Observer Pattern : Library -> JNI

Setting an observer for asynchronous communication from the library to the JNI wrapper.

```
JNIEXPORT void JNICALL
Java_com_stappone_neolib_NeoLibWrapperImpl_setLibraryResponseMessageObserve
r(JNIEnv *env,

jobject thiz) {
    auto comms = getStapponeCommsHandle(env, thiz);
    jobject observer = env->NewGlobalRef(thiz);
    globalMessageRef = observer;

    JavaVM *jvm;
    env->GetJavaVM(&jvm);

    comms->setMessageObserver([jvm, observer](uint8_t *msgPtr, uint8_t
msgLen) {
        runInJavaEnvironment(jvm, [observer, msgPtr, msgLen](JNIEnv *env) {
            jclass clazz = env->GetObjectClass(observer);
            jmethodID meth = env->GetMethodID(clazz, "onMessageData", "
([B)V");
            jbyteArray jdata = env->NewByteArray(msgLen);

            env->SetByteArrayRegion(jdata, 0, msgLen, (jbyte *) msgPtr);
            env->CallVoidMethod(observer, meth, jdata);
            env->DeleteLocalRef(jdata);
        });
    });
}
```

This is the function signature. It tells the JNI framework that this function corresponds to the Java method setLibraryResponseMessageObserver in the class com.stappone.neolib.NeoLibWrapperImpl. It takes two parameters: env, a pointer to the JNI environment, and thiz, a reference to the Java object that called this method.

```
Java_com_stappone_neolib_NeoLibWrapperImpl_setLibraryResponseMessageObserve
r(JNIEnv *env, jobject thiz)
```

It retrieves a handle (comms) to a C++ object of some communication class by calling the getStapponeCommsHandle function. This handle allows the C++ code to interact with the communication object.

```
auto comms = getStapponeCommsHandle(env, thiz);
```

The implementation of getStapponeCommsHandle.

```
NEOLibrary::StapponeComms *getStapponeCommsHandle(JNIEnv *env, jobject obj)
{
    jlong handle = env->GetLongField(obj, getStapponeCommsHandleField(env,
obj));
    return reinterpret_cast<NEOLibrary::StapponeComms *>(handle);
}
```

It creates a global reference to the thiz object (the Java object that called this method). This global reference ensures that the Java object is not garbage collected while this C++ code is still using it.

```
jobject observer = env->NewGlobalRef(thiz);
```

It assigns the global reference observer to a global variable called globalMessageRef. This allows other parts of the C++ code to access the Java object.

```
globalMessageRef = observer;
```

It gets a pointer to the Java Virtual Machine (JVM) by calling env->GetJavaVM(&jvm). This is necessary to later create a Java environment for running Java code from within C++.

```
JavaVM *jvm; env->GetJavaVM(&jvm);
```

It sets a message observer for the communication object (comms). This observer is a C++ lambda function that takes a uint8_t pointer msgPtr and a uint8_t msgLen as arguments.

```
comms->setMessageObserver(...)
```

Inside the lambda function, runInJavaEnvironment(jvm, ...) is called, which is a utility function in your codebase. It's used to execute a block of Java code within the Java environment.

Inside the `runInJavaEnvironment` block:

- `jclass clazz = env->GetObjectClass(observer);`: It gets the Java class of the `observer` object.
- `jmethodID meth = env->GetMethodID(clazz, "onMessageData", "([B)V");`: It obtains the method ID of the Java method named `onMessageData` that takes a byte array (`[B`) and returns `void` (`V`).
- `jbyteArray jdata = env->NewByteArray(msgLen);`: It creates a new Java byte array of the specified length.

- `env->SetByteArrayRegion(jdata, 0, msgLen, (jbyte *) msgPtr);`: It copies the data from the C++ `msgPtr` into the Java byte array `jdata`.
- `env->CallVoidMethod(observer, meth, jdata);`: It calls the `onMessageData` method on the `observer` object, passing in the `jdata` byte array.
- `env->DeleteLocalRef(jdata);`: It deletes the local reference to the `jdata` byte array to avoid memory leaks.

In summary, this code sets up a callback mechanism in C++ to notify a Java object (`observer`) of incoming messages. It uses JNI to bridge the gap between Java and C++. When a message is received, it copies the message data into a byte array and calls the `onMessageData` method of the Java object, passing the message data as an argument.

runInJavaEnvironment

```
void
runInJavaEnvironment(JavaVM *jvm, std::function<void(JNIEnv *)>
executable) { //convenience function
    runInJavaEnvironment(jvm, NULL, [executable](JNIEnv *env,
    jclass clazz) { executable(env); });
}
```

```
void runInJavaEnvironment(JavaVM *jvm, jobject observer,
                          std::function<void(JNIEnv *, jclass clazz)>
executable) {
    bool attachedHere = false; /* know if detaching at the end is
necessary*/
    try {
        JNIEnv *env;
        if (getAttachedJavaEnvFromJvm(&env, jvm, &attachedHere)) {
            /* Execute code using env */
            jclass clazz = NULL;
            if (observer != NULL) {
                clazz = env->GetObjectClass(observer);
            }
            executable(env, clazz);
            if (attachedHere) { /* Key check */
                /* Done only when attachment was done here */
                jvm->DetachCurrentThread();
            }
        }
    } catch (...) {
        if (attachedHere) { /* Key check */
            /* Done only when attachment was done here */
            jvm->DetachCurrentThread();
        }
    }
}
```

1. `void runInJavaEnvironment(JavaVM *jvm, std::function<void(JNIEnv *)> executable)`: This is the first overloaded version of the function. It takes two parameters:

   - `jvm`: A pointer to the Java Virtual Machine (JVM).
   - `executable`: A `std::function` that represents a block of code which takes a `JNIEnv*` (JNI environment pointer) as its argument and returns `void`.

2. `runInJavaEnvironment(jvm, NULL, [executable](JNIEnv *env, jclass clazz) { executable(env); });`: This line is a call to another overloaded version of `runInJavaEnvironment`, which takes three arguments:

   - `jvm`: The JVM pointer passed from the outer function.
   - `NULL`: This is not used in the provided code.
   - A lambda function that takes a `JNIEnv*` pointer (`env`) and a `jclass` (`clazz`) as arguments. Inside the lambda, it calls the `executable` function with the `env` argument. This lambda essentially adapts the `std::function` interface to match the three-argument version of `runInJavaEnvironment`.

3. `void runInJavaEnvironment(JavaVM *jvm, jobject observer, std::function<void(JNIEnv *, jclass clazz)> executable)`: This is the second overloaded version of the `runInJavaEnvironment` function. It takes three parameters:

   - `jvm`: The JVM pointer.
   - `observer`: A `jobject` (Java object reference). This is typically used to determine the class of the object, allowing the code block to interact with Java methods related to that object.
   - `executable`: A `std::function` that represents a block of code that takes a `JNIEnv*` pointer (`env`) and a `jclass` (`clazz`) as its arguments. The `jclass` represents the class of the `observer` object.

4. Inside the function, there is a try-catch block. The purpose of this block is to handle exceptions that might occur during the execution of the code block.

5. `bool attachedHere = false;`: This flag is used to keep track of whether the current thread is attached to the JVM within this function. It starts as `false`.

6. `if (getAttachedJavaEnvFromJvm(&env, jvm, &attachedHere))`: This condition checks whether the current thread is already attached to the JVM. If it's not attached, the function `getAttachedJavaEnvFromJvm` is called to attach the thread and obtain the `env`. The `attachedHere` flag is set to `true` to indicate that the attachment was done within this function.

7. `jclass clazz = NULL;`: A `jclass` variable is initialized as `NULL`. This will be used to hold the class of the `observer` object.

8. `if (observer != NULL) { clazz = env->GetObjectClass(observer); }`: If the `observer` object is not `NULL`, it retrieves the class of the `observer` object using `GetObjectClass` and assigns it to the `clazz` variable.

9. `executable(env, clazz);`: Finally, the `executable` function is called with the `env` and `clazz` arguments, allowing it to execute within the Java environment and interact with Java objects and methods.

10. After the execution of the code block is complete, there is a check: `if (attachedHere) { jvm->DetachCurrentThread(); }`. If the current thread was attached to the JVM within this function (`attachedHere` is `true`), it is detached from the JVM to clean up the thread attachment.

In summary, the `runInJavaEnvironment` function is a utility that ensures a block of code is executed within the Java environment, handling thread attachment and detachment as needed. It provides the `JNIEnv` to the code block for interaction with Java objects and methods.

Potential Alternatives

1. Java Native Access (JNA)
2. JNI Wrappers
3. Android NDK

JNA

create the Java interface

```
public interface JavaInterface {
    void someJavaMethod(String message);
}
```

create the Java class

```
public class JavaClass implements JavaInterface {
    public void someJavaMethod(String message) {
        System.out.println("Java received: " + message);
    }
}
```

1. Compile the Java class, which generates a .class file.
2. In your C++ code, use JNA to load the Java class and call its methods.

```
#include <jni.h>
#include <dlfcn.h> // On Linux/macOS

typedef void (*JavaMethod)(const char*);

int main() {
    void* jvmLibrary = dlopen("libjvm.so", RTLD_NOW); // Load the JVM library
    if (jvmLibrary == nullptr) {
        // Handle error
        return 1;
    }
```

```cpp
    JNIEnv* env;
    JavaVM* jvm;
    JavaMethod javaMethod;

    // Initialize the JVM and get the JNIEnv
    typedef jint (*CreateJavaVMFunc)(JavaVM**, JNIEnv**, void*);
    CreateJavaVMFunc createJavaVM = (CreateJavaVMFunc)dlsym(jvmLibrary, "JNI_CreateJavaVM");
    if (createJavaVM(&jvm, &env, nullptr) != JNI_OK) {
        // Handle error
        return 1;
    }

    // Load the Java class and method
    jclass javaClass = env->FindClass("JavaClass");
    jmethodID javaMethodID = env->GetMethodID(javaClass, "someJavaMethod", "(Ljava/lang/String;)V");

    // Create an instance of the Java class
    jobject javaObject = env->NewObject(javaClass, env->GetMethodID(javaClass, "<init>", "()V"));

    // Call the Java method
    env->CallVoidMethod(javaObject, javaMethodID, env->NewStringUTF("Hello from C++"));

    // Clean up
    jvm->DestroyJavaVM();
    dlclose(jvmLibrary);

    return 0;
}
```

JNI Wrapper

1. The Java class

```java
// JavaClass.java
package com.example.myapp;

public class JavaClass {
    public void someJavaMethod(String message) {
        System.out.println("Java received: " + message);
    }
}
```

2. C++ wrapper class

```cpp
// JavaWrapper.h
#include <jni.h>

class JavaWrapper {
public:
    JavaWrapper(JNIEnv* env);
    ~JavaWrapper();
    void callJavaMethod(const char* message);

private:
    JNIEnv* env_;
    jobject javaObject_;
};
```

3. Implement the C++ wrapper class

```cpp
// JavaWrapper.cpp
#include "JavaWrapper.h"

JavaWrapper::JavaWrapper(JNIEnv* env) : env_(env) {
    jclass javaClass = env_->FindClass("com/example/JavaClass"); // Replace with your Java class package
    jmethodID constructor = env_->GetMethodID(javaClass, "<init>", "()V");
    javaObject_ = env_->NewObject(javaClass, constructor);
}

JavaWrapper::~JavaWrapper() {
    env_->DeleteLocalRef(javaObject_);
}

void JavaWrapper::callJavaMethod(const char* message) {
    jmethodID javaMethod = env_->GetMethodID(env_->GetObjectClass(javaObject_), "someJavaMethod", "(Ljava/lang/String;)V");
    jstring javaMessage = env_->NewStringUTF(message);
    env_->CallVoidMethod(javaObject_, javaMethod, javaMessage);
    env_->DeleteLocalRef(javaMessage);
}
```

4. create an instance of the wrapper class and call the Java Method

```cpp
#include "JavaWrapper.h"

int main() {
    JavaVM* jvm; // Initialize and set up the JVM as needed
    JNIEnv* env; // Obtain JNIEnv
    // ...

    JavaWrapper javaWrapper(env);
    javaWrapper.callJavaMethod("Hello from C++");
    // Clean up JNI and JVM

    return 0;
}
```

Android NDK

1. Create a Java class with the function you want to call from C++. For example, let's create a simple Android project with a Java class named JavaBridge:

```java
// JavaBridge.java
package com.example.myapp;

public class JavaBridge {
    public static void javaFunction() {
        System.out.println("Java function called from C++");
    }
}
```

2. In your CMakeLists.txt file, include the necessary NDK module, and specify the source files for your C++ code. For example:

```cmake
cmake_minimum_required(VERSION 3.10)

project(MyNDKApp)

# Specify the minimum version of the NDK your project depends on
set(CMAKE_CXX_STANDARD 14)
set(CMAKE_CXX_STANDARD_REQUIRED ON)

# Specify the Android platform and version
set(CMAKE_SYSTEM_NAME Android)
set(CMAKE_ANDROID_NDK "path/to/ndk-bundle")
set(CMAKE_ANDROID_STL_TYPE "c++_shared")
set(CMAKE_ANDROID_API_MIN 21)

# Add your C++ source files
add_library(
    mynative
    SHARED
    mynative.cpp
)

# Link the required libraries (e.g., log)
target_link_libraries(
    mynative
    android
    log
)
```

3. Write your C++ code that will call the Java/Kotlin function. In this example, we'll create a file named mynative.cpp:

```
#include <jni.h>
#include <android/log.h>

extern "C" {

// Declare a JNI function to call the Java/Kotlin function
JNIEXPORT void JNICALL
Java_com_example_myapp_JavaBridge_callJavaFunction(JNIEnv* env, jobject thiz) {
    jclass clazz = env->FindClass("com/example/myapp/JavaBridge"); // Replace with your package and class name
    if (clazz != nullptr) {
        jmethodID methodID = env->GetStaticMethodID(clazz, "javaFunction", "()V");
        if (methodID != nullptr) {
            env->CallStaticVoidMethod(clazz, methodID);
        }
    }
}

} // extern "C"
```

4. Android Side

```
// MainActivity.java (or any other suitable class)
package com.example.myapp;

import androidx.appcompat.app.AppCompatActivity;
import android.os.Bundle;

public class MainActivity extends AppCompatActivity {

    static {
        System.loadLibrary("mynative");
    }

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        // Call the JNI function to invoke the Java/Kotlin function
        JavaBridge.callJavaFunction();
    }
}
```