HarmonyArchitecture

Translations available:

- HarmonyArchitectureItalian Architettura Proposta Per Harmony (Overview) (incompleta)
- HarmonyArchitectureChinese Proposed Harmony Architecture (Overview)

Proposed Harmony Architecture

This is based on "*Harmony: The Apache J2SDK Project*" (http://www.jguru.se/jguru/control/Developers/Harmony) with a few changes taken from the *Harmony mailing list* (http://mail-archives.apache.org/mod_mbox/incubator-harmony-dev). Please direct any comments and discussions there, and include changes on this page when consensus is reached.

Harmony is a project from the Apache Foundation aimed at creating a stable and open-source implementation of the Java 2 SDK. It is not a Java 2 Enterprise Edition (J2EE) application server, nor does it handle the Java 2 Micro Edition (J2ME) applications.

Initial Requirements

The major requirements for Project Harmony are evident and given by the JDK Technology Compatibility Kit (TCK). Several other requirements are, however, required:

1. Simple porting and use of OS native facilities. This requirement can be managed by introducing the OS Abstraction Layer (OAL) which is the only point of contact between the JVM and the native OS, or the tool library and the native OS respectively. It is vital that the OAL is thin and that its boundaries may be moved within the JVM to deliver optimum performance on all or most OSes. 2. Reduce required library clutter and promote augmentation of JDK tools. This requirement can be met by using a common library for all tools. By reducing the number of top-layer dependencies and keeping a clean API, we may increase developer throughput in project aiming to provide, say, GUI versions of JDK standard tools.

Please provide more suggestions for requirements - and let us keep them measurable from now on. 🙂

Initial Architecture and Subprojects

I propose initial architecture and subprojects for Harmony as outlined below. Each of the (sub)projects should adhere to the standard Apache project model, unless a particularly good reason tells us not to. Originating from the early discussions on the Harmony mailing list, I have created some structure sketches on the project.

http://www.jguru.se/jguru/Channel_Html/generic/images/developers/harmony/overall.jpg

The main internal structure of the JVM and class library project is illustrated above. According to discussions on the development list:

- **The classpath project** aims at creating a cleanroom, open source implementation of the Java standard API. GNU Classpath (http://www.gnu.org/software/classpath) may be used to fill the need of the classpath project, given that licensing may be honored on both sides, and proper credits are given.
- **The JVM project** aims at creating a cleanroom, open source implementation of the Java Virtual Machine. There have been discussions of implementing it in Java, C/C++, or using a framework in which each component can be written in different languages. We currently lack a good choice for JVM code baseline. Potentially, we face reimplementing from scratch.
- **The OS Abstraction Layer (OAL) project** aims at defining and enforcing a thin wrapper layer which connects the JVM and the classpath to the services of the native OS. This layer is the only part directly interacting with the OS for non-development versions of the JVM. Currently the Apache Portable Runtime (APR) (http://apr.apache.org) seems the best choice.
- **The development facilities project** aims at creating APIs and tools to facilitate debugging and development-time profiling in order to track state within the JVM. Development facilities should be stripped from the build for non-development releases. Development facilities must be adapted to the JVM, and will therefore be defined pending the choice of JVM.

JVM subprojects

The main Harmony project in terms of volume is likely the JVM project as illustrated in the overall project structure above. We should therefore split the development effort of the JVM project into a few subprojects. as illustrated below. http://www.jguru.se/jguru/Channel_Html/generic/images/developers/harmony/jvm_projects.jpg

It seems reasonable that we can identify (at least) three subprojects with a number of development goals inside the JVM. These are

- ClassLoading. This project provides facilities for loading Java bytecode streams, verifying integrity and optionally performing pre-runtime optimizations. (I.e. all activities before any bytecode is actually executed).
    - Loader. Loads bytecode streams as requested by a java.lang.ClassLoader
    - Verifier. Verifies the integrity of the loaded bytecode stream.
    - Optimizer. Performs simple optimizations on loaded bytecode in order to reduce its size or rearrange operations as required by the native processor or OS.
- Runtime. This project provides facilities for executing Java code already verified by the ClassLoading facilities. This includes performing runtime optimizations if applicable, and interfacing with requested native OS services as requested by the Java class.
    - MemoryManager. Automatic Java heap management, including the supports of object allocation and dead object reclamation.
    - ThreadManager. Manages memory and services on a per-thread basis.
    - JIT. Standard Just-In-Time compiler converting pre-optimized bytecode to executable machine code for the platform in question.
    - Native Interface. The Java Native Interface connection to the services of the native OS.

- Statistics. No JVM is complete without facilities to measure and tune its performance in runtime. This is the task of the statistics project.
    - Profiling. Standard profiling interface which provides information about resource allocation and usage for the running JVM. I recommend that we provide facilities to modify runtime parameters on the fly through this interface.
    - Crash Talkback. It is important to receive crash dump information from JVMs to provide statistical information about runtime bugs, faulty optimizations or other such data. The talkback services sends this information over the internet to our statistics collection server.

As usual, these may not be the optimal or final project or development streams. Your comments are welcome.

Development Facilities Subprojects

Development facilities exist to enhance the overall development process, and should be stripped from the build in production releases (or, at least, non-development and -debug releases). Development facilities may explore state or data flow in any layer of the JVM, class library or OAL.

http://www.jguru.se/jguru/Channel_Html/generic/images/developers/harmony/devfacilities_projects.jpg

I recommend (at least) the following projects under the Development Facilities umbrella.

- Internal Tools. This project provides development facilities checking state and delivering development state profiling information in the running JVM.
    - Trace. Investigates state in all parts of the JVM; a "debugger-on-steroids" interface.
    - Thread Profiler. Collects statistics information from all running threads in the JVM and provides mechanisms to alter Thread runtime parameters and/or priorities.
- Analyzer. Creates simple-to-understand reports from reports retrieved from the net or local JVM.
    - Talkback Analyzer. Reads a set of talkback reports and collects relevant statistics from them. This developer stream should perhaps not be located in the Development Facilities project, but has been placed here for now and for lack of a better place.
    - Introspection GUI. The biggest problem with obtaining profiling information from several current JVMs is the complexity of the information interface. I propose that we create a slim and simple GUI for the Harmony development facilities where such information may be visualized in a simple manner.

Again, feedback is welcome - but use the Harmony mailing list for the feedback.

Modular Structure JVM Components

- **Standard class libraries** - The Java code included in the Apache Harmony distribution which implements the required class libraries (e.g., J2SE 5). Standard class library code interacts with the outside world and with native code through the services provided by the virtual machine. This is just like any Java code, but it needs specialized access to the JVM and the platform for implementing low level functionality and attaining acceptable performance.
- **Platform Accessors** - Provide Java code in the standard class libraries the ability to efficientely pass parameters and query results to and from native code.
- **VM Accessors** - Provide Java code in the standard class libraries the ability to efficientely access internal Java data structures and functionality (e.g., objecs, arrays, stack traces).
- **Java Apps** - The Java code of applications and applets being executed by the JVM.

- **VM Core** - VM Core is a centric part of overall JVM design. The VM Core manages component loading, versioning and inter-component coordination, and includes common sub-components such as Class Loader, Verifier, JVMTI framework, JNI framework, Exception handling and Stack walking. Details on some of the Class Loader interfaces are in the following links: field_access.txt method_access.txt method_access_java_conv.txt vm_class_manipulation.txt Details of some of the GC/VM interfaces can be found in the following links. The VM calls into the GC using gc_interface.txt. The GC calls into the VM using vm_gc_interface.txt. A proposal for a simple component model in C using function pointer tables can be found here [ComponentModelFunctionPointers].

- **Thread Manager** - Provides the functionality required to implement the Java threading and locking capabilities. The thread manager exposes a high-level threading interface inspired by the platform independent Java threading model and it is built on the top of an internal thread portability layer which hides low level OS specifics.
- **Garbage Collector** - Provides automatic management of the Java heap.
- **OS Portability Layer** - Provides portable wrappers for OS calls and hides differences between OS interfaces. This component may be based on the Apache Portable Runtime (APR), updated to incorporate the functionlaity that may be needed to support the requirements of a JVM.
- **Execution Manager** - A “container” for profile collectors and execution engines. It implements the logic for defining which execution engine is to be used for which method and handling profile collection events and triggering recompilation using various execution engines and/or phases.
- **Execution Engine** - The component which actually provides execution of Java bytecodes, such as interpreter and JIT compilers. The interpreter typically executes (interprets) Java bytecodes directly, whereas JITs compils the bytecodes to machine code directly executable by the target CPU.
- **Profile Collector** - Implements profile collection techniques. A single profile collector can be reused by several other components (VM, execution engines, the execution manager or even a garbage collector).

The diagram below is is a standard UML component diagram: each component is represented as a box and exposed interface groups are shown as annotated "lollipops." The annotation identifies the corresponding interface group. Colors represent functional areas associated with components, interface groups and dependencies.

Modular Virtual Machine Interfaces Component Diagram ModularJVM.jpg

VM Core Sub-components

- JNI native interface module
- JNI invocation interface module
- Global/local references manager
- Reflection manager (i.e., used by reflection native code)

- Invocation manager - required to dynamically invoke methods from within the VM itself
- Exception manager - can unwind stacks to throw exceptions
- Bootstrapper - code to get the VM started
- Resolver - performs field, class, method resolution
- Verifier
- Class initializer - invokes <clinit>'s
- Boot loader - the bootstrap class loader (reads ZIP files, etc)
- Class deriver - defines runtime types (including array types)
- Native library manager - dlopen()'s native libs, finds functions
- Properties manager - sets system properties (e.g., "java.vm.name").
- String manager - converts between String <-> char *utf8; interning
- Memory allocator - allocator for non-Java-heap memory
- Finalizer module

J2SDK Toolkit Architecture

In addition to the JVM and classpath, the standard JDK tools must be implemented for Harmony. These tools includes:

- An appletviewer
- A Browser Java plug-in
- A Java compiler (javac). Possible candidates is Eclipse and Jikes.
- A jar utility & signer
- A javadoc utility
- more? JNLP-implementation (>Webstart/OpenJNLP/netx), i18n-tool (>native2ascii/Resource Bundle Manager)

These utilities can either be written in Java using the classpath project, or written in C/C++. In the later case we should, to promote reuse and facilitate deployment, strive for a single distribution library (called Common Tools Library or CTL in the image below) on top of the OAL from the JVM project.

http://www.jguru.se/jguru/Channel_Html/generic/images/developers/harmony/tools.jpg

The overall tool goal is to permit or promote augmented clients in addition to the standard-named ones. I believe most of us would like to see an augmented keytool that could - say - provide native OpenSSL CA management directly to the default keystore. Given a decent Swing-based GUI for the job, key management annoyances would likely be greatly reduced compared to today's text-based equivalent. Alas, we would strive for the situation illustrated below:

http://www.jguru.se/jguru/Channel_Html/generic/images/developers/harmony/tools_example.jpg