JAVA INTERFACE FOR THE TRAJECOTRY SYNTHESIZER

Danish Vaid

Universities Space Research Association
Summer 2015

Table of Contents

What is the Trajectory Synthesizer (TS)?

TS is a generation software written and used for CTAS to generate trajectories for scheduling and conflict prediction/resolution. TS operates by taking in input files and generating trajectories from those, currently in C/C++.

Task and Motivation

My main task was to research and develop approaches for Java to access C++.

My motivation for this task was to create a Java interface to access the TS software that would allow Java research software platforms (ex. ACES, FACET, etc) to use the TS as an alternative trajectory generator.

Task Stages

My Task stages were as follows:

Stage 1:	Research different tools/libraries
Stage 2:	Test research results, learn, and determine best option
Stage 3:	Inital prototyping
Sub - Stage 3.1:	Pass TsInput file name and process
Sub - Stage 3.2:	Prototype different data structure types
Stage 4:	Design and Implementation (*Create TsInput Java Object and pass it into C++ TS)*
Sub - Stage 4.1:	TS Class to Struct conversion
Sub - Stage 4.2:	Java (JNA) declaration and linking
Sub - Stage 4.3:	Struct to Class Constructor
Sub - Stage 4.4:	Return results to Java side

Stage 1: Possible Tools/Libraries

I looked into 5 main interface libraries for my project: Google Protocol Buffers (GPB), BridJ, Java Native Interface (JNI), JavaCPP, Java Native Access (JNA).

Google Protocol Buffers (GPB)

Google Protocol Buffers are a language-neutral, platform-neutral extensible mechanism for serializing structured data, just as XML serialization but smaller, implementation-aly faster, and simpler.

It works off of you (the user) defining how you want your data to be structured, then you write that structure and Google's "proto-buff" language and use that special generated source code to easily write and read your structured data to and from a variety of data streams and using a variety of language.

I chose not to follow this choice because it would be slow, working off of XML type serialization and deserialization, it would require hand maintenance of 4 separate layers of connections (source code, proto-buff language, and the constructors and de-constructors for sending to and from the buffer socket). These and the fact that GPB serialization is in binary wire format that is not self-describing (meaning without an external specification in the listening socket that stream would be worthless) lead me to not use GPB.

BridJ

BridJ is a JNA inspired layover to JNI meant to be a fast and very usable interface between Java and C++. In its current state it provides limited support for C++ classes and subclasses (inheritance inclusive), however the project page states their main goal is to provide complete support.

This would have been the easiest and best choice to use for our project, however it is mainly theoretical at this point and their documentation is incomplete at best. This project is widely untested and I did not find much support for it. These would all make maintenance too difficult and this path had not guarantee of working since there were no cases of it ever being used to wrap something as extensive as TS.

Java Native Interface (JNI)

Java Native Interface (JNI) is a programming framework that enables Java code running in JVM to call and be called by native applications in C/C++. JNI enables programmers to write native methods to handle situations when an application cannot be written entirely in the Java programming language, e.g. when the standard Java class library does not support the platform-specific features or program library.

JNI requires a specific, convoluted, syntax on the C side to use the native function, you must use the framework's header generator to manually create header links and then declare the java code. Upon this, one of the biggest disadvantage for JNI was that it does not support class or struct passing, meaning that all of the data of a TS object would have to be destructed to primitive types and passed in, either one-by-one or through arrays (both ways being inefficient and horrible implementations).

JavaCPP

JavaCPP is a JNI complement built to make C++ code available to Java without having to constantly re-type the overhead command that are demanding. It provides a quick way to call existing native libraries, by exploiting the syntactic and semantic similarities between Java and C++.

This option really only makes the syntax a bit easier when compared to JNI, it does not really solve anything else and would still pose many of the same issues to us that JNI posses.

Java Native Access (JNA)

Java Native Access (JNA) is a community developed library that provides Java programs easy access to native shared libraries without using JNI. It uses foreign function interfaced libraries to dynamically invoke native code. Basically, it uses native functions allowing code to load a library by n ame and retrieve a pointer to a function/struct/C object.

For this method the developer uses a Java interface to describe funtions and structures in the target native libraries and JNA handles the linking/mapping automatically.

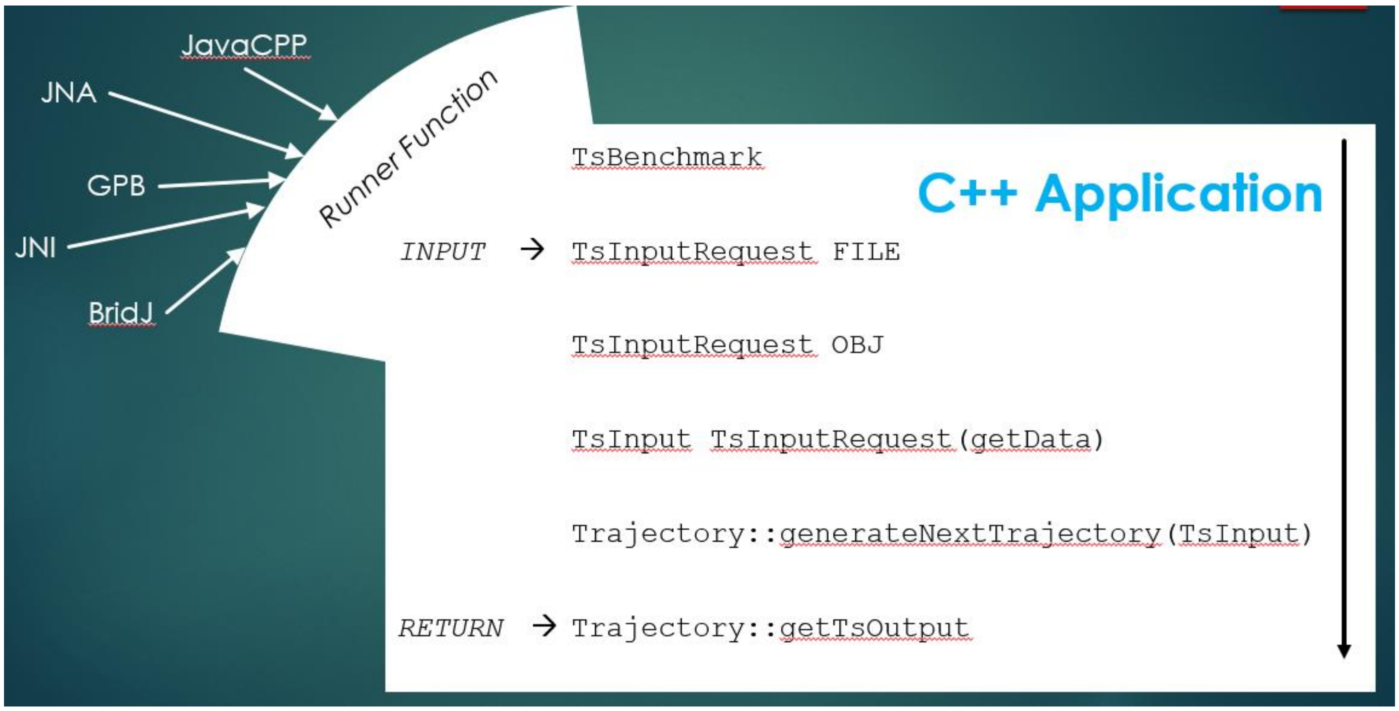
Summary of Possibilities

At this point in my project JNI and JNA were the top most contestants for my tools to write this interface. The summary of all of the above mentioned tools ease and performance is;

Implementation Ease: JNA > GPB > JavaCPP/JNI

Performance/Speed: JNI > JavaCPP > JNA > GPB

Data Flow Chart


JavaCPP
JNA
GPB
JNI
BridJ
Runner Function
C++ Application
TsBenchmark
INPUT → TsInputRequest FILE
TsInputRequest OBJ
TsInput TsInputRequest(getData)
Trajectory::generateNextTrajectory(TsInput)
RETURN → Trajectory::getTsOutput

The chart above displays my initial plan for the interface. My chosen method of communication would access a runner function that would call my C++ application.

My C++ application would start by creating a TsBenchmark instance, taking the inputted file and parsing it to populate a TsInputRequset object that would then be used to generate and return a trajectory.

JNI vs JNA

Having written my C++ application, my next step was to pick my chosen method of interface from my top 2 contenders: JNI and JNA. Their differences are shown below:

JNI	**JNA**
Framework enables code running in JVM to access C/C++ code	Community developed layover to JNI
Allows Native Method access	Uses foreign function native interface for dynamic invocation using proxying
Mapped through (machine generated) header file	Mapping handled automatically during declaration
Does not support custom object passing	Allow C structure development and passing
Runs in native JVM	Requires running through JNA in JVM *(. . . jna.jar:. filename)*
	Some reports say JNA is 10x slower than JNI at large matrix data structures

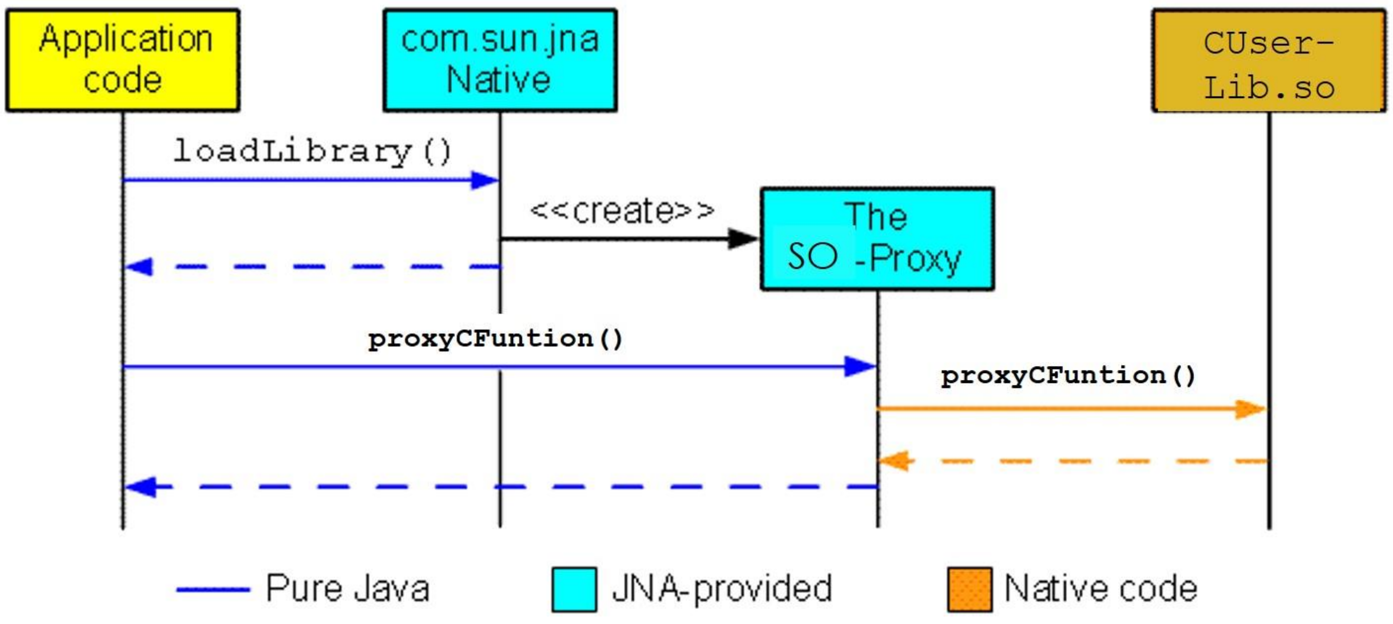
This comparison lead me to choose JNA over JNI because even though JNI is faster and runs in the native JVM, JNA handles mapping and all run-time aspects by itself, allows struct passing, and is easier to implement and maintain.

What is a wrapper?

A wrapper is an interface method that encapsulates the functionality of another component. It is designed to provide a level of abstraction from underlying application.

I like to think of a wrapper acting as a bridge. As if you have the Java environment on one side of a river and the C/C++ side of the other side of the river. The wrapper acts as a connection between the two that allows stuff to go back and forth between the two environments.

How does JNA work?



JNA works off of a proxy pattern, a software design pattern that is a class functioning as an interface to something else. The proxy design pattern allows you to provide an interface to other objects by creating a wrapper class as the proxy. The JNA obtains the proxy-ed object and methods from SO (shared object file). JNA makes proxy linking easier by automatically handles all run-time aspects. The only caveat requirement is that the code must extend: com.sun.jna.Library.

Stage 2: JNA Basic Example

C:

```
int  example1(int val)
{
    return val * 2;
}
```

JAVA:

```
Public interface Clibrary extends Library{
    public int example1(int val);
}
Clibrary clib = (Clibrary)Native.loadLibrary("testlib", CLibrary.class);
int newVal = clib.example1(23);
System.out.println("example 1: " + newVal);
```

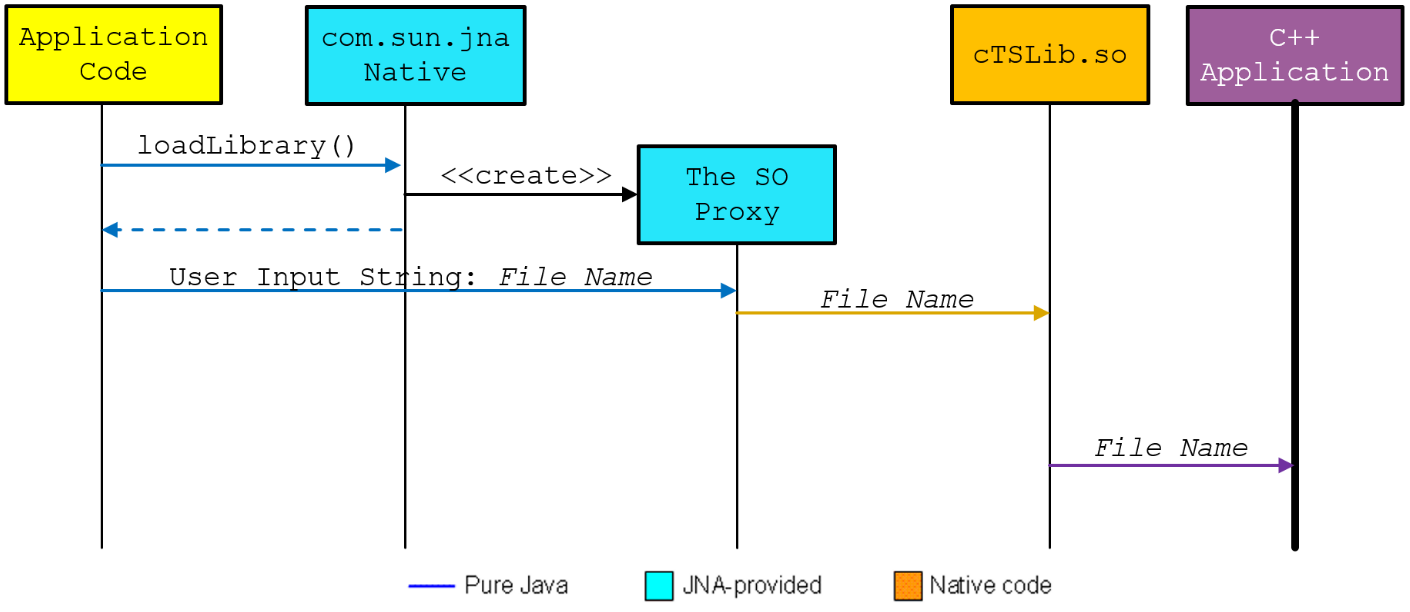
The diagram above shows an example C code function and its corresponding Java (JNA) declaration. The three steps below show compilation and execution:

```
gcc – o libName –shared fileName.c          → Create shared Object
Javac –classpath jna.jar fileName.java      → Compile .java files
Javac –classpath jna.jar:. filename         → Run Java Classes
```

Bash File Written for .so Linking and Running

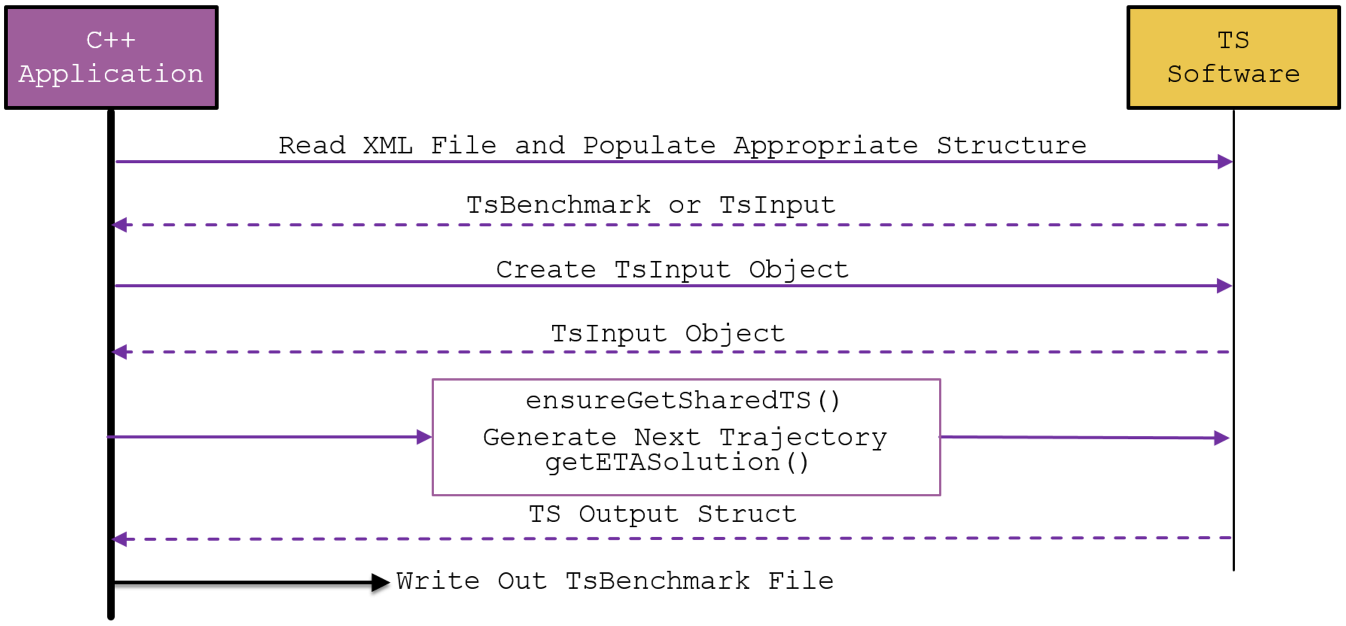
I have written an automated bash file that uses the TS shared library, imports JNA packages in libTS.so, compiles the Java files with JNA and the necessary flags and links that to .so and then runs the classes with the JNA environment on top of JVM.

Stage 3: Develop Java Wrapper Passing TsInput File Name String to TS



This is the Java side of my initial JNA wrapper milestone. This wrapper implementation takes in a filename (location) as a string input in the Java side and passes that to the C side. This is shown in the graph above.

The application begins by loading a dynamic library up using JNA that creates the SO proxy. Then the application sends the user inputted file name to the SO proxy that then forwards the string and function call to the shared object library that then invokes the function call in the C/C++ side with the string that has been passed through.



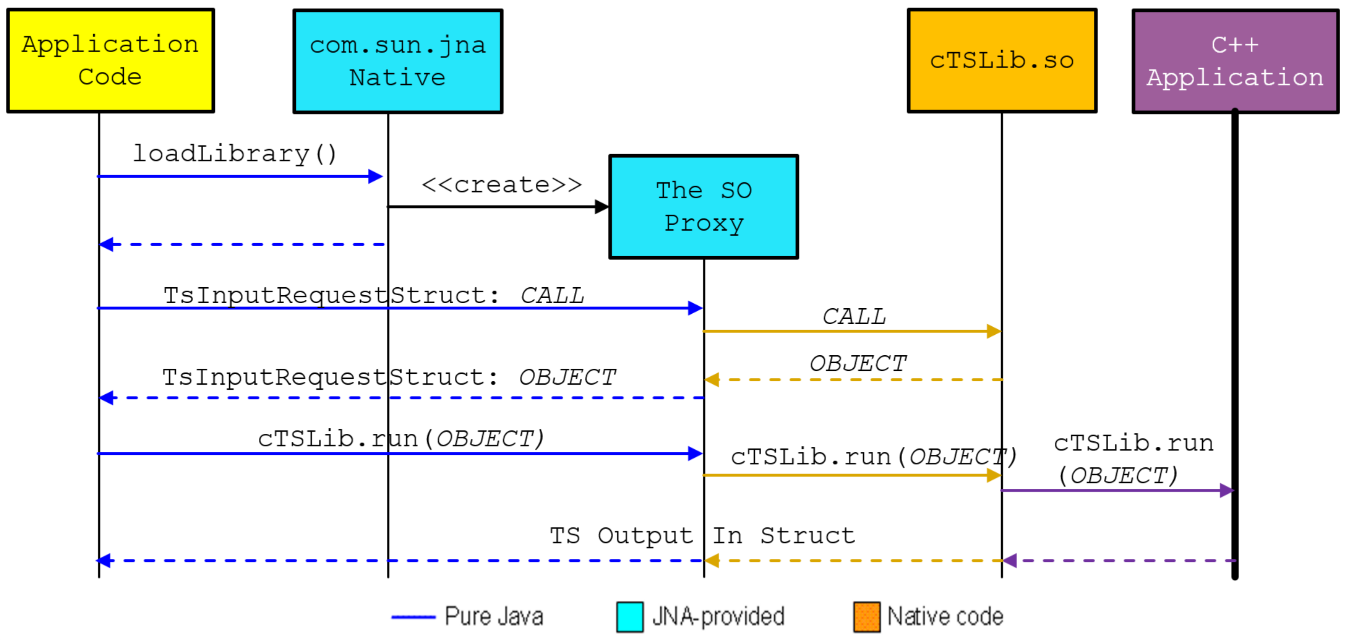
The diagram above shows the C/C++ application of my initial wrapper. This side of the wrapper does all of the hard lifting at this stage. It takes in a file name, loads the file and parses it to populate the appropriate structure and get that return. It then creates a TsInput Object from that returned structure and using that object generates the trajectory. Once it gets the trajectory back it writes out a TsBenchmark XML file.

Stage 3.2: Prototype Different Data Structure Types

In this stage I tested and implemented many different data structures to see what JNA could handle and how to implement those. I tried and JNA successfully handled the passing and returning of:

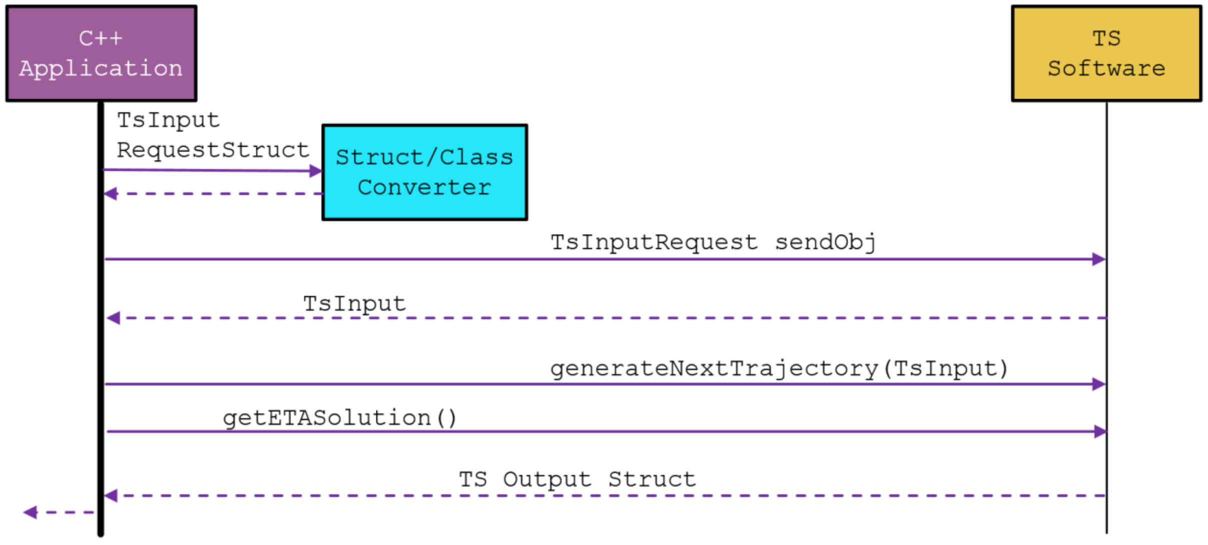
- Plain old data types (int, float, double, etc.)
- C Strings
- Arrays of PODs
- Arrays of Structs
- Unions
- Arrays of Unions
- Arrays of Structures containing Unions

Stage 4: Create TsInput Java Object and Pass it into C++ TS



This is the Java side of the main goal of my project. To create a TS object in the Java side, populate it, and pass it through to the C++ TS application. This application starts by loading up the dynamic library which causes JNA to create the SO proxy link. Next I must have a constructor call to create a TsInputRequestStruct which get forwarded from the Java application to the SO proxy, and then to the shared object library. Which creates a linked TsInputRequestStruct object and returns it. Next the java application send this TsInputRequestStruct object with a call to run my C application to the SO proxy that forwards it to the shared object library, which in turn calls the C++ application with the passed in object. The C/C++ side of the wrapper will be explained next but as the bottom of the diagram shows, the C/C++ application returns a TS Output In Struct.

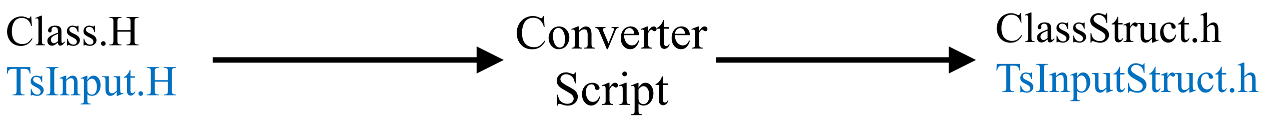
A main point to notice in JNA is that it only supports passing of C structs, not C++ structs or classes, so the TS data structures must be converted to C structs and then my application will require a constructor from struct to class.



This is the C/C++ side of the final wrapper. It starts by sending the inputted Struct through a converter/constructor to create a appropriate TS class object. Then my application passes that request object to the TS software to get back the TsInput; which is in turn sent to the TS software to generate a trajectory. The result of said trajectory generation is returned and sent back to the Java side through the shared object, which forwards it to the SO proxy, which returns it to the Java application.

Stage 4.1: TS Class to Struct Conversion

As mentioned above, JNA only allows passing of C structs, not C++ structs or classes, so we must start by converting all of TS data members into C structs. This is a daunting and time consuming task and would make maintenance unnecessarily more convoluted. However, I wrote a script that does the conversion for you. It works off the following form:



My conversion script handles file instantiation and uses the regular expressions module for the line-by-line logic.

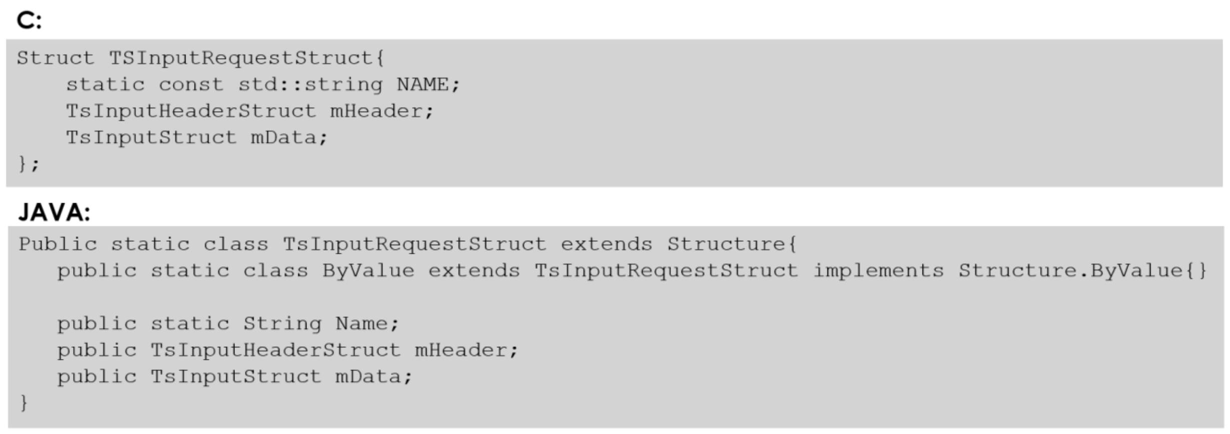
My script's step outline is:

1. Import necessary modules
2. Open original and temporary writing copy files
3. Compile regular expressions code
4. Set up line loop for file
5. Split away the "//" comments
6. Take away lines with function operators "()"
7. Copy data
8. Close files
9. Load up temporary file into memory
10. Get rid of all lines with pure whitespace and copy to final file
11. Close final output fiel and temp file
12. Delete temp file
13. Print success and end of script

Stage 4.2: Java (JNA) Declaration and Linking

This part of my project was to write all of TS's data member (struct version) in the JNA-Java declaration for JNA linking and proxy creation. Basically you have to write all of the structs (we created in the last step) in JNA's format for Java.

Example:



C:

```
Struct TSInputRequestStruct{
    static const std::string NAME;
    TsInputHeaderStruct mHeader;
    TsInputStruct mData;
};
```

JAVA:

```
Public static class TsInputRequestStruct extends Structure{
   public static class ByValue extends TsInputRequestStruct implements Structure.ByValue{}

   public static String Name;
   public TsInputHeaderStruct mHeader;
   public TsInputStruct mData;
}
```

Stage 4.3: Structs to Class Constructor

Since TS operates off of its defined classes that we turned into structs for passing through to our C/C++ application, we now have to have a constructor to create a class version of that object for the passed in struct. This should be one function that handles all of the depend data members. An example of a struct constructor/converter is:

C/C++:

Struct Defined as:

```
    Struct TsInputRequestStruct{
        static const std::string NAME;
        TsInputHeaderStruct mHeader;
        TsInputStruct mData;
    };
```

Class Constructor:

```
    TsInputRequest structToClass(TsInputRequestStruct inp){
        return  TsInputRequest():
                NAME(inp.NAME)
                mHeader(inp.mHeader)
                mData(inp.mData);
    }
```

Current Progress

Completed

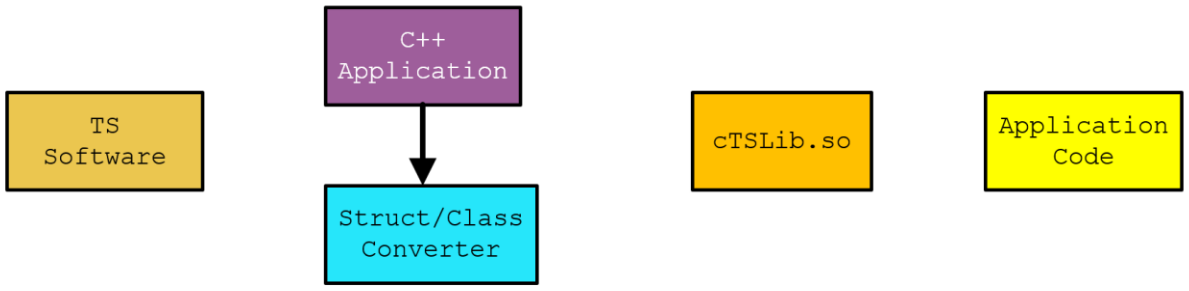
All of Stage 1, Stage 2, and Stage 3 have been completed; Sub - Stage 4.1 has also been completed. We know the tool we want to use, JNA, it has been using for an initial wrapper, its been tested for all data types, and we have completed designing the final wrapper. I ran my conversion script to change TS classes into C struct data members. However, I have run out of time and have not been able to finish the rest.

Still Needs to Be Done

There are still TS data members in struct that needs to be declared and defined in Java using JNA syntax. After that, the struct to class constructor/converter must be written so that TS can accept our input type. And lasted, the wrapped must be tested in its entirety for returned results to java side and be de-bugged completely.

Code Maintenance.

The stages to be re-followed for any code changed/maintenance are Sub – Stages 4.1 to 4.3. Any changed in one of the layer boxes below must be reflected in the others to maintain the correct linking and usage.


TS Software
C++ Application
Struct/Class Converter
cTSLib.so
Application Code