Swift-Tcl

Peter da Silva

Tcl 2016

October 24, 2016

Bridging between Swift and Tcl

This talk will examine Swift Tcl, a bridge between Swift and Tcl, providing deep interoperability between the two languages. Swift developers can use Tcl to make new and powerful ways to interact with, debug, construct automated testing for, and orchestrate the high level activities of their applications. Tcl developers can use Swift to gets its sweet, expressive, high performance, scripting-language-like capabilities wherever they would like while retaining all of their existing code. Developers can extend Tcl by writing new commands in Swift and extend Swift by writing new commands in Tcl. Tcl commands written in Swift are tiny and simple compared to those written in C. They are a joy.

Why Swift? - Swift vs C and Swift vs Tcl.

Swift is a compiled language, like C, so it's not suited to dynamic and interactive environments where program logic is in flux... but unlike C it's type-safe and can be written in such a way as to eliminate virtually all of the kind of run-time errors that C is notorious for, such as type mismatches, null pointer dereferencing, and of course array bounds errors like buffer overruns.

As a compiled language, it's very fast: not as fast as C, due to unavoidable run-time memory checks, but even in basic string handling it's faster than Tcl and for numeric code it's orders of magnitude faster. This means it's easily fast enough to write native Tcl extensions.

Swift can parse and interpret native C include files and does a very good job of bridging to native C libraries, like the Tcl runtime, and other libraries one might want to call from Tcl. This makes it an excellent glue between Tcl and the underlying OS.

Performance of Swift, Tcl, and C code.

Primarily to get a feel for the performance and limitations of the Swift environment, we have created a Swift library that provides similar functionality to the Tcl "speed tables" package. The weaker introspection and reflection capabilities introspection in Swift means that the database-like API can't be effectively implemented, but the core functionality of the indexed tables was possible using Swift generics.

The performance of Swift is good, but the overhead of Swift's memory management was considerable. A fully managed Skip-list implementation in Swift 3 is about 16 times slower than the optimized C. This overhead is due to a couple of causes.

First, walking skip-lists using generic managed objects performed an implicit retain and release as each list element was brought into scope and left it again. Switching to the use of Unmanaged and Unsafe interfaces, and abandoning Swift type safety, we were able to reduce the overhead to about a factor of four.

Second, Swift performs a lot of bookkeeping in the allocation and deallocation of generic objects that we were able to avoid in our native speed tables by generating custom C code for managing each speed table. This overhead is probably unavoidable in any managed language.

Design of Swift Tcl.

The top level of a Swift-Tcl application is a Swift program that imports or creates a Tcl interpreter, and hands control over to Tcl when necessary using eval. There is no mechanism at present for loading it as a Tcl package: this is partly a result of the current state of development and the use of the Apple XCode environment. Once we can host Swift in our server environment that may change. Getting Swift running under FreeBSD 11 is a current project.

Swift Tcl is exposed to the user through three basic classes - the interpreter (TclInterp), the Tcl object (TclObj), and a utility class for dealing with arrays (TclArray).

The TclInterp class.

Swift-Tcl provides deep access to Tcl object from the Swift environment. The main interface is through the TclInterp class. Creating a new interpreter object can produce a new Tcl interpreter:

```
let interp = TclInterp()
```

Or we can pass an already created interpreter to the Swift environment:

```
let interp = TclInterp(rawTclInterpreter)
```

The TclInterp class provides access to variables and arrays, the interpreter's result object and error code, and allows the execution of Tcl code and the creation of Swift extensions in the interpreter.

The TclObj class.

The TclObj class encapsulates a Tcl object. Every TclObj object is created from a Tcl interpreter and maintains a reference to its corresponding TclInterp object. While there are operations on the underlying Tcl_Obj that could be implemented without an interpreter, error handling and management are more complex without one. An earlier version of the library allowed for interpreter-less objects, but it was more cumbersome to use and we really couldn't see enough applications for the capability to keep it.

A TclObj can be initialized or assigned from a number of Swift types: not just simple types like strings, integers, and floating point values, but also arrays, dicts, and sets of these. For example:

```
let list = interp.newObject(["Larry", "Curly", "Moe", "Shemp"])
```

This initializes a Tcl object containing the list `{Larry Curly Moe Shemp}`.

```
list.set("Groucho Harpo Chico Karl")
```

This initializes a Tcl object containing the string `"Groucho Harpo Chico Karl"` this can of course be treated as a list and Tcl will transparently create a list version of the object behind the stage:

```
print(list[0] as String?); // prints "Optional(Groucho)"
```

What's that "String?" and "Optional()" business?

Swift has a really amazing approach to errors and undefined values. It's kind of obsessive about them, and it takes a while to get used to it, but once you're used to it, it really makes it easier to write safe code.

Every error generates an exception, there are no (ahem) exceptions to this. Every function or method that might generate an exception is flagged with the "throws" keyword. When calling such a function you have to flag the call with a "try":

```
// scarycall1() may throw an exception
do {
    try scarycall1() // try calling it
} catch error {
    // deal with the error
}

// scarycall2 returns an optional result instead of throwing
// an error
let optvaluevalue = try? scarycall2() // may produce an optional

// If you know that at this point in the program
// scarycall2() can never fail, you can unconditionally
// unwrap the result. This is not recommended, since if you
// were wrong you get a runtime error at this point.
let stringvalue = scarycall2()!

// Finally, you can also ignore the possibility of scarycall1()
// producing an error. Again, not recommended, because if you
// guess wrong the results are bad.
try! scarycall1()
```

In Swift, there are no "null" values: rather any variable, argument, or function return that might return an "unknown" value returns an `Optional`. An optional basically wraps a value as a Swift enumeration.

Swift enumerations are a bit more complex than C enumerations, and are more akin to a typed C union, where each case of the enumeration can have associated values. In this case, an Optional variable has the possible values `.None` (effectively a NULL) or `.Some(T)` where `T` is some type. Thus, an optional of type `T` would be equivalent to this enumeration:

```
enum Optional<T> {
    case None
    case Some(T)
}
```

Since it's possible for a Tcl list index to fail (for example if you pass something that isn't a list to lindex), indexing a Tcl list as an index to a Swift object returns an `Optional(String)`, that is an `Optional` where the type (`T`) is `String`, written as `String?` and having the possible values `.None` or `.Some(String)`. To convert this to an actual value the simplest way is to conditionally unwrap it, generally using this idiom:

```
// If value is not .None, then extract the wrapped String
// and assign it to the contant "s"
if let s = value? {
      // do something with the String s, now safely unwrapped.
}
```

The TclObj class declares most of its functions to throwing exceptions, since that preserves any Tcl error that was generated, but a few cases where that's not practical it returns an optional instead.

The TclArray class.

The TclArray class encapsulates an array. It allows Tcl arrays to be used in Swift as if they were Swift arrays.

```
let brothers = try? interp.newArray("brothers", ["Marx": "Groucho
Harpo Chico", "Smothers": "Tom Dick"])

print(try brothers.names())
// prints ["Marx", "Smothers"]
print(brothers["Smothers"] as String?)
// prints Optional("Tom Dick")
```

Since the elements of the array are actually Tcl objects, they can be treated as lists even if the original list was imported from Swift as a string.

```
let list = brothers["Marx"] as TclObj?
print(list![0] as String?) // prints Optional("Groucho")
```

Calling Tcl from Swift.

The simplest way to call Tcl is through "eval". This can be done naively, simply constructing a Tcl command as a string the way scripting languages and interpreters have been embedded since the '60s and '70s. But eval also accepts an array as an argument which will be passed directly to Tcl_EvalObj as a list without being converted to a string and re-parsed:

```
try eval(["puts", "String containing unbalanced {{braces}"])
// prints "String containing unbalanced {{braces}"
```

In addition, we have Tcl code that generates direct Swift wrappers for calling Tcl procs with typed arguments, avoiding a trip through the Tcl string interpreter and shimmering the arguments. It infers the type of arguments from things like the default values of a proc, for example, feeding it:

```
proc import_file {file {first 1} {step 1}} {
    ...
    return $result
}
```

Generates:

```
func tcl_import_file (springboardInterp: TclInterp, file: String,
first: Int = 1, step: Int = 1) throws -> String {
    let vec = springboardInterp.newObject()
    try vec.lappend("import_file")
    try vec.lappend(file)
    try vec.lappend(first)
    try vec.lappend(step)
    Tcl_EvalObjEx(springboardInterp.interp, vec.get(), 0)
    return try springboardInterp.getResult()
}
```

Which can be inserted into a .swift file and imported into your project.

Calling Swift from Tcl

While calling Tcl from Swift is primarily about passing objects to the interpreter, calling Swift from Tcl jumps straight into actually creating a new Tcl extension that provides a new native Tcl command.

A swift function called from Tcl will be called with two arguments: a Tcl interpreter, and a list of Tcl objects. Its return value can be any of Int, String, Double, Boolean, or a Tcl object or a Tcl return code. The function and its name is passed to `interp.createCommand`, which creates a new Tcl command using Tcl_CreateObjCommand.

The Swift function is not called directly from Tcl. Instead, the function and its inferred type is stored in a TclCommandBlock which gets passed as a ClientData pointer, along with a the wrapper function swift_tcl_bridger, to Tcl_CreateObjCommand.

Let's start with a typical example Tcl extension, a function to average a list of numbers. Here's the Swift code:

```
func avg (interp: TclInterp, objv: [TclObj]) -> Double {
    var sum = 0.0
    var num = 0
    for obj in objv[1...objv.count-1] {
        if let val: Double = try? obj.get() {
            sum += val
            num += 1
        }
    }
    return(sum / Double(num))
}
```

And our typical example Tcl extension has typical Tcl extension syntax and semantics. The argument list starts with the function name, so we skip over that and add up the numeric arguments in the list, and divide by the number of numeric arguments we found.

The function is added to the Tcl interpreter with the Swift call to createCommand:

```
interp.createCommand(named: "avg", using: avg)
```

This is implemented as a wrapper around TclCreateObjCommand, that comes in a set of variants each accepting a slightly different typedef that maps to a function like `avg` returning a corresponding type. These are matched by Swift's function overloading, which is also used to select the correct init function for the TclCommandBlock structure. The function takes a name and a reference to a function (declared with the `@escaping` keyword to let the compiler know that the function reference may still be called after the function `Interp.createCommand` returns).

```
public func createCommand(
    named name: String,
    using swiftTclFunction: @escaping SwiftTclFuncReturningDouble
) {
    let cmdBlock = TclCommandBlock(function: swiftTclFunction)
    let cData = Unmanaged.passRetained(cmdBlock).toOpaque()
    Tcl_CreateObjCommand(interp, name, swift_tcl_bridger, cData, nil)
}
```

This creates a new TclCommandBlock initialized with a `SwiftTclFuncReturningDouble`, increments its reference count via `passRetained` and converts it into an Opaque Unmanaged pointer (more or less a "void *"). Then the native C `Tcl_CreateObjCommand` is called to add the new command to the Tcl interpreter. Or, to be precise, to add the function `swift_tcl_bridger` with the actual Swift function squirreled away behind a `ClientData` pointer as a `TclCommandBlock`, which looks like:

```
// Part of "class TclCommandBlock" with most copies of the
// initializer and invoker removed:
class TclCommandBlock {
    let swiftTclCallFunction: SwiftTclFunctionType
    // [...]

    init(function: @escaping SwiftTclFuncReturningDouble) {
        swiftTclCallFunction = .double(function)
    }
    // [...]

    func invoke(Interp: TclInterp, objv: [TclObj]) throws -> Double {
        switch swiftTclCallFunction {
        case .double(let function):
            return try function(Interp, objv)
        case .int(let function):
            return try function(Interp, objv)
        // [...]
        default:
            abort()
        }
    }
}
```

SwiftTclFunctionType is an enumeration that encapsulates both the function and its type in an enumeration. Each case of the enumeration has an associated function reference that matches it:

```
public enum SwiftTclFunctionType {
    case tclReturn(SwiftTclFuncReturningTclReturn)
    case int(SwiftTclFuncReturningInt)
    case double(SwiftTclFuncReturningDouble)
    case string(SwiftTclFuncReturningString)
    case bool(SwiftTclFuncReturningBool)
    case tclObj(SwiftTclFuncReturningTclObj)
}
```

Finally, when the Tcl command is invoked from Tcl, the `swift_tcl_bridger` is invoked with the usual arguments for a Tcl extension. Note that other than the "void *" ClientData, all the C arguments are fully typed and used directly as typed Swift values:

```
func swift_tcl_bridger (
    clientData: ClientData?,
    interp: UnsafeMutablePointer<Tcl_Interp>?,
    objc: Int32,
    objv: UnsafePointer<UnsafeMutablePointer<Tcl_Obj>?>?
) -> Int32 {
    let tcb = Unmanaged<TclCommandBlock>.fromOpaque(clientData!).takeUnretainedValue()
    let Interp = TclInterp(interp: interp!, printErrors: false)

    // construct an array containing the arguments
    var objvec = [TclObj]()
    if let objv = objv {
        for i in 0..<Int(objc) {
            objvec.append(TclObj(objv[i]!, Interp: Interp))
        }
    }
    // invoke the Swift implementation of the Tcl command and return its value
    do {
        switch tcb.swiftTclCallFunction {
        case .double:
            let result: Double = try tcb.invoke(Interp: Interp, objv: objvec)
            Interp.setResult(result)

        case .int:
            let result: Int = try tcb.invoke(Interp: Interp, objv: objvec)
            Interp.setResult(result)

        // [...]
        }
    } catch TclError.errorMessage(let message) {
        Interp.result = message.message
        try! Interp.setErrorCode(message.errorCode)
        return TCL_ERROR
    // more error cases skipped
    } catch (let error) {
        Interp.result = "unknown error type \(error)"
        return TCL_ERROR
    }
    return TCL_OK
}
```

The operations previously performed on the `TclCommandBlock` to create an opaque unmanaged pointer are reversed. `ClientData` is converted into a reference to a `TclCommandBlock` and is dereferenced (without decrementing the reference count) via `takeUnretainedValue()`.

The passed `Tcl_Interp` is imported into a `TclInterp` class, and the vector of arguments is imported into a Swift array of `TclObj`.

Finally, it drops into a case statement that calls the right version of the `invoke()` method in the `TclCommandBlock` that was squirreled away back in `CreateCommand` (remember `createCommand`?)... which proceeds to call the original `func avg() throws -> Double`, and handles any errors resulting from the call (in this case, there can't be any).

Conclusion

Swift-Tcl gives you a nice type-safe language to write Tcl extensions in, and hides all the inconvenient type-casting and memory management glue behind a set of lightweight wrappers that take advantage of both Swift's and Tcl's strengths.