Panama: A case study

With OpenSSL and Apache Tomcat ™

Rémy Maucherat

Software engineer at Red Hat

Working on Red Hat JBoss Web Server

Apache Tomcat committer since 2000

ASF member

Contents

Panama

The Panama project

Project developed by Oracle for inclusion in java.base

Replacement for JNI

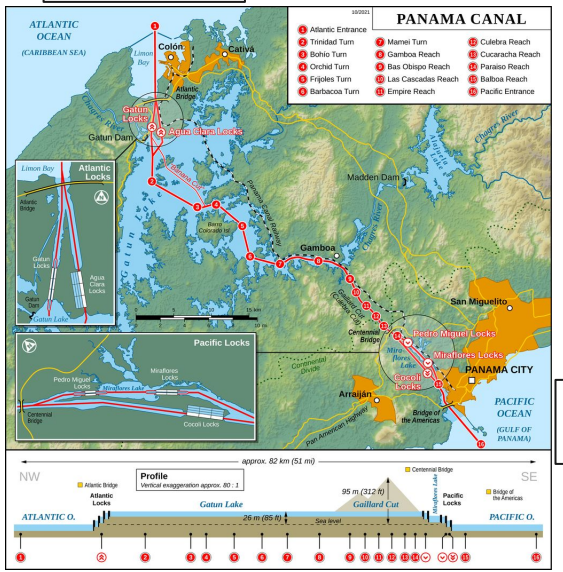
Improve safety and reliability

Better management of native memory

Replacement for various Unsafe based hacks

Incubating and JEPs since Java 14

Java code



Native code

MemorySession

Handles native access lifecycle

Allocations and deallocations

Explicit close or GC close

MemorySegment

The main API in Java 20 (previously also MemoryAddress and Addressable)

Native or heap memory, with associated segment size

Basically a pointer

Tied to an associated session

Does size and lifecycle checks -> safety

Memory layouts and native types

Allows modelling types and structures

Simple types are easy

Valhalla will provide support for additional types

Function descriptors and Method handles

FunctionDescriptor API allows describing the native calls to the JVM

Associate a native function to a matching method handle

Use the Linker API for that

Downcalls

Calls from Java to native

Use lookup to get the native symbol

Use the linker to get the method handle

Simply call the method handle with the right arguments

Downcall example

```
System.loadLibrary("ssl");

MemorySegment OpenSSL_versionSymbol = SymbolLookup.loaderLookup().find("OpenSSL_version").get();

MethodHandle OpenSSL_version = Linker.nativeLinker().downcallHandle(OpenSSL_versionSymbol,
FunctionDescriptor.of(ValueLayout.ADDRESS, ValueLayout.JAVA_INT));

System.out.println("Hello " + ((MemoryAddress) OpenSSL_version.invokeExact(0)).getUtf8String(0));
```

Upcalls

Calls from native to Java

Get a method handle from your Java method

The linker gives a memory segment containing a function pointer

Call the appropriate native downcall to set the function pointer

Upcall example

```
FunctionDescriptor openSSLCallbackVerifyFunctionDescriptor = FunctionDescriptor.of(ValueLayout.JAVA_INT, ValueLayout.JAVA_INT, ValueLayout.ADDRESS); /* typedef int (*SSL_verify_cb)(int preverify_ok, X509_STORE_CTX *x509_ctx); */

MethodHandle openSSLCallbackVerifyHandle = MethodHandles.lookup().findStatic(OpenSSLContext.class, "openSSLCallbackVerify", MethodType.methodType(int.class, int.class, MemorySegment.class));

MethodHandle SSL_CTX_set_verify = ...; // The OpenSSL downcall to set the function pointer

MemorySegment openSSLCallbackVerify = Linker.nativeLinker().upcallStub(openSSLCallbackVerifyHandle, openSSLCallbackVerifyFunctionDescriptor, state.contextMemorySession);

SSL_CTX_set_verify(state.sslCtx, /* int */ validationMode, openSSLCallbackVerify);
```

OpenSSL

All things TLS

Everything

Even more later

Extremely useful to Tomcat

Tomcat needs

TLS 1.3 with PHA

More key formats, ciphers, protocols as they are appear

Better performance

High level API for QUIC

Also used through Tomcat-native

Legacy code

Modernized and cleaned up with 2.0

Some platforms need static linking of OpenSSL (= CVEs !)

Productization needed

Cloud annoyances

OpenSSL API style

Factories and destructors for everything

Accessors and setters for everything

No need to model structures in Panama

Many callbacks needed

Tooling

Jextract

Helps write boilerplate code, handle library updates

Upcalls

Downcalls

Structures

Helps track Panama API changes across Java versions

Using jextract

Now easier to build from https://github.com/openjdk/jextract

Uses native library headers

Generates sources or classes

Skips functional macros

Big library means huge very verbose blob

Trimming down the blob

List native APIs actually used

Limit jextract to these APIs

Time consuming …

Using jextract

- With OpenSSL
- Config file

Design and coding

Java code we had

Working OpenSSL implementation for JSSE (not a full provider but close)

Rather well tested

Support for OpenSSL 3.0

Significant amount of Java code using a thinner native API (compared to APR connector)

Tomcat-native code

Needs translation to Java code using Panama

Then integrate into the Tomcat OpenSSL code

Lots of wrapper code, needless structures and state tracking

Surprisingly large amount of logic inside the JNI layer in some places (certs handling, init, OCSP)

Lifecycle and Memory management

Was already appropriate for Panama

Needs to remain GC based for safety

Process

Generate the full OpenSSL API with jextract

Write OpenSSL init code and test out Panama

Translate the rest of the tomcat-native code into the existing Java classes

Add state tracking for upcalls

Test, fix, improve, test, fix, improve, etc

Challenges

No #ifdef and precompiler

This is bad for us

Also no functional macros support

Have to target real API functions

OpenSSL API compatibility

Initial target was 1.1

API changes in 3.0, resolved by macros (which don't work with Panama)

Manual downcall declarations needed, with runtime version check

Extensive API changes would need a fully separate implementation

For example, support of LibreSSL is not doable as part of the same module

Memory leaks are easy

OpenSSL structures need cleanup

Memory segments point to the session

Bound downcalls unfortunately pins the session

API changes

Java 17 is the first LTS release with the incubating Panama (JEP 412)

Some changes in Java 18 (JEP 419)

Same in Java 19, moving to preview status (--enable-preview is enough) (JEP 424)

Same in Java 20, still preview, heavy API changes

Java 21 LTS is the target for the stable API

Targeting Java 17

Needs scary command line arguments (users will panic)

API is less refined than current

Functionally equivalent for us

Panama seems surprisingly stable

Ok for in house projects that cannot wait for Java 21, not ok for real productization

Instability

Instability when initially trying with Java 18

Mysterious behaviors, debugging was inconclusive

Had to report to the Panama team

Was caused by stack corruption with (lots of) upcalls

Thankfully fixed

Updating to a new Java

Monitor Panama API changes in openjdk/panama-foreign and evaluate impact

Code merge into next Java openjdk/jdk

Wait for jextract updates to target new API (needed for a large library like OpenSSL)

Migrate and test !

MemoryAddress removal in Java 20

Was simply a long, so now the hacks will be easy to spot again !

Must use MemorySegment for everything, which references its associated session

Good since it holds a reference to the session, so safer

But cleanup tasks on session close must hold on a long or a segment from another session

Refactoring ahead !

Results

It worked !

Surprisingly stable on Java 17

Fast enough on Java 17

Works out of the box if OpenSSL is a system library

With new Java versions, only "--enable-preview" needed (with startup warnings)

Needs to have multiple branches

Many API differences with each JEP

https://github.com/apache/tomcat/tree/main/modules/openssl-foreign : Tracks the Panama API (Java "20" right now, hopefully final API in Java 21), support for OpenSSL 1.1 and 3.0

https://github.com/apache/tomcat/tree/main/modules/openssl-java17 : Frozen for Java 17, support for OpenSSL 1.1 and 3.0, "stable"

Performance

A bit over 10% slower than JNI across Java 17 and 18

Custom Java 19 builds were about 5% slower

Probably not a glitch since all other testing results was unchanged

OpenSSL Demo

- With Java 17 (JEP 412)
- With Java "20"

Conclusion

Roadmap to stable

Good performance already with safer code

Expected stable with Java 21 (09/2023), so one year to go

Will then be integrated into Tomcat's source tree and ready to use

Should work out of the box on Java 21

tomcat-native will be maintained for a long time (EOL when Java 21 becomes required ?)