Framework for Video Editor with Support for Many Virtual Machines

Angel R. Kanchev[1], Antoaneta Popova[2]

Abstract **– Big software like video editor needs strong and stable framework. The attempt to make that framework at most open led to creation of plug-in system that can load both JavaVM [1] and .NET [2] components. Considering the multiplatform developing approach and the ability for distributed callings, the framework has become attractive for any big application.**

Keywords **– Java, .NET, plugins, plug-ins, multiplatform, framework.**

I. Introduction

The framework that is described here is made for Audio/ Video Editor but is universal enough – it can be used in any big application.

By "big" application, we will understand an application that is built by many modules and is big memory and CPU consumer. This is the reason to put on first place the requirement for fastness and size of the framework.

All requirements for such framework could be summarized this way:

1. To be fast, small and stable.
2. To be easily supported and easy for use.
3. To be maximum open and extensible.
4. To be easily ported for different platforms.

The described framework could be written for some Virtual Machine (VM) in order to be stable and multiplatform.

Currently there are two widespread virtual machines: Java VM [1] and .NET CLR (Common Language Runtime) [2].

Both virtual machines have disadvantages when they are used for big applications:

1. Interpretation of VM code (only for Just-In-Time and Install-Time compilations – managed native code is not a problem according this point).
2. Validation and verification of the VM code (correctness and security check)
3. No control over the memory usage (there is a "magical" tool – Garbage Collector).
4. Reflection – too universal type descriptions and calling conventions, which makes it too heavy.

The argument that writing native (unmanaged) code is dangerous is not strong enough. Actually there are two reasons for dangerous code – the programmer is not good (the team is not well formed) or the schedule is too tight. Neither the technology nor the language make the code more secure.

The good things in VM approach are the fast source compilation (to VM code), small size of the executables, platform independence, nice exception system and multi-language interoperation. So writing managed code is faster and easier – it is good for rapid application development (r.a.d.). For big, heavy applications, where full control over the hardware is needed, native code is still the best (however, if we put commercial arguments here the situation will be a little bit different).

The hybrid approach – like .NET with unsafe code, has the disadvantages of both systems (managed and unmanaged)… The best approach is to use one technology for the whole framework.

As a result, it is created a simple unmanaged framework that can load different virtual machines (see Table I).

TABLE I
REASONS FOR IMPLEMENTATION APPROACH

Decision	Reason
Simple	Small, stable, easy to support and use
Unmanaged	Fast and full control over the hardware
Loading of VM	It can support plug-ins for different Virtual Machines

The goal of the framework is to ease writing of open Video Editor. The key word here is open – the application must be dynamic (change of active modules at runtime) and extensible. Change of active modules means change of different modules that can do similar work.

Multiplatform approach is used in the framework's implementation (multiplatform libraries are used and platform-dependant code is isolated).

There is plug-in system with cross-call capabilities (it will be described later).

II. Overall description

The framework is like mini virtual machine – it has:

- Application loaders
- GUI (Graphical User Interface) – the multiplatform library wxWidgets is used [3].
- Reflection-like system for class creation (there are identifications only for modules and class names)
- Plug-in system – the plug-in interfaces are statically hard-coded: no reflection, metadata, Interface Definition Language (IDL) or any of these universal systems.

[1] Angel R. Kanchev, is with the Faculty of Communications and Communications Technologies, Technical University, Kliment Ohridski 8, 1000 Sofia, Bulgaria, E-mail: angel_kanchev@mail.bg
[2] Antoaneta A. Popova, is with the Faculty of Communications and Communications Technologies, Technical University, Kliment Ohridski 8, 1000 Sofia, Bulgaria, E-mail: antoaneta.p@komero.net

Note that any reflection-like system is heavy – even the current one should be used for major interfaces and for plug-ins only. On Fig.1, you can see the major interfaces (and the data transfer) for Video Editing system.

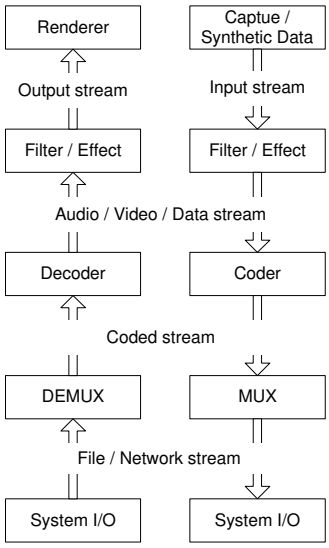

Fig. 1. Main parts of Audio/Video editor

The framework is abstract enough not to limit with Video Editor's needs, so we will talk for "Application" instead of Video Editor from now on.

From distribution point of view the application consists of three parts: loader, core and plug-ins. The application core has the major functionality of the application. The module that has the binding system between core modules, that dispatches calls to all core modules and that has the startup code for the application core is called Core. We will distinguish application core and module Core by the first capital letter.

On Fig. 2, you can see the major parts of the framework and their time relation (Core and PluginManager are modules in the application core). Arrow directions show function calls (i.e. at loading time the callings are unidirectional).

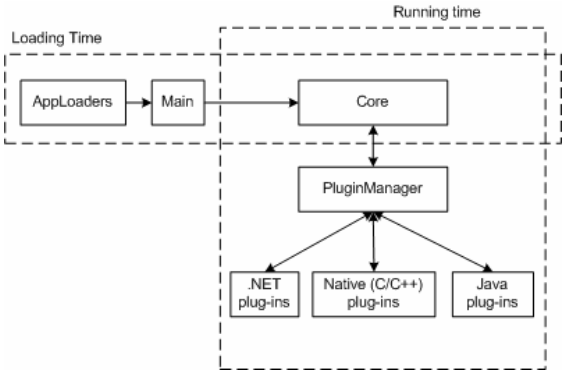

Fig. 2. Parts of the framework and their interaction in time

Note that plug-ins and PluginManager have bidirectional calls. That is – when the core needs a plug-in it uses PluginManager to initiate a call. If the plug-in needs something from the core, it can call back PluginManager, which in turn will call the necessary function in the core. This is the cross-call capability of PluginManager. For example, it allows calling .NET plug-in from inside Java plug-in (it will go through module Core – see Fig. 3). If it is necessary, PluginManager can load a VM – for now are supported JavaVM [1] and .NET CLR [4].

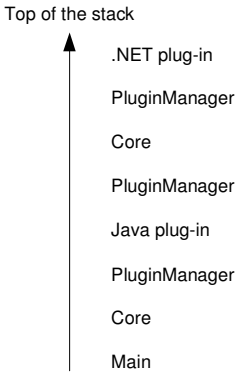

Fig. 3 Cross call stack (between Java and .NET)

On Fig.4, you can see the application loading in more details. There are many AppLoaders which goal is to find and load one dynamic library – the "Main" module. Main module has exactly one exported function – ExecuteApp. This function returns when the application must exit so the loader should exit when the function returns. There is parameter to ExecuteApp, which is "Environment Data". The application loader must build and pass correct environment data depending on what is the loader's type.

Before loading the core, Main have to check that all necessary components are available and with correct versions. After that, Main loads Core and gives it the environment data. On its turn, Core passes the environment data to PluginManager.

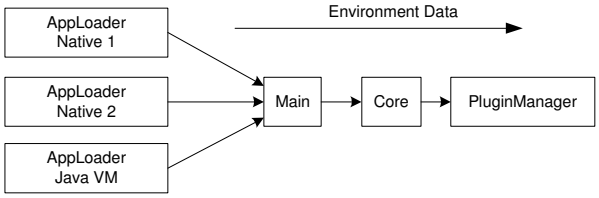

Fig. 4. Application loading

If the loading environment is different from the one necessary for a plug-in, PluginManager creates the necessary environment (i.e. loads the necessary virtual machine).

III. Framework Organization

The framework organization from developer's point of view can be seen on Fig. 5. The arrow means "use" or "links to".

There are three major (or distribution) groups: application loaders (AppLoaders), application core and application plug-ins. The application core consists of many dynamic libraries (modules). On Fig. 5 are shown three of them: CoreUtilities,

Core and PluginManager. These three modules are the modules from the core that participate in the framework. The application core contains additional modules that use the framework to add functionality to the core.

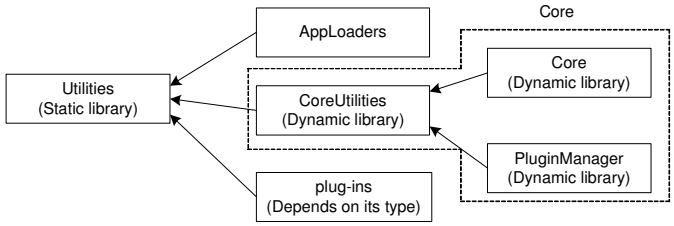

Fig. 5. Framework organization

The module CoreUtilities is used to export the classes in Utilities as dynamic library so the size of the core modules will not increase. Utility modules are modules like: .ini file parser, configuration loader, dynamic libraries manipulator and version support.

IV. Features of the Application Core

A. Component / Proxy architecture

The module Core exports the binding system for the application core. It is implementation of design pattern called "Component / Proxy" [5]. That pattern says: "component is a class that is hidden behind another class – proxy". Components can be accessed only through their proxies.

As you can see on Fig.6, one component can have many proxies but one proxy can point to one component.

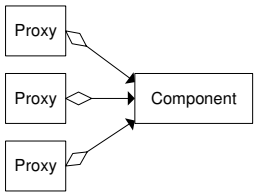

Fig. 6. Component – Proxy relations

It is very important to note – proxies are requested for creation and destruction while creation and destruction of components is automatic. This way it is possible that there are components without proxies as well as proxies without components (dead proxies). It is normal for component to be without proxies but dead proxy is "bad" thing. Such proxy has to simulate some work when its functions are called…

According creation, there are two types of components: singleton (it can live without proxies and there can exist only one instance of these components) and non-singleton (for each requested proxy, one component is created). Component can be destroyed after all its proxies are destroyed (but even though, the component could be left alive). There is manager that controls the lifetime of components and takes requests for proxies – ComponentManager.

The component and proxy creation is done via string identification that contains: Name of the Module (a dynamic library) and Name of the Component. Creation of classes using string identification is similar to reflection systems in Java and .NET.

In the current implementation in addition to the reference counting system, proxy and component lists are used. The benefit is that a component can understand when any of its proxies is destroyed and a proxy can understand if its component is destroyed.

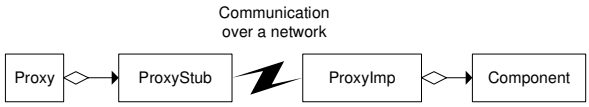

Fig. 7. Remote call between Proxy and Component

"Component / Proxy" architecture has one more benefit – it can be used for remote calls as shown in Fig. 7. For class Proxy, ProxyStub looks like a component. The same way, for class Component, ProxyImp seems to be ordinary proxy. This way Proxy and Component never understand that they are on different computers and are communicating via network. The job of ProxyStub and ProxyImp is to convert function calls to protocol requests / responses.

Static class hierarchy for the described architecture is shown on Fig. 8 (an arrow means inheritance). Class BaseComponentProxy has one pointer to IBaseComponent. This way Proxy and ProxyImp can point to either Component or ProxyStub. Class BaseNetworkProxy has basic functionality for data exchange through network.

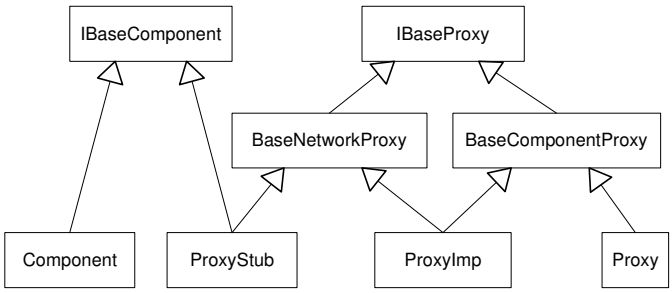

Fig. 8. Class hierarchy of C / P architecture

B. Configuration file

If you look again at Fig. 2 you will see three independent modules that are involved in the application loading: AppLoader, Main and Core. The interface between them is very tight – one function (ExecuteApp, exported by both Main and Core). In order to parameterize the loading and to create storage with common data for these modules there is a little configuration file. It could be parsed up to three times (each of the involved modules may need to parse it). Once the module Core has the information in the configuration file, the whole application core will have it.

In the main configuration file there is reference to another configuration file – for the logging system.

C. Log system

According the destination of the log strings there are 3 output types:

1. File – the log strings are written in a file
2. GUI – the log strings are put in a place where the user can see them
3. Debug – in the debugger output (for debug builds only).

The string formatting for each output type can be different. This way the developer and the user are eased at most.

There are four logging levels: Info, Warning, Recoverable Error and Fatal Error. The application exits on fatal errors…

The log configuration supports different combinations between levels and output types. In addition, there is filter on per-module basis (which modules to include / exclude from logging).

In order to be fast, the log system defines for each possible configuration different (optimized) function. There are 4 function pointers – for each log level. During initialization, these pointers are set according the configuration. The logging is done via call to the necessary function pointer…

D. Plug-in system

The plug-in system was discussed on different places in this article and was well described ideologically. It consists of core module (PluginManager) and export declarations. PluginManager exports strictly defined interfaces for use by different Virtual Machines; or by native plug-ins, written in C or compatible language (C++, Borland's Pascal). The plug-in system is tested with native (C, C++) and VM (Java and .NET) plug-ins.

If the core is loaded by loader for some VM and a plug-in for the same VM is called, PluginManager use the environment of the loader. Otherwise, it loads the VM first and then proceeds with loading and calling the plug-in. The VM loading code is called only if corresponding plug-in is requested. Once VM is loaded, it is used for all plug-ins of its type. The code is protected against missing VM so the user is not obliged to have any VM.

V. EXPERIMENTS AND CONCLUSIONS

The framework that was described so far is implemented in native C++ and is used in Video Editing software (see Fig. 1). The experiments with that software have shown that creating new module for the application core is time-consuming task. In order to ease the creation of new core modules, there is created custom wizard for Microsoft Visual Studio called "CorePackage". This wizard generates project that can be directly compiled to a dynamic library that covers the requirements for core module.

The experiments have shown the following advantages of the framework:

1. It is very open and supports native (C compatible), JavaVM and .NET CLR plug-ins (all plug-in types are tested)

2. It gives optimized, configurable and easy-to-use log system.

3. It has reflection-like system that allows module loading and class creation at runtime determined by string. When used with predefined interfaces (like the interfaces in Fig. 1) the system is powerful enough without being as heavy as Java or .NET reflection.

Allowing each block on Fig. 1 to be plug-in makes the software very dynamic and extensible. In addition, concentrating interfaces and data flow in one place (the application core) gives full control over the data.

Future work – after finishing the framework for the Video Editor, features implementation can be started (stream editor with lazy algorithm and after that – editor for each level on Fig. 1).

REFERENCE

[1] Liang, Shen: "The Java™ Native Interface", Addison Wesley Longman Inc., 1999.

[2] Microsoft Corporation: "Technical Overview of the Common Language Runtime", 2006.

[3] Smart, Julia; Robert Roebling; Vadim Zeitlin; Robin Dunn: "wxWidgets 2.5.5: A portable C++ and Python GUI toolkit", April 2005.

[4] Chakraborty, Ranjeet: "Creating a Host to the .NET Common Language Runtime", article in "The Code Project", October 2001

[5] Gamma, Erich; Richard Helm; Ralph Johnson; John Vlissides: "Design Patterns: Elements of Reusable Object-Oriented Software", Addison Wesley Longman Inc., 1998.