



vlcj-pro

User Guide

Version 2.0.0 (Beta 2)

www.capricasoftware.co.uk

March 2021

Table of Contents

Introduction.....	3
Changes Since vlcj-pro 1.x.....	3
Background.....	4
Features.....	6
Supported Platforms.....	8
OSX Not Supported.....	8
OSX Workaround.....	8
Future OSX Support.....	8
Getting Started.....	9
Installation.....	9
Dependencies.....	9
Client API.....	10
Component Factory.....	10
Out-Of-Process Media Player Component.....	10
Component Listeners.....	11
Media Player Event Listeners.....	12
Video Surface.....	12
Out-Of-Process Media Player.....	13
Media Player Exceptions.....	13
Latency.....	14
User Interface Responsiveness.....	15
Special Cases.....	15
Port Acquisition.....	16
Application Termination.....	17
Shut-down Hook.....	17
Configuration.....	18
Client Configuration.....	19
Server Configuration.....	21
Additional Configuration Options.....	22
Logging.....	23
System Properties.....	23
Logging Considerations.....	24
Logback.....	24
Application Design Recommendations.....	25
Demo.....	26
Screenshots.....	27
Demo Application with Snapshot Capture.....	27
Independent Volume Controls.....	28
Demo Logging Configuration.....	28
Footprint of vlcj-pro Code in the Demo.....	28
Limitations.....	29
Native Media Player Resources.....	29
Native Resource Leaks.....	29
System Hangs.....	29
Summary of Limitations.....	30
Release Notes.....	30
Licensing.....	31
Contact.....	31

Introduction

This document describes the vlcj-pro application programming framework and how to use it in your own applications to build robust media player applications.

The intended audience for this document is for experienced Java developers, ideally developers who are already familiar with vlcj, who need to use vlcj-pro with their own applications.

The document is split into five major sections. The first deals with the background of vlcj-pro, why it is needed and what it provides; the second deals with how to implement an application that uses vlcj-pro; the third section deals with the demo/reference application provided with vlcj-pro; the fourth section describes some limitations of vlcj-pro that application developers need to be aware of; the final section contains details on licensing.

This guide is neither a Java nor Swing development tutorial and assumes a level of familiarity with the same.

Changes Since vlcj-pro 1.x

The vlcj-pro 2 series represents a significant investment in development effort to bring vlcj-pro up-to-date with the latest vlcj and LibVLC versions.

For vlcj-pro 2, you must use vlcj 4.0.8 or later, and you must use VLC 3.0.0 or later. Earlier versions are not supported and likely will not work *at all*.

The major changes are:

- update all dependencies to latest
- redesign API to mirror the major API changes that came with vlcj-4's API rework
- rework of out of process media player component start-up and recovery
 - port number acquisition now works on a round-robin basis, rather than always grabbing the first port in the range – this is important for the next item...
 - if some other process is using one of the ports configured for vlcj-pro, this will now be detected, giving the application a chance to try again with a new port
 - creating the resources required for a remote media player is now more robust and has a cleaner “fail” state, which means it is much easier and more reliable to try and restart a previously failed component (this is explained in detail later in this document)
- added support for almost all new features that came with vlcj-4 and LibVLC 3.0.0
- pass additional LibVLC initialisation options from the client for remote media players
- general code-hardening
- minimum requirement is now Java 8

Background

The vlcj project provides a way to embed a native VLC media player inside a Java application using LibVLC.

Such applications almost always have the media player executing in the same process, in fact the same Java Virtual Machine, as the application itself. This is the normal thing to do, it is very simple and for most typical application use-cases this works just fine.

Using native code has consequences.

When a Java application interfaces with native code this native code interaction is “unmanaged” - the Java Virtual Machine has no real control over the native code and it can not guarantee the correct operation of that native code. If an error occurs during execution of the native code, the entire Java Virtual Machine will terminate instantly with no chance for the Java application to handle the error and recover.

In practical terms for a vlcj application, what this means is that if an error occurs in the native VLC media player that vlcj embeds, then the entire application will crash instantly with no chance for recovery.

Such crashes are basically non-existent if a vlcj application is implemented correctly, and especially where the vlcj application only has a single media player embedded within it, but nevertheless there is no cast-iron guarantee that the native code will not fail.

When building vlcj applications that need multiple media players in the same application ordinarily each of those media players will also run in the same process, and the same Java Virtual Machine, as the application itself. This is potentially problematic because now there are concurrency and re-entrancy issues that must be considered.

Take for example the vlcj method to play new media. When this method is invoked it gets translated to a native call. That native call creates various native media player resources, loads various VLC plug-ins and eventually the media will start playing. The key point is that the native method executes *asynchronously*.

There is now the possibility that while this native method is executing asynchronously another method invocation from the Java side is made that will cause another native method to be called concurrently. Since the native library is in the same process as the first media player, this second method invocation causes the native library to be re-entered.

Re-entrancy on its own is not necessarily a problem, and indeed LibVLC itself is claimed to be safely re-entrant. However, VLC and its plug-ins use a lot of third-party native libraries and experience shows that one or more of these libraries are not re-entrant.

A library is not re-entrant if, for example, it has some global state (global to the current process) – if more than one call is executing concurrently the state at any particular point in time might be inconsistent, or overwritten, or whatever else ultimately leading to a crash and an instant

termination of the Java Virtual Machine.

The solution to mitigate these problems is to have each native media player run inside its own operating system process (its own Java Virtual Machine). Each unique process then gets its own copy of the native libraries and there are therefore no longer the same re-entrancy and concurrency problems.

If one of the external processes crashes, the main application running in its own Java Virtual Machine is isolated from that crash and will keep on running. The external process failure can be detected and the main application can try and recover, most often by creating yet another external process and restarting the media.

Clearly using these out-of-process media players is most beneficial for applications that require multiple media players, but it can also be used in applications that have only a single media player to provide a safety net in case the native code does actually fail – remember when native code is involved, there is no cast-iron guarantee from the Java Virtual Machine.

There is an additional major benefit of running each media player inside a separate process – it is the only way to get independent volume controls. For multiple in-process media players all of those media players share the same volume controls, changing the volume or muting one media player will make that same volume change or mute on all media players. With out-of-process media players each media player is effectively its own independent application and the volume and muting can be controlled independently.

The major disadvantage of using out-of-process media players is that, simply put, they are much more complex to implement than in-process media players. This is where vlcj-pro comes in.

The vlcj-pro application programming framework provides a way to easily implement robust out-of-process vlcj media players.

Features

Some of the key features provided by the vlcj-pro application programming framework are:

- The media player interface for an out-of-process media player is (almost) exactly the same, in terms of specification, as a regular vlcj media player – this means if you know how to use a vlcj media player, the same methods will (for the most part at least) work in exactly the same way.
- The vast majority of vlcj media player features are made available to out-of-process media players, using the exact same API as normal media players – playback controls, volume controls, track information, logo, marquee, video snapshots, rewind, skip, chapter navigation, audio equalizer, video adjustments and so on are all supported;
- There are a small number of features that are *not* supported, this is mainly where a handle to a native resource may be required (e.g. a native media instance) – in these cases it is not possible to send or receive those native handles from the external media player process, and indeed it makes no real sense to do so. Nevertheless if it is possible to sensibly support a particular media player method, it will be supported;
- Media player events are mostly the same as with a normal vlcj application, there is however a key difference. The media player events now originate from a media player running inside an external process. These remote events must be sent from the external media process back to the main application. This happens transparently, but a client application can not register for events directly on the media player instance it has a reference to (the events do not come from that client-side media player, it is basically a stub) it must instead register media player event listeners on the media player component not the media player - this is termed “deferred” event registration in vlcj-pro;
- When registering event listeners using the deferred approach there is an added benefit. Event listeners are usually registered by a client application on a media player instance once. One of the features of vlcj-pro is that if media player crashes a new one can be restarted in its place, but an application really does not want to have to re-register all of those event listeners again. The deferred registration can transparently handle a switch-over from a broken media player to a new one without the client application needing to do anything else.
- An out-of-process media player component with a simple interface – the client application is never exposed to the remote process configuration;
- Lazy initialisation of external media players. Creating a new process and doing the necessary hand-shake between the client application and the remote process is quite a heavyweight process. On its own it does not take a significant amount of time to start an external media process, the problem is when the number of media player instances increases the overall time taken to create all of them can become significant. So when creating a media player component the actual stub media player inside that component and the components required for the remote communications are created lazily on first access. If an

application wants instead to eagerly force creation of all media players it simply needs to iterate them and request the media player instance;

- Easy, clean, shut-down. A media player component is managed by a factory, that factory keeps track of all of the processes it has launched. On shut-down of the application it is important to do this cleanly and part of this process means telling all of the external media player processes to shut themselves down. There can be significant overhead doing this, and again as the number of media players increases there may be a significant lag when shutting down the application. For this reason the media player component factory spawns threads to concurrently shut-down the external media player processes – this is much quicker than shutting down each one individually. A client application makes a single call on the factory to free the external processes. The application can even choose to background thread this and provide a progress dialog box if it wants to, although it is likely to be quick enough that this is not necessary.
- Each external media player process implements a heartbeat that calls back to the main application to make sure it is still alive. This is useful to prevent “orphaned” processes from hanging around if something goes wrong or some disorderly shut-down occurs in the main client application. If the media player process does not receive a heartbeat response it assumes the main application has gone away and shuts itself down.

Despite the huge complexity involved with robust out-of-process media players, the killer feature of vlcj-pro is that it hides just about everything it can from the client application. A client application mostly deals with a mirror of a regular vlcj media player interface as if it were in the same application. The only real fault recovery that an application should implement itself is to listen for a notification that a media player failed, and to start a new one in its place.

Supported Platforms

vlcj-pro is designed for Java 1.8 and later, on Linux and Windows.

The vlcj-pro 2.x series requires at least vlcj 4.0.1 and at least VLC 3.0.0 is required. No earlier versions are supported at all.

OSX is not supported.

OSX Not Supported

The reason that OSX is not supported is two-fold.

First, the way vlcj-pro works is that a main application hosts all of the video surfaces for the media players and each media player is in a separate process. Each media player process is told where to render the video. On OSX, it is not permitted for one process to access a user interface component (and hence a video surface) in another process.

Second, with Java 1.7 and later on OSX there is no longer any heavyweight AWT – vlcj, through LibVLC, requires a heavyweight component (an AWT Canvas) to render the video into.

For multiple media player instances on OSX, the recommended approach is to use the so-called “direct rendering” media players provided by vlcj. Direct rendering media players are more limited than regular vlcj media players but they have one distinct advantage – multiple instances of direct media players all in the same process works reliably and does not suffer the same problems as the native heavyweight media players. There are however plenty of down-sides to using direct media players so they are generally not recommended for other platforms.

OSX Workaround

A solution that has been mooted for Java 1.7 and later on OSX is to run a Windows Java Virtual Machine on OSX using Wine.

This approach comes without any guarantees and is only for the brave or desperate to consider.

Future OSX Support

There is a possibility for some limited support of OSX in the future, but even so it is unlikely that there will be feature parity with Linux and Windows.

The **Window** component is a heavyweight component on modern OSX so in theory it would be possible to build an application with vlcj-pro that was based on windows rather than canvases. However a window solution has severe compromises such as not being able to embed the window into another view, and inability to handle resizing of the video.

This is not expected to be officially supported any time soon by vlcj-pro.

Getting Started

This section will describe how to get started developing robust out-of-process media player applications using vlcj-pro and vlcj.

Installation

vlcj-pro is a commercial product so it is not available in any public maven repository.

You should unpack your commercial vlcj-pro distribution package and either install the vlcj-pro jars, and the dependencies if needed, into your own local maven repository, or simply add the jar files to your project in your IDE.

Dependencies

vlcj-pro currently has the following direct dependencies:

<i>GroupId</i>	<i>ArtifactId</i>	<i>Version</i>
com.google.guava	guava	28.0-jre
javax.annotation	javax.annotation-api	1.3.2
net.bytebuddy	byte-buddy	1.9.13
org.apache.commons	commons-exec	1.3
org.apache.commons	commons-lang3	3.8.1
org.apache.thrift	libthrift	0.12.0
org.slf4j	slf4j-api	1.7.26
org.yaml	snakeyaml	1.23
uk.co.caprica	native-streams	1.0.0
uk.co.caprica	vlcj	4.0.8

Generally, at the time a vlcj-pro release is made, the latest available release versions of these dependencies are used.

The jar files for these dependencies, and their own transitive dependencies, need to be added to your project.

The recommended way is to manually install the vlcj-pro artefacts into your own repository and use maven.

Client API

The client API is contained in the `uk.co.caprica.vlcjpro.client.player` package.

There are a small number of classes in this package, but two classes in particular are key: the media player component factory and the media player component implementation.

In fact, the client API exposes less than 10 classes!

Component Factory

The first step is to create an instance of the out-of-process media player component factory – this factory hands out out-of-process media player components and keeps track of them so that they may be easily shut down properly when your application terminates.

This class is named `OutOfProcessMediaPlayerComponentFactory` and can be created thus:

```
OutOfProcessMediaPlayerComponentFactory factory =  
    new OutOfProcessMediaPlayerComponentFactory();
```

Your application should keep a reference to the factory so it can be properly shut down later.

Out-Of-Process Media Player Component

Now you have your factory instance, it is easy to create media player components. Create as many as you need (this example arbitrarily creates 4):

```
OutOfProcessMediaPlayerComponent[] components =  
    new OutOfProcessMediaPlayerComponent[4];  
for (int i = 0; i < 4; i++) {  
    components[i] = factory.newOutOfProcessMediaPlayerComponent();  
}
```

It will likely be most convenient for your application to keep references to these media player components since your application will be using them a lot later, but you can always get them again from the factory (if you know the index):

```
OutOfProcessMediaPlayerComponent component = factory.component(2);
```

At this point you have a reference to what could be considered a “stub” component – the external process has not yet been created since it is lazily created on first access. This may be a convenient time to add the various out-of-process component listeners and media player event listeners that your application uses.

Component Listeners

A component listener is used to notify life-cycle changes:

```
component.addOutOfProcessMediaPlayerComponentListener(componentListener);
```

The listener specification is simple:

```
public interface OutOfProcessMediaPlayerComponentListener {  
  
    void componentStarted(OutOfProcessMediaPlayerComponent component);  
  
    void componentStopped(OutOfProcessMediaPlayerComponent component,  
        MediaPlayerProcessExitCode exitCode);  
}
```

The listener interface only has two methods, but for convenience an adapter class is available for you to subclass and implement only the event you are interested in:

```
public class OutOfProcessMediaPlayerComponentAdapter  
    implements OutOfProcessMediaPlayerComponentListener {  
  
    @Override  
    public void componentStarted(OutOfProcessMediaPlayerComponent component) {  
    }  
  
    @Override  
    public void componentStopped(OutOfProcessMediaPlayerComponent component,  
        MediaPlayerProcessExitCode exitCode) {  
    }  
}
```

The `componentStarted` method is invoked when a new external process for the out-of-process media player has been created and properly initialised, confirmed by a synchronisation handshake between the component and the remote process.

The `componentStopped` method is invoked when the external process goes away for whatever reason, or there is some other failure when trying to communicate with the external process. At the time this event is raised, the component has already cleaned up all of the resources previously allocated to manage the external process so the component is ready for a re-start.

When an application receives a `componentStopped` notification, to recover the situation and create a new external media player process, all the application need do is invoke either the `start()` or `mediaPlayer()` method on the component instance that failed. All previously registered listeners will still work and will be associated with the new process – an application need not re-establish any listeners.

A `componentStopped` notification will be raised during normal, orderly, shut-down of the component factory, the type of failure can be discriminated by inspecting the value of the `exitCode` parameter – in this case, the exit code will be the value `NORMAL`.

You can add any number of such listeners.

You can optionally share listener implementations amongst many different component instances since each event callback method contains a reference to the component that raised the event.

Note that there is no guarantee which thread will deliver the event notifications. If you update user interface state in response to an event notification you must take the usual steps and use the `invokeLater()` method on the `SwingUtilities` class.

Media Player Event Listeners

The media player event listener is a mirror of the corresponding interface as specified by `vlcj`.

Rather than adding the listener directly to the media player instance, it must be added to the component instance:

```
component.addMediaPlayerEventListener(mediaPlayerEventListener);
```

The listener implementation, or extension of the corresponding event adapter class, is almost exactly the same as if you were using a regular `vlcj` in-process media player event listener.

You can add any number of such listeners.

You can optionally share listener implementations amongst many different media players since each event callback method contains a reference to the media player that raised the event.

Note that there is no guarantee which thread will deliver the event notifications. If you update user interface state in response to an event notification you must take the usual steps and use the `invokeLater()` method on the `SwingUtilities` class.

Video Surface

So far the out-of-process media player component is not associated with a video surface, usually the `Canvas`, that will be used to render the video. The video surface needs to be associated with the media player component and this must be done before requesting the media player for the first time (as explained in the next section).

```
component.setVideoSurface(videoSurface);
```

The video surface is set indirectly, i.e. it is set on the component, not the media player itself.

There are a number of video surface implementations provided, the most commonly used is the `ComponentOutOfProcessVideoSurface` that wraps an AWT `Component`, but it is possible to use any implementation of an `OutOfProcessVideoSurface` which return a native window handle, for example an SWT `Composite` could be used.

A video surface is optional – you can still get the benefits of using out of process media players even if you are only playing audio.

It is possible, but not recommended, for an application to explicitly and directly set the video surface on the media player, using the regular `mediaPlayer.setVideoSurface()` method, at any time if it so wishes.

Out-Of-Process Media Player

At this point, you have a factory, one or more out-of-process media player component instances, registered component life-cycle listeners and registered media player event listeners.

The creation of the out-of-process media player itself is pending the first call to the component `mediaPlayer()` or `start()` method. Invoking this method, the first time, transparently spawns a new process, creates the necessary components to handle the remote process communication and returns a reference to the new media player. Subsequent invocations of this method will simply return the media player reference.

```
OutOfProcessMediaPlayer mediaPlayer = component.mediaPlayer();
```

If the external process fails in some way, the out-of-process media player component will clean up in an orderly fashion and return the component to an initial state so that it can be re-started. Re-starting the component simply means the next invocation of the `mediaPlayer()` method.

It is possible that this `mediaPlayer()` method fails and returns `null` – this will only happen if it was not possible to start a new process. If this happens there is likely a serious underlying problem that can not be recovered from, although the application could try again later if it so desired.

The media player reference returned by this method is to all intents and purposes the same as if you were using a regular vlcj in-process media player.

Even though behind this interface there is a remote procedure invocation being made to an external process, your own application never needs to concern itself with that. In fact, your own application does not even need to catch any exception that may be raised during this remote procedure call since the exception handling is transparently built-in to the object referenced by the media player reference. If any such error occurs, the external process will be shut down in an orderly manner and the application will receive a life-cycle event (as described previously).

Media Player Exceptions

An exception in the out-of-process media player is treated as a *total failure* of the component and will cause the media player process to terminate.

In most cases this is exactly what you would expect.

Latency

You need to be aware that even though the interface for an out-of-process media player looks the same as that for a regular vlcj media player, there is much more going on behind the scenes to complete the method invocation. The method invocation and its arguments must be serialised and sent via a network socket to the external process. This is then deserialised by the external process and translated into a method call on an actual native media player instance. The result of the method call is then serialised and sent back through the network socket to the calling application. The calling application then deserialises the response so the value can be returned to the caller.

This processing obviously will introduce latency. For the vast majority of cases this latency will not cause any problems and likely will not even be noticeable.

However, there is a common use-case where this latency must be explicitly considered and that is handling slider value change notifications in a user interface (e.g. volume, position, and video adjustments). There are two ways to handle such change notifications: the first is to track the value of the slider while the user is dragging it, and on each notification make a method call; the second is to ignore the intermediate value changes and only make a method call when the user has finished dragging. There will clearly be more latency with the first approach since it will be making many more method invocations.

Swing provides the `slider.getValueIsAdjusting()` method so you can make a determination.

It is up to you as the developer of the client application to decide which approach is acceptable for you.

User Interface Responsiveness

If you discover a situation where the latency encountered when making a media player method call from your application to an out-of-process media player becomes noticeable you may need to consider pushing at least some of your out-of-process media player method calls to a background thread.

Consider what might happen if you include in your user interface a button that is used to capture a video snapshot and then 'download' that snapshot from the external process back to your application. Clearly this is one of the more significant situations in which latency may occur due to the captured snapshot image being serialised and sent via a network communications channel back to your application. In this case what you might see is that when you press the snapshot button, the user interface becomes momentarily unresponsive while the remote method invocation is being made.

Perhaps the most significant situation where this might arise is when creating the out of process media player components and starting them for the first time. There is a lot going on when a component is started and it can take a significant amount of time to complete.

There is a severe constraint that you must take into account if you intend to push your out-of-process media calls to a background thread: ***invocations of methods on access to media player must be single-threaded or serialised***. This is critical as those method invocations, their method arguments and their results are serialised and transferred via a network socket – the communications protocol simply does not allow for 'multiplexing' messages concurrently. It is up to your application to ensure single-threaded or serial access to the out-of-process media player.

In practice the latency might not even be noticeable. Nevertheless it illustrates the point that you, after appropriate evaluation, may need to push some of your remote method invocations to a background thread.

Special Cases

When dealing with logo and marquee, there can be many method invocations needed to achieve the desired effect – for example setting the position, the opacity, for the logo there is the actual logo image and for the marquee there is the text, the text size, the time-out and so on. Ordinarily these are separate method calls.

It is however possible to construct a **Logo** or **Marquee** instance and apply them as one method call each rather than multiple individual method calls for their individual attributes:

```
Logo logo = ...
mediaPlayer.setLogo(logo);

Marquee marquee = ...
mediaPlayer.setMarquee(marquee);
```

These are new methods added to vlcj to support vlcj-pro.

Port Acquisition

The communications between a client application and the external media player processes take place over network sockets. Your application uses configuration (this is described in a later section) to specify a range of port numbers that are available for vlcj-pro to use, and behind the scenes these port numbers are allocated to each particular out-of-process media player component as they are needed.

Each out-of-process media player component requires three network ports: one, for the media player process to receive remote method invocations; two, for the client application to receive media player event callbacks; and three, for the client application to receive and respond to heartbeat requests.

This all happens transparently, your application is not involved other than to allocate the range of available ports to use.

It is your responsibility as an application developer to make sure that the range of ports you have configured are available for vlcj-pro to use and are not in use by some other server process running in the same environment.

During the initial handshake when invoking `start()` or `mediaPlayer()` on the media player component, the component creates a new process for the remote media player and tells it which port number to use.

Ordinarily this is straightforward, however it is possible that some other application already owns that port. This can not be reliably known in advance, it is only known when the media player process tries to open a server socket on that port and it gets an exception. If this situation arises, the media player process will terminate itself and therefore this component start-up process will fail.

Meanwhile, the out of process media player component in the client application is waiting for notification that the remote process has started successfully. In the present scenario, this event will never arrive because the process terminated due to the port not being available. The client will timeout and report the error.

When the client application subsequently invokes `start()` or `mediaPlayer()` again, the whole process will be started from scratch, with a new port number which hopefully this time will not be in use.

This error detection and restarting of the component can be invoked any number of times until the component is successfully started. It is the responsibility of the client application to deal with the failures and to retry, the out of process media player component will not automatically perform the retries.

Application Termination

When an application shuts down, it is important to shut down the vlcj-pro components in an orderly manner. This clean, orderly, shut-down contributes to the robustness offered by using vlcj-pro, and should help primarily in making sure that there are no orphaned media player processes still running when the main application exits.

Shutting down the out-of-process media players is trivial, and is achieved by invoking a method on the factory that was created earlier:

```
factory.release();
```

By default, this method invocation will spawn multiple threads to shut down the media player processes concurrently. This is generally much faster than shutting down each process in turn.

Note that this method safely handles the case where an external process was never created in the first place – releasing these components is always done in a safe, orderly, way and will tolerate and swallow any exceptions that may be thrown during the shut-down routine.

Due to the multi-threaded nature of this shut-down, the entire shut-down routine should complete quite quickly.

Nevertheless, a client application might still like to invoke this shut-down routine on a background thread and perhaps show a busy dialog to the user.

It is possible to explicitly release a particular media player instance at any time but this is not recommended.

Shut-down Hook

When an application terminates, vlcj-pro does not itself instigate the shut-down of the managed out-of-process media player components, it relies on the application to do it.

The most obvious place to put this code is in a `windowClosing` method inside a `WindowListener` implementation attached to your application frame.

Instead of this it may be preferable for your application to install its own Java Virtual Machine shut-down hook – this hook is supposed to run even if your application is abruptly and unexpectedly terminated rather than, for example, having a user click on the window close button.

```
Runtime.getRuntime().addShutdownHook(new Runnable() {  
    @Override  
    public void run() {  
        factory.release();  
    }  
});
```

Configuration

There are two aspects to configuration in vlcj-pro. The first is the configuration of the media player 'client' component in the main application. The second is the configuration of the media player 'server' in the external process. The configuration files use YAML as the file format.

vlcj-pro does not itself use a configuration file, it has reasonable hard-coded default configuration that will be used in the absence of an application-specific configuration file.

The configuration loader for both client and server processes first attempts to load the configuration file from the class-path, if not found then an attempt is made to load the file from the file-system, and finally if that fails the hard-coded default configuration is used.

Client Configuration

The default client configuration file is `vlc-j-pro-client.config`. This can be overridden by setting a value for the `vlc-jProClientConfig` system property when launching your application.

The available client configuration properties are:

<i>Property</i>	<i>Type</i>	<i>Description</i>	<i>Default Value</i>
launcherProperties	List of String	List of values to specify as system properties when launching the external media player process Java Virtual Machine (this can be used e.g. to pass logging configuration)	
javaExecutable	String	Full path to the Java executable used to launch the external media player process	{java.home}/bin/java
serverClassPath	List of String	List of values to pass as classpath when launching the external media player process	inherit client classpath
fromPort	Integer	Minimum socket port number to use for remote communications	9500
toPort	Integer	Maximum socket port number to use for remote communications, or -1 for no maximum (up to 65535)	-1
clientConnectionRetries	Integer	Maximum number of times the client component will attempt to connect to the external media player process	8
clientConnectionDelay	Integer	Amount of time, in milliseconds, to wait when trying the initial connection to the external media player process	250
serverReadyWaitTimeout	Integer	Amount of time, in milliseconds, to wait for the handshake from the remote media player process	5000
serverReadyWaitPeriod	Integer	Amount of time, in milliseconds, to wait while polling the remote media player process during the handshake	50
maximumShutdownThreads	Integer	Maximum number of threads to spawn when shutting down the out-of-process media players	-1
maximumShutdownWaitTime	Integer	Maximum amount of time, in milliseconds, to wait when shutting down the out-of-process media players or -1 to wait indefinitely for each process to report its termination. If a value other than -1 is provided, then after that time period expires the application will terminate	-1

An example YAML client configuration file:

```
launcherProperties:
- logback.configurationFile=logback-server.xml

javaExecutable=/home/jvm/1.8/bin/java

serverClassPath:
- /home/install/my-app/vlcj-pro.jar
- /home/install/my-app/my-app.jar

fromPort: 9500
toPort: 9600

clientConnectRetries: 8
clientConnectDelay: 250

serverReadyWaitTimeout: 5000
serverReadyWaitPeriod: 50

maximumShutdownThreads: -1
maximumShutdownWaitTime: -1
```

Server Configuration

The default client configuration file is `vlc-j-pro-server.config`. This can be overridden by setting a value for the `vlcJProServerConfig` system property when launching your application.

The available server configuration properties are:

Property	Type	Description	Default Value
clientConnectionRetries	Integer	Maximum number of times the server components will attempt to connect to the main application (for event callback and heartbeat)	8
clientConnectionDelay	Integer	Amount of time, in milliseconds, to wait when trying the initial connection to the main application (for event callback and heartbeat)	250
heartbeat	Integer	Time period, in milliseconds, for the heartbeat. The heartbeat is sent repeatedly at this interval.	1000
libVlcArgs	List of String	List of VLC arguments (effectively command-line switches) used to initialise LibVLC	
libVlcAutoDiscovery	Boolean	Enable/disable attempted automatic discovery of LibVLC from well-known locations in the file-system	true
libVlcPath	String	Specific file-system path to load LibVLC from – <i>if this is provided the auto-discovery will be disabled</i>	
suppressNativeStreams	Boolean	Enable/disable the suppression of native log messages in the process output and error streams (Linux only)	false

An example YAML server configuration file:

```
libVlcPath: /home/linux/vlc/install/lib
libVlcAutoDiscovery: false

suppressNativeStreams: false

libVlcArgs:
- --no-osd
- --no-snapshot-preview

clientConnectRetries: 8
clientConnectDelay: 250

heartbeat: 1000
```

Additional Configuration Options

When the client application creates an out of process media player component there is a mechanism to programmatically tweak the configuration of that remote media player.

First, it is possible to specify additional options for the process command-line – this can be used to pass JVM arguments, system properties (using the normal `-Dproperty=value` syntax) and so on.

This mechanism can even be used to override the previously described `vlcJProServerConfig` system property to provide a different configuration file for a particular remote media player.

Second, if the `vlcJProLibVLC` system property is specified then the value of that property is used to specify additional options used to configure LibVLC for the remote media player.

For example:

```
-DvlcJProLibVLC=--video-filter=rotate --rotate-angle=235
```

Care may need to be taken when using these additional process options with regards to any options that might contain spaces and whether such strings need to be quoted or not.

A malformed option will likely result in a general failure of the remote media player to initialise and start up correctly.

Logging

vlcj-pro uses the SLF4J API to implement logging.

As is normally the case with SLF4J, vlcj-pro does not depend on any particular logging implementation so you are free to decide how to configure and use logging.

The recommended library for the logging implementation is Logback.

With a client application and multiple separate out-of-process media player processes the logging situation is not as straightforward as normal. The recommended approach is to configure separate logging for the main application and each of the media player processes – i.e. one isolated log configuration per process.

This enables you to have your client logging in one file, and each individual media player process logging each in their own files. There is rarely any benefit to be gained from trying to log all of the separate processes in the same log files, even if your logging implementation of choice supports all of those different processes writing to the same log files.

This raises a question of how to uniquely identify each log file name – for the client this is easy as there is only one log file, but for the media player processes what should the log file names be?

An obvious candidate to use in the log file name is the separating system media player process identifier. When a media player process starts up, it discovers its own process identifier and sets it as a Java system property. This can then be used by your logging configuration (e.g. by using token/variable substitution) to generate a unique file-name, perhaps also including a time-stamp.

The point is that vlcj-pro helps to some extent, but the logging configuration is exclusively the domain of your application.

If you do not provide a run-time dependency on a logger implementation library, vlcj-pro will run happily without it, and your application performance will be optimal (at least with regard to logging).

System Properties

For server logging configuration, a number of additional system properties are made available that you can use as you fit or not at all with your logging configuration:

Property	Description
VLCJ_PRO_PID	Operating system process identifier
VLCJ_PRO_SERVER	Media player server communications port number
VLCJ_PRO_CALLBACK	Media player event listener callback communications port number
VLCJ_PRO_HEARTBEAT	Media player process heartbeat communications port number

Logging Considerations

Client configuration is straightforward but the impact of server configuration requires more careful consideration.

This is because, as described repeatedly in this document, each media player instance is running in its own process in its own Java Virtual Machine. This means that each time a media player instance is started, a new Java Virtual Machine must be created and initialised, and the logging configuration must be applied each time.

If you need to pass any specific logging configuration to the server process you can do this as Java Virtual Machine arguments by setting the `launcherProperties` configuration in the client configuration file.

Depending on the logging configuration this can impact significantly the start-up time of the media player instance, and this added latency may be an issue for your application. It is true that this added latency is a one-time hit per new media player, but nevertheless it might be significant.

Logback

The demo project provided with the vlcj-pro distribution describes one way to configure vlcj-pro logging using Logback as the logging implementation.

Logback enables configuration via XML or via Groovy. The Groovy configuration is the preferred option for Logback, but for vlcj-pro you might like to consider the XML option instead.

If you decide to use the Groovy configuration mechanism, this can add the aforementioned start-up latency since the Groovy classes must be loaded and then the configuration file processed as a Groovy script.

Anecdotally using Groovy configuration over XML configuration might double the start-up time for the external media player process, but it is still the order of a couple of seconds so this might or might not be significant for your own application.

Application Design Recommendations

In general, all of the out-of-process media player component factory and all of the out-of-process media player components your application needs should be created immediately when your application starts up. Due to the lazy initialisation of the heavyweight part of the component this should not be a problem, all you are creating at this point is effectively a lightweight “stub”.

Generally an application should register life-cycle and media player event listeners at this point, but it is not absolutely necessary to do so as listeners can be added and removed at any time

A client application should keep the factory reference and all of the components for the entire lifetime of the application.

A client application should set the video surface as soon as the video surface is created, or at the very least it must be set before the media player instance is requested for the first time from the out-of-process media player component via the `mediaPlayer()` call. The video surface should be set once and never changed.

The out-of-process resources should only be released when the application terminates.

The resources should only be released by invoking the `release()` method on the factory.

Always prefer re-using components rather than destroying them and creating new ones.

Demo

A complete demo application is provided with the vlcj-pro distribution package.

The demo application shows how to create an application with multiple vlcj media players implemented via vlcj-pro out-of-process media player components.

The demo does not pretend to be the perfect implementation of a fully-featured Swing application – the purpose of the demo is to illustrate how vlcj-pro can be used, how to embed the media player video surface, how to invoke methods, how to register listeners and handle events, and how to perform orderly clean-up during application termination. To keep the demo as simple as possible and to focus on the vlcj-pro aspects, some compromises on application design may have been made.

The demo application includes the following features:

- Creating the out-of-process media player component factory;
- Creating multiple out-of-process media player components;
- Registering component life-cycle listeners and responding to events;
- Restarting a component if the remote media player process is killed, or fails to start (e.g. because of a port being already in use)
- Registering multiple media player event listeners and responding to events;
- Setting a logo and marquee;
- Capturing a video snapshot from the native media player as a buffered image;
- Playback controls, play, pause, skip, rewind;
- Independent volume and mute controls;
- Clean and orderly shut-down of the out-of-process media player components when the application terminates;
- How to configure logging generally and per-external process logging, in particular with Logback.

There are many more features of vlcj that could be incorporated into the demo application, things like the audio equalizer, video adjustment, track descriptions will all work if you choose to use them in your own application.

The demo application accepts on the command-line up to two parameters: if no arguments are provided then two media players will be used (1 row X 2 columns); if one argument is provided then this is used as the number of columns to use (1 row X n columns); if two arguments are provided then this is rows and columns (n rows X m columns).

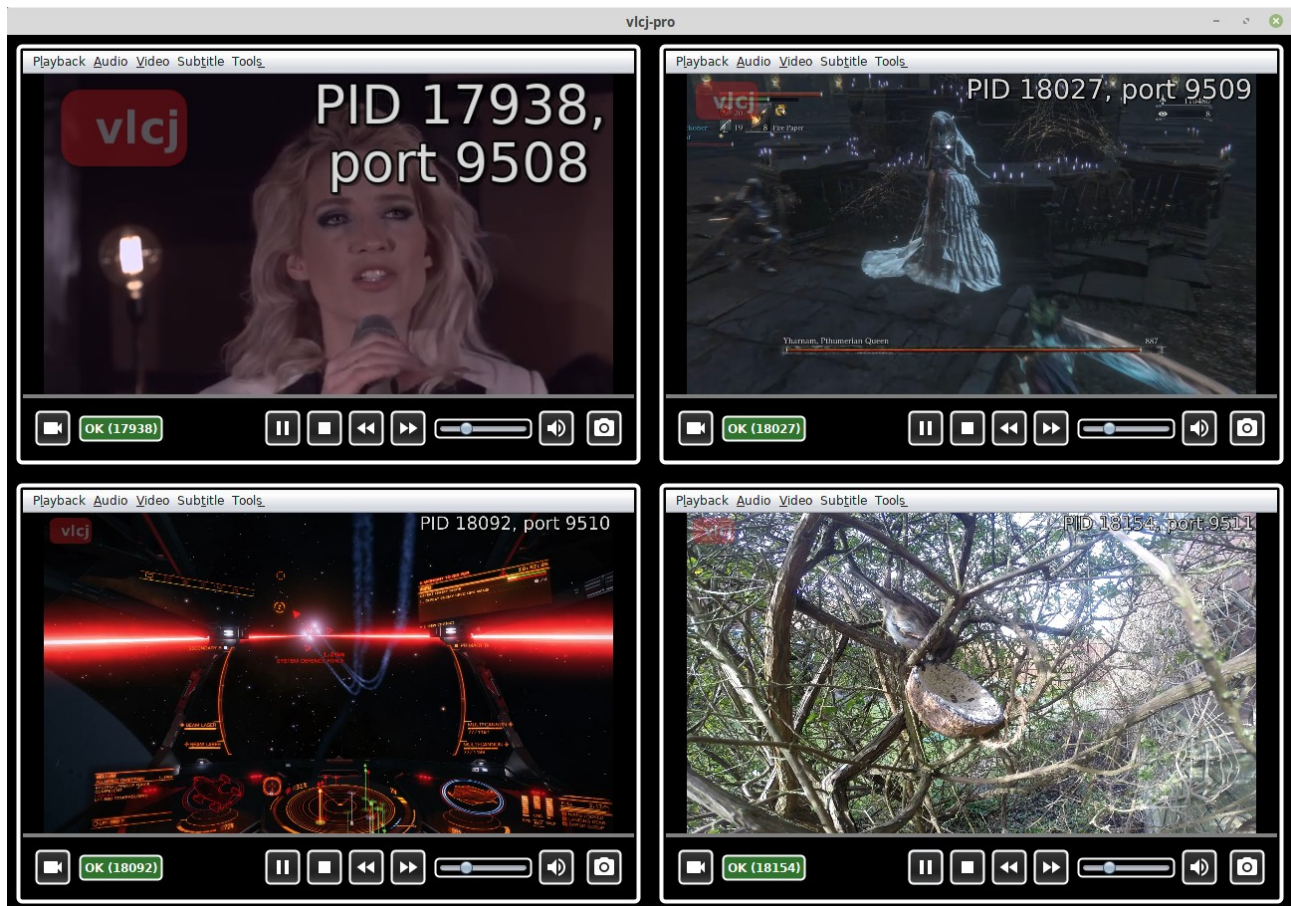
You can run the demo as a standalone application, but it is recommended that you import the demo project into your Integrated Development Environment and run it from there.

Screenshots

Demo Application with Snapshot Capture

This screenshot shows the demo application with four out-of-process media players.

The screenshot also shows use of the logo and marquee.



When the application starts up, the media players are lazily created – you will see a “Not Connected” label.

The out-of-process media player will be created the first time the media player instance is requested. You can trigger this in the demo application by playing media (drag and drop a media file onto the media player component you want to use), or by pressing the button with the video camera icon.

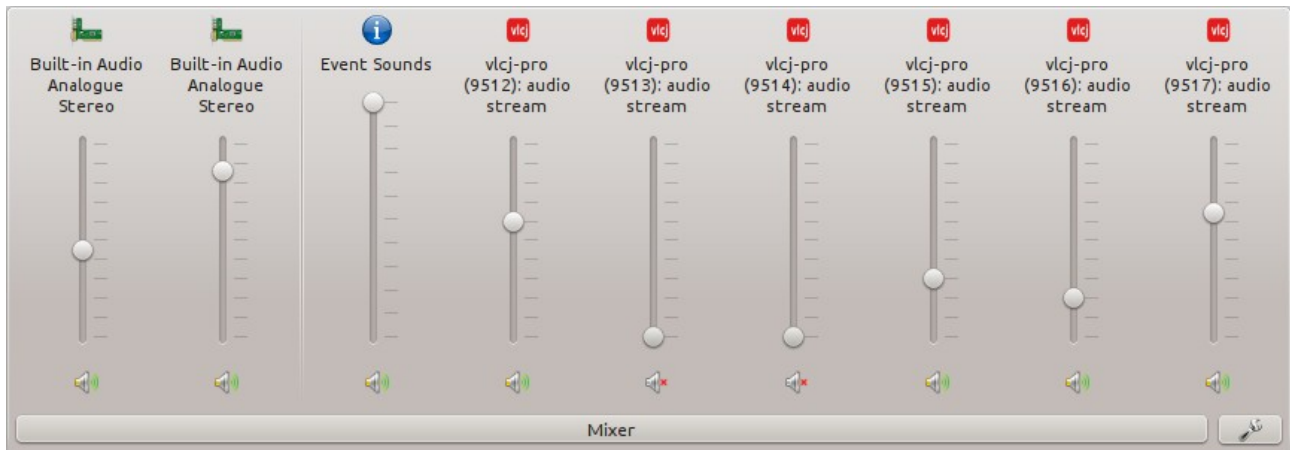
When video starts playing, if you press the photo camera button a snapshot will be generated and returned to the application as a `BufferedImage` and displayed in a separate frame.

Use operating system tools to kill the remote process (the PID is displayed in the video window) to test the component restart functionality.

Independent Volume Controls

This screenshot shows another configuration of the demo application, this time with six out-of-process media players.

The screenshot below shows the operating system volume mixer control panel applet and you can see in this case independent volume and mute controls for those six media players.



Demo Logging Configuration

The default logging configuration used by the demo will create log files in a “vlcj-pro-logs” directory in the current user's home directory.

Footprint of vlcj-pro Code in the Demo

If you study the demo code you will see that the vast majority of the code is standard Java Swing code.

In fact, the amount of code required to integrate vlcj-pro and out-of-process media player applications is *very small indeed*.

This was a key design goal of vlcj-pro.

Your own application deals with media players, event listeners and video surfaces as though they were all contained within your application – your application never really knows or has to deal with the external media player processes in any significant or onerous way.

For the most part, using a media player interface or media player event listeners is exactly the same as compared to using vlcj without vlcj-pro.

Despite the simplicity of the API, behind the scenes vlcj-pro hides a lot of complexity.

Limitations

It is possible using vlcj-pro to build robust multi-instanced out-of-process media player applications that can survive and recover from failures (crashes) in the native media player.

However, there are still circumstances where it may be impossible to recover from such a failure.

Native Media Player Resources

In particular, as you increase the number of native media player processes in your application you must be aware that you putting increased demands on your system, each native media player consumes CPU and GPU resources – vlcj-pro has no influence on this, so it is your responsibility as an application developer to ensure that you scale the number of media players in your application according to the available resources.

In short, vlcj-pro is not a magic bullet that enables you to use infinite numbers of native media players in your application. vlcj-pro can not give you hardware resources that you do not already have.

Native Resource Leaks

It is possible that the native code, i.e. VLC or any of its many plug-ins, causes memory or resource leaks. With a single media player instance such leaks may not be noticeable or significant, even when playing media for an extended period of time.

However, when scaling up the number of concurrent media players in your application you are effectively multiplying the number or rate of resource leaks. After an extended period of time this can become a significant factor and may eventually lead to irrecoverable system hangs.

There is nothing that vlcj-pro can do to recover any leaked native resources.

System Hangs

Following on from resource leaks there may be circumstances where eventually your entire system will hang. This may be caused by something like a buggy graphics driver.

Again, increasing the number of native media players will increase the rate or chances of a system hang occurring, especially after an extended period of time.

If your system hangs, vlcj-pro can not help you to recover.

Summary of Limitations

vlcj-pro will help you build robust out-of-process media player applications that deal with the use-case of multiple media players in the same application. You will get isolated native media players so there will be no concurrency or re-entrancy issues. You will get independent volume controls. For the external media player processes you will get crash detection and recovery.

The chances of an external process crash will be greatly reduced, but in any case any such crash will not take down your application – you will have a chance to recover.

However, vlcj-pro can not conjure up infinite CPU, GPU or other resources nor infinite native media players for your application to use. Nor can it help you reclaim resource leaks due to native code bugs. Nor can it help you recover from a hard system hang.

Do not be tempted to simply allow any number of media players to be created and used by your application. You need to understand the limits of your platform and the amount of system resources available to you, and you need to test appropriately before allowing the number of native media players to scale up.

Resource leaks by their very nature take time to accrue before failure, you need to soak test your applications, especially as the number of media players you use increases.

Despite the limitations, for the vast majority of use-cases and application scenarios vlcj-pro offers a high quality, robust, out-of-process media player solution for vlcj, Java and VLC.

Release Notes

Additional limitations pertaining to a particular release of vlcj-pro may be described in the release notes. Please review the release notes documentation provided in the vlcj-pro distribution bundle.

Licensing

vlcj-pro is a commercial product.

You can apply for a time-limited evaluation license to ensure vlcj-pro is fit for your purposes before making a purchasing decision. Please use the contact details below.

Use of vlcj-pro requires a commercial license.

vlcj-pro licensing is separate from and different to vlcj licensing.

If you never redistribute vlcj then you do not need a commercial license for vlcj – you can use it under the terms of GPL 3+.

If you use vlcj-pro, even if you only use it internally and even if you do not redistribute it, you need a commercial license for vlcj-pro.

A vlcj-pro license does not include a license to redistribute vlcj.

If you want to use vlcj-pro and redistribute vlcj then you will need a commercial license for vlcj-pro and a commercial license for vlcj.

A vlcj-pro license, and indeed a vlcj commercial license, is for a single named application or product that you produce.

If you want to use vlcj-pro, or vlcj, in multiple different applications then you will need multiple licenses. If you have a product suite with multiple different applications within it using vlcj-pro and/or vlcj, then for licensing purposes each such application is considered a separate application and multiple licenses would be needed.

In all cases, there is no per-user runtime royalty to pay, nor is there any per-developer seat cost to use vlcj-pro or vlcj. There are in fact no recurring costs at all.

Contact

vlcj-pro is developed by Caprica Software Limited:

www.capricasoftware.co.uk

For more information and support or to request a commercial license quotation or an evaluation license please contact:

mark.lee@capricasoftware.co.uk