Automated Logging Framework for Java

User Guide

Version 0.9.1

Introduction

Debugging is time-consuming and difficult. Good design and effective unit testing help to eliminate errors, but bugs which are invariably discovered after code has been built and unit tested are often more difficult to resolve. To determine points of failure, the path taken through your code before the problem occurs need to be known. Traditionally, there are two methods of achieving this: by using adebugger or by adding trace statements to your code. A debugger will allow you to interactively step through each line of code, but requires you to set up a debugger in the deployment environment and can be also very tedious. Adding trace statements requires editing your code to include print or logging statements, recompiling and redeploying. Even after all this, you might not have covered enough code to locate the exact point of failure.

ALF aims to provide an alternative to these methods by providing automatic tracing for your Java code. It does not require you to add code or even recompile. ALF works with the Java Virtual Machine(JVM) to provide valuable tracing information as your program is running. You can also use the ALF logging library to incorporate logging statements into the tracing output.

Supported Platforms

ALF is written in Standard C++ using the Standard Template Library, with the aim of achieving extensive portability. As of version 0.9.1, ALF has been successfully built on the following platforms:

Red Hat Linux 7.2 for x86, Kernel 2.4.7-10, Sun Java SDK 1.3.1
Compiled with GCC Version 2.96

Windows 98, 2000 and XP Professional, Sun Java SDK 1.3.1 and 1.4.0
Compiled with Borland C++ Builder 4.0 using Perl Compatible Regular Expressions(http://www.pcre.org)

ALF should compile or other platforms. e.g. Solaris, Darwin. The target platform must have a JVM with support for the Java Virtual Machine Profiling Interface (JVMPI). If you succeed (or fail!) in building and running ALF on any platform please let us know. ALF is currently dependent on a POSIX regular expression library, but this will change in coming versions.

Getting ALF

ALF is hosted at SourceForge and is available via CVS and HTTP.

Download Source or Binaries for you platform via HTTP at:
http://sourceforge.net/project/showfiles.php?group_id=49358

Source can be retrieved via anonymous CVS as follows.

```
cvs -d:pserver:anonymous@cvs.alfj.sourceforge.net:/cvsroot/alfj login
cvs -z3 -d:pserver:anonymous@cvs.alfj.sourceforge.net:/cvsroot/alfj co alfj
```

The CVS repository may be browsed at http://cvs.sourceforge.net/cgi-bin/viewcvs.cgi/alfj

Installing

Installing from Source

Source installation of ALF involves two steps: Java compilation and C++ compilation. The first step should produce three JARs for ALF logging, the second step produces a single native shared library. Once compilation is complete, see *Installing ALF Binaries*.

Windows and UNIX Users:

Use ANT(http://jakarta.apache.org/ant) to build the JARs. The `build.xml` file is located in the `javasrc` folder.

If you do not have ANT installed, you can build the JARs manually by performing issuing the **javac**, **jar** and **javah** commands in sequence. For details of each step, examine the `build.xml` fie.

UNIX Users Only:

The native shared library (`libalfj.so`) can be built using the Makefile in the root directory. To use the Makefile, type the following command.

```
make JAVA_HOME=/opt/sun/jdk
```

Change the `JAVA_HOME` value to point to wherever your JDK installation is. If you are using a platform other than Linux, specify the name present in`JAVA_HOME/include` as follows.

```
make JAVA_HOME=/opt/sun/jdk PLATFORM=solaris
```

Once ALF has been built and linked, the shared library, `libalfj.so` will be present in the root directory.

Windows Users Only:

Currently, the only tested method for building on Windows is to use Borland C++ Builder. It should be possible to use Borland's Free Compiler and use the project as a Makefile. The project is `src\win32\alfj.bpr`. You will need to specify a regular expression library such as PCRE in your include and library path. When ALF has been built successfully, the linker should produce the shared library,`alfj.dll`.

Don't forget to let us know if you have any success building with any other compilers.

Installing ALF Binaries

UNIX Users:

The native shared library, `libalfj.so` should be placed anywhere it can be loaded by the JVM. On UNIX, this includes:

- In `JAVA_HOME/lib/i386` (Linux x86) or `JAVA_HOME/lib/sparc` (Solaris)
- Anywhere specified in your`LD_LIBRARY_PATH` environment variable

Windows Users:

The native DLL, `alfj.dll`, should be placed anywhere it can be loaded by the JVM. On Windows, this include:

- In `JAVA_HOME/bin`
- In Windows system directories
- Anywhere specified in your`PATH` environment variable

Using ALF

Testing Your Install

To test you installation, simply issue the following command.

```
java -Xrunalfj:help
```

You should see the ALF banner followed by a brief explanation ofALF's command-line arguments.

If you see something like this instead, check the location of your shared library as described above.

```
Error occurred during initialization of VM
Could not find -Xrun library: alfj.dll
```

If you see something else entirely, then you might have found a bug. Pleasesubmit a bug report.

Running Your Application with ALF

To run an existing application with ALF, simply add the -Xrunalfj option to your *java* command. This will run ALF with tracing turned on for *all* packages. You will rarely need this level of tracing, as it includes every tracing for every method in the Java language and Java libraries. Here's a few lines of ALF's output.

```
Automated Logging Framework (ALF) v0.1.0
Sat May  4 14:08:24 2002
14:08:24 [main] > java.lang.Thread.<init>( String ) : void
14:08:24 [main] >> java.lang.Thread.init( String ) : void
14:08:24 [main] >>> java.lang.Thread.currentThread() : Thread
14:08:24 [main] <<< java.lang.Thread.currentThread() : Thread
14:08:24 [main] >>> java.lang.ThreadGroup.checkAccess() : void
14:08:24 [main] >>>> java.lang.System.getSecurityManager() : SecurityManager
14:08:24 [main] <<<< java.lang.System.getSecurityManager() : SecurityManager
```

As you can imagine, this will produce millions of lines of tracing, making tracing for your application impossible to find. In addition, the I/O required to produce such output will cause your application to be unacceptably slow. This is where package and class filtering comes in. ALF has two options to support this: *exclude* and *include*. For instance, to turn off all tracing you may use the following command.

```
java -Xrunalfj:exclude=* <classname>
```

If all your code is in the package `org.acme`, you can use the *include* option to limit tracing to that package:

```
java -Xrunalfj:exclude=*,include=org.acme
```

The following output is from a sample which creates 3 threads (*ACMETasks*) and runs them. (*ACMETask*'s *run()* method calls the *getText()* method, prints its return values and sleeps for a random amount of time.)

```
Automated Logging Framework (ALF) v0.9.1
Sat May  4 14:25:42 2002
14:25:42 [main] > org.acme.ACMEApp.main( String[] ) : void
14:25:42 [main] >> org.acme.ACMETask.<init>( int ) : void
14:25:42 [main] << org.acme.ACMETask.<init>( int ) : void
14:25:42 [ACME Task No: 0] > org.acme.ACMETask.run() : void
14:25:42 [ACME Task No: 0] >> org.acme.ACMETask.getText( int ) : String
14:25:42 [ACME Task No: 0] << org.acme.ACMETask.getText( int ) : String
Iteration: 0
14:25:42 [main] >> org.acme.ACMETask.<init>( int ) : void
14:25:42 [main] << org.acme.ACMETask.<init>( int ) : void
14:25:42 [main] >> org.acme.ACMETask.<init>( int ) : void
14:25:42 [main] << org.acme.ACMETask.<init>( int ) : void
14:25:42 [main] < org.acme.ACMEApp.main( String[] ) : void
14:25:42 [ACME Task No: 1] > org.acme.ACMETask.run() : void
14:25:42 [ACME Task No: 1] >> org.acme.ACMETask.getText( int ) : String
```

```
14:25:42 [ACME Task No: 1] << org.acme.ACMETask.getText( int ) : String
Iteration: 0
14:25:42 [ACME Task No: 2] > org.acme.ACMETask.run() : void
14:25:42 [ACME Task No: 2] >> org.acme.ACMETask.getText( int ) : String
14:25:42 [ACME Task No: 2] << org.acme.ACMETask.getText( int ) : String
Iteration: 0
14:25:42 [ACME Task No: 0] >> org.acme.ACMETask.getText( int ) : String
14:25:42 [ACME Task No: 0] << org.acme.ACMETask.getText( int ) : String
Iteration: 1
14:25:42 [ACME Task No: 0] >> org.acme.ACMETask.getText( int ) : String
14:25:42 [ACME Task No: 0] << org.acme.ACMETask.getText( int ) : String
Iteration: 2
14:25:43 [ACME Task No: 2] >> org.acme.ACMETask.getText( int ) : String
14:25:43 [ACME Task No: 2] << org.acme.ACMETask.getText( int ) : String
Iteration: 1
14:25:43 [ACME Task No: 1] >> org.acme.ACMETask.getText( int ) : String
14:25:43 [ACME Task No: 1] << org.acme.ACMETask.getText( int ) : String
Iteration: 1
14:25:43 [ACME Task No: 0] >> org.acme.ACMETask.getText( int ) : String
14:25:43 [ACME Task No: 0] << org.acme.ACMETask.getText( int ) : String
Iteration: 3
14:25:43 [ACME Task No: 2] >> org.acme.ACMETask.getText( int ) : String
14:25:43 [ACME Task No: 2] << org.acme.ACMETask.getText( int ) : String
Iteration: 2
14:25:43 [ACME Task No: 1] >> org.acme.ACMETask.getText( int ) : String
14:25:43 [ACME Task No: 1] << org.acme.ACMETask.getText( int ) : String
Iteration: 2
14:25:43 [ACME Task No: 1] >> org.acme.ACMETask.getText( int ) : String
14:25:43 [ACME Task No: 1] << org.acme.ACMETask.getText( int ) : String
Iteration: 3
14:25:44 [ACME Task No: 2] >> org.acme.ACMETask.getText( int ) : String
14:25:44 [ACME Task No: 2] << org.acme.ACMETask.getText( int ) : String
Iteration: 3
14:25:44 [ACME Task No: 2] < org.acme.ACMETask.run() : void
14:25:44 [ACME Task No: 0] < org.acme.ACMETask.run() : void
14:25:44 [ACME Task No: 1] < org.acme.ACMETask.run() : void
```

If your application does not use packages, you can still turn of unwanted packages by excluding system packages explicitly as follows.

```
java -Xrunalfj:exclude=java:javax:com:sun MyApp
```

ALF Output Explained

Each line of ALF's output consists of the following parts:

- **Time:** The time the method enter/exit event occurred
- **Thread Name:** The name of the thread the method was executed in. This is illustrated in

the mult-threaded example above.

- **Stack Depth and Direction:** The > and < symbols are used to indicate a method enter or exit. The number of symbols indicates the stack depth for the method and can be used to link method exits to their corresponding enter events. *Note:* The stack depth is not the full thread stack depth, but the stack depth for just the methods which are being traced.
- **Method Name:** The full package, class name and method name with parameter types followed by the method's return type

Using ALF Logging

Most users will want to include logging of general events in their applications. As an alternative to using a logging framework such as your own implementation or a separate third-party product, ALF allows you to integrate logging into your method tracing. To do this you must add one of the ALF Jars to your classpath.

- **alfj_native.jar:** This uses the same native code as ALF tracing for seamless integration. If you are using ALF tracing, this is the best option, but you must always use the `-Xrunalfj` option to load the native code.
- **alfj_noauto.jar:** This is a pure Java implementation of ALF logging. If you wish to deploy your application without using ALF tracing, but*with* ALF logging, use this library.
- **alfj_stub.jar:** This turns off ALF logging completely so yo don't have to remove ALF logging statements from your code if you wish deploy*without* ALF logging.

ALF supports different levels of logging: Information, Warning, Error and Exception. The methods *logInfo()*, *logWarn()*, *logErr()* and *logExcep()* are documented in the ALF Logging API.

Using ALF with J2EE Application Servers

To use ALF with a J2EE Application Server, locate the part of the start script which executes the java command and add the `-Xrunalfj` argument as above. You will probably want to add *exclude* arguments for the J2EE vendor's classes.

ALF has been tested with JBoss 2.4.4 and Weblogic 6.1. Executing the JVM with ALF will cause a loss of performance and this is especiallynoticeable during startup of the Application Server. Once the startup sequence is complete, this will not be as apparent.

Options

By using -Xrunalfj:<option>, you can control ALF'sbehaviour. The following options are now supported.

- **help**
  Displays ALF usage
- **exclude=<package>:<class>:...**
  Turns off tracing for a package or class. Exclusion includes sub-packages and may be used many times along with*include* to specify a full set of packages and classes. e*xclude=** turns off tracing for all packages.
- **Include=<package>:<class>:...**
  Turns on tracing for a package or class. Can be used many times along with*exclude.*

- **File=<filename>**
  ALF tracing is sent to STDOUT by default. This options causes ALF output to be sent to a file so it may be browsed or parsed later. If you use the UNIX command *tail* with the *-f* argument you can view the file's contents as it is being written to have the same effect as STDOUT tracing.

Contact

E-Mail

donalg@users.sourceforge.net

WWW

http://alfj.sourceforge.net