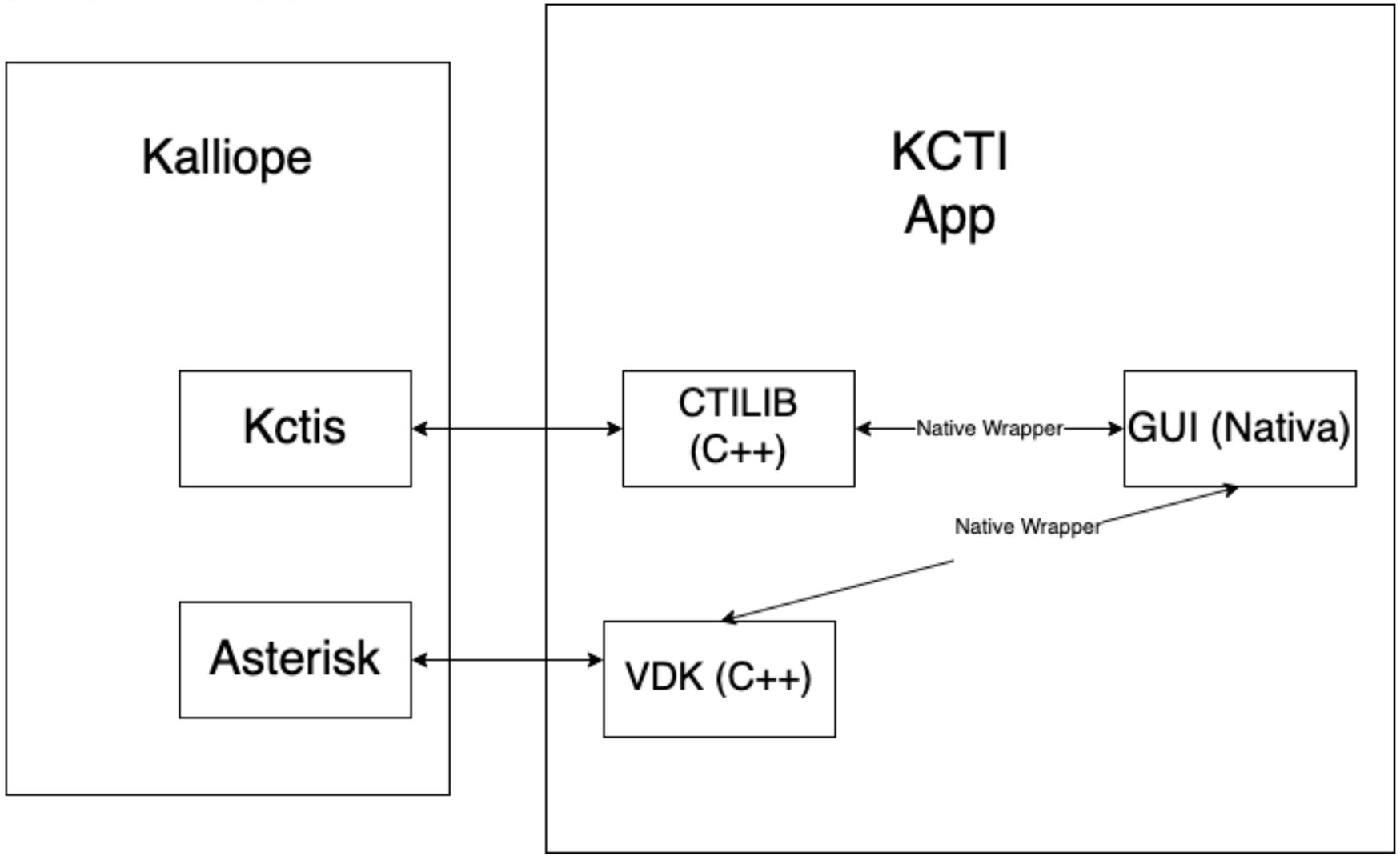
# Kalliope CTI Mobile

## Architecture & Technical solutions

**Francesco Lamonica**

# Architecture

## KCTI Common architecture



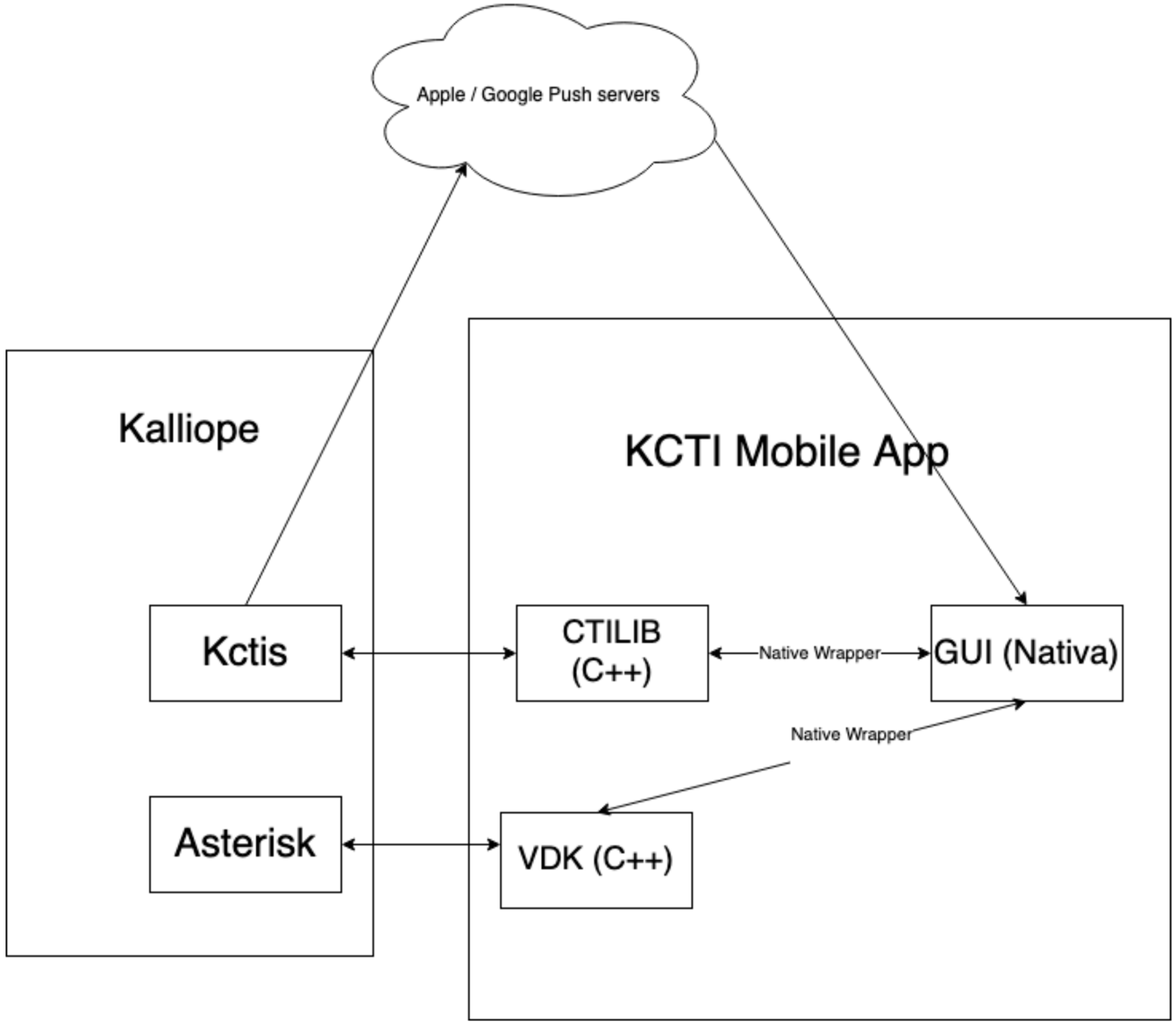
# Architecture

## Differences with desktop

- Mobile OSes have their own quirks
  - battery saving
  - connection roaming
  - notifications to wake up apps
- Lack of Qt Signals / Slots (architectural choice to go native)

# Architecture

## Mobile



# No (Qt) Signal

# Lack of Signal & Slots

## Going native (or non-Qt)

- Synchronous vs Event-driven flow-control
- Qt Signals
  - Every Qt Application has an “event loop” thread that dispatches signals to their connected slots
  - Calling an async method is as simple as:
    - Defining the connection between an event and what to do when event occurs: “connect(srcobj, eventname, destobj, func2)”
    - Triggering the event in the “srcobj”: “emit eventname”

# Lack of Signal & Slots

## Going native (or non-Qt)

- function callbacks (not present in every language)
- Old-style IPC
  - Localhost server listening in our Qt libs / objects
  - Symmetric-key encryption (optional)
  - Synchronous API to get the listening port
  - Native language socket reading code (eventually wrapped in native language high-level async structures: i.e. Specific Java Listeners)

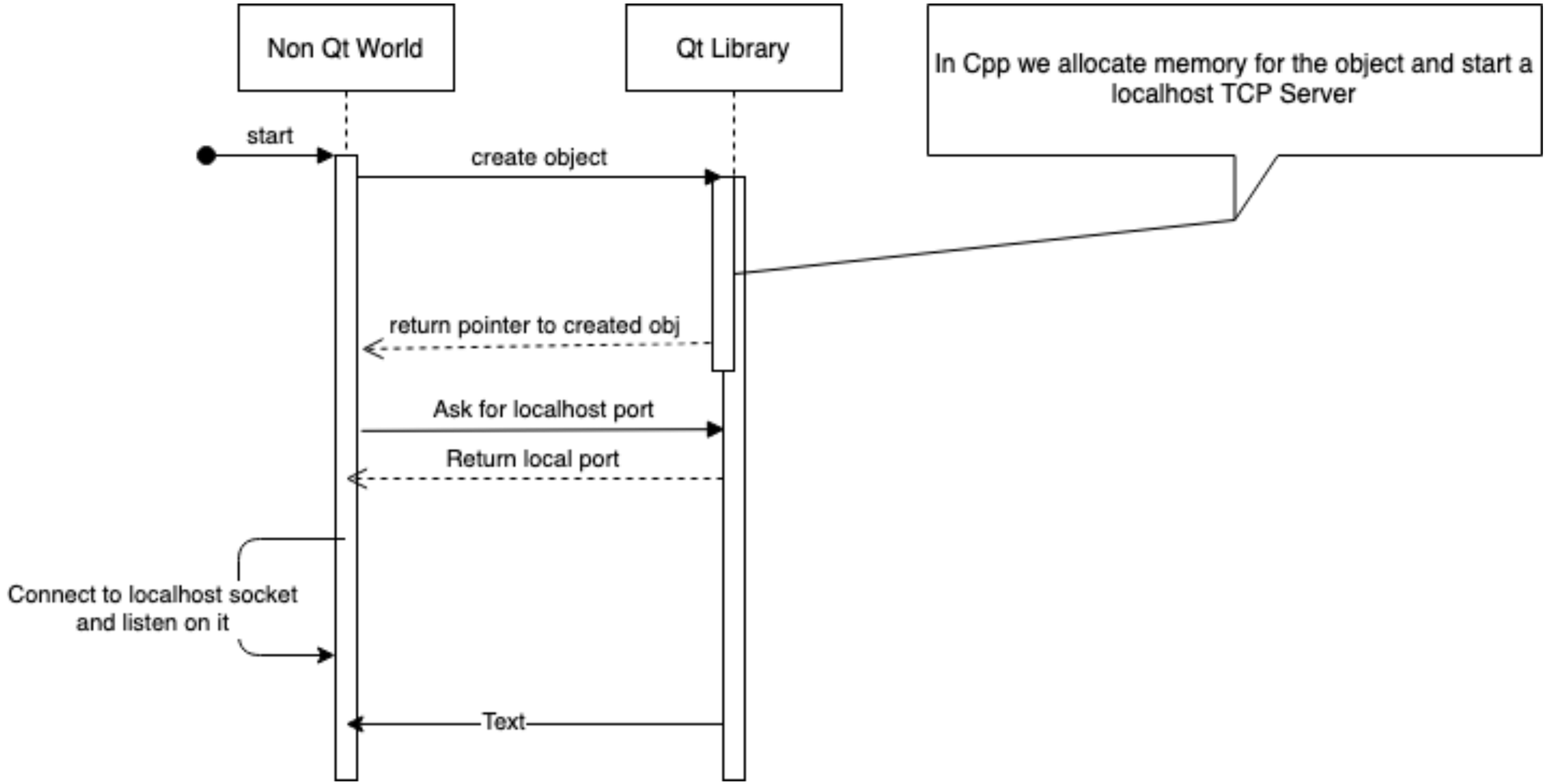
# Lack of Signal & Slots

## Going native (or non-Qt)

- Are we losing something else?
  - No Application -> No QEventLoop
  - Our Qt-libs will not function properly in native apps!!!
- How we deal with a missing QEventLoop?
  - We try to detect missing QtApplication and create one accordingly
  - What if there are more than one library?
  - Beware of ephemeral params (read the docs / chagelogs!)

# Lack of Signal & Slots

## Going native (or non-Qt)



# Questions?

# Native wrapping

# Native wrapping

## Android: Java / JNI

- Java Native Interface
  - java -> C / C++ / asm and vice versa
- Adjust native code to be called
- Write Java code to call the native one

# JNI

## Java side

- Write a package / class that handles the native code calling
  - loadLibrary(“vdk”); //automatically chooses the right prefix and suffix (libvdk.so / libvdk.dylib /  vdk.dll)
  - define the “native” methods to be called
    - they will be called by this package / class
    - they will be searched in the native libraries loaded

# JNI

## hello world (java side)

```
package it.netresults.test;

public class HelloWorldJNI {

    static {
        System.loadLibrary(“helloworld"); //libhelloworld.so
    }

    public static void main(String[] args) {
        new HelloWorldJNI().hello();
    }

    private native void hello(); //this will be called in c space
}
```

# JNI

## C++ side

- We need to link some specific code to our c++ code

- Basically we need C-style prototype functions to be called

- include jni.h (that provides functions and types to map the 2 worlds)

# JNI

## hello world (c++ side)

Header file

```
JNIEXPORT void JNICALL Java_it_netresults_test_HelloWorldJNI_hello(JNIEnv*, jobject);
```

Implementation file

```
JNIEXPORT void JNICALL Java_it_netresults_test_HelloWorldJNI_hello(JNIEnv*, jobject)
{
  std::cout << "Hello world C++" << std::endl;
}
```

# JNI

## welcome to the real world

- Problems of helloworld
  - hello() didn’t have parameters
  - return type was void
  - code called was more C than C++ (no objects and more importantly there was no internal state to be kept.)

# JNI

## parameters

- POD parameters are mostly mapped 1-1 (see oracle documentation)
  - int -> jint
  - bool -> jboolean
  - long -> jlong
- Other types (i.e. strings need more “manipulation”)

# JNI

## Dealing with state and c++ objects

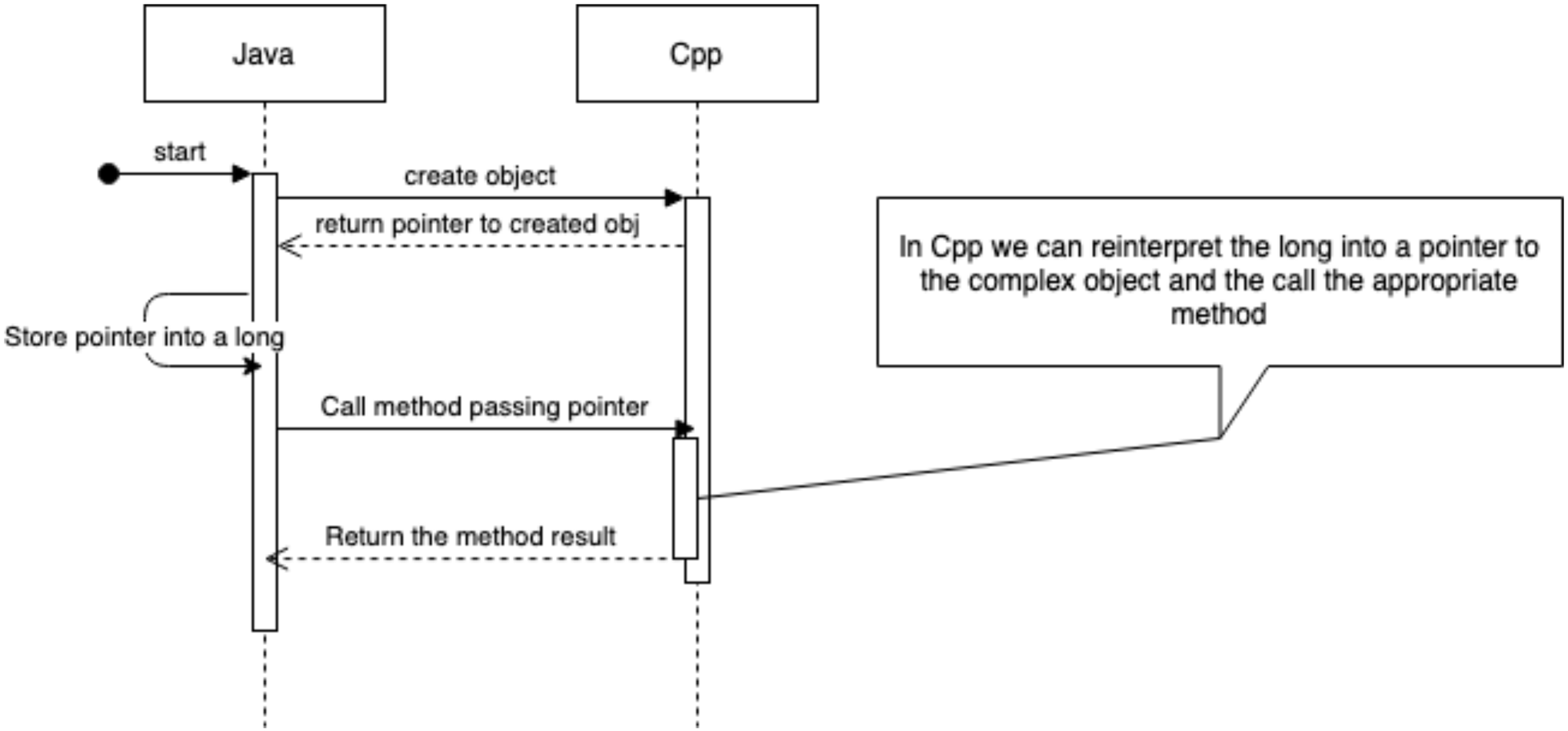
- What shall we do to create and keep around complex objects and their states (including socket connections, db, etc.)

- Create the object and pass the pointer to be kept in Java class as a simple memory address (number)

- And when we need to call a method on the complex object wrap it passing back the pointer from Java to Native

# JNI

## Dealing with state and c++ objects



# JNI

## VDK real example

- VDKJEngine.h / VDKJEngine.cpp (compiled in VDK library)
  - contain all the wrapped VDK methods
  - dynamically generated (wrappergen script)

- VDKJavaEngine.java (class used in Android)

# Questions?