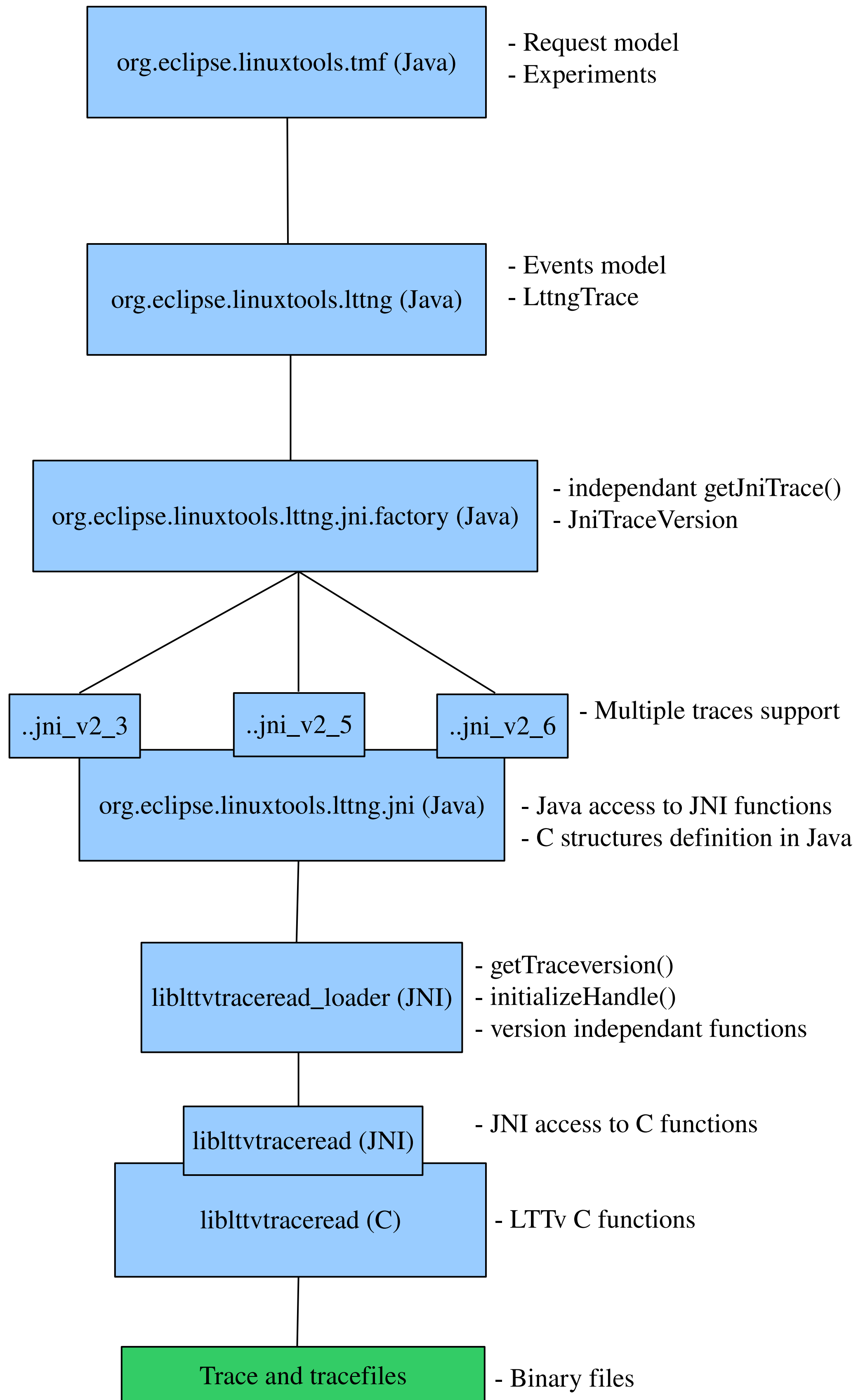
JNI in LTTng

Overview Diagram....................2
Description of the architecture....................3
liblttvtraceread....................3
liblttvtraceread_loader....................4
1) ltt_getTraceVersion(jstring tracepath)....................4
2) ltt_initializeHandle(jstring libname)....................4
3) Version independant access to JNI functions....................5
org.eclipse.linuxtools.lttng.jni....................6
1) org.eclipse.linuxtools.lttng.jni 6
2) org.eclipse.linuxtools.lttng.jni_vX_Y....................6
3) org.eclipse.linuxtools.lttng.jni.common....................6
4) org.eclipse.linuxtools.lttng.jni.exception....................7
5) org.eclipse.linuxtools.lttng.jni.factory....................7
Annexes....................8
Java native functions name mangling....................8
Note on Java/C types : 8
Java internal storage of function names and missing System.unloadLibrary()....................9
How to suppport a new LTTng trace version....................10
Usual case : no API change in LTTv....................10
1 - libltvtraceread and liblttvtraceread_loader :....................10
2 - org.eclipse.linuxtools.lttng.jni :....................10
More complex case : API change in LTTv....................11
1 - libltvtraceread and liblttvtraceread_loader :....................11
2 - org.eclipse.linuxtools.lttng.jni :....................11

Overview Diagram



Description of the architecture

liblttvtraceread

This is contained in files named like "*liblttvtraceread-X.Y.so*" when X and Y are the major and minor number of the trace version supported.
Also, during build, a symlink from "*liblttvtraceread.so*" to the latest version build should be made.

This library is compiled from the following code :

- marker.c
- event.c
- tracefile.c
- jni_interface.c (only if flag "--with-jni-interface" is given)
- lttvtraceread_loader.c (only if flag "--with-jni-interface" is given)

Contains two main components :

- Basic C functions and structures.
- First layer of JNI Code

The C functions are always present in the library but the JNI code is only build into the library when the flag "--with-jni-interface" is given to the configure (or autogen.sh) script.

All the JNI functions name respect java's name mangling (like : *Java_org_eclipse_linuxtools_lttng_jni_JniTrace_ltt_1openTrace*), so it is still possible to bind with them directly from java, if the functions are in the correct java package (see notes about java name mangling below).

liblttvtraceread_loader

This is contained in files named like "*liblttvtraceread_loader-X.Y.so*" when X and Y are the major and minor number of the trace version supported.
Also, during build, a symlink from "*liblttvtraceread_loader.so*" to the latest version build should be made.

Note : This library is build ONLY when the flag "--with-jni-interface" is given to the configure script).

Contains three main components :

1. ltt_getTraceVersion(jstring tracepath)
2. ltt_initializeHandle(jstring libname)
3. version independant access to JNI functions

1) ltt_getTraceVersion(jstring tracepath)

ltt_getTraceVersion make it possible to know the version number of a trace given its path.
Mangling is "*Java_org_eclipse_linuxtools_lttng_jni_factory_JniTraceVersion_ltt_1getTraceVersion*", so function expect to be reached from a cal to "*ltt_ getTraceVersion*" from the class "*JniTraceVersion*" in "*org.eclipse.linuxtools.lttng.jni.factory*" package.
The function will then callback the java function "*setTraceVersionFromC(int major, int minor)*" in the "*JniTraceVersion*" class to set the version number.

To make this function independant to library version installed on the system, it will always try to use dlopen call to bind to "*liblttvtraceread_loader.so*", if this library is not available/reachable, a segfault will occurs.
In case of an error (path is not a trace or could not get the version), "0" will be returned for both numbers.

2) ltt_initializeHandle(jstring libname)

ltt_initializeHandle load (if needed) the passed .so library and keep its function pointers into a special structure. This make it possible to use more than one "liblttvtraceread" library at the same time, at the cost of a slight overhead (around 15% for fully parsed events).
Mangling is "*Java_org_eclipse_linuxtools_lttng_jni_JniTrace_ltt_1initializeHandle*", so function expect to be reached from a cal to "*ltt_1initializeHandle*" from the class "*JniTrace*" in "*org.eclipse.linuxtools.lttng.jni*" package.

The function need the exact .so name (something like "*liblttvtraceread-2.6.so*") and could even take path. The function will then return an integer that represent the "handler" to the loaded library, making possible access the same functions in different libraries.
If the library does not exist or is not reachable, "-1" is returned.

Note : calling this function allocate memory in C that can't be free automatically by java. The function "*ltt_freeHandle*" (full mangling "*Java_org_eclipse_linuxtools_lttng_jni_JniTrace_ltt_1freeHandle*") need to be called by java so the memory get freed. This should be done by the "*finalize*" function in "*JniTrace*".

3) Version independant access to JNI functions

Once the library is (or libraries are) initialized using "*ltt_initializeHandle*", it is possible to access the same function in different version of the library without having to unload either one.
The call to "*ltt_initializeHandle*" should have returned an integer handle, this handle should then be given as first argument of any JNI functions. That ways, the loader should be able to call the correct version of the function using the function pointer that were loaded by "*ltt_initializeHandle*".

Passing the incorrect handle will result in unexpected behavior and could segfault the JVM. In case the handle was lost, calling "*ltt_initializeHandle*" again will return the existing handler (if any) or instanciate a new one (if the library was unloaded meanwhile).

org.eclipse.linuxtools.lttng.jni

First layer of the "java" side of the JNI.

Contains five components :

1. org.eclipse.linuxtools.lttng.jni
2. org.eclipse.linuxtools.lttng.jni_vX_Y
3. org.eclipse.linuxtools.lttng.jni.common
4. org.eclipse.linuxtools.lttng.jni.exception
5. org.eclipse.linuxtools.lttng.jni.factory

1) org.eclipse.linuxtools.lttng.jni

Main part of the plugin, contains the definition in Java of the C structure and architecture. The main entry point should always be "*JniTrace*" and the correct version of the trace should be returned by the "*JniTraceFactory*" in "*org.eclipse.linuxtools.lttng.jni.factory*".
All the functions that are calling the C side have the prefix "ltt_" and are declared with "native".
To ensure correct performance, frequent occuring objects such as event avoid memory allocation if it can be avoided and avoid calling to C unnecessary.

To ensure the Java name mangling can find the correct function even so we are using several libraries version at the same time, the classes in this package that need to access the C are declared "abstract" and "native" functions (C functions) take an extra integer as first argument to bind to the correct library handle (see the part "version independant access to jni function" in liblttvtraceread_loader part of this document).

2) org.eclipse.linuxtools.lttng.jni_vX_Y

Concrete (as opposition to "abstract") definition of the classes in "*org.eclipse.linuxtools.lttng.jni*".
These classes contain (for now) very few code. Their main use is to :
1- Ensure the java mangling is consistent so we can access transparently functions with the same name but in different C libraries.
2- Transparently (hopefully) support changes in the liblttvtraceread API.

Note that since we can't force "client" to Override constructor, each classes that need to allocate objects in "*org.eclipse.linuxtools.lttng.jni*" is forced to reimplement methods named "allocateNew...".
These methods are expected to allocate a new object of the expected type with the correct version and to return its reference. Althought not very elegant this solution should be flexible enought to support any trace version.

3) org.eclipse.linuxtools.lttng.jni.common

Common stuff shared between all JNI objects. Structure that old C pointers or timestamp are kept here.

4) org.eclipse.linuxtools.lttng.jni.exception

Class exception that need to be shared between different objects of the package.

5) org.eclipse.linuxtools.lttng.jni.factory

Structure needed to support multiple trace versions simultaneously.

Contain two components :
1 - JniTraceFactory :
Usual entry point to the “*JniTrace*”, throught “*getJniTrace(path, showDebug)*”. From the given path, the factory will attempt to determine to correct JNI version to use using “*JniTraceVersion*”. If successful, it will open a “*JniTrace*” of this version and return it to the client.

2 - JniTraceVersion :
This class perform necessary call to the JNI to find a library trace version.

Annexes

Java native functions name mangling

To understand the "why" of the architecture described above, we need to understand the way Java perform its name mangling from Java "native" function to C function names.

To illustrate it with an example, let's suppose we have the following code :

```
package com.test.my;
public class JniIsPainful {
	native void some_function(int x, long y, String str, JniIsPainful myobj);

	public void() {
		someFunction(1, 2L, "3", this);
	}
}
```

Althought the function name declared here is "some_function", Java expect the function name in C to be something like "*Java_com_test_my_JniIsPainful_some_1function*" (this can be obtained using the "javah" or "javap -s" utility on command line).
There does seem to be any way to control the name mangling java will be using, so the code, either on C and Java side, need to be ajusted accordingly. This, obviously, is a real problem in the case of advanced object oriented design like in the "factory" design pattern.

However, classes that derive from abstract classes does not override their parents name mangling; so to bypass this limitation, it is possible to declare all the "native" functions into an "abstract" class.

Note on Java/C types :

Also, note that JNI code between C and Java needs to use its own types. So from the example above, the complete function generated with its parameters would be :
"*Java_com_test_my_JniIsPainful_some_1function(JNIEnv *, jobject, jint, jlong, jstring, jobject)*".
Using the wrong type or using them without having properly converted them would result in a segfault of the JVM. This is especially important when playing with String/Char* or with Object/pointer.

Java internal storage of function names and missing System.unloadLibrary()

Note : everything here is pure speculation, as I did not read java's code, but this is how I understand the internal part of the JNI.

When a developper is loading a C library in Java (using System.loadLibrary call), the JVM is performing a number of things.

1. First, it checks if a library by that name can be found in system.java.path and if so, it will load it.
2. Then, when a “native” functions is called, java will look into an internal table to check if it knows what C function to call.
3. If no correlation is found, java will then try to find the function in the loaded library at the *expected name mangling* and save the correlation *Java function -> C function in an internal table* .
4. This table will be kept at least as long as the object is not garbage collected.

Question : What's happen if we try to access several functions with the same name (name mangling), but that are contained in different libraries ?
Answer : It is **not possible** to do such a thing using JNI, only the first library loaded will ever be considered, unless the garbage collector free (“unload”) it.
There is also very few guarantee “*unloadLibrary()*” will get called as the expected time (i.e. if the object still has an used reference, it will not get garbarge collected and so unload will not be done).

Why this is dumb :
Since the library is kept as long the object is not garbage collected, a workaround could have been to use different objects everytime we use different libraries. However, as the name mangling take into account the name of the class as well, this mean the function name in C would need to be different in each library as well.
This is not only a maintenance burden (the need to ajust every library versions in the case of a function name change) but it also prevents developpers to use advanced design patterns like factory.

Workaround :
The C programming language, however, is able to support different functions with the same name contained in different library. The functions “*dlopen*”, “*dlsym*” and “*dlclose*” (all part of “*dlfcn.h*”) can be used to manipulate (open, bind functions and close) libraries on the C side.
So to fool Java, we can use a “loader” library; this library is responsible to load either this or this version of a certain C library and to call the correct version depending on which one Java try to access, to the cost of a little overhead in performance.

How to suppport a new LTTng trace version

Usual case : no API change in LTTv

1 - libltvtraceread and libltvtraceread_loader :

First, make sure the numbering is correctly set in the C. The version number that control the build of the library can be found in "*ltt/Makefile.am*" of the LTTv directory. It should be like the following :

```
liblttvtraceread_la_LDFLAGS = -release 2.6
liblttvtraceread_loader_la_LDFLAGS = -release 2.6
```

As long the number after "-release" is the correct LTTng version, the build process should build the librairies correctly.

2 - org.eclipse.linuxtools.lttng.jni :

In Eclipse, the only needed step is to create a JNI directory for the new version, like the one org.eclipse.linuxtools.lttng.jni_v2_6.
There should be very few codes changes to do here, unless there is new special cases to support, in which case you might want to override some functions.
The only important line to change is the one that control which library is loaded; in "*JniTrace_vX_Y.java*", you should have, as example :

```
public class JniTrace_v2_6 extends JniTrace {
      private static final String LIBRARY_NAME = "liblttvtraceread-2.6.so";
```

This line tell the loader which C library to load, the numbering should be changed for the correct version.
Then, in factory, you need to add support for the version you wish to add; in "*JniTraceFactory*", in the package "*org.eclipse.linuxtools.lttng.jni.factory*", you should have :

```
public class JniTraceFactory {
      static final String TraceVersion_v2_3 = "2.3";
      static final String TraceVersion_v2_5 = "2.5";
      static final String TraceVersion_v2_6 = "2.6";

      static public JniTrace getJniTrace(String path) throws JniException {

            try {
                  JniTraceVersion traceVersion = new JniTraceVersion(path);
                  if (traceVersion.getVersionAsString().equals(TraceVersion_v2_6))
                  {
                        return new JniTrace_v2_6(path);
                  }
                  else if ...
```

Obviously, you need to adjust this code to support the new version of the library.

More complex case : API change in LTTv

This is a theoritical case as it never hapen yet, but the actual architecture should be able to support major API changes without much problems. Here is some hints of what should be done in such case.

1 - libltvtraceread and liblttvtraceread_loader :

The most important thing to verify is that "*ltt_get_trace_version()*" function in LTTv (in the file "*ltt/tracefile.c*" of the LTTv directory) is always able to return the version number of the trace, for the old format as well as for the "new" ones.

It is also important to make sure that the file "*ltt/jni_interface.c*" in the LTTv directory reflects the change and give a correct access to LTTv from Java.

Assuming that the new API changes conflict wih the older version, "liblttvtraceread_loader" will need to adjusted to support both. If the API changes are extreme, a suffixe might need to be added to all the JNI access to the new functions so that clients could easily (and knowingly) select the new or the old version of a certain function.

2 - org.eclipse.linuxtools.lttng.jni :

The current architecture is able to support multiple traces version with very few changes (as long as the API of LTTv is stable) but provide flexibility for the developper; if special cases are needed for certain versions, it can easily added by overriding methods of the abstract classes with a version dependant implementation, without disturbing older implementation.

However, if the architecture on the Java side need to be seriously changed to support complex API changes, it is important to remember that the name mangling of the C functions depends on the Java packages, classes and function names. Thus, in the case that the current Java JNI architecture can not support the new version, new abstract classes with new name (like a suffixe pointing to the new version) should be created to make sure the old ones are still accessible in the same ways.