JJack – Using the JACK Audio Connection Kit with Java

Jens GULDEN
Formal Models, Logic and Programming (FLP),
Technical University Berlin
jgulden@cs.tu-berlin.de

Abstract

The JACK Audio Connection Kit is one of the de-facto standards on Linux operating systems for low-latency audio routing between multiple audio processing applications. As sound processing in an interconnected setup is not originally supported by the Java language itself, it has proven to be useful to provide interoperability between JACK and Java via the JJack bridge API. This allows creating music applications for use in a professional audio environment with the Java programming language.

Keywords

JACK Audio Connection Kit, Java, low-latency, Linux audio

1 Introduction

This article introduces JJack ([1]), a JACK-to-Java bridge API, which consists of a native bridge library providing access to the JACK ([2]) audio processing system, and a Java-side model API which reflects features of the native JACK system into the object-oriented world of Java. With JJack, audio processing in an interconnected virtual studio environment can be achieved on Linux and MacOS with software written in Java.

First, the JACK Audio Connection Kit gets briefly introduced in section 2. Section 3 shows the main principles of creating JACK clients with Java using JJack. An object-oriented model of basic client interconnectivity is provided by the high-level API of JJack as discussed in section 4. Section 5 sketches possible further applications by pointing out the JavaBeans-compatibility of JJack clients. In section 6, sample JJack clients from the JJack distribution archive are presented. Section 7 gives a note on how new features since JDK 1.4 have helped to optimize JJack's performance, and section 8 provides a short comparison between the core features of Java's internal Java Sound API and the JJack bridge API. Finally, a real-world application using JJack, the music sequencer *Frinika*, is presented in section 9, and section 10 gives a short concluding summary.

2 The JACK Audio Connection Kit

JACK ([2]) is a low-latency audio server which provides interconnectivity between different audio applications, such as software-synthesizers, multi-track sequencers, effect-modules, mixers, recording applications etc. While physically being assigned to usually just a single audio device, on the software-side the JACK server provides a virtual studio environment for any number of different software applications, and allows to interconnect them with virtual cables.

A possible JACK system setup, with one JJack-based application acting as a JACK client, is schematically displayed in Fig. 1.

3 Processing Audio with JACK and Java

Each native software application that is capable of acting as a JACK client provides a `process()`-function. This function gets called by the JACK system regularly to pass in audio data as input, and probably receive audio data as output from the `process()`-method.

The very basic idea of how a JACK-client works in Java is the same: a method `process()` is invoked repeatedly, providing audio data as input for processing. This data can then be used by the `process()`-method to be analyzed or transformed in any way, and to finally generate output audio data. Clients can also choose to restrict themselves to only process audio input data without generating any output (e. g. recording the audio input), or to

ignore any audio input and exclusively generate output (e. g. sound-generators, synthesizers etc.).

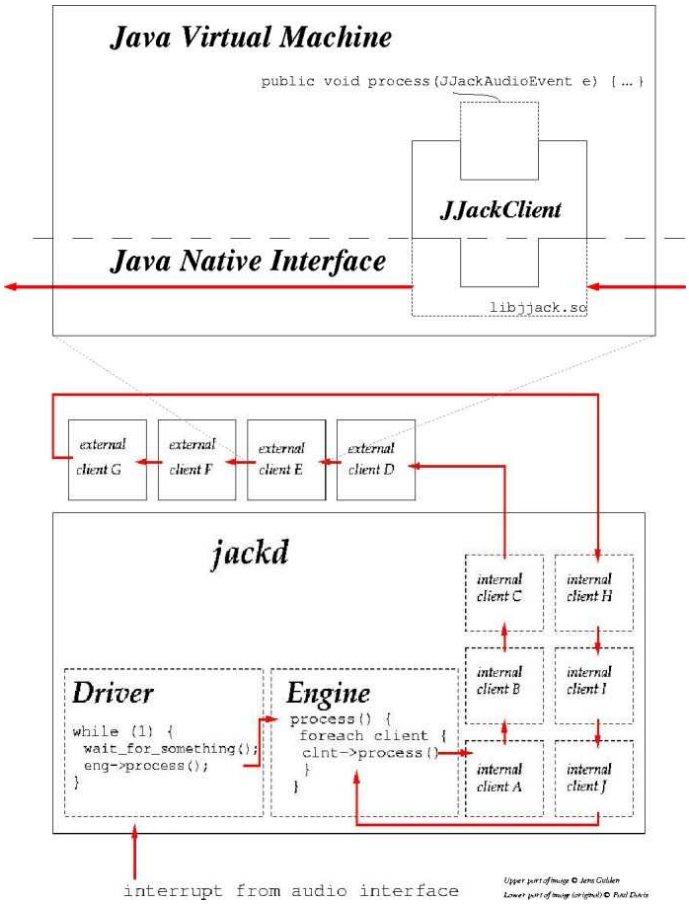

Fig. 1: JACK processing loop and JJack client

Each implementation of the interface `de.gulden.framework.jjack.JJackAudioProcessor` owns a method `process(JJackAudioEvent e)` which is responsible for processing audio data in this basic way. Example 1 gives an idea of how the `process()`-method can be implemented.

```
public void process(JJackAudioEvent e) {
    float v = getVolume(); // parameter from gui
    for (int i=0; i<e.countChannels(); i++) {
        FloatBuffer in = e.getInput(i);
        FloatBuffer out = e.getOutput(i);
        int cap = in.capacity();
        for (int j=0; j<cap; j++) {
            float a = in.get(j);
            a *= v;
            if (a>1.0) {
                a = 1.0f;
            } else if (a<-1.0) {
                a = -1.0f;
            }
            out.put(j, a);
        }
    }
}
```

Example 1: `process()`-method in Java

In the most simple scenario, a Java-written JACK client implements one basic `process()`-method, which will get called by a native bridge library with every callback from the JACK processing thread. The native bridge library appears as one native JACK client to the overall JACK system. (This situation had been depicted in Fig. 1.) Audio input data is provided via `JJackAudioEvent.getInput(channel)` as a Java-accessible `DirectFloatArray`. Audio output data is written to another `DirectFloatArray` retrieved by `JJackAudioEvent.getOutput(channel)`, which in turn will be passed as a memory-addressed native float-array back to the JACK server again.

Initializing JJack to be used in such a single-`process()`-method setup is done as demonstrated in example 2.

```
public class MyJJackClient implements JJackAudioProcessor {

    public static void main(String[] args) {

        // get JACK system's sample rate, initialize
        int sampleRate = JJackSystem.getSampleRate();
        System.out.print("Sample-rate: "+ sampleRate);

        // set single processing client
        MyJJackClient client = new MyJJackClient();
        JJackSystem.setClient(client);

        // ...
    }

    public void process(JJackAudioEvent e) {
        // ... (example 1) ...
    }

}
```

Example 2: using a single `process()`-method

4 JJack high-level API

Besides the 1:1 reflection of a native JACK client's `process()`-method as described in the previous section, JJack also contains a basic object-oriented framework for interconnecting multiple `JJackAudioProcessors` inside one running virtual machine, making them appear as one complex JACK client to the native JACK system. The high-level API is an add-on to JJack's basic functionality. It can be used to model complex audio processors which internally consist of multiple `JJackAudioProcessors` (or derived classes) in a Java-style, object-oriented manner on the Java side.

When using JJack's high-level API, the audio data is internally routed among the connected `JJackAudioProcessors` within one Java virtual machine. The overall combined setup of JJack-

processors appears as one single native JACK client to the overall native JACK system. This situation is shown in Fig. 3.

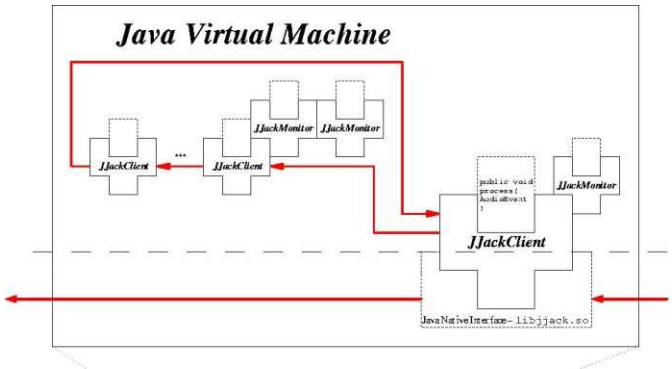

Fig. 3: Using the high-level API with multiple interconnected clients

Some core elements of the high-level API are displayed by the UML class-diagram in Fig. 4.

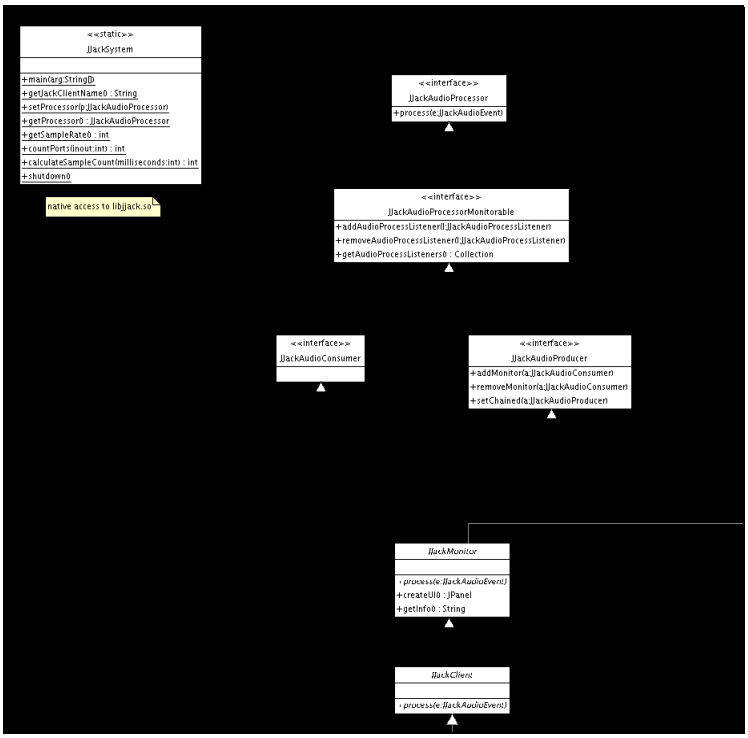

Fig. 4: JJack API UML class-diagram (partial)

5 JavaBeans compatibility

Interconnecting JJack clients is achieved by a JavaBeans-compatible event mechanism. As a consequence, it is possible to configure and interconnect multiple JJack clients inside a JavaBeans development-environment like Sun's *Bean Builder* ([4]). All clients combined inside one Java virtual machine appear as a single native client to the JACK system. Figure 5 demonstrates the use of the *Bean Builder* for creating setups of multiple JJack clients.

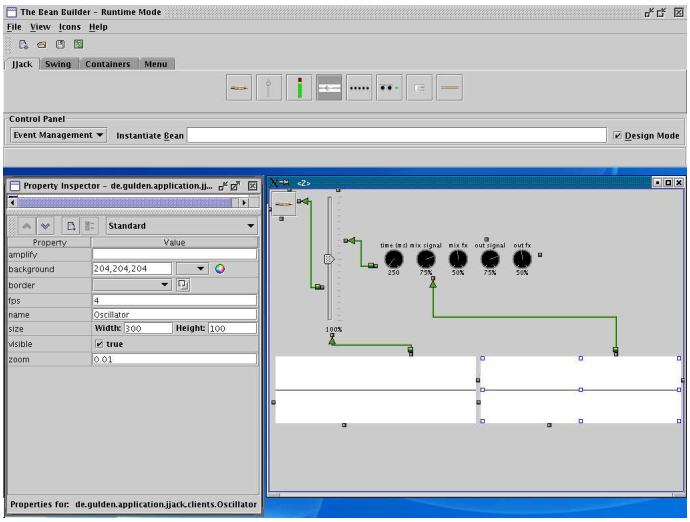

Fig. 5: JJack clients configured as JavaBeans in the BeanBuilder

6 Example clients

Some example JJack clients are included in the distribution archive, together with source-code. They can act as a starting point to learn how to use the JJack API. The examples are

GAIN:
class `de.gulden.application.jjack.clients.Gain`, a simple volume control (linear multiplication of the signal amplitude).

VU:
class `de.gulden.application.jjack.clients.VU`, a monitor client (input only) for displaying the average signal amplitude.

OSCILLATOR:
`de.gulden.application.jjack.clients.Oscillator`, a monitor client (input only) for displaying the signal as a waveform graph.

DELAY:
class `de.gulden.application.jjack.clients.Delay`, adds an echo effect to the audio signal.

GATE:
class `de.gulden.application.jjack.clients.Gate`, a noise gate that suppresses audio signal below a given threshold value. Only if the signal is loud enough, it will be passed through.

CHANNEL:
`de.gulden.application.jjack.clients.Channel`,

selects one channel from a multi-channel input and routes it to the mono output channel #0.

CABLE:
class `de.gulden.application.jjack.clients.Cable`, passes the audio signal through without any change. This is just a null-client for demonstration.

Fig. 6 shows a combination of example clients in operation.

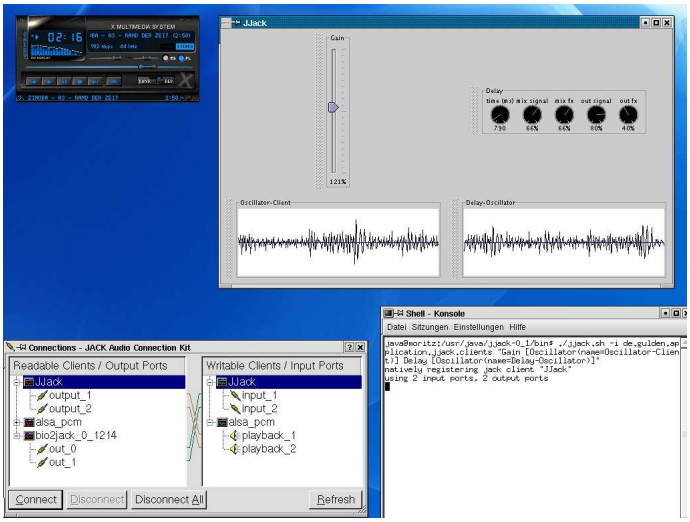

Fig. 6: Screenshot showing a combination of example clients in operation

The source-code implementing the DELAY client might help giving insight in how digital audio processing is done with Java and JJack. The source-code of the `process()`-method is listed in example 3.

7 Used features from JDK 1.4

Bridging between native JACK and Java can be achieved efficiently thanks to new “hardware-near” extensions to the Java standard API. Since JDK version 1.4, Java contains the package `java.nio.*` which allows the mapping of physical memory to Java-accessible arrays of choosable data-types ([4]). By using the interface `java.nio.FloatBuffer` and its implementation `DirectFloatBuffer`, no data-format conversion and even no copying of data is required to interface JACK's processing thread with code running inside the Java virtual machine.

This major advantage of JDK 1.4's features are the reason why JJack requires Java version 1.4 or above, versions up to 1.3 cannot be used with JJack.

```
/**
 * class:  de.gulden.application.jjack.clients.Delay
 *         (implements
 *         de.gulden.framework.jjack.JJackAudioProcessor)
 * method: void process(JJackAudioEvent e)
 *
 * Applies a classical digital delay, seperately
 * for each channel.
 */
public void process(JJackAudioEvent e) {
  int delaytime = getTime();
  float mixSignal = (float)getMixSignal() / 100;
  float mixFx = (float)getMixFx() / 100;
  float outSignal = (float)getOutSignal() / 100;
  float outFx = (float)getOutFx() / 100;
  int sampleRate = getSampleRate();
  int diff = delaytime * sampleRate / 1000 ;
  int channels = e.countChannels(); // number of channels
  if (ring == null) { // first call, init ringbuffers
    ring = new RingFloat[channels];
    for (int i = 0; i < channels; i++) {
      ring[i] = new RingFloat(diff);
    }
  }
  for (int i=0; i < channels; i++) {
    RingFloat r = ring[i];
    r.ensureCapacity(diff);
    FloatBuffer in = e.getInput(i);   // input buffer
    FloatBuffer out = e.getOutput(i); // output buffer
    int cap = in.capacity(); // number of samples available
    for (int j=0; j<cap; j++) {
      float signal = in.get(j); // read input signal
      float fx = r.get(diff);
      float mix = signal * mixSignal + fx * mixFx;
      float ou = signal * outSignal + fx * outFx;
      r.put(mix); // remember for delay
      out.put(j, ou); // write output signal
    }
  }
}
```

Example 3: `process()`-method of the Delay example client

8 Comparison between JJack and the Java Sound API

Since JDK version 1.3, the Java Sound API ([5]) is part of the Java language standard. As JJack and the Java Sound API differ in several aspects, it is important to be aware of the differences when deciding which audio processing approach is best suited for a software project's needs.

Table 1 gives an explanatory overview on the most relevant differences between JJack and the Java Sound API.

	JJack	JavaSound
interoper-ability	high, virtual cable-connectivity	low, device I/O oriented
timing latency[1]	JACK timing, potential low-latency below 5 ms on fast machines	device-dependent, varying among OSs and JDK versions (up to 200 ms latency JDK1.4/Linux, below 5 ms with JDK1.5 and higher)
realtime-capability	yes (application can run in a user-thread, JACK in a realtime thread)	no (unless the whole JVM runs in a realtime thread)
process architecture	pull-architecture (the JACK thread calls the `process()`-method)	push-architecture (application is responsible for delivering data on time)
internal data -format	32-bit float (less aliasing and higher performance expected, less conversions between interconnected applications)	16-bit integer (faster processing for simple clients, additional floating-point conversion required for complex tasks)
number of channels	any	according to hardware driver or software mixer
operating system	Linux or Mac (JACK required)	Any (JDK >= 1.3)

Table 1: Comparison JJack – Java Sound API

[1]Latency tests have been performed with a two-computers setup, using the benchmark application `de.gulden.framework.jjack.util.benchmark.AudioBenchmark` (available via JJack's concurrent versions system, CVS).

9 Real-world application

The music sequencer application *Frinika* ([7]) is entirely written in Java and serves as a real-world example for the use of JJack. On Linux systems, *Frinika* detects the available audio system, and if JACK can be accessed successfully, the application runs on top of JJack to handle its audio I/O.

Since version 0.3.0, *Frinika* supports multi-track audio recording, the accuracy of which is again based on JACK through JJack on Linux systems.

Fig. 7 shows a screenshot of the *Frinika* music sequencer in operation.

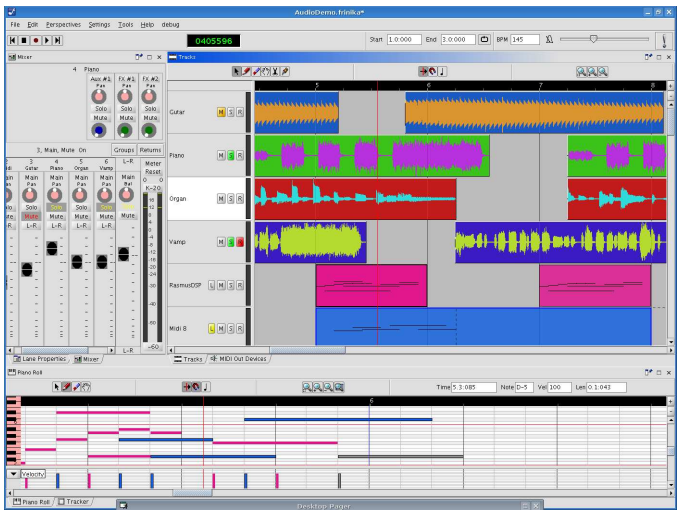

Fig. 7: Screenshot of the *Frinika* music sequencer

10 Conclusion

JJack has proven to be a serious alternative to the standard Java Sound API natively shipped with the Java programming language. Especially music applications for Linux and MacOS that are supposed to interoperate with other audio-processing software can benefit from interfacing with the JACK audio system instead of relying on the audio support internally provided by Java.

The overall architecture of a JACK-based audio application appears more elegant through the continuous use of floating-point arithmetic. It also is expected to be more stable with regard to timing issues, which is suggested by the pull-approach of the JACK processing architecture and the potential use of realtime-capabilities. Depending on the system configuration and driver support, using JJack may also offer lower latency times.

References

[1] Gulden, J., *JJack – JACK to Java bridge API*, software, http://jjack.berlios.de/, licensed under the GNU Lesser General Public License (LGPL)

[2] Davis, P. et al, *JACK – JACK Audio Connection Kit*, software, http://jackaudio.org/, licensed under the GNU General Public License (GPL) and GNU Lesser General Public License (LGPL)

[3] Sun Microsystems, *Bean Builder,* a visual programming environment demonstrating the assembly of applications by joining live instances of JavaBeans, software, https://bean-builder.dev.java.net/, licensed under the Berkeley Software Distribution (BSD) License

[4] Sun Microsystems, *New I/O APIs*, http://java.sun.com/j2se/1.4.2/docs/guide/nio/

[5] Sun Microsystems, *Java Sound API,* http://java.sun.com/products/java-media/sound/

[6] Schmeder, A., *PyJack,* JACK client for Python, http://a2hd.com/software, licensed under the GNU General Public License (GPL)

[7] Salomonsen, P. et al, *Frinika music workstation*, software, http://www.frinika.com/, licensed under the GNU General Public License (GPL)

[8] GNU Software Foundation, *GNU General Public License (GPL)*, legal license, http://www.gnu.org/licenses/gpl.txt

[9] GNU Software Foundation, *GNU Lesser General Public License (GPL)*, legal license, http://www.gnu.org/licenses/lgpl.txt

Figure 1 partially by Paul Davis