Eagle: Tcl Integration with the CLR

Joe Mistachkin <joe@mistachkin.com>

1. Abstract

Eagle [1] (Extensible Adaptable Generalized Logic Engine) is an implementation of the Tcl [2] scripting language [3] for the Common Language Runtime (CLR) [4]. It is designed to be a universal scripting solution for any CLR-based language.

In addition to being an implementation of the Tcl language, Eagle has the ability to function as a bidirectional "bridge" between a Tcl library and the CLR.

This paper explains the details of how Eagle integrates Tcl with managed code [5] and the CLR. It then describes the specifics of how to dynamically [6] load, use, manage state for, respond to events generated by, and unload a Tcl library [7].

2. Introduction

This paper presents the Tcl integration [8] features of Eagle. The goal of these features is to make Tcl libraries fully accessible to any application running on the CLR. The original goal of the Eagle project was to provide a Tcl-compatible scripting solution without being dependent on any native libraries. However, as the project progressed, it became clear that it would be extremely useful for Eagle to have the ability to dynamically load and use a Tcl library.

3. Rationale and Motivation

Historically, it has been very difficult to safely use Tcl from a CLR-based application, primarily because of the complex and error prone nature of P/Invoke [9] (e.g. portability to 64-bit [10] operating systems). Another issue is that the best practices for dynamically loading and using a Tcl library are poorly documented, and rarely followed with total compliance. This is in sharp contrast to the very well documented best practices for writing a portable, stubs-enabled [11] Tcl extension.

As an added benefit to the Tcl community, Eagle now provides a cross-platform [12], scriptable way to unit-test [13] dynamically loading and interacting with a Tcl library.

4. Design Philosophy

While the Tcl integration features of Eagle borrow heavily from the underlying API [14] design patterns in Tcl itself, there are differences:

- Hides complexities that are not applicable to managed code (e.g. allocation of temporary objects), resulting in a relatively high-level API; however, it does not abstract away [15] necessary details.

- Omits support for things that can easily be accomplished using managed code alone.

- Tries very hard not to crash [16], hang [17], or throw exceptions [18] from any of the publically exposed methods [19], even when supplied with invalid arguments. This is a fundamental difference between native code and managed code; it must be assumed that any caller of an API in managed code may be untrusted, or partially trusted. Uses the strong guarantee [20] (i.e. all or nothing) whenever possible to ensure that the application state [21] remains consistent.

The Tcl integration functionality is divided into the following components:

- The wrapper component (i.e. the **TclWrapper** class) – Provides a high-level interface to the core Tcl library.

- The module component (i.e. the **TclApi** class) – Provides a low-level container for the native module handle and delegates [22]. This is a low-level component that interacts directly with the Tcl C API [23], primarily via the delegates it manages.

- The worker thread component (i.e. the **TclThread** class) – Provides an interpreter in an isolated worker thread [24] which can be used to evaluate scripts and process events [25] without affecting the main application thread.

- The command bridging component (i.e. the **TclBridge** class) – Provides a bidirectional communication channel between the Tcl library and commands implemented in managed code. This component receives inbound native calls from the Tcl library, makes outbound managed method calls via the IExecute interface, and handles marshalling of the command arguments and interpreter result.

- The build information component (i.e. the **TclBuild** class) – Holds version and build information for a particular Tcl library. Typically, applications will not deal directly with this component.

- The collection of Tcl C API related delegate types (i.e. **TclDelegates**).

- The collection of Tcl C API related enumerations (i.e. **TclEnumerations**).

- The command implementation. This includes the **tcl** command itself and the supporting methods in the **Interpreter** class. This allows scripted access to the Tcl integration features and serves to demonstrate the key use cases [26] for the other components.

Of the components listed above, perhaps the most significant from the perspective of an application developer is the wrapper. It is specifically designed to be suitable for direct use by applications as it focuses on wrapping the high-level facilities provided by the Tcl library.

5. General Features

The major features include:

- Supports the .NET Framework Version 2.0 [27] on Windows [28] and Mono 2.0 [29] on both Windows and Unix [30] platforms. The .NET Framework Version 2.0 Service Pack 2 [31] or Mono 2.4 [32] is highly recommended. Having the .NET Framework 2.0 Software Development Kit [33] and/or Visual Studio 2008 [34] available is very useful.

- Supports 32-bit [35] and 64-bit [36] operating systems.

- Supports any build of Tcl/Tk, version 8.4 [37] or higher, including "basekits" [38] and "stardlls" [39]. Tcl/Tk version 8.6 [40] is highly recommended.

- Supports worker threads with their own event loop processing.

- Supports exit handler [41] creation and deletion.

- Supports script cancellation via TIP 285 [42].

- Supports a unified stack trace [43] for errors via the Eagle **errorInfo** [44] variable.

- Supports error line number management via TIP 336 [45].

- Prevents a deleted interpreter [46] from being used.

- Prevents an interpreter with activations from being deleted via TIP 335 [47].

- Automatically manages the interpreter result [48] and returns it for use in managed code.

6. Build Location and Selection

Before a Tcl library can be loaded, it must be located. This can be done manually by the host application, or the wrapper can attempt to handle it automatically during the loading process. The search performed by the wrapper includes the following locations:

- The directory the wrapper is running from.

- The directory the application executable is running from.
- The base directory of the application domain [49].
- The file specified by the "Tcl_Dll" environment variable.
- The Windows Registry [50] (i.e. installed instances of ActiveTcl [51]).
- The executable search path [52].
- On Unix, the "/usr/lib" [53] [54] and "/usr/local/lib" directories.

After the search has been conducted, one of the candidate builds must be selected. The wrapper chooses the "best" Tcl library among the candidates found using the following criteria:

- Always attempt to select the library with the highest version. All other factors being equal, if multiple libraries share the highest version (i.e. does not take into account the patch level, as the file names do not contain that information), the precise library selected is unspecified; however, it will be one that shares the highest version.
- Always favor threaded libraries, if this information can be deduced from the file name.
- Always favor libraries compiled for debugging, if the wrapper is compiled for debugging and this information can be deduced from the file name.
- Refuse to consider any library that does not match the current operating system architecture; the architecture flag not being set means that this restriction is waived.
- Refuse to consider any library that does not meet the minimum or maximum required versions; null means (in either case) that the restriction is waived.

7. Loading and Initialization

The **Load** method of the wrapper handles the loading and initialization requirements for a Tcl library. The loading and initialization process is divided roughly into the following steps:

1. The Tcl library is loaded using the facilities provided by the operating system.

2. The entry points for the necessary functions are resolved. During this process, the delegates (i.e. managed function pointers) for these entry points are created and stored. All future interaction with the Tcl library will take place using these stored delegates.

3. The **GetVersion** delegate is called. This is done to verify that the Tcl library does indeed meet the version requirements, including patch level, as specified by the caller.

4. The **FindExecutable** delegate is called to initialize various subsystems of the Tcl library.

5. The **Kit_SetKitPath** delegate is called if it is valid (i.e. in the case of a "stardll").

6. The **CreateInterp** delegate is called to create the initial or "master" interpreter.

7. The newly created interpreter is checked to make sure it is not incorrectly thought to be in use (i.e. it should not be in use at this point because it was just created). This is also the first real opportunity to double-check that the Tcl library was built for a threaded environment.

8. The **Kit_AppInit** delegate is called to initialize the initial interpreter if it is valid; otherwise, the **Init** delegate is called instead.

8. Usage

Tcl libraries are loaded and initialized using the **Load** method. The loading process is explained in the "Loading and Initialization" section. They are finalized and unloaded using the **Unload** method. The unloading process is explained in the "Finalization and Unloading" section.

Interpreters are created and deleted using the **CreateInterpreter** and **DeleteInterpreter** methods, respectively. Interpreters should be deleted when they are no longer required.

Interpreters may be created in a worker thread using the worker thread component (i.e. the **TclThread** class). The static **Create** method is used to create and start a worker thread and the **Dispose** method is used to terminate that worker thread. Events are queued to the worker thread, either synchronously or asynchronously, using the **QueueEvent** method. Event types include interpreter creation and deletion, command creation and deletion, script evaluation, script cancellation, variable management, and execution of arbitrary **EventCallback** delegates.

Commands are created and deleted using the command bridging component (i.e. the **TclBridge** class). The static **Create** method is used to create a command and the **Dispose** method is used to delete that command.

Scripts contained in a string are evaluated using the **EvaluateScript** method. Scripts contained in a file are evaluated using the **EvaluateFile** method. Expressions contained in a string are evaluated using the **EvaluateExpression** method. These methods return a standard completion code and a string result copied from the interpreter result.

Evaluations in progress are canceled using the **CancelEvaluate** method. This method returns an error if the Tcl library does not support TIP 285 (e.g. Tcl 8.5 and earlier).

Variable values are read, written, and unset using the **GetVariable**, **SetVariable**, and **UnsetVariable** methods, respectively.

Events are processed using the **DoOneEvent** method.

9. Integration and Extensibility

Integration with a Tcl library is accomplished using delegates. These delegates are annotated with the appropriate attributes, and are resolved dynamically for use with P/Invoke.

Extensibility is accomplished using classes that implement the **IExecute** interface, which consists of a single method named **Execute** defined as:

```
ReturnCodes Execute(
     IInterpreter interpreter,
     IClientData clientData,
     ArgumentList arguments,
     ref Result result);
```

The first argument is the Eagle interpreter associated with the command. If no such association exists, this value will be null. The second argument is the user data that was used when creating the command. The third argument consists of a list of argument objects which mirrors the arguments provided by Tcl. The fourth argument is passed by reference and is used to supply the interpreter result with the result of the command invocation. This method must return a standard completion code.

10. Finalization and Unloading

The **Unload** method of the wrapper handles the finalization and unloading requirements for a Tcl library. The wrapper does not maintain any state, such as lists of created interpreters, commands, or worker threads. Therefore, it is the responsibility of the application to maintain this state; to clean up all interpreters (except the master interpreter), commands, and worker threads it creates prior to calling this method. The finalization and unloading process is divided roughly into the following steps:

1. Make sure that there are no outstanding calls to the Tcl library.

2. Make sure that the master interpreter has already been deleted, or that its thread matches the current thread.

3. The **DeleteInterp** delegate is called to delete the master interpreter.

4. The **_Finalize** (note the leading underscore) delegate is called to finalize the subsystems of the Tcl library.

5. The Tcl library is unloaded using the facilities provided by the operating system.

11. Conclusion

It was not among the original goals of the Eagle project to facilitate access to a Tcl library from the CLR. However, it opens up an exciting new realm of possibilities for heterogeneous integration. The Tcl integration features of Eagle do not try to provide exhaustive access to APIs provided by a Tcl library; instead, they focus on a core subset of features required for scripting CLR-based applications. In addition, they allow Tcl to evaluate Eagle scripts and vice-versa, as well as allowing the lifetime of the Tcl library to be scripted in Eagle.

The official source code and binary distributions for the Eagle project may be downloaded from either the CodePlex web site [55] or the project web site and any issues encountered may be reported via the CodePlex issue tracker. Eagle may be used, modified, and redistributed at no cost, with minimal restrictions [56] (i.e. the same ones that apply to Tcl itself).

12. Acknowledgements

The author wishes to thank Dawson Cowals, Stuart Cassoff, Cameron Laird, Donal Fellows, Pat Thoyts, Jeffrey Hobbs, and Matthew Wilson for keeping him honest and for their suggestions, testing, and bug reports.

13. References

[1] Eagle, http://eagle.to/
[2] Tcl Developer Xchange, http://www.tcl.tk/
[3] "Scripting language", http://en.wikipedia.org/wiki/Scripting_language
[4] "Common Language Runtime", http://en.wikipedia.org/wiki/Common_Language_Runtime
[5] "Managed code", http://en.wikipedia.org/wiki/Managed_code
[6] "Dynamic loading", http://en.wikipedia.org/wiki/Dynamic_loading
[7] Tcl library, http://www.tcl.tk/man/tcl8.6/TclLib/contents.htm
[8] "Systems integration", http://en.wikipedia.org/wiki/System_integration
[9] "P/Invoke", http://en.wikipedia.org/wiki/Platform_Invocation_Services
[10] "64-bit computing", http://wiki.tcl.tk/17435
[11] "Stubs", http://wiki.tcl.tk/285
[12] "Cross-platform", http://en.wikipedia.org/wiki/Cross-platform
[13] "Unit testing", http://en.wikipedia.org/wiki/Unit_testing
[14] "Application programming interface", http://en.wikipedia.org/wiki/Application_programming_interface
[15] "Is Over Abstraction Java's Achilles Heel?", http://www.manageability.org/blog/stuff/over-abstraction-java-achilles-heel
[16] "Crash (computing)", http://en.wikipedia.org/wiki/Crash_(computing)
[17] "Hang (computing)", http://en.wikipedia.org/wiki/Hang_(computing)
[18] "Exception handling", http://en.wikipedia.org/wiki/Exception_handling

[19] "Method (computer science)", http://en.wikipedia.org/wiki/Method_(computer_science)
[20] "Exception guarantees", http://en.wikipedia.org/wiki/Exception_guarantees
[21] "State (computer science)", http://en.wikipedia.org/wiki/State_(computer_science)
[22] "Delegates (C# Programming Guide)", http://msdn.microsoft.com/en-us/library/ms173171.aspx
[23] "Tcl C API", http://wiki.tcl.tk/14054
[24] "Thread (computer science)", http://en.wikipedia.org/wiki/Thread_(computer_science)
[25] "Tcl event loop", http://wiki.tcl.tk/1527
[26] "Use case", http://en.wikipedia.org/wiki/Use_case
[27] Microsoft .NET Framework Version 2.0 RTM (x86), http://www.microsoft.com/downloads/details.aspx?familyid=0856eacb-4362-4b0d-8edd-aab15c5e04f5
[28] Microsoft Windows, http://www.microsoft.com/windows/
[29] Mono, http://www.mono-project.com/
[30] Unix, http://www.unix.org/
[31] Microsoft .NET Framework Version 2.0 Service Pack 2, http://www.microsoft.com/downloads/details.aspx?familyid=5b2c0358-915b-4eb5-9b1d-10e506da9d0f
[32] Mono 2.4, http://www.mono-project.com/Release_Notes_Mono_2.4.2.3
[33] Microsoft .NET Framework 2.0 Software Development Kit (x86), http://www.microsoft.com/downloads/details.aspx?familyid=fe6f2099-b7b4-4f47-a244-c96d69c35dec
[34] Microsoft Visual Studio 2008, http://msdn.microsoft.com/vs2008/
[35] "32-bit", http://en.wikipedia.org/wiki/32-bit
[36] "64-bit", http://en.wikipedia.org/wiki/64-bit
[37] Tcl/Tk 8.4, http://www.tcl.tk/software/tcltk/8.4.html
[38] "basekit", http://wiki.tcl.tk/15985
[39] "stardll", http://wiki.tcl.tk/15969
[40] Tcl/Tk 8.6, http://www.tcl.tk/software/tcltk/8.6.html
[41] Tcl exit handlers, http://www.tcl.tk/man/tcl8.6/TclLib/Exit.htm
[42] "TIP 285: Script Cancellation with [interp cancel] and Tcl_CancelEval", http://tip.tcl.tk/285
[43] "Stack trace", http://en.wikipedia.org/wiki/Stack_trace
[44] "errorInfo", http://wiki.tcl.tk/1645
[45] "TIP 336: Supported Access To interp->errorline", http://tip.tcl.tk/336
[46] Tcl interpreter lifetime, http://www.tcl.tk/man/tcl8.6/TclLib/CrtInterp.htm
[47] "TIP 335: An API for Detecting Active Interpreters", http://tip.tcl.tk/335
[48] Tcl interpreter result, http://www.tcl.tk/man/tcl8.6/TclLib/SetResult.htm
[49] "Application Domain", http://en.wikipedia.org/wiki/Application_Domain
[50] "Windows Registry", http://en.wikipedia.org/wiki/Windows_Registry
[51] ActiveTcl, http://www.activestate.com/activetcl/
[52] "PATH (variable)", http://en.wikipedia.org/wiki/PATH_(variable)
[53] "Filesystem Hierarchy Standard", http://en.wikipedia.org/wiki/Filesystem_Hierarchy_Standard
[54] "hier -- layout of file systems", http://www.freebsd.org/cgi/man.cgi?hier
[55] Eagle on CodePlex, http://eagle.codeplex.com/
[56] Eagle License, http://eagle.to/standard/license.html