementation will define *how* the types are accepted and received in the functions we define.

We already discussed the C++ bindings for number and string types. However, these types are sent and received from JavaScript as `pp::Var` objects. To convert them, we define a generic `RPCType` class, which has static `AsVar` and `Extract` methods. Other types then inherit from `RPCType`. We define a type class for each WebIDL type. For example, Listing 4.14 shows the `AsVar` and `Extract` methods of the WebIDL long type.

```
pp::Var LongType::AsVar(const ValidType<int32_t>& v){
  return pp::Var((int) v.getValue());
}

ValidType<int32_t> LongType::Extract(const pp::Var& v){
  if(v.is_int()){
    return ValidType<int32_t>((int32_t)v.AsInt());
  } else {
    return ValidType<int32_t>();
  }
}
```

Listing 4.14: An example of WebIDL type marshalling

We do this for each WebIDL type. Now, complex types such as Dictionary and Sequence types use these classes to extract each of the keys and values of the dictionary. For example, Listing 4.15 shows how we convert a `XYZ` dictionary type into an `XYZ` struct or a `pp::Var`. Notice how we use the `Extract` methods of other WebIDL types, for example `FloatType::Extract`. In this way, type extraction is recursive - it is simple to extract dictionaries which have values which are dictionaries, and so on. It also allows generating these classes simpler, since each type is *only* concerned about extracting its own type, thus giving a general, uniform pattern for converting types. In fact, the code shown in Listing 4.15 is actually generated by our generator.

Notice also the use of the `ValidType` class. This class is simply a wrapper over the C++ type. The reason we have it is to be able to check if a type is valid. If it is constructed without a value, then it is invalid. If it is constructed with a value, then it is valid. Therefore, we can check if the type marshalling and de-marshalling happened successfully by using the `isValid` method on the `ValidType` wrapper. We do this, for example, in the runtime layer, where we return an error if the type extraction failed, which can happen if an incorrect type was sent to the layer from JavaScript.

```
ValidType<XYZ> XYZType::Extract(const pp::Var& v){
  ValidType<XYZ> invalid;
  if(v.is_dictionary()){
    pp::VarDictionary vDict(v);
    XYZ r;

    /* member: x */
    if(!vDict.HasKey("x")) return invalid;
    const ValidType< float >& xPart = FloatType::Extract(vDict.Get("x"));
    if(!xPart.isValid()) return invalid;
    r.x = xPart.getValue();

    /* member: y */
    if(!vDict.HasKey("y")) return invalid;
    const ValidType< float >& yPart = FloatType::Extract(vDict.Get("y"));
    if(!yPart.isValid()) return invalid;
    r.y = yPart.getValue();

    /* member: z */
    if(!vDict.HasKey("z")) return invalid;
    const ValidType< float >& zPart = FloatType::Extract(vDict.Get("z"));
    if(!zPart.isValid()) return invalid;
    r.z = zPart.getValue();


    return ValidType<XYZ>(r);
  }
  return ValidType<XYZ>();
}

pp::Var XYZType::AsVar(const ValidType<XYZ>& v){
  XYZ value = v.getValue();
  pp::VarDictionary r;
  /* member: x */
  r.Set("x", FloatType(value.x).AsVar());

  /* member: y */
  r.Set("y", FloatType(value.y).AsVar());

  /* member: z */
  r.Set("z", FloatType(value.z).AsVar());

  return r;
}
```

Listing 4.15: An example of marshalling and de-marshalling a dictionary type called ‘XYZ’, which is defined to have three float members: x, y, and z.

4.2.6 Implementation in JavaScript

Unlike C++, JavaScript doesn't have strict typing, so it is perfectly normal to send a number to a function which was intending to receive a string. However, in our library, this is bad. In fact, the runtime layer will return an error callback. In order to make it more convenient to the JavaScript developer, the Native Calls JavaScript library also provided type checking. In other words, instead of just sending an object to the C++, we can just throw a JavaScript error to tell the developer that that's an illegal call because we know what type the C++ function is expecting through the IDL.

To do this, we use *JSON schemas*. JSON schemas are a declarative notation for defining JavaScript object types. We can use a JSON schema validator to check that a JavaScript object agrees with a JSON schema. The JSON schema notation is defined in a specification by the Internet Engineering Task Force (IETF) [26]. Instead of writing our own validator, we used the open source tv4 validator [27]. To get tv4 to work with our library, it needed to support AMD (see background section 2.7.2 on page 25), so we created a patch for it on GitHub, and the pull request was merged successfully.

Now that we have the notation and the validator, all we needed to do is use it in our library. We created a TypeChecker class that encapsulated a tv4 validator instance and is used to check parameter types. We have a JSON schema for each WebIDL type. Listing 4.16 shows some of these schemas.

```
{
  "byte"               : {"type": "integer", "maximum": 127,
                                             "minimum": -128},
  "octet"              : {"type": "integer", "maximum": 255,
                                             "minimum": 0},
  "short"              : {"type": "integer", "maximum": 32767,
                                             "minimum": -32768},
  // ... similar definitions for other number types ...
  "any"                : {},
  "float"              : {"type": "number"},
  "double"             : {"type": "number"},
  "DOMString"          : {"type": "string"},
  "boolean"            : {"type": "boolean"},
  "object"             : {"type": "object"},
  "null"               : {"type": "null"},
  "void"               : {"type": "null"}
};
```

Listing 4.16: JSON Schemas of WebIDL types

To define dictionary types, we simply used schema references. For example, Listing 4.17 shows a dictionary definition as a JSON schema.

```
{
  "name": "XYZ",
  "required": ["x","y","z"],
  "properties": { "x": {"$ref":"float"},
                  "y": {"$ref":"float"},
                  "z": {"$ref":"float"} }
}
```

Listing 4.17: JSON schema of a dictionary definition

The `$ref` key references another schema. In this case, we referenced the `float` schema which we defined earlier and showed in Listing 4.16. The validator will recursively look up the schemas and validate the objects. So nested dictionaries will be checked recursively.

The benefit of this is that the type checking code is done entirely declarative and therefore easy to generate automatically. In fact, the schema in Listing 4.17 was generated automatically using a WebIDL definition. In section 4.1.4.1 on page 48, we discuss how we use this notation to define a whole RPC function, interface, and module in JavaScript.

4.3 Generating RPC Code

To convert from a WebIDL file to a JavaScript and C++ RPC library, we need four main ingredients:

- WebIDL type bindings to JavaScript and C++
- The WebIDL file(s) that define the types and interfaces of our module
- A WebIDL parser
- A generator that produces the relevant JavaScript and C++ files

In the previous section, we discussed the WebIDL bindings. In this section, we discuss the parser and generator needed to produce the relevant code.

4.3.1 WebIDL Parser

The WebIDL parser takes as input a WebIDL file, and returns as output an Abstract Syntax Tree (AST) representation of the file. For more information about how parsers work, please read the background section 2.5 on page 19.

Several open source WebIDL parsers exist, so we had a choice of using an existing parser or building our own. The advantage of building our own is that we can define the format of the AST so that it can be used directly with our generator. The disadvantage is the time and effort involved in writing the parser, as well as updating it if the WebIDL specification changes. The advantage of using an existing parser is that it will be kept relatively up to date and probably more stable, as most available parsers are unit tested to ensure they work properly. An existing parser will also be more complete, meaning they support most if not all of the WebIDL syntax and specification. For these reasons, we decided to use an existing WebIDL parser. However there were a few popular parsers to choose from. We compare and contrast the different implementations below.

- Open source browser vendor implementations such as Chromium's Blink WebIDL parser [15] or Mozilla's WebIDL Parser [28].
- Robin Berjon's WebIDL2.js [29].

The advantages of using either the Chromium or Mozilla parser is that we know it's used in an actual web browser implementation. Therefore, they are reliable and are probably maintained to be up to date with the specification. Both of these implementations are written in Python, and so will run fast. However, the disadvantages are that both have very little to no documentation, making both of them hard to work with. Because they are embedded in the source code of another project, it is difficult to include them in our project without copying the source code and versioning the file in our project. This means every time the parser is updated, we have to update the file separately.

The advantages of using the WebIDL2.js parser is that it is a node.js package which can be easily added as a dependency. This means we don't need to worry about updating it in our source code. Also, the parser has detailed documentation about how the abstract syntax tree is defined. This is useful for our generator. Although the parser isn't written by a browser vendor, it is well tested against the actual WebIDL specification. In fact, it is written by Robin Berjon at W3C, so we can be confident that the parser will work well at least for the majority of cases. The disadvantage is that it might be slightly slower, although this has not been measured and is not noticeable.

In the end, I decided to go for WebIDL2.js because of its good documentation. I found the parser in JavaScript a bit easier to work with, especially since the unit tests for the generator are written in JavaScript too - so we ended up with a common testing language for the whole project.

4.3.2 Code Generators

The generator essentially does the reverse of a parser - it takes in an AST and returns a string representing the relevant code. However, sometimes the AST information was not in the format which we required, so we do a single pass through the AST, augment it, then use the augmented AST to generate the code. Figure 4.6 shows an illustration of this.

We use the augmented AST as a *context* to be passed to the template engine. We also add some helper functions on top of the template engine, to simplify the actual template. The template engine then uses the context to substitute the relevant strings into the correct places. For example, Listing 4.18 is a context that is used with the template shown in Listing 4.19 to generate the output shown in Listing 4.20. The rest of the code generator implementation is essentially augmenting the context to allow easy access for the template engine to generate all the files we needed.

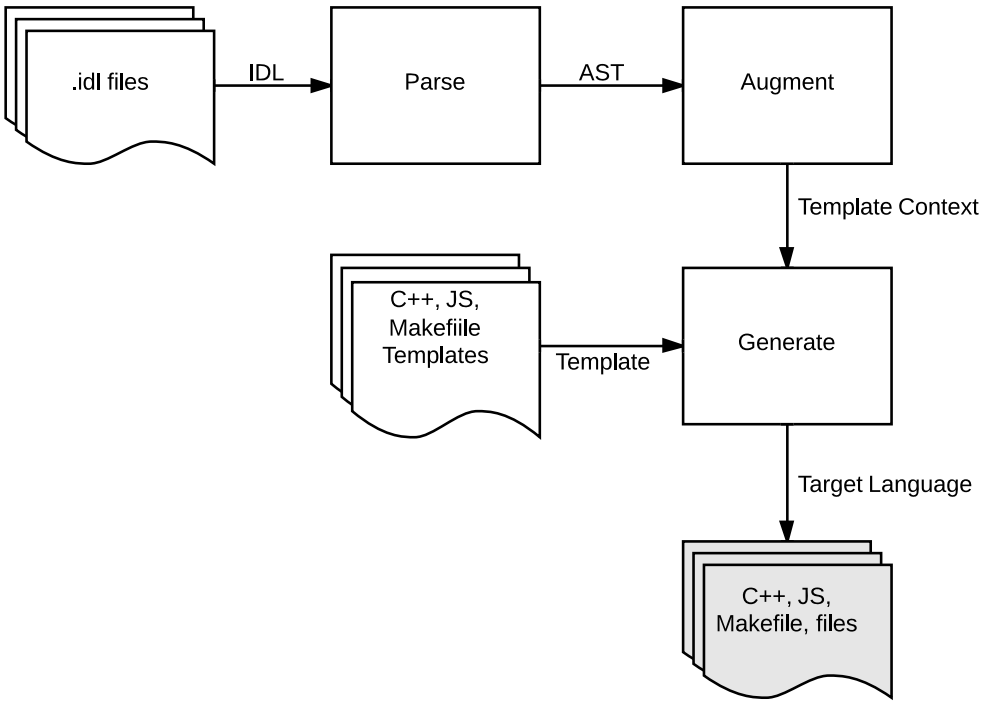


Figure 4.6: Flow diagram showing the role of the generator

Because the parser we used produces an AST in JavaScript, it made sense to have the generator in JavaScript too. This means we had to find a JavaScript template engine. There is more than a dozen template engine available to choose from, however, to simplify the templates and make them as human-readable as possible, we decided to limit our search only to templates with simple markup and logics. One template notation stood out, mustache [30]. Mustache is an elegant, simple, templating language used across many languages. We decided to go with it because of its good documentation on its notation, popularity, and simplicity. However, several implementations of mustache existed. There was mustache.js [31], handlebars [32], and hogan [33] by twitter. After considering each implementation, we found that they were very similar. We decided to choose hogan for its simplicity, speed, and extra features.

```
{
  timestamp: "Thu May 08 2014 21:38:16 GMT+0100 (BST)",
  moduleName: "Bullet",
  dictionaries: [{
    name: "XYZ",
    members: [
      { name: "x", STDTypeName: "float"},
      { name: "y", STDTypeName: "float"},
      { name: "z", STDTypeName: "float"}
    ]
  }]
}
```

Listing 4.18: An example of a template context

```
/* AUTOMATICALLY GENERATED ON {{timestamp}} */

#ifndef PPRPCGEN_{{moduleName}}_TYPES_H_
#define PPRPCGEN_{{moduleName}}_TYPES_H_

#include <string>
#include <vector>

namespace pprpcgen{
{{#dictionaries}}
typedef struct {
  {{#members}}
  {{^typeIsSequence}}{{STDTypeName}}{{/typeIsSequence}}
  {{#typeIsSequence}}std::vector<{{STDTypeName}}>{{/typeIsSequence}}
  {{name}};
  {{/members}}
} {{name}};

{{/dictionaries}}

}

#endif /* PPRPCGEN_{{moduleName}}_TYPES_H_ */
```

Listing 4.19: An example of a template

```
/* AUTOMATICALLY GENERATED ON Fri Jun 06 2014 20:11:41 GMT+0100 (BST) */

#ifndef PPRPCGEN_Bullet_TYPES_H_
#define PPRPCGEN_Bullet_TYPES_H_

#include <string>
#include <vector>

namespace pprpcgen{

typedef struct {
  float x;
  float y;
  float z;
} XYZ;

}

#endif /* PPRPCGEN_Bullet_TYPES_H_ */
```

Listing 4.20: An example of the output of the generator

4.4 Test Driven Development

Test Driven Development (TDD) is a software development approach whereby the developer writes unit tests that describe some functionality first, then implements the functionality in order to make the tests pass.

The project includes several tests for each component of the system. Unit tests are written to test fine-grained functionality (e.g. functions), while end-to-end (E2E) tests have been written to test large parts of the system as a whole.

Because the project is implemented on both C++ and JavaScript, tests had to be written for each of these languages. Thus, the project includes the following tests:

- JavaScript library tests: These test the functionality of each component of the JavaScript library. The tests run on the browser.
- Generator tests: These test the functionality of the JS and C++ generators.
- C++ library tests: These test the functionality of the C++ RPC library.
- E2E tests: These are test applications written to test the 'full stack': starting from code generation, compilation, and all the way down to individual RPC call requests.

4.4.1 Karma Test Runner

Karma test runner [34] is a test runner framework implemented at Google that makes running JavaScript tests easy. It was designed to simplify and speed up test-driven development for JavaScript. It works by letting the developer specify a configuration that states which files should be loaded, then the tests are run directly from the command line. This means the developer does not need to open a browser manually every time they want to run the tests.

Karma has been used extensively in this project to test the client side JavaScript and C++.

4.4.2 JavaScript Testing Framework

Many testing frameworks exists for JavaScript. The most popular are Jasmine [35], QUnit [36] and Mocha [37]. All of them support a similar set of features and APIs.

- Jasmine: A generic unit testing framework that gives a behavioural driven development (BDD) approach to testing. Jasmine is easy to set up, with little to no configuration. It can work on any JavaScript runtime (browser or node.js).
- QUnit: A simple unit testing framework, similar to junit for Java. QUnit is used by the popular jQuery library.
- Mocha: A testing framework that allows powerful configuration - any assertion library can be used, including Jasmine.

In the end, I decided to use Jasmine for its straight-forward set up, BDD approach, and easy configuration with Karma.

4.4.3 C++ Testing Framework

Again, many unit testing frameworks and libraries exist for C++. The most popular are CppUnit, googletest, and the Boost test library.

Google Mock (gmock) is a powerful library that allows mocking classes in C++. Mocking makes it easier to test different components of a system without requiring the actual implementation. gmock can be used with any testing framework, and it is one of the most mature mocking libraries for C++ available.

I decided to use googletest for its simple syntax, portability (as it was included with the NaCl SDK) and the fact it was easy to integrate gmock for mocking classes.

4.4.4 Creating a unified testing environment

During the initial stages of the project, a lot of time was spent finding a way to efficiently run unit tests for the generator, JavaScript library, and C++ library.

A sort of framework built on top of Jasmine, googletest, and karma was developed in order to test the C++ library. Essentially, how this worked was as follows:

- A single JavaScript test written for Jasmine was created "`it should not fail any C++ tests`".
- The body of the test loaded a Native Client module. The Native Client module was a single executable which linked *all* the C++ library tests together.

- When a test passed or failed, a message was sent to the JavaScript using `PostMessage`. This told the test suite the progress of the test.
- When a C++ test *failed*, the message received in JavaScript caused the test to throw an error and therefore the JavaScript test would fail.
- When *all* the C++ tests passed, a message is sent to JavaScript which caused the test to pass

So in the end, passing the JavaScript test was equivalent to passing all the C++ tests. When a C++ test failed, detailed information would be provided in the terminal to indicate what assertion was broken, etc.

Now, when we create a karma configuration to run the JavaScript test, it means we are able to run the C++ tests from the terminal. In other words, we are able to test the C++ Native Client module in the browser without having to open a browser manually. This saved a tremendous amount of time when doing TDD.

The final issue which we had to solve was how to test the C++ code for the different NaCl SDK tool chains available (for information about tool chains, read background section 2.6 on page 20). The issue was how to test the NaCl module which was built using a specific tool chain using the terminal (e.g. `make TOOLCHAIN=newlib CONFIG=Debug`), given that the *JavaScript* code that loaded the module hard coded this information?

The solution was to pass this information down to the JavaScript test using karma, through the main Makefile. We create a Makefile target called `cpptest`. When we run `make cpptest`, karma is started, the JavaScript test runs - which loads the C++ test module, which runs the C++ tests. Now, when we run `make cpptest TOOLCHAIN=newlib CONFIG=Debug`, karma is started with some extra command line options. Karma passes the command line options to the JavaScript by embedding them inside the HTML page. Now, our JavaScript test can use this information to correctly find and load the correct NaCl manifest and pexe/nexe.

These techniques were also done for the end to end tests, which loaded modules of different tool chains and configurations too, based on the command line arguments.

In the end, we created a Makefile target `make test` which automatically ran all the JavaScript, generator, C++, and end to end tests automatically from the terminal.

4.5 Getting Started Guide

Now that we have discussed the design and implementation of the RPC Framework and generators, we show here a concrete example that illustrates how the implementation works. This is in the form of the Native Calls getting started guide. The guide was published on GitHub in order to get anyone to easily use the Native Calls framework from scratch. The original guide includes introductions to Native Client, the idea, and so on as well as setup instructions. But we remove these details for brevity.

This guide shows how to create a simple C++ library using Native Calls. We will create a complex number calculator using a C++ native module. I've written this tutorial in a way such that you can follow along and write the module yourself.

4.5.1 Writing our interface using Web IDL

Native Calls works by generating JS and C++ that handles communication between your native module and any JavaScript application. To do this, you will need to tell Native Calls what functions you want to expose to JavaScript. You do this by writing the interface using Web IDL (which is very simple). Native Calls then takes this IDL file and generates the code for you.

Let's begin by creating a folder for our complex number calculator project.

```
mkdir complexCalculator
cd complexCalculator
vim complex.idl
```

Now, we write our complex number calculator IDL file (`complex.idl`):

```
dictionary complex {
  double r;
  double i;
};

interface Calculator {
  complex add(complex x, complex y);
  complex subtract(complex x, complex y);
  complex multiply(complex x, complex y);
```

```
  complex sum_all(sequence<complex> contents);
  complex multiply_all(sequence<complex> contents);
  sequence<double> map_abs(sequence<complex> contents);
};
```

Before moving on, let's take a closer look at the interface.

4.5.1.1 Defining dictionary types

Dictionary types get converted into C++ `structs`. In JavaScript, they define JavaScript objects. In the above, the `complex` dictionary defines a struct in C++ that has the fields `r` and `i`. It also defines the JavaScript object with the properties `r` and `i`.

Once a dictionary is defined, it can be used as a type. Native Calls allows many types, as defined in the Web IDL specification.

4.5.1.2 Defining interfaces

An interface can include definitions for many functions. These functions will be exposed to the JavaScript (i.e. we'll be able to call these functions directly from Javascript). In the IDL above, we defined `add`, `subtract` and `multiply` which all take two paramaters of `complex` type and return a `complex` type.

Meanwhile, `sum_all` takes a `sequence` type. Sequences get converted into C++ `std::vectors`, and on JavaScript, they're arrays.

4.5.2 Generating the RPC module

Now that we've defined the interface for the module, we now pass it to the generator. The generator lives in `native-calls/generator/pprpcgen.js` and can be executed directly. In the `complexCalculator` folder, generate the code like this:

```
~/native-calls/generator/pprpcgen.js --package=Complex complex.idl
```

NOTE: If you installed the package using `make install`, you should have the `pprpcgen` command installed globally. If so, you can just type `pprpcgen --package=Complex complex.idl`, and use `pprpcgen` from any directory. You can also install only the generator globally, without cloning the repo, by typing `npm install -g native-calls`.

`pprpcgen` will create a folder called Complex (matching the `--package` option). Let's take a look inside.

```
cd Complex
ls
```

Using the IDL file, we can see that the generator generated the following files:

- `PPRPCGEN_Calculator.h` This is the C++ interface that we need to implement
- `ComplexRPC.cpp` This is the C++ RPC library, specific to our Complex number calculator
- `ComplexRPC.js` The javascript file that we can include in our HTML to interface with the C++ library.
- `PPRPCGEN_ComplexTypes.h` Since we defined some extra types, (the `complex` dictionary type), this file is generated and includes the corresponding C++ `struct`.
- `Makefile` Finally, a makefile is generated for us to be used as a template.

Take a look at each file to see how the RPC library works. Most importantly let's see what's inside `Calculator.h` and `ComplexTypes.h`.

```
less PPRPCGEN_ComplexTypes.h
```

```
//...
typedef struct {
  double r;
  double i;
} complex;
```

As we expected, the dictionary was converted into an equivalent `struct`.

```
less PPRPCGEN_Calculator.h
```

```
#include "ComplexTypes.h"
#include "nativecalls/RPCType.h"
#include <vector>
namespace pprpcgen{
```

```
namespace Calculator{
complex add( complex x,  complex y);

complex subtract( complex x,  complex y);

complex multiply( complex x,  complex y);

complex sum_all( std::vector<complex> contents);

complex multiply_all( std::vector<complex> contents);

std::vector<double> map_abs( std::vector<complex> contents);

}
}
```

We can see that the generator created a header file for us to implement. The header file is entirely standard C++, using no extra libraries.

4.5.3 Implementing the interface

We can now start writing our implementation. The generated `Makefile` requires us to write the implementation in a file called `Calculator.cpp`, matching our interface name (`Calculator.h`), in the same folder (`~complexCalculator/Complex/`). Feel free to skip over the actual implementation. I've placed it here so you can copy it if you've been following along with the tutorial.

```
vim Calculator.cpp

#include "PPRPCGEN_Calculator.h"

#include <complex>
#include <vector>
namespace pprpcgen{
namespace Calculator{
complex add(complex x, complex y){
  complex cd;
  std::complex<double> std_cd = std::complex<double>(x.r, x.i)
```

```
                              + std::complex<double>(y.r, y.i);
  cd.r = std_cd.real();
  cd.i = std_cd.imag();
  return cd;
}

complex subtract(complex x, complex y){
  complex cd;
  std::complex<double> std_cd = std::complex<double>(x.r, x.i)
                              - std::complex<double>(y.r, y.i);
  cd.r = std_cd.real();
  cd.i = std_cd.imag();
  return cd;
}

complex multiply(complex x, complex y){
  complex cd;
  std::complex<double> std_cd = std::complex<double>(x.r, x.i)
                              * std::complex<double>(y.r, y.i);
  cd.r = std_cd.real();
  cd.i = std_cd.imag();
  return cd;
}

complex sum_all(std::vector<complex> contents){
  std::complex<double> currentSum(0,0);
  complex sum;
  for(std::vector<complex>::iterator it = contents.begin();
                                        it != contents.end(); ++it) {
    complex current_cd = *it;
      currentSum += std::complex<double>(current_cd.r, current_cd.i);
  }
  sum.r = currentSum.real();
  sum.i = currentSum.imag();
  return sum;
}


complex multiply_all(std::vector<complex> contents){
```

```
  std::complex<double> currentSum(1,0);
  complex sum;
  for(std::vector<complex>::iterator it = contents.begin();
                                        it != contents.end(); ++it) {
    complex current_cd = *it;
    currentSum *= std::complex<double>(current_cd.r, current_cd.i);
  }
  sum.r = currentSum.real();
  sum.i = currentSum.imag();
  return sum;
}


std::vector<double> map_abs(std::vector<complex> contents){
  std::vector<double> r;
  for(std::vector<complex>::iterator it = contents.begin();
                                        it != contents.end(); ++it) {
    complex current_cd = *it;
    r.push_back(abs(std::complex<double>(current_cd.r, current_cd.i)));
  }
  return r;
}

}
}
```

Without delving too much into the implementation details, what we wrote here was all pure C++. We didn't need to use any libraries (other than `std`), and we simply returned the results, just like we're used to doing.

Now, all we have to do is compile and include the library in our html file.

4.5.4 Building our RPC Module

For the most part, our generated `Makefile` will do everything for us. Depending on how complex our RPC functions are, we might need to tweak it a bit. But for our complex number calculator project, we can simply call `make` and everything will work.

```
make
```

```
  CXX  pnacl/Release/ComplexRPC.o
  CXX  pnacl/Release/Calculator.o
  LINK pnacl/Release/Complex_unstripped.bc
  FINALIZE pnacl/Release/Complex_unstripped.pexe
  CREATE_NMF pnacl/Release/Complex.nmf
```

This build process is actually included from `$(NACL_SDK_ROOT)/tools/common.mk`, which is used to build the NaCl SDK's examples. We use it here to make it easy to change toolchain and configuration. The default toolchain is `pnacl` and the default config is `Release`, but we could use any of the available toolchains (`pnacl`, `newlib`, and `glibc`). For example, we can build the same application using newlib by running `make TOOLCHAIN=newlib`. You can read more about the NaCl supported toolchains here.

In the end, a `.pexe` file is generated along with the NaCl Manifest (`Complex.nmf`).

Interestingly, we can package the whole Complex folder into a zip or tar file and distribute it for any JavaScript developer to use on their website, without even needing to compile it.

4.5.5 Using our library from JavaScript

We now have a binary native client application that we can include into our web application. To include it, we will use the Native Calls JavaScript library. The Native Calls JavaScript library was generated when we installed the Native Calls library using `make install`. The generated file can be found in `~/native-calls/scripts/build/NativeCalls.j`
We need to put this file into our html file, along with the generated RPC module (`complexCalculator/Complex/ComplexRPC.js`).

We copy the built `NativeCalls.js` file into our `complexCalculator` folder, and then we can write our html file that uses the library.

```
cp ~/native-calls/scripts/build/NativeCalls.js ./
vim index.html
```

```
<!DOCTYPE html>
<html>
```

```
<head lang="en">
    <meta charset="UTF-8">
    <title>Complex Number Calculator</title>
    <script type="text/javascript" src="require.min.js"></script>
    <script>
    require(['./Complex/ComplexRPC.js'], function (ComplexRPC) {
        window.Complex = new ComplexRPC();
    });
    </script>
</head>
<body>
<h1>Complex Number Calculator</h1>
</body>
</html>
```

Now, all that's left is to see the library in action! To do this, Native Client requires that the files are hosted on a server. Let's install a configure-free server such as `serve`.

```
npm install -g serve
```

Great, now we can host a server in our `complexCalculator` folder, by simply running serve.

```
serve
```

```
serving ~/complexCalculator on port 3000
```

Now, open chrome and navigate to `http://localhost:3000/`, then open the console to start using the library.

4.5.6 Making remote procedure calls from JavaScript

With the console open, we can try out some remote procedure calls. You'll notice that the functions we exposed using the idl file are available to us in the console window. The functions work as you would expect, but they are always completely **asynchronous** since `postMessage` is used as the underlying transfer layer. We can see what data is sent and received in the console. Let's call `add` to add two numbers.

```
Complex.Calculator.add({r:10, i:10}, {r:5, i:-10}, function(result){
  console.debug(result);
});
```

We can see the data being transferred in the console:

```
[NaClModule:Complex] ->  {
                           "jsonrpc":"2.0",
                           "method":"Calculator::add",
                           "id":3,
                           "params":[{"r":10,"i":10},{"r":5,"i":-10}]
                         }

[NaClModule:Complex] <-  {
                           "jsonrpc":"2.0",
                           "id":3,
                           "result":{"i":0,"r":15}
                         }
```

And finally the result being logged:

```
Object {i: 0, r: 15}
```

This is the expected result. In fact, all remote procedure calls from JavaScript take in an extra, optional, 2 paramaters: a success callback and an error callback. But what happens if we do not provide the correct number of paramaters? The RPC library is able to detect this, by throwing an error:

```
Complex.Calculator.add();
```

```
TypeError: The function add expects 2 parameter(s) but received 0
```

The RPC library also has reasonable type checking. For example if we did not pass in an object:

```
Complex.Calculator.add("hello", "world");
```

```
TypeError: Parameter 0 has invalid type: string (expected object)
```

Client side type-checking is also recursive:

```
Complex.Calculator.add({r:12, i:23}, {r:3 ,i:"not a number"});
```

```
TypeError: Parameter 1 has invalid type: string (expected number) [at .i]
```

Type checking also happens on the C++ side. In that case, the error callback is called.

4.5.6.1 Configuration

Before the module constructs, we can specify some configuration. We can specify the toolchain, config, and whether or not we want client-side type checking (as shown above). To do this, we edit the `script` tag that loads the module, and set the global NaClConfig object.

```
<script>
window.NaClConfig = {
  VALIDATION: false // can also set TOOLCHAIN and CONFIG
};
require(['./Complex/ComplexRPC.js'], function (ComplexRPC) {
    window.Complex = new ComplexRPC();
});
</script>
```

Now, when we refresh and make a remote procedure call with incorrect types, the callback will be called - i.e. type checking in the C++ has rejected the call.

```
var success = function(result){ console.log(result); };
var error = function(error){ console.error("ERROR! "+error.message); };
Complex.Calculator.add({r:12,i:23},{r:3,i:"not a number"}, success, error);
```

```
ERROR! Invalid Params: Param 1 (y) has incorrect type. Expected complex
```

Turning off JS Validation can increase performance, especially for applications that perform many requests per second.

4.6 Future Extensions

This section discusses possible future extensions to the implementation of the project.

4.6.1 Transferring contiguous number types as binary

In section 4.2 we discussed how to transfer a sequence of any type. This is represented in WebIDL as `sequence<T>`, where `T` could be any type, including dictionary types. But there is one case where it makes sense to send the data as binary data, through the use of ArrayBuffers. This is when we want to send a contiguous array of numeric type, for example, an array of floats.

Sending binary data in that case is efficient for two reasons. The first is the fact you don't need to marshal the data into a `pp::VarArray` type, since the binary buffer can be sent directly using the `pp::VarArrayBuffer` class. The second reason is how binary data is transferred in NaCl. When we send an ArrayBuffer to/from JavaScript, instead of the data being copied, it is shared. Only when the data is written to does the data get copied. This makes transferring ArrayBuffers very efficient - instead of O(n) time, it will probably take O(1) time.

Now, considering the performance gains, if we decide to send and receive contiguous number arrays as ArrayBuffers, a few questions arise. The first is how will the data be represented in JavaScript, and whether or not this representation makes sense in every context. The answer is that in JavaScript, the data will need to be sent and received as an `ArrayBuffer`. It's difficult to do anything with an ArrayBuffer though, so in JavaScript, a few more classes were made to help with reading buffers of certain types. These are called ArrayBufferViews. Currently available ArrayBufferViews are `Int8Array`, `Int16Array`, `Int32Array`, `Float32Array`, and `Float64Array`, and also their unsigned counterparts. These classes allow accessing the data of a buffer as though it was a normal JavaScript array. So, when we relate these ArrayBufferViews to IDL types, these make sense for byte, short, long, float, and double WebIDL types. The `long long` type will be unsupported, but that is understandable, considering JavaScript's number size limitations (as described earlier). In conclusion, the answer to the first question is "the binary data will be represented in JavaScript as an appropriate ArrayBufferView, and this representation makes sense for most number array types".

The second question is *when* do we send binary data? To answer the question, we consider when it's possible to send arrays of numbers in general:

- As a parameter
- As a result
- Embedded inside a dictionary or array

We could choose to send and receive binary for *all* the above scenarios, or some. To figure out when to send, we need to run some benchmarks to find how much of a performance improvement it might give.

The third question is how do we accept binary data in C++? The possibilities are either to accept it as a buffer, or a vector. As discussed earlier, however, accepting it as a buffer is problematic since we need to provide the length of the array. Luckily, we can easily and efficiently construct a vector with the same data, by providing a pointer to the data in the constructor of the vector. When sending it back, we use the `std::vector::data()` method to efficiently get a pointer to the buffer, that we can then use to send.

The fourth and final question we need to ask is how the data is transferred from C++ to JavaScript. The answer is through the `pp::VarArrayBuffer` interface. But there arises a problem to do with copying memory. To illustrate the problem, consider how we plan to send the array buffer:

- In the server stub, the concrete function implementation is called, and the result - a `std::vector` of numbers - is returned.
- A pointer to the buffer of the returned data is retrieved `std::vector::data()`.
- The number of elements in the buffer is retrieved using `std::vector::size()`.
- We now want to send the data pointed to by the buffer. To send the data, we need to use the `pp::VarArrayBuffer` class. But this will just allocate its own memory.
- Problem: we already have the `std::vector` allocating memory to the buffer we wish to send, but `pp::VarArrayBuffer` can only be constructed by allocating its own memory.

One solution would be to copy the memory contents of the vector's buffer to the `pp::VarArrayBuffer` buffer. But this seems like a slow solution.

Another solution is to specify the array length in the IDL file, and pass the result vector by reference to the function. So, before the function is called, a `pp::VarArrayBuffer` is

constructed with the correct number of bytes according to the length of the array. A pointer to that buffer is retrieved using `pp::VarArrayBuffer::map()`. The pointer is used to construct a `std::vector`. The `std::vector` is passed by reference to the concrete function implementation. Instead of the function returning, it returns void but alters the `std::vector`. When the function returns, the server stub unmaps the `pp::VarArrayBuffer` again and sends it to JavaScript.

Chapter 5

Evaluation

The project has a qualitative evaluation as well as a quantitative evaluation.

The qualitative part is to do with how “developer friendly” the system is, as a whole. To measure it, we look at the code written by the developer, as well as the learning curve required to write a complete library from JavaScript to C++.

The quantitative part is to do with the performance characteristics of the RPC library. To measure it, we measure the average time it takes to do a native computation, the time spent in the RPC library code, and the time spent in the JavaScript library code. Therefore, we can calculate roughly how much of an overhead using the library will impact on the performance.

To study these two characteristics in a real world scenario, we will use two applications:

- *Bullet Physics*: A rigid-body physics simulation using the bullet physics [38] library
- *Oniguruma*: A regular expression engine written in C++ using the Oniguruma [39] library.

We also measure the general performance of the framework by the use of micro benchmarks.

The section gives the results as well as implementation details of both the qualitative and quantitative evaluation of the project. In the end, we analyse the overall performance and usability of the RPC framework, and compare it to other currently available methods.

5.1 Performance Testing Environment

To test the framework and applications, we use the following machine specification:

- Processor: 2.3 GHz Intel Core i7 (1 processor, 4 cores)
- Memory: 8GB 1600 MHz DDR3. 256KB L2 cache per core; 6 MB L3 cache
- Graphics: NVIDIA GeForce GT 650M 1024 MB
- OS: Mac OS X 10.9.3
- Google Chrome version: 35.0.1916.153 (Official Build 274914)
- V8 version: 3.25.28.18
- Blink version: 537.36 (@175075)
- PPAPI version: 34

We use Benchmark JS [40] to easily and accurately measure the amount of time taken to run JavaScript code. Benchmark JS uses high-resolution timers (microsecond precision) and automatically runs the function we wish to test enough times so that it returns statistically significant results.

5.2 Application Performance Evaluation

5.2.1 Bullet Physics Performance

The simplest way to measure how well the physics engine performs using Native Calls RPC is to analyse the frames per second (FPS) for a range of scenes of varying complexity. We identify what the biggest impact to the FPS is by measuring how long it takes to make a request and get a response for each frame rendered. We also measure how long it takes to perform the actual simulation step. Finally, we compare these measurements with the original implementation, which was implemented using Native Client Acceleration Modules and tweaked for performance using ArrayBuffers.

5.2.1.1 Setup

To test the implementation, we use seven physics scenes. A screenshot of each scene is shown in Figure 5.2. Apart from the Jenga scenes, each scene starts with all the objects falling from the sky and crashing on the ground.

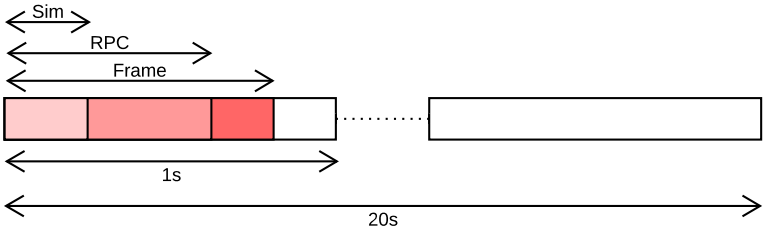


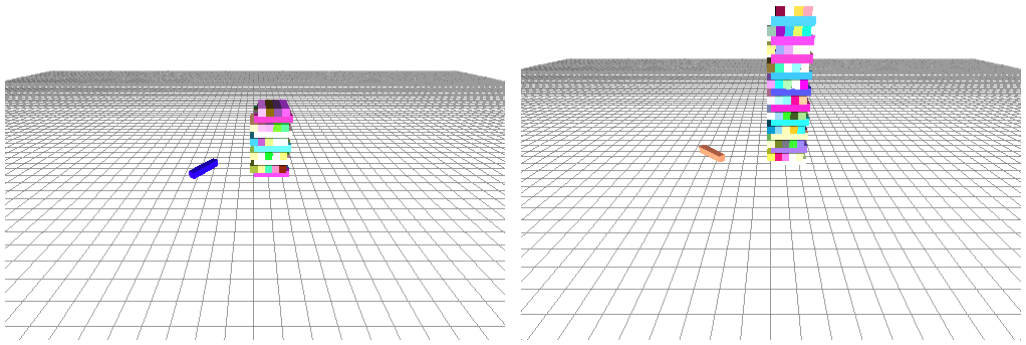
Figure 5.1: A graphical representation showing the times measured.

We measure three things - the frame rate, the simulation time, and the round trip time. To measure the frame rate, we simply add to a total of frames requested. A frame is requested by the browser automatically in order to achieve a frame rate of 60 frames per second. This is done using the `window.requestAnimationFrame` API, which conveniently takes the computation time for rendering and processing the frame into account. This is why the frame rate drops when the round trip time and simulation time increases. We measure the total simulation time inside the C++ application by taking time stamps between and after the simulation step and calculating the difference. We send it back with the results every time we do the simulation. Finally, we calculate the round trip time by taking timestamps before and after the RPC call and taking a difference. We average all this data over a period of one second and send it to be plotted.

For each second, the average time per frame is calculated. This is the inverse of frames per second, and we call this period the *frame time*. During that time, a RPC request is made and a response is received. We call the period between making a request and receiving a response the *RPC time*. We calculate these times and average them for each second for a period of 20 seconds - the *elapsed time*. Figure 5.1 shows a visualisation of this terminology.

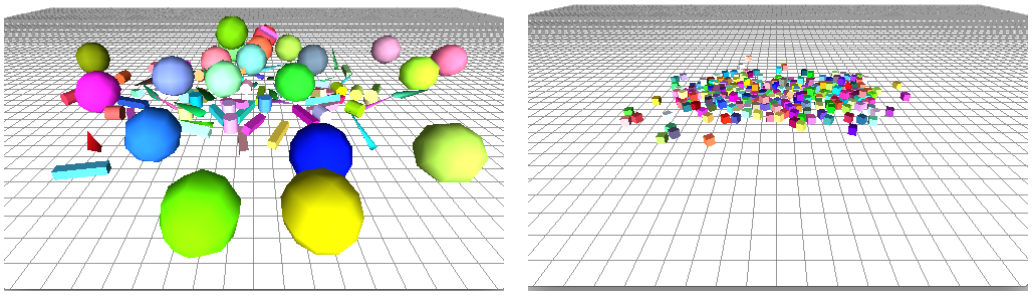
The graph is plotted in real time in the browser, but we make sure not to use the same `window` object - as this will impact JavaScript performance and give us skewed results. Instead, we send the data to be plotted in a different window - which in Chrome corresponds to a different process. HighCharts was chosen to quickly and easily create the realtime graph with very little configuration.

The actual implementation of the RPC library for the physics simulation is discussed in the qualitative evaluation section 5.4.1 on page 99.



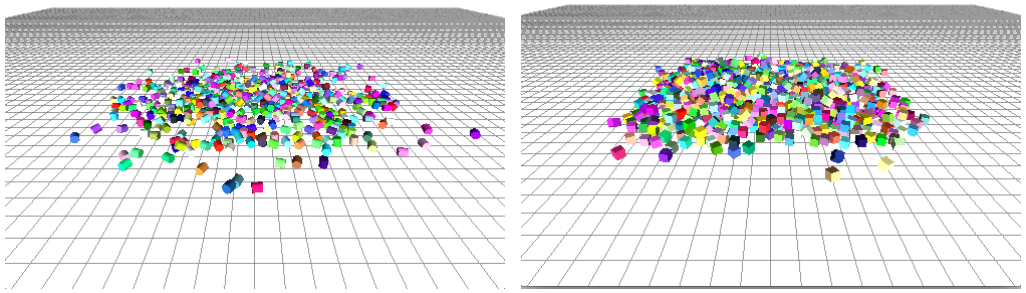
(a) Jenga 10

(b) Jenga 20



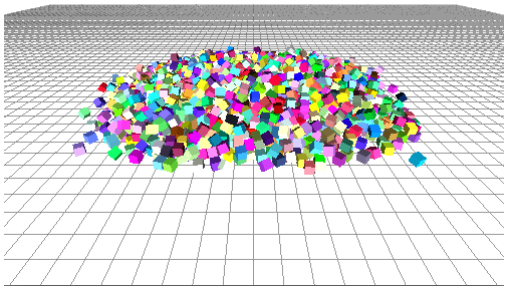
(c) Random Shapes

(d) 250 Cubes



(e) 500 Cylinders

(f) 1000 Cubes



(g) 2000 Cubes

Figure 5.2: The bullet physics scenes used in the benchmark

Scene	Time / ms									FPS		
	Frame			RPC			Simulation					
	μ	Max	Min	μ	Max	Min	μ	Max	Min	μ	Max	Min
A	17	18	16	17	18	16	0.11	0.13	0.10	60	61	55
B	17	18	16	17	18	16	0.35	2.93	0.08	60	61	56
C	17	17	16	17	17	16	0.45	0.90	0.23	60	61	59
D	17	21	16	17	21	16	0.67	1.80	0.20	59	61	48
E	21	23	17	21	24	17	2.46	5.73	1.08	49	58	44
F	39	45	17	38	43	17	10.80	14.56	0.11	25	60	22
G	72	91	34	73	91	34	23.73	41.08	0.91	14	29	11

(a) Using Native Calls

Scene	Time / ms									FPS		
	Frame			RPC			Simulation					
	μ	Max	Min	μ	Max	Min	μ	Max	Min	μ	Max	Min
A	17	18	16	17	18	16	0.13	0.14	0.12	60	61	55
B	17	18	16	17	18	16	0.38	3.37	0.10	60	61	57
C	17	17	16	17	20	16	0.60	0.93	0.27	60	61	59
D	17	19	16	17	20	17	1.14	1.87	0.22	60	61	52
E	17	18	16	17	26	16	2.64	5.82	0.97	60	61	55
F	19	21	17	19	24	17	12.06	15.49	1.09	53	59	48
G	35	40	17	35	42	18	30.77	37.93	8.86	29	58	25

(b) Using NaClAM

Table 5.1: Time measurements for the bullet physics demo, using different implementations

5.2.1.2 Results and comparison

Table 5.1 shows the results of the measurements taken during the simulation every second, over a duration of 20 seconds. Figure 5.2 shows which scene corresponds to which capital letter (for example, scene 'E' is referring to the simulation of 500 cylinders).

Because the simulation is affected by time, the table shows the range of values taken for each measurement.

We can see that scenes A, B, C, D, and E have roughly similar results for both implementations. Scenes F and G show some more interesting results. Figures 5.3, 5.4, and 5.5 show some plots showing how the simulation time and frame time change per second for scenes E, F and G respectively, using the Native Calls and NaClAM implementations.

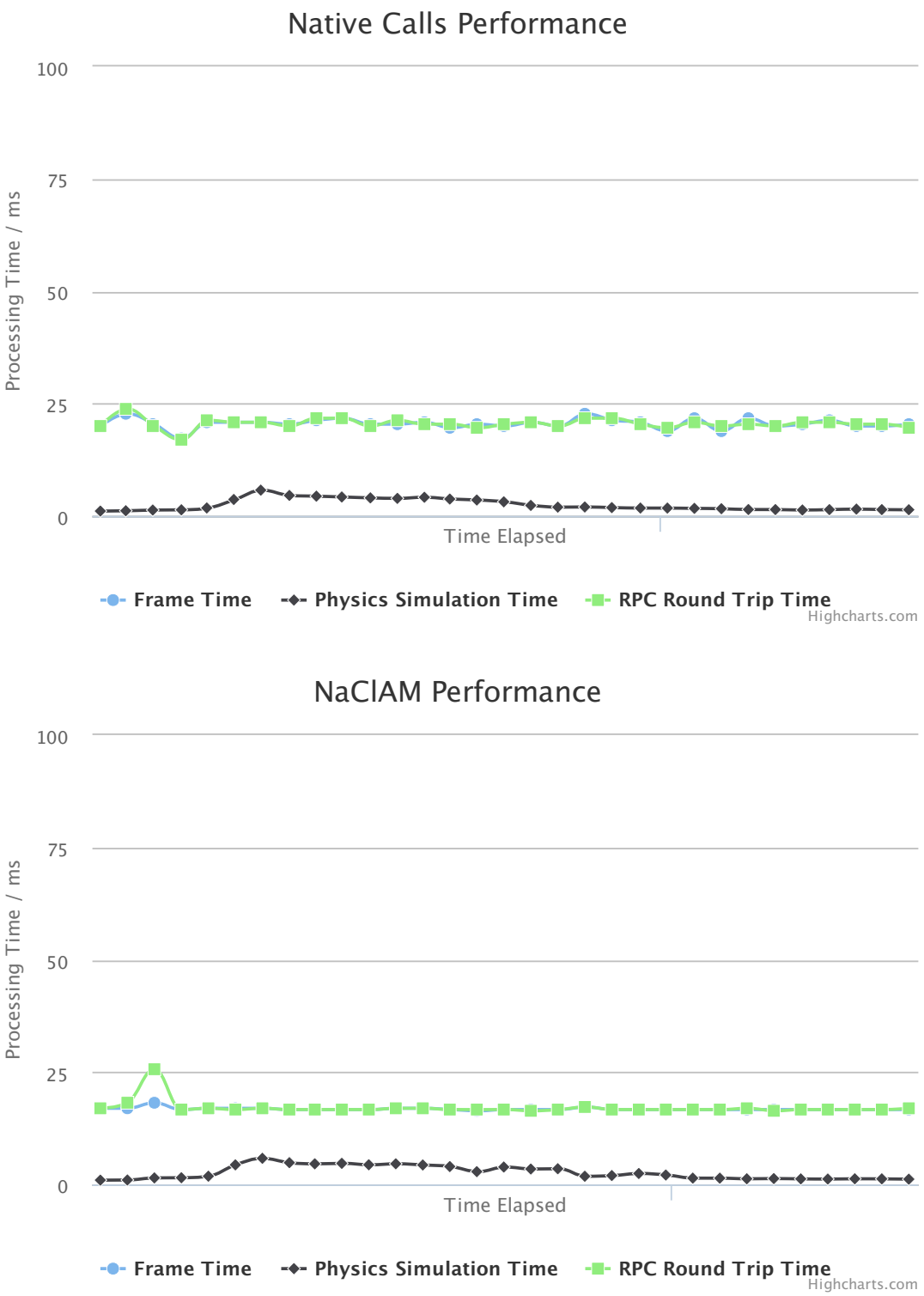


Figure 5.3: The mean processing times per second over a period of 20 seconds, for Scene E: 500 cylinders. The two graphs show the results using the same scene but different implementations.

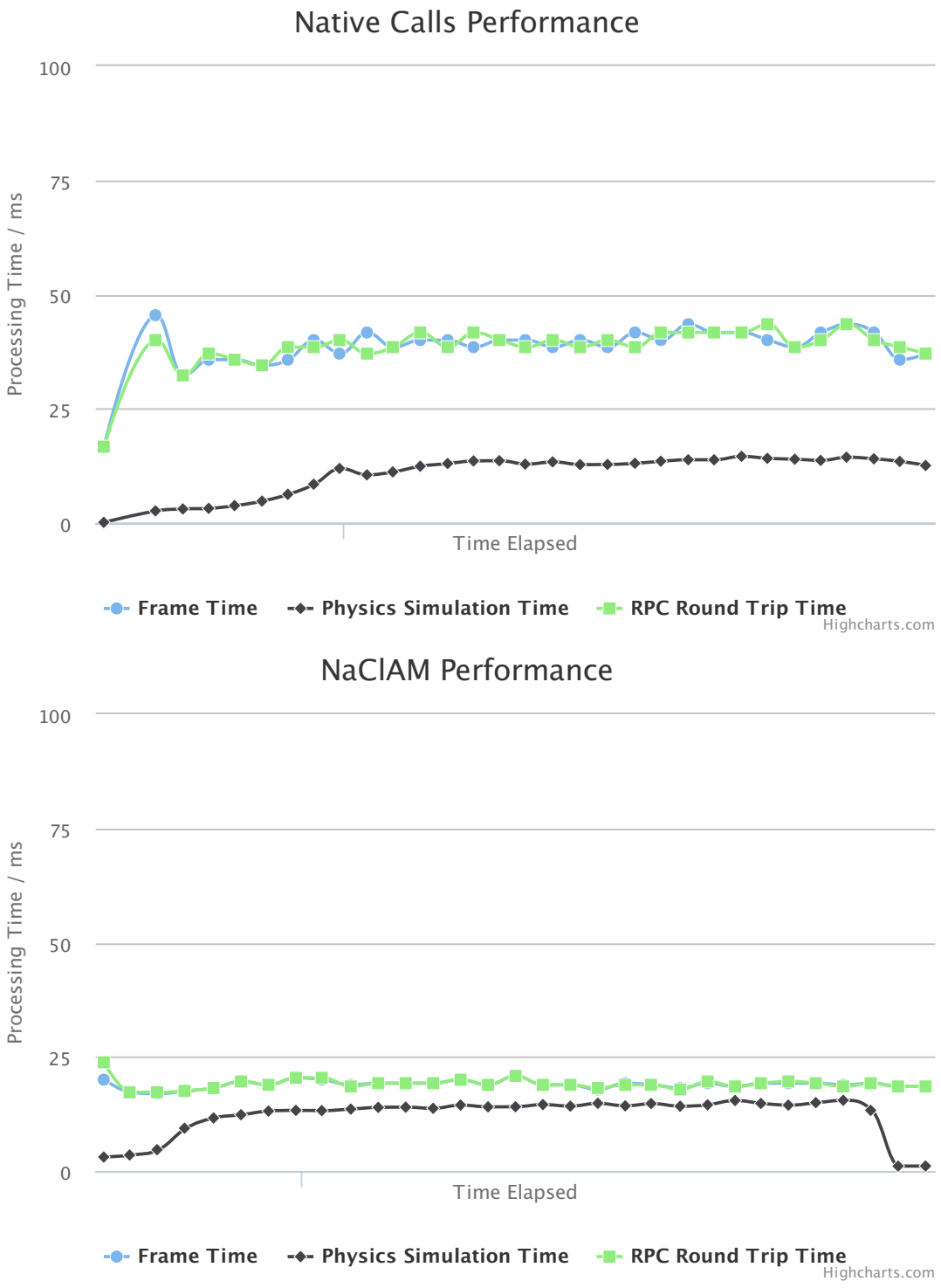


Figure 5.4: The mean processing times per second over a period of 20 seconds, for Scene F: 1000 cubes.

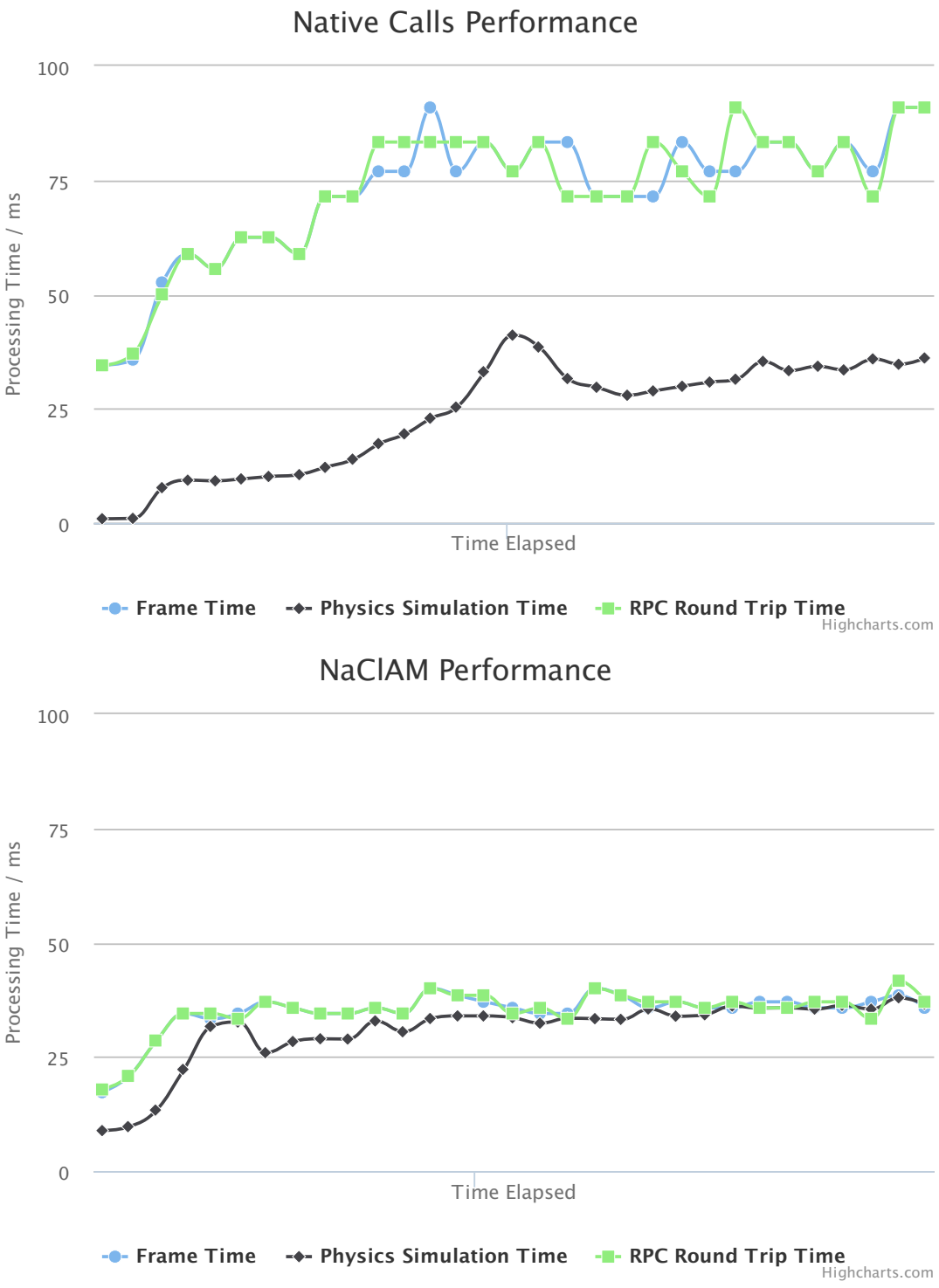


Figure 5.5: The mean processing times per second over a period of 20 seconds, for Scene G: 2000 cubes.

5.2.1.3 Analysis

From the graphs and tables, we can obviously see how the performance impact of our framework gets higher and higher with more and more objects being sent. We can see how for large scenes such as 2000 cubes, the Native Calls implementation performs almost two times worse than the NaClAM implementation. However for smaller scenes, the two implementations have very similar performance.

From the graphs we can see how the Native Calls framework contributes to the performance impact. For example, in Figure 5.5, the large space between the physics simulation time line and the RPC round trip time line is an indication that the framework is doing most of the processing in the RPC call, not the simulation. When we compare this to the NaClAM version in the same figure, it is clear that most of the processing happens in the physics simulation.

The most likely reason for these two observations is that the data marshalling on the C++ Native Calls RPC framework implementation negatively impacts the performance. In our implementation, the data is processed in O(n) time in order to convert the received `pp::VarArray` array of objects into a `std::vector` of `struct`s. In section 5.3 (page 94), we find that converting WebIDL dictionaries is also quite slow, compared to converting an array of numbers. When we compare this to the NaClAM implementation, we can see that the NaClAM version has almost O(1) time, since the data is only being read and is shared between the module and JavaScript. This is explained in section 3.1 on page 27.

5.2.2 Oniguruma Regular Expressions Performance

To get an insight of the performance of the oniguruma library for Native Client using Native Calls, we shall count the number of regular expression matches. We compare this to running the engine in Node JS server natively, and then using WebSockets to get functionality in the browser.

5.2.2.1 Setup

Again, the actual implementation of the library is discussed later on in the evaluation, in section 5.4.2 on page 99. In this section, we discuss how we measure the time it took to find all matches.

For the benchmark, we searched a large code base. We took part of the jQuery implementation in JavaScript and stored it in a string. We then performed regular

expression searches on that string. The search string was 3467 characters. We split it into lines. For each line, we found all instances of `"this"`, `"var"`, `"selector"`, and `"window"` using regular expressions. In total, there was 237 matches, and each implementation gave the correct output. However, we measured the amount of time it took to find all these matches.

For the browser implementations (Native Calls and web sockets), we measure it using BenchmarkJS, to get an accurate running time, and a relative error margin. Using BenchmarkJS, we measured the time it took from initiating a request to receiving a response. BenchmarkJS performed the tests hundreds of times to get an accurate running time.

For the server implementation (using node.js natively on the server), we simply timed and performed all the searches 1000 times and got the mean of the running time.

5.2.2.2 Results and comparison

Table 5.2 shows the results of running the benchmark application using Native Calls RPC and node-oniguruma.

Method	**Time Taken / s**
Native Calls	0.709 ±0.47%
node-oniguruma with web sockets	0.375 ±0.32%
node-oniguruma (native)	0.045

Table 5.2: A comparison of the time taken to find all matches of a regular expression using different implementations

5.2.2.3 Analysis

We can see that the node-oniguruma version performs much better. This is due to a number of reasons:

- The node-oniguruma implementation is *native*, in the sense that it uses JavaScript types directly. No marshalling nor transfer happens.
- The node.js V8 engine usually performs better than V8 in Chrome, as it is optimised for server side performance.
- The web socket implementation has a very simple runtime on both client and server sides. No error checking, type checking, etc. happens.
- Strings are the slowest primitive types to marshal, send and de-marshal in Native Calls. See section 5.3 for details.

5.3 Framework Performance Evaluation

We used the IDL file shown in Listing 5.1 to test transfer and processing performance of individual operations:

```
dictionary dict {
  DOMString str;
  double d;
  boolean b;
};

dictionary nestedDict {
  DOMString topStr;
  double topD;
  boolean topB;
  dict nested;
};

interface Benchmark{
  long bench_long(long v);
  double bench_double(double v);
  DOMString bench_DOMString(DOMString v);
  dict bench_dict(dict v);
  nestedDict bench_nestedDict(nestedDict v);

  sequence<long> bench_seq_long(sequence<long> v);
  sequence<double> bench_seq_double(sequence<double> v);
  sequence<DOMString> bench_seq_DOMString(sequence<DOMString> v);
  sequence<dict> bench_seq_dict(sequence<dict> v);
  sequence<nestedDict> bench_seq_nestedDict(sequence<nestedDict> v);
};
```

Listing 5.1: WebIDL file used for benchmarking

We will use the generated RPC library to test the framework's performance. We will do this by making RPC calls and measuring how long it takes.

5.3.1 Round trip performance

We measure the number of round trips performed in one second (round trips per second, RT/s).

One round trip corresponds to a full remote procedure call, starting from JavaScript, reaching the target function, returning from the function, and going back to the JavaScript. This is illustrated in Figure 5.6.

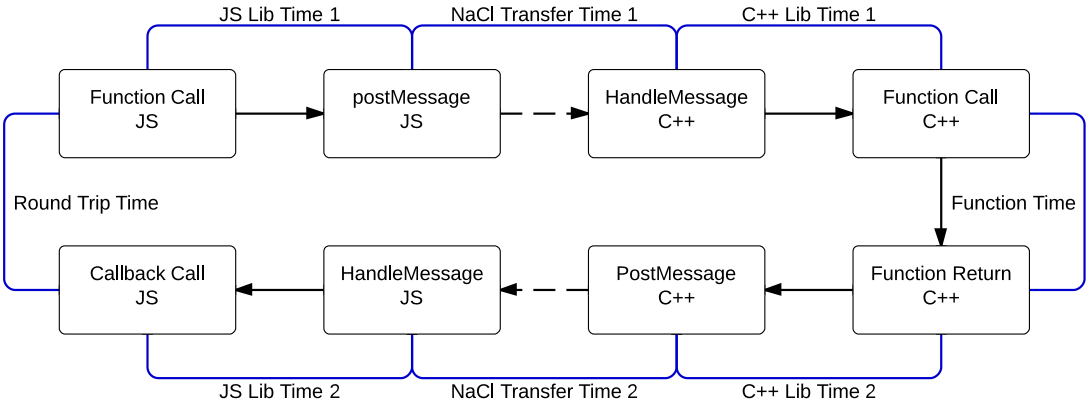


Figure 5.6: A depiction of the round trip time from JavaScript to C++

Type	Mean RT/s	Uncertainty	Number of runs
long	418	±1.79%	56
double	423	±2.08%	48
DOMString	420	±1.25%	43
dict	415	±2.39%	44
nestedDict	385	±1.29%	47

Table 5.3: Round trip performance of sending a single parameter

Array Length	Round trips per second				
	long	double	DOMString	dict	nestedDict
10	403	403	378	317	244
45	379	384	309	182	112
100	354	347	234	110	60.07
450	237	235	102	32.82	15.83
1000	163	160	55.41	15.39	7.43
4500	49.39	48.93	14.60	3.62	1.68
10000	24.68	24.50	6.62	1.62	0.75
45000	5.99	5.98	1.28	0.33	0.15

Table 5.4: Round trip performance for arrays of different lengths and types

Tables 5.4 and 5.3 show the number of round trips performed in a second for RPC calls with a single parameter and different array lengths. To compare these times, Figure 5.7 shows a column chart visualisation of the data.

5.3.2 C++ Library Time

We measure the number of microseconds taken to handle a RPC call. This is the time it takes to detect it is an RPC call, extract parameters, convert them, find the method, call it, pack the result, and post the message back to JS.

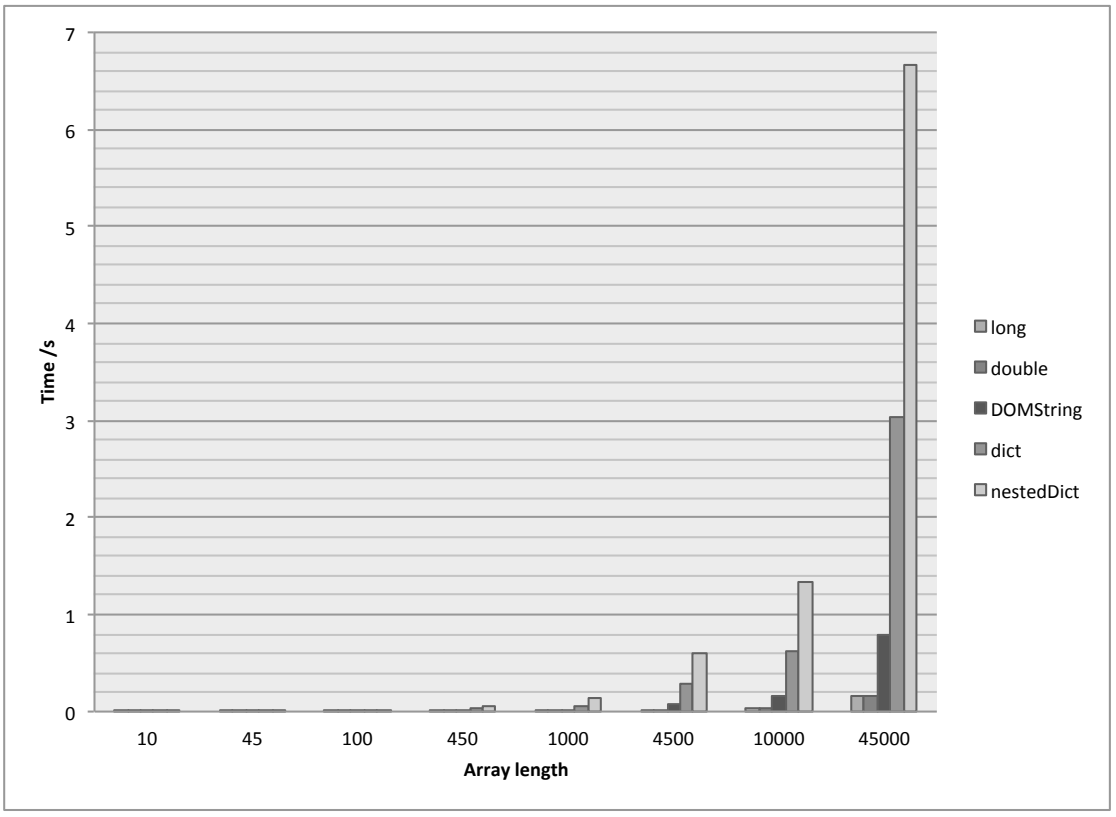


Figure 5.7: The round trip time for arrays of different lengths and types

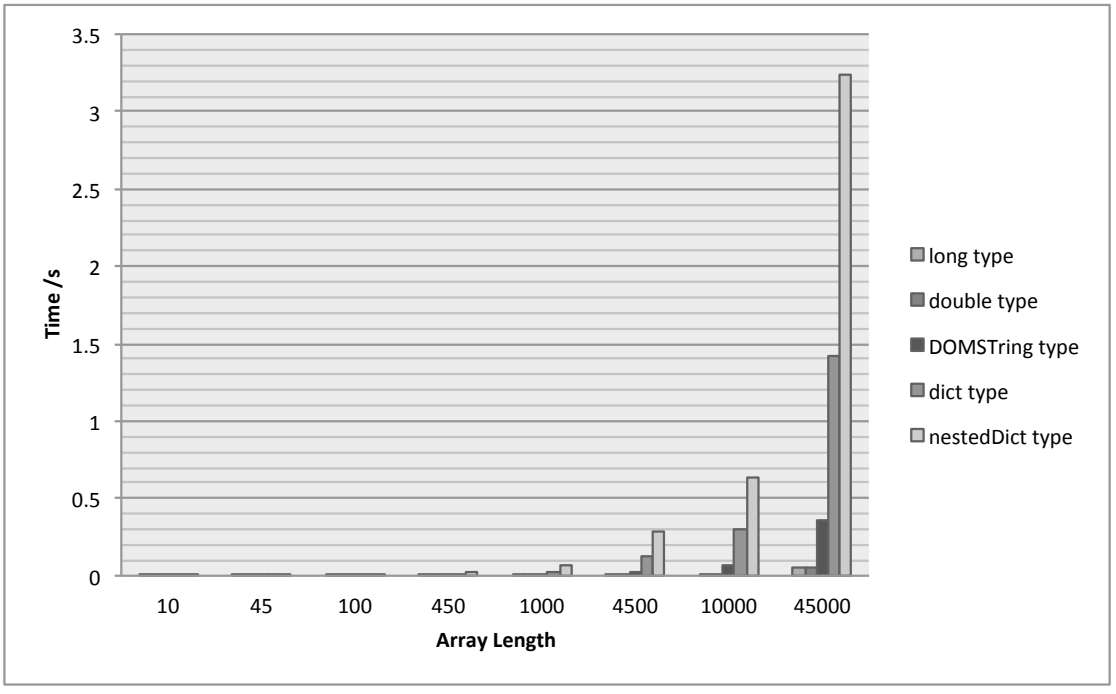


Figure 5.8: The C++ library time for arrays of different lengths and types

The results are measured and averaged for the same runs that were performed above. Tables 5.5 and 5.6 show the results, and Figure 5.8 shows a visualisation of the data.

Type	**Mean lib time/μs**	**Uncertainty (1 sd)**
long	106	18
double	104	17
DOMString	104	18
dict	136	23
nestedDict	180	28

Table 5.5: Mean C++ library time for sending and receiving single parameters of different types

Array Length	**Time / μs**				
	long	**double**	**DOMSTring**	**dict**	**nestedDict**
10	125	126	197	453	861
45	169	163	484	1513	3273
100	243	246	906	3445	7010
450	705	704	3735	13629	28198
1000	1276	1355	7607	29582	63494
4500	5548	5564	30485	132122	292828
10000	11283	11376	68444	301438	632956
45000	50532	50843	359134	1418792	3242286
100000	104087	114020	799743	3319250	7347985

Table 5.6: Mean C++ library time for sending and receiving arrays of different lengths and types

5.3.3 JS Library performance

The JS library performance **without** client-side type validation has also been measured, however its performance impact is negligible. The slowest benchmark was found to take approx 3 microseconds (269,253 ops/sec $\pm$ 1.90%).

5.3.4 Analysis

From the data, we can see that for small types, the most contributing factor to performance is the browser (e.g. event system, etc.) and PPAPI libraries (how PPAPI implements postMessage). For example, sending a single long type takes 2392.34 microseconds (.002 seconds), but our library only spends 105.5 microseconds processing the call (less than 5% of the time).

For large and complicated data, the impact of using the library becomes higher and higher. For example, sending 45000 nested objects (which are actually quite simple)

has a total round-trip time of 6.67s, and a whole 3.24 seconds of this is spent in our library (i.e. half the time).

The most likely reason for this is that the C++ library takes O(n) time to process the data in order to marshal it, by converting it from a `pp::VarArray` into a `std::vector`.

We also notice that the DOMString and Dictionary types are the slowest to marshal and de-marshal. One reason for this could be that number types have a standard representation so perhaps the PPAPI was able to improve how they are transferred, whilst the string types and dictionary types would need marshalling even by the PPAPI. One obvious reason for why the dictionary types take longer to marshal is that they contain multiple types, each type is individually marshalled. So sending a dictionary with multiple keys and values is almost equivalent to sending multiple values separately.

5.4 Usability Evaluation

To get some insight as to how usable and useful the library and generator is, we analyse the number of lines the developer would need to write to build the same application. We take a look at two different applications: a bullet physics simulation which uses the C++ module to calculate simulation steps, and a regular expression library which uses a native module to do execute regular expressions. The table below shows how many lines the developer had to write to achieve the same program.

5.4.1 Implementation: Bullet

We use the IDL shown in Listing 5.2 as our interface. This allows us to send normal JavaScript objects to the C++ implementation, and in C++, the dictionaries are automatically marshalled as structs.

The rest of the implementation is taken from the original implementation. This is basically a static object which holds all the information about the scene, allows loading a scene, and calculates simulation steps, all using the Bullet Physics library.

5.4.2 Implementation: Oniguruma

We use the IDL shown in Listing 5.3. Again, the rest of the implementation is taken from the node-oniguruma original project. There is a critical difference though, which is to do with the object-oriented nature of the regular expressions. To implement this, we have a static map of Scanner ids to C++ references to Scanner objects. RPC requests call the C++ class methods, using the id to to get the actual instance. An example of how this works is shown in the C++ Listing 5.4.

```
dictionary XYZ{
        float x;
        float y;
        float z;
};

dictionary Cube{
        DOMString name;
        float wx;
        float wy;
        float wz;
};

dictionary Convex{
        DOMString name;
        sequence<XYZ> points;
};

// Sphere, Cylinder, and Body  defined similarly.

dictionary Scene{
        sequence<Cube> cubes;
        sequence<Convex> convices;
        sequence<Sphere> spheres;
        sequence<Cylinder> cylinders;
        sequence<Body> bodies;
};

dictionary SceneUpdate{
        sequence<float> transform;
        unsigned long long delta;
};

interface BulletInterface {
        double LoadScene(Scene scene);
        SceneUpdate StepScene(XYZ rayTo, XYZ rayFrom);
        boolean PickObject(double index, XYZ pos, XYZ cpos);
        boolean DropObject();
};
```

Listing 5.2: WebIDL for Bullet

```
dictionary CaptureIndex{
        unsigned long start;
        unsigned long end;
        unsigned long length;
        unsigned long index;
};

dictionary OnigMatch{
        unsigned long index;
        sequence<CaptureIndex> captureIndices;
};

interface Scanner{
        unsigned long newScanner(sequence<DOMString> patterns);
        OnigMatch findNextMatch(unsigned long scannerID, DOMString string,
                                            unsigned long startPosition);
};
```

Listing 5.3: WebIDL for the Oniguruma implementation

```
uint32_t newScanner( std::vector<std::string> patterns){
        return OnigScanner::newInstance(patterns);
}

OnigMatch findNextMatch( uint32_t scannerID,  std::string string,
                                             uint32_t startPosition){
        OnigScanner* scanner = OnigScanner::getInstance(scannerID);
        return scanner->findNextMatch(string, startPosition);
}
```

Listing 5.4: Wrapping C++ instance methods with RPC functions, in the Oniguruma implementation

5.4.3 Results: Bullet

Line count	Original	Native Calls	Difference
C++	404	331	73
JS	979	811	168
IDL	0	54	-54
Total	1383	1196	187

Table 5.7: The number of lines of code needed to write the Bullet demo

Table 5.7 shows the number of lines that the developer would have to write to implement the bullet physics simulation, based on the original implementation. We can see a total of 187 lines were saved by using the Native Calls library and generators. In the C++ code, most of these differences occurred because the NaClAM version required the user to marshal and de-marshal the messages manually. For example, Listings 5.5 and 5.6 shows how both implementations handle the `PickObject` RPC call.

```
bool PickObject(double index, XYZ pos, XYZ cpos) {
  if (!bulletScene.dynamicsWorld) {
    return false;
  }
  index++;
  if (index < 0 ||
      index >= bulletScene.dynamicsWorld->getNumCollisionObjects()) {
    bulletScene.pickedObjectIndex = -1;
    return false;
  }
  bulletScene.pickedObjectIndex = index;
  bulletScene.addPickingConstraint(btVector3(cpos.x, cpos.y, cpos.z),
                                   btVector3(pos.x, pos.y, pos.z));
  return true;
}
```

Listing 5.5: Native Calls Implementation of PickObject

```
void handlePickObject(const NaClAMMessage& message) {
  if (!scene.dynamicsWorld) {
    return;
  }
  const Json::Value& root = message.headerRoot;
  const Json::Value& args = root["args"];
  const Json::Value& objectTableIndex = args["index"];
  const Json::Value& pos = args["pos"];
  const Json::Value& cpos = args["cpos"];
  float x = pos[0].asFloat();
  float y = pos[1].asFloat();
  float z = pos[2].asFloat();
  float cx = cpos[0].asFloat();
  float cy = cpos[1].asFloat();
  float cz = cpos[2].asFloat();
  int index = objectTableIndex.asInt();
  index++;
  if (index < 0 ||
            index >= scene.dynamicsWorld->getNumCollisionObjects()) {
    scene.pickedObjectIndex = -1;
    return;
  }
  scene.pickedObjectIndex = index;
  scene.addPickingConstraint(btVector3(cx, cy, cz), btVector3(x,y,z));
  NaClAMPrintf("Picked \%d\n", scene.pickedObjectIndex);
}
```

Listing 5.6: NaClAM implementation of PickObject

In the JavaScript code, most of the lines saved were type checking code and separate functions for the RPC calls and handlers. For example, the whole of `world.js` (140 lines) is type checking code similar to Listing 5.7. Notice how this doesn't even check the actual types, it just checks if the fields are defined. On the other hand, the generated Native Calls library produces full, structure recursive, convenient type checking without extra effort from the developer. Listing 5.8 shows an example of how separate handlers had to be written for the NaClAM implementation, while Listing 5.9 shows how the same effect is achieved by the use of callbacks.

```
function verifyCubeDescription(shape) {
  if (shape['wx'] == undefined) {
    return false;
  }
  if (shape['wy'] == undefined) {
    return false;
  }
  if (shape['wz'] == undefined) {
    return false;
  }
  return true;
}
```

Listing 5.7: NaClAM implementation's JavaScript type checking example

```
// somewhere in scene.js...
function loadWorld(worldDescription) {
  clearWorld();
  //... some JS implementation
  NaClAMBulletLoadScene(worldDescription); // RPC request
  lastSceneDescription = worldDescription;
}

// somewhere in NaClAMBullet.js...
function NaClAMBulletInit() {
  aM.addEventListener('sceneloaded', NaClAMBulletSceneLoadedHandler);
  // other handlers
}

function NaClAMBulletLoadScene(sceneDescription) {
  aM.sendMessage('loadscene', sceneDescription);
}

function NaClAMBulletSceneLoadedHandler(msg) {
  console.log('Scene loaded.');
  console.log('Scene object count = ' + msg.header.sceneobjectcount);
}
```

Listing 5.8: NaClAM implementation of requests and response handlers

```
function loadWorld(worldDescription, callback ) {
  clearWorld();
  //... some JS implementation
  bullet.BulletInterface.LoadScene(rpcScene, function(result){
    console.log("Scene loaded "+ result);
    if(callback)callback();
  });
  lastSceneDescription = worldDescription;
}
```

Listing 5.9: Native Calls Implementation of requests and response handlers

We can see that for both the JavaScript and C++ developer, using Native Calls saves a lot of time. The examples also show how the developer does not need to get used to another paradigm (i.e. using listeners, requests, and handlers) or library (i.e. JsonCpp). Instead, using Native Calls, the developer on both JavaScript and C++ can focus on the main logic of the actual native module!

5.4.4 Results: Oniguruma

Line count	Original	Native Calls	Difference
C++	666	601	65
JS	100	134	-34
IDL	0	15	-15
Total	766	750	16

Table 5.8: The number of lines of code needed to write the Oniguruma library

Table 5.8 shows the number of lines that the developer would have to write to implement the Oniguruma library, based on the original implementation. Note: the original implementation used *CoffeeScript*, we used the CoffeeScript compiler to get the equivalent JavaScript code. The generated JavaScript code is what is counted - and although it is computer generated, it maps directly to human-readable JavaScript that would have been written. We consider this fair when we compare the number of lines, as CoffeeScript introduces a lot of *syntax sugar* which can greatly reduce the number of lines in JavaScript. The number of CoffeeScript lines in the original implementation is 51.

We can see that only a few number of lines are saved in the Oniguruma implementation. This is because of the fact that Native Calls does not currently support object oriented RPC. The JavaScript developer lost quite a few lines because of this. We needed to create a class that makes the RPC calls and keeps references to objects in the C++ code. Listing 5.10 shows this. The `_whenScannerReady` member function

allows calling different RPC functions with a scannerID. Scanners are instantiated dynamically in the C++ using the `newScanner` RPC call. All member functions then take in an extra parameter, the ID, which is used to find the instance in the C++ code. We compare this to the node-oniguruma implementation. Here, the C++ implementation extends JavaScript, so a C++ object corresponds directly to a JavaScript object. This is explained in the related work section 3.2 on page 29.

```
function OnigScanner(sources){
  this.ready = false;
  this.scannerID = -1;
  this.sources = sources;
  this.scanner = scanner; // the RPC interface
}

OnigScanner.prototype._whenScannerReady = function(callback){
  var thisRef = this;
  if(this.ready){
    // call straightaway.
    if(typeof callback == "function"){
      callback.apply(this);
    }
  } else {
    // new scanner then call. does RPC request
    this.scanner.newScanner(this.sources, function(scannerID){
      thisRef.ready = true;
      thisRef.scannerID = scannerID;
      if(typeof callback == "function"){
        callback.apply(thisRef);
      }
    });
  }
};

OnigScanner.prototype.findNextMatch = function(string, startPosition,
                                                    callback) {
  this._whenScannerReady(function(){
    var thisRef = this;
    // RPC with scannerID which is used to find the C++ instance
    this.scanner.findNextMatch(this.scannerID, string, startPosition,
                                                  function(match){
      if(match.captureIndices.length == 0){
        match = null;
      }
      if(match != null){
        match.scanner = thisRef;
      }
      if(typeof callback == "function"){
        callback(match);
      }
    });
  });
};
```

Listing 5.10: Augmenting RPC: Wrapping RPC methods with a JS class

5.5 Evaluation Conclusion

After looking at two different applications, we can see that for static or singleton-based applications, such as the bullet physics application, the Native Calls library and generators saves a lot of development time and provides a natural, straight forward method of implementing a C++ library usable from JavaScript. However, for object oriented applications, we find that the JavaScript and C++ developer must think about implementing low level details such as instance look up in order to give a object oriented RPC system.

As for performance, we find that the performance impact is negligible for RPC calls with little data. The impact increases with increasing data size.

This makes using Native Calls perfect for singleton-based applications which send and receive little data, while a trade-off must be made for object-oriented applications or applications that send and receive large data.

Chapter 6

Conclusion

In this project, we have provided a solution for writing high performance JavaScript applications that use a C++ Native Client modules, in a way that is natural to both the JavaScript developer and C++ developer. To do this, we provide a code generator that produces C++ and JavaScript code in a package that the C++ developer can change and tweak for performance and feature extension. The C++ developer will have to write an interface in WebIDL, then the generator will produce all the boiler-plate code for both JavaScript and C++.

The main challenges were how to map WebIDL types and interfaces to C++ and JavaScript language features, and using a parser to produce human readable JavaScript and C++ code. Other challenges included using PostMessage as a transport layer for a layered RPC framework on both JavaScript and C++, as well as creating a testing framework and a build system based on the Native Client examples to efficiently build and test C++ Native Client modules in the browser.

In the end, we developed the generator and framework and wrote demo applications to test its usability and performance. We concluded that the framework performed well when little data is passed to the RPC functions, however, had a larger performance impact when large amounts of data are sent and received. We found that the code generator saved a lot of development time when compared to previous methods of implementing the same application, but developing object oriented RPC functionality required some more time.

6.1 Future Work

- Provide C++ to JavaScript RPC. Whilst we originally set out to provide a multi-directional RPC framework, because of time constraints this was not possible and only JavaScript to C++ RPC was implemented. However, implementing RPC in the other direction is symmetrical in most cases, as several layers including transport and RPC will remain the same.
- Improve performance by sending binary data when we can. In section 4.6 on page 81, we mentioned some design considerations and possible implementation of sending binary data when an array of contiguous number types are sent. This will greatly improve performance since binary data is shared between JavaScript and C++.
- Experiment with different RPC protocols and data types, such as Google Protocol Buffers to see if a performance improvement can be achieved. Because of our layered approach to RPC, this should be feasible and could provide some interesting results.
- Improve performance by allowing the RPC framework to spawn a new thread per request, thus allowing concurrent RPC calls.
- Object oriented generated code. For the evaluation, we showed how it is possible to create an object oriented RPC library by wrapping the RPC calls in JavaScript classes which hold a C++ instance identifier. One future extension to the project would be to allow these classes to be generated automatically.
- Produce a JavaScript fall-back when Native Client is not supported in the browser. This can be done using PepperJS [41], a library by Google that uses emscripten [42] to transpile machine code produced by the Native Client compilers, into JavaScript code.

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/.

[14] The Chromium Developers. *Web IDL in Blink*. Google Developers, 2014. http://www.chromium.org/blink/webidl.

[15] The Chromium Developers. *Chromium WebIDL Parser*. Chromium, 2014. https://code.google.com/p/chromium/codesearch#chromium/src/tools/idl_parser/idl_parser.py.

[16] ES Web Browser Project. *esidl: The Esidl Web IDL compiler for C++11*, 2014. https://github.com/esrille/esidl.

[17] Extensible Web Community Group. *webidl.js: An implementation of WebIDL in ECMAScript*, 2014. https://github.com/extensibleweb/webidl.js.

[18] Google Developers. *Google Protocol Buffers*. Google Developers, April 2012. https://developers.google.com/protocol-buffers/docs/overview.

[19] Daniel Wirtz. *ProtoBuf.js: Protocol Buffers for JavaScript*, 2014. https://github.com/dcodeIO/ProtoBuf.js.

[20] FireBreath. *FireBreath: A browser plugin framework*. FireBreath, November 2013. http://www.firebreath.org/.

[21] Apache Software Foundation. *Apache Thrift Website*. Apache Software Foundation, 2014. http://thrift.apache.org/.

[22] ahilss. *thrift-nacl: Thrift for Native Client*, 2014. https://github.com/ahilss/thrift-nacl.

[23] Ivan Žužak. *pmrpc: HTML5 inter-window and web workers RPC and publish-subscribe communication library*, 2014. https://github.com/izuzak/pmrpc.

[24] Sebastien Vincent. *JsonRpc-Cpp: JSON RPC Protocol implementation in C++*. Sourceforge, 2014. http://jsonrpc-cpp.sourceforge.net/.

[25] Wikipedia. *JSON-RPC — Wikipedia, The Free Encyclopedia*, 2014. URL http://en.wikipedia.org/w/index.php?title=JSON-RPC&oldid=612788472. [Online; accessed 16-June-2014].

[26] Francis Galiegue and Kris Zyp. *JSON Schema Specification*. Internet Engineering Task Force, 2013. http://json-schema.org/latest/json-schema-core.html.

[27] Geraint. *tv4: Tiny Validator for JSON Schema v4*, 2014. https://github.com/geraintluff/tv4.

[28] Mozilla. *Mozilla WebIDL Parser*. Mozilla, 2014. http://mxr.mozilla.org/mozilla-central/source/dom/bindings/parser/WebIDL.py.

[29] Robin Berjon. *webidl2.js: JavaScript WebIDL parser*, 2014. https://github.com/darobin/webidl2.js.

[30] Mustache. *Mustache: logic-less templates*. Mustache, 2014. http://mustache.github.io.

[31] Jan Lehnardt. *Mustahce.js: Minimal templating with mustache in JavaScript*, 2014. https://github.com/janl/mustache.js.

[32] Yehuda Katz. *Handlebars: A mustache implementation in JavaScript with extra features*, 2014. https://github.com/wycats/handlebars.js.

[33] Twitter. *hogan.js: A mustache compiler*, 2014. https://github.com/twitter/hogan.js.

[34] Karma Runner. *karma: Spectacular Test Runner for JavaScript*, 2014. http://karma-runner.github.io.

[35] Pivotal Labs. *Jasmine: Behavior-Driven JavaScript*, 2014. http://jasmine.github.io.

[36] jQuery. *QUnit: A JavaScript Unit Testing framework*, 2014. http://qunitjs.com.

[37] TJ Holowaychuk. *mocha: simple, flexible, fun javascript test framework*, 2014. http://visionmedia.github.io/mocha.

[38] Erwin Coumans. *Bullet physics real-time simulation library*, 2014. http://bulletphysics.org/.

[39] K. Kosako. *Oniguruma regular expressions library*, 2013. http://www.geocities.jp/kosako3/oniguruma.

[40] BestieJS. *BenchmarkJS: A JavaScript benchmarking library*, 2014. https://github.com/bestiejs/benchmark.js.

[41] Google. *PepperJS: Pepper applications as Native Client or JavaScript*, 2014. https://github.com/google/pepper.js.

[42] Alon Zakai. *Emscripten: An LLVM-to-JavaScript Compiler*, 2014. https://github.com/kripken/emscripten.