
valkka*examplesDocumentation*

Sampsa Riikonen

Sep 08, 2023

Contents

1	About Valkka	3
1.1	Why this library?	3
1.2	Valkka API	3
1.3	The Project	4
2	Supported hardware	7
2.1	IP Cameras	7
2.2	USB Cameras	7
2.3	Codecs	7
2.4	Linux clients	8
2.5	Hardware Acceleration	8
3	Installing	11
3.1	A. Install using PPA	11
3.2	B. Install using releases	11
3.3	C. Compile yourself	12
3.4	Test your installation	12
3.5	Numpy	12
3.6	Install the testsuite	12
3.7	GTK	13
3.8	OpenCV	13
3.9	Development version	13
4	The PyQt testsuite	15
4.1	test_studio_1.py	17
4.2	test_studio_detector.py	19
5	Tutorial	21
5.1	Using the tutorial	21
5.2	Prerequisites	21
5.3	Lessons	22
6	Decoding	63
6.1	Single thread	63
6.2	Multithread	63
6.3	GPU Accelerated	64
6.4	Queueing frames	64

7	Integrating with Qt and multiprocessing	67
7.1	Qt integration	67
7.2	Drawing video into a widget	67
7.3	Python multiprocessing	68
7.4	C++ API	70
8	Multi-GPU systems	71
8.1	Introduction	71
8.2	Our approach	72
8.3	Configuration	72
8.4	Note on \$DISPLAY	73
9	ValkkaFS	75
9.1	VMS Architecture	75
9.2	Filesystem	76
9.3	Multiple Streams per File	77
9.4	Using an entire partition	77
10	Cloud Streaming	79
11	OnVif & Discovery	81
11.1	Intro	81
11.2	Python OnVif	81
11.3	OnVif with Zeep	82
11.4	Service classes	83
11.5	Example call	83
11.6	Notes	84
11.7	Discovery	84
12	Common problems	85
12.1	Pitfalls	85
12.2	Bottlenecks	86
12.3	System tuning	87
12.4	FAQ	87
13	Debugging	89
14	Repository Index	91
15	Licence & Copyright	95
16	Authors	97
17	Knowledge Base	99
17.1	General	99
17.2	OpenCV & OpenCV contrib	99
17.3	Jetson Nano	101

Valkka is a python media streaming framework

Create video streaming and surveillance solutions purely in Python. No need to go C++ ever again.

Some highlights of Valkka

- Python3 API, while streaming itself runs in the background at the cpp level. Threads, semaphores, frame queues etc. are hidden from the API user.
- Works with stock OnVif compliant IP cameras
- Create complex filtergraphs for your streams - send stream to screen, to disk or to your module of choice via shared memory
- Share decoded video with python processes across your Linux system
- Plug in your python-based machine vision modules : libValkka is framework agnostic, so everything goes - pytorch, tensorflow, you name it!
- Designed for massive video streaming : view and analyze simultaneously a large number of IP cameras
- Recast the IP camera video streams to either multicast or unicast
- Build graphical user interfaces with PyQt, interact machine vision with Qt's signal/slot system and build highly customized GUIs

If you are not a developer and wonder why all this is supposed to be cool and the economic benefits of it (an investor perhaps), please take a look at the Valkka VMS [whitepaper](#)

Devs already involved knee-deep in the video streaming / machine vision business, might want to take a look at [this presentation](#) to see some of the typical video streaming / machine vision problems libValkka can solve for you.

This documentation is a quickstart for [installing](#) and developing with Valkka using the Python3 API.

This is the recommended learning process:

- Start with the [tutorial](#)
- Valkka streams video and images between python multiprocesses, so reading [this article](#) is a must
- If you're into creating PyQt/PySide2 applications, take a look at [the PyQt testsuite](#)
- If you're more into cloud apps, check out [valkka-streamer](#)

For a desktop demo program using Valkka, check out [Valkka Live](#)

CHAPTER 1

About Valkka

1.1 Why this library?

So, yet another media player? I need to stream video from my IP camera into my python/Qt program and I want something that can be developed fast and is easy to integrate into my code. What's here for me?

If you just need to stream video from your IP cameras, decode it and show it on the screen, we recommend a standard media player, say, VLC and its python bindings.

However, if you need to stream video and *simultaneously* (1) present it on the screen, (2) analyze it with machine vision, (3) write it to disk, and even (4) recast it to other clients, stock media players won't do.

Such requirements are typical in large-scale video surveillance, management and analysis solutions. Demand for them is growing rapidly due to continuous decline in IP camera prices and growing computing power.

As a solution, you might try connect to the *same* camera 4 times and decode the stream 4 times - but then you'll burn all that CPU for nothing (you should decode only once). And try to scale that only to, say, 20+ cameras. In order avoid too many connections to your IP cameras (this is typically limited by the camera), you might desperately try your luck even with the multicast loopback. We've been there and it's not a good idea. And how about pluggin in your favorite machine vision/learning module, written with OpenCV or TensorFlow?

1.2 Valkka API

Valkka will solve the problem for you; It is a programming library and an API to do just that - large scale video surveillance, management and analysis programs, from the comfort of python3.

With Valkka, you can create complex pipings ("filtergraphs") of media streams from the camera, to screen, machine vision subroutines, to disk, to the net, etc. The code runs at the cpp level with threads, thread-safe queues, mutexes, semaphores, etc. All those gory details are hidden from the API user that programs filtergraphs at the python level only. Valkka can also share frames between python processes (and from there, with OpenCV, TensorFlow, etc.)

If you got interested, we recommend that you do the [tutorial](#), and use it together with the [PyQt testsuite](#), and the [example project](#) as starting points for your own project.

This manual has a special emphasis for Qt and OpenCV. You can create video streaming applications using PyQt: streaming video to widgets, and connect the signals from your machine vision subprograms to the Qt signal/slot system - and beyond.

For more technical information, check out the [library architecture page](#)

Finally, here is a small sample from the tutorial. You'll get the idea.

```

main branch, streaming
(LiveThread:livethread) --> -----+
                                   |
                                   |
{ForkFrameFilter: fork_filter} <---- (AVThread:avthread) << ---+  main branch, decoding
                                   |
                                   |
      branch 1 +--->> (OpenGLThread:glthread) --> To X-Window System
                                   |
      branch 2 +---> {IntervalFrameFilter: interval_filter} --> {SwScaleFrameFilter:
↪sws_filter} --> {RGBSharedMemFrameFilter: shmем_filter}
                                   |
↪
                                   |
↪      To OpenCV <-----+

```

1.3 The Project

In Valkka, the “streaming pipeline” from IP cameras to decoders and to the GPU has been completely re-thought and written from scratch:

- No dependencies on external libraries or x window extensions (we use only glx)
- Everything is pre-reserved in the system memory and in the GPU. During streaming, frames are pulled from pre-reserved stacks
- OpenGL pixel buffer objects are used for texture video streaming (in the future, we will implement fish-eye projections)
- Customized queueing and presentation algorithms
- etc., etc.

Valkka is in alpha stage. Even so, you can do lot of stuff with it - at least all the things we have promised here in the intro.

Repositories are organized as follows:

valkka-core : the cpp codebase and its python bindings are available at the [valkka-core github repository](#). The cpp core library is licensed under LGPL license see [here](#).

valkka-examples : the python tutorial and PyQt example/testsuite are available at the [valkka-examples github repository](#). MIT licensed.

For more, see [here](#).

All functional features are demonstrated in the [tutorial](#) which is updated as new features appear. Same goes for the [PyQt testsuite](#).

Near-term goals for new features are:

- Interserver communications between Valkka-based server and clients
- ValkkaFS filesystem, designed for recording large amounts of video (not yet fully operational / debugged)

- Synchronized recording of video
- Fisheye projections
- Support for sound

Valkka is based on the following opensource libraries and technologies:

Supported hardware

2.1 IP Cameras

OnVif compliant IP cameras (supporting the RTSP protocol)

Initial configuration of IP cameras can be a hurdle:

Many IP cameras typically require a half-broken Active-X (!) web-extension you have to download (to use with some outdated version of internet explorer).

Once you have sorted out those manufacturer-dependent issues you are good to go with OnVif and the RTSP protocol (as supported by libValkka).

Axis cameras, on the other hand, have a decent (standard javascript) web-interface for camera initial configuration.

Once you have done succesfull initial configuration, you can test the rtsp connection, with, say, using ffmpeg:

```
ffmpeg rtsp://user:password@ip-address
```

2.2 USB Cameras

USB Cameras capable of streaming H264

We have tested against *Logitech HD Pro Webcam C920*

2.3 Codecs

For the moment, the only supported codec is H264

2.4 Linux clients

libValkka uses OpenGL and OpenGL texture streaming, so it needs a robust OpenGL implementation. The current situation is:

- Intel: the stock **i915** driver is OK
- Nvidia: use **nvidia** proprietary driver
- ATI: **not tested**

OpenGL version 3 or greater is required. You can check your OpenGL version with the command `glxinfo`.

2.5 Hardware Acceleration

Please first read this *word of warning*.

VAAPI

Comes in the basic libValkka installation (and uses ffmpeg/libav infrastructure) - no additional packages needed.

First of all, the user using VAAPI, must belong to the “video” user group:

```
groups $USER
# run the following command if the user does not appear in the video group
sudo usermod -a -G video $USER
# after that you still need to logout & login
```

In order to use the VAAPI acceleration, just replace *AVThread* with *VAAPIThread*, i.e. instead of

```
avthread = AVThread("avthread", target_filter)
```

use this:

```
avthread = VAAPIThread("avthread", target_filter)
```

For more details about VAAPI, you can read [this](#), [this](#) and [this](#).

WARNING: VAAPI, especially the intel implementation, comes with a memory leak, which seems to be feature, not a bug - see discussions in [here](#) and [here](#). I have confirmed this memory leak myself with libva 2.6.0.

The opensource Mesa implementation (“i965”) is (surprise!) more stable and libValkka enforces i965 internally by setting the environment variable *LIBVA_DRIVER_NAME* to *i965* (this happens when you do *from valkka.core import **).

If you *really* want to use other *libva* implementation, you can set

```
export VALKKA_LIBVA_DRIVER_NAME=your-driver-name
```

If you wish to use VAAPI in a docker environment, you should start docker with

```
--device=/dev/dri:/dev/dri
```

And be sure that the host machine has all required vaapi-related libraries installed (the easiest way: install libValkka on the host as well).

Finally, you can follow the GPU usage in realtime with:

```
sudo intel_gpu_top
```

NVidia / CUDA

Provided as a separate package that installs into the *valkka.nv* namespace and is used like this:

```
from valkka.nv import NVThread  
avthread = NVThread("avthread", target_filter, gpu_index)
```

Available [here](#)

Huawei / CANN

Provided as a separate package. *Very* experimental and not guaranteed to work.

Available [here](#)

CHAPTER 3

Installing

The debian package includes the core library, its python bindings and some API level 2 python code. The python part is installed “globally” into `/usr/lib/python3/dist-packages/`

Note: LibValkka comes precompiled and packaged for a certain ubuntu distribution version. This means that the compilation and it’s dependencies assume the default python version of that distribution. Using custom-installed python versions, anacondas and whatnot might cause dependency problems.

3.1 A. Install using PPA

the preferred way

For recent ubuntu distributions, the core library binary packages and python bindings are provided by a PPA repository. Subscribe to the PPA repo (do this only once) with:

```
sudo apt-add-repository ppa:sampsa-riikonen/valkka
```

Install with:

```
sudo apt-get update
sudo apt-get install valkka
```

When you need to update valkka, do:

```
sudo apt-get update
sudo apt-get install --only-upgrade valkka
```

3.2 B. Install using releases

if you don’t like PPAs

You can download and install the required .deb packages “manually” from the [releases page](#)

```
sudo dpkg -i Valkka-*.deb
sudo apt-get install -fy
```

The last line pulls the dependencies.

Repeat the process when you need to update.

3.3 C. Compile yourself

the last resort

If you’re not using a recent Ubuntu distro and need to build libValkka and it’s python bindings yourself, please refer to the [valkka-core github page](#).

3.4 Test your installation

Test the installation with:

```
curl https://raw.githubusercontent.com/elsamposa/valkka-examples/master/quicktest.py -
  →o quicktest.py
python3 quicktest.py
```

3.5 Numpy

Valkka-core binaries has been compiled with the numpy version that comes with the corresponding Ubuntu distro, i.e. the numpy you would install with `sudo apt-get install python3-numpy`.

That version is automatically installed when you install valkka core with `sudo apt-get`, but it might be “shadowed” by your *locally* installed numpy.

If you get errors about numpy import, try removing your locally installed numpy (i.e. the version you installed with `pip install --user`).

3.6 Install the testsuite

First, install some debian packages:

```
sudo apt-get install python3-pip git mesa-utils ffmpeg vlc
```

some of these will be used for benchmarking Valkka against other programs.

The testsuite and tutorials use also imutils and PyQt5, so install a fresh version of them locally with pip:

```
pip3 install --user imutils PyQt5 PySide2 setproctitle
```

Here we have installed two flavors of the Qt python bindings, namely, [PyQt5](#) and [PySide2](#). They can be used in an identical manner. If you use PyQt5, be aware of its licensing terms.

Finally, for tutorial code and the PyQt test suite, download **valkka-examples** with:


```
git clone https://github.com/elsampsa/valkka-examples
```

Test the installation with:

```
cd valkka-examples
python3 quicktest.py
```

and you're all set.

When updating the python examples (do this always after updating *valkka-core*), do the following:

```
git pull
python3 quicktest.py
```

This checks that **valkka-core** and **valkka-examples** have consistent versions.

In the case of a numerical python version mismatch error, you are not using the default numpy provided by your Ubuntu distribution (from the debian package *python3-numpy*). Remove the conflicting numpy installation with *pip3 uninstall* or setting up a virtualenv.

Next, try out the *PyQt test/demo* suite or learn to program with the *tutorial*.

3.7 GTK

If you wan't to use **GTK** as your graphical user interface, you must install the PyGObject python bindings, as instructed [here](#), namely:

```
sudo apt-get install python-gi python-gi-cairo python3-gi python3-gi-cairo gir1.2-gtk-
↳ 3.0
```

3.8 OpenCV

Install with:

```
pip3 uninstall opencv-python
sudo pip3 uninstall opencv-python # just in case!
sudo apt-get install python3-opencv
```

The first one deinstall anything you may have installed with pip, while the second one installs the (good) opencv that comes with your linux distro's default python opencv installation.

3.9 Development version

As described above, for the current stable version of *valkka-core*, just use the repository.

For the development version (with experimental and unstable features) you have to compile from source. You might need to do this also for architectures other than *x86*.

Follow instructions in [here](#).

CHAPTER 4

The PyQt testsuite

So, you have installed `valkka-core` and `valkka-examples` as instructed [here](#). The same hardware requirements apply here as in the [tutorial](#).

The PyQt testsuite is available at

```
valkka_examples/api_level_2/qt/
```

The testsuite is intended for:

- Demonstration
- Benchmarking
- Ultimate debugging
- As *materia prima* for developers - take a copy of your own and use it as a basis for your own Valkka / Qt program

If you want a more serious demonstration, try out [Valkka Live](#) instead.

Currently the testsuite consists of the following programs:

File	Explanation
test_studio_1.py	<ul style="list-style-type: none"> - Stream from several rtsp cameras or sdp sources - Widgets are grouped together - This is just live streaming, so use: <pre>rtsp://username:password@your_ip</pre> - If you define a filename, it is interpreted as an sdp file
test_studio_2.py	<ul style="list-style-type: none"> - Like <i>test_studio_1.py</i> - Floating widgets
test_studio_3.py	<ul style="list-style-type: none"> - Like <i>test_studio_2.py</i> - On a multi-gpu system, video can be sent to another gpu/x-screen pair
test_studio_4.py	<ul style="list-style-type: none"> - Like <i>test_studio_3.py</i> - A simple user menu where video widgets can be opened - Open the <i>File</i> menu to add video on the screen
test_studio_detector.py	<ul style="list-style-type: none"> - Like <i>test_studio_1.py</i> - Shares video to OpenCV processes - OpenCV processes connect to Qt signal/slot system
test_studio_file.py	<ul style="list-style-type: none"> - Read and play stream from a matroska file - Only matroska-contained h264 is accepted. - Convert your video to “ip-camera-like” stream with: <pre>ffmpeg -i your_video_file -c:v h264 -r 10 -preset ultrafast -profile:v baseline -bsf h264_mp4toannexb -x264-params keyint=10:min-keyint=10 -an outfile.mkv</pre>
16	<ul style="list-style-type: none"> - If you’re recording directly from an IP camera, use: <pre>ffmpeg -i rtsp://username:password@your_ip -c:v copy -an outfile.mkv</pre>

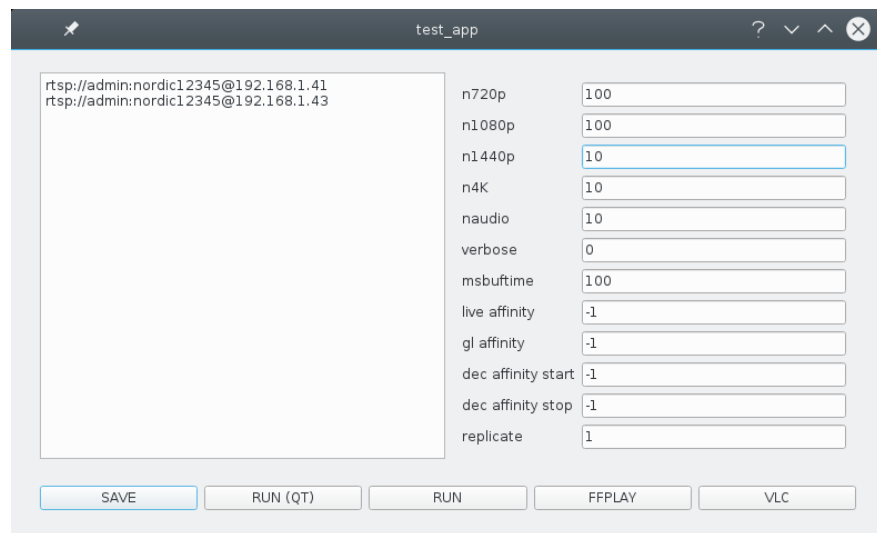
Before launching any of the testsuite programs you should be aware of the *common problems* of linux video streaming.

4.1 test_studio_1.py

Do this:

```
cd valkka_examples/api_level_2/qt/
python3 test_studio_1.py
```

The program launches with the following menu:



The screenshot shows a window titled "test_app" with a dark header bar containing a search icon, a question mark, and window control buttons. The main area is divided into two sections. On the left, there is a text area with two lines of RTSP stream sources: "rtsp://admin:nordic12345@192.168.1.41" and "rtsp://admin:nordic12345@192.168.1.43". On the right, there is a list of configuration options, each with a corresponding input field. The options and their values are: n720p (100), n1080p (100), n1440p (10), n4K (10), naudio (10), verbose (0), msbuftime (100), live affinity (-1), gl affinity (-1), dec affinity start (-1), dec affinity stop (-1), and replicate (1). At the bottom of the window, there are five buttons: "SAVE", "RUN (QT)", "RUN", "FFPLAY", and "VLC".

rtsp://admin:nordic12345@192.168.1.41	n720p	100
rtsp://admin:nordic12345@192.168.1.43	n1080p	100
	n1440p	10
	n4K	10
	naudio	10
	verbose	0
	msbuftime	100
	live affinity	-1
	gl affinity	-1
	dec affinity start	-1
	dec affinity stop	-1
	replicate	1

Buttons: SAVE, RUN (QT), RUN, FFPLAY, VLC

The field on the left is used to specify stream sources, one source per line. For IP cameras, use “rtsp://”, for sdp files, just give the filename. In the above example, we are connecting to two rtsp IP cams.

The fields on the right are:

Field name	What it does
n720p	Number of pre-reserved frames for 720p resolution
n1080p	Number of pre-reserved frames for 1080p resolution
n1440p	etc.
n4K	etc.
naudio	(not used)
verbose	(not used)
msbuftime	Frame buffering time in milliseconds
live affinity	Bind the streaming thread to a core. Default = -1 (no binding)
gl affinity	Bind frame presentation thread to a core. Default = -1
dec affinity start	Bind decoding threads to a core (first core). Default = -1
dec affinity stop	Bind decoding threads to cores (last core). Default = -1
replicate	Dump each stream to screen this many times
correct timestamp	1 = smart-correct timestamp (use this!) 0 = restamp upon arrival
socket size bytes	don't touch. Default value = 0.
ordering time millisecs	don't touch. Default value = 0.

As you learned from the [tutorial](#), in Valkka, frames are pre-reserved on the GPU. If you're planning to use 720p and 1080p cameras, reserve, say 200 frames for both.

Decoded frames are being queued for “msbuftime” milliseconds. This is necessary for de-jitter (among other things). The bigger the buffering time, the more pre-reserved frames you'll need and the more lag you get into your live streaming. A nice value is 300. For more on the subject, read [this](#).

Replicate demonstrates how Valkka can dump the stream (that's decoded only once) to multiple X windows. Try for example the value 24 - you get each stream on the screen 24 times, without any performance degradation or the need to decode streams more than once.

In Valkka, all threads can be bound to a certain processor core. Default value “-1” indicates that the thread is unbound and that the kernel can switch it from one core to another (normal behaviour).

Let's consider an example:

Field name	value
live affinity	0
gl affinity	1
dec affinity start	2
dec affinity stop	4

Now LiveThread (the thread that streams from cameras) stays at core index 0, all OpenGL operations and frame presenting at core index 1. Let's imagine you have ten decoders running, then they will be placed like this:

Core	Decoder thread
core 2	1, 4, 7, 10
core 3	2, 5, 8
core 4	3, 6, 9

Setting processor affinities might help, if you can afford the luxury of having one processor per decoder. Otherwise, it might mess up the load-balancing performed by the kernel.

By default, don't touch the affinities (simply use the default value -1).

Finally, the buttons that launch the test, do the following:

Button	What it does?
SAVE	Saves the test configuration (yes, save it)
RUN(QT)	Runs THE TEST (after saving, press this!)
RUN	Runs the test without Qt
FFPLAY	Runs the streams in ffplay instead (if installed)
VLC	Runs the streams in vlc instead (if installed)

RUN(QT) is the thing you want to do.

FFPLAY and *VLC* launch the same rtsp streams by using either ffplay or vlc. This is a nice test to see how Valkka performs against some popular video players.

4.2 test_studio_detector.py

The detector test program uses OpenCV, so you need to have it *installed*

Launch the program like this:

```
cd valkka_examples/api_level_2/qt/
python3 test_studio_detector.py
```

This is similar to *test_studio_1.py*. In addition to presenting the streams on-screen, the decoded frames are passed, once in a second, to OpenCV movement detectors. When movement is detected, a signal is sent with the Qt signal/slot system to the screen.

This test program is also used in the *gold standard test*. Everything is here: streaming, decoding, OpenGL streaming, interface to python and even the posix shared memory and semaphores. One should be able to run this test with a large number of cameras for a long period of time without excessive memory consumption or system instabilities.

5.1 Using the tutorial

Once you have installed **valkka-examples** with git as instructed in [here](#), the example codes of this tutorial can be run like this:

```
cd valkka_examples/api_level_1/tutorial
python3 lesson_1_a.py
```

5.2 Prerequisites

Before starting with the tutorial, you need at least:

- A decent desktop/laptop linux box with Ubuntu 16 installed
- At least 4 GB of total memory (as modern linux distros take around 2 GB out of that)
- Valkka and its python bindings installed (instructions [here](#))
- OpenCV installed (instructions [here](#))
- An RTSP camera connected to your router
 - Valkka uses standard protocols (RTSP, RTP, etc.), so it works with most of the cameras on the market
 - If your camera is “OnVif compliant”, then it support RTSP and RTP
- For the USB camera example, a H264 streaming USB camera is required
- Basic knowledge of media streaming in linux:
 - How to connect to an rtsp camera (e.g. “ffplay rtsp://username:passwd@ip_address”)
 - What are RTSP, SDP, etc., what is H264 and how video is streamed, etc.
- A media player installed, say, vlc and/or ffplay

5.3 Lessons

5.3.1 Lesson 1 : Receiving frames from an IP camera

A single FrameFilter

Download lesson [\[here\]](#)

Import the valkka level 1 API:

```
import time
from valkka.core import *
```

Create a starting point for a FrameFilter chain:

```
live_out_filter = InfoFrameFilter("live_out_filter")
```

This is the “entry point” where we receive all the frames.

InfoFrameFilter does nothing fancy - it just prints out the frames it receives.

However, as you will learn during this tutorial, FrameFilters can do a lot of stuff. You can chain them together. They can be used to fork and copy the stream into complex graphs, etc.

Next we need a thread that feeds the frames into our FrameFilter, so we instantiate a LiveThread:

```
livethread = LiveThread("livethread")
```

We also need a context describing the connection to an IP camera:

```
ctx = LiveConnectionContext(LiveConnectionType_rtsp, "rtsp://admin:nordic12345@192.168.
↪1.41", 1, live_out_filter)
```

The first parameter defines the device type, which in this case is an IP camera using the rtsp protocol. Note that we include the “entry point” live_out_filter. The integer parameter “1” is the slot number - it will be discussed in detail later on in this tutorial.

Finally, we can start streaming frames from the IP camera:

```
livethread.startCall()
livethread.registerStreamCall(ctx)
livethread.playStreamCall(ctx)
time.sleep(5)
livethread.stopCall()
print("bye")
```

The output looks like this:

```
InfoFrameFilter: live_out_filter start dump>>
InfoFrameFilter: FRAME    : <SetupFrame: timestamp=1525870891068 subsession_index=0_
↪slot=1 / media_type=0 codec_id=27>
InfoFrameFilter: PAYLOAD  : []
InfoFrameFilter: timediff: 0
InfoFrameFilter: live_out_filter <<end dump
InfoFrameFilter: live_out_filter start dump>>
InfoFrameFilter: FRAME    : <BasicFrame: timestamp=1525870891068 subsession_index=0_
↪slot=1 / payload size=45 / H264: slice_type=7>
InfoFrameFilter: PAYLOAD  : [0 0 0 1 103 100 0 42 173 132 1 12 32 8 97 0 67 8 2 24 ]
```

(continues on next page)

(continued from previous page)

```

InfoFrameFilter: timediff: 0
InfoFrameFilter: live_out_filter <<end dump
InfoFrameFilter: live_out_filter start dump>>
InfoFrameFilter: FRAME    : <BasicFrame: timestamp=1525870891068 subsession_index=0_
→slot=1 / payload size=9 / H264: slice_type=8>
InfoFrameFilter: PAYLOAD  : [0 0 0 1 104 238 49 178 27 ]
InfoFrameFilter: timediff: -1
InfoFrameFilter: live_out_filter <<end dump
...
...

```

InfoFrameFilter simply prints the frame type and first few bytes of it's payload (if there is any).

The first frame we get is a setup frame. This is a key feature of Valkka: the stream of frames that flows from source to the final sink, consists, not only of payload (say, H264 or PCMU), but of frames that are used to inform the system about the stream type, codec, etc.

Note: The code itself (LiveThread, InfoFrameFilter) runs in c++, while the connections are programmed here, at the python level

Chaining FrameFilters

Download lesson [[here](#)]

In the previous example, we had a thread (LiveThread), feeding a single FrameFilter (InfoFrameFilter). The “filter-graph” for this case looks like this:

```
(LiveThread:livethread) --> {InfoFrameFilter:live_out_filter}
```

In this notation, threads are marked with normal parenthesis (), and FrameFilters with curly brackets {}. Both class and instance names are included.

Next, let's chain some FrameFilters like this:

```
(LiveThread:livethread) --> {InfoFrameFilter:live_out_filter} ->
→{InfoFrameFilter:filter_2} -> {InfoFrameFilter:filter_3}
```

That chain can be created in python like this:

```

filter_3      =InfoFrameFilter("filter_3")
filter_2      =InfoFrameFilter("filter_2",filter_3)
live_out_filter =InfoFrameFilter("live_out_filter",filter_2)

```

Note that creating the filtergraph programmatically is started from the last framefilter (“filter_3”): we need to create “filter_3” first and pass it as a parameter (output target) to “filter_2”, etc. If you get confused with this, when dealing with more complex filtergraphs, just follow this rule of thumb: when instantiating framefilters, follow the filtergraph from end-to-beginning.

The output when running the python code looks like this:

```

InfoFrameFilter: live_out_filter start dump>>
...
InfoFrameFilter: live_out_filter <<end dump
InfoFrameFilter: filter_2 start dump>>

```

(continues on next page)

(continued from previous page)

```
...
InfoFrameFilter: filter_2 <<end dump
InfoFrameFilter: filter_3 start dump>>
...
InfoFrameFilter: filter_3 <<end dump
```

So, live_out_filter gets frame from livethread. It prints out info about the frame. Then it passes it to filter_2 that again prints information about the frame. filter_2 passes the frame onto filter_3, etc.

Note: LiveThread has an internal FrameFilter chain that is used to correct the timestamps given by your IP camera

Forking FrameFilters

Download lesson [\[here\]](#)

As a final trivial example for this lesson, we fork the FrameFilter chain into two:

```
filtergraph:
                                                    branch 1
                                                    +----->
↪{GateFrameFilter: gate_filter}
main branch
(LiveThread:livethread) --> {ForkFrameFilter:live_out_filter} --> |
↪+--- {InfoFrameFilter: info_filter}                               |
                                                                    |
                                                                    +----->
↪{FileFrameFilter: file_filter}
                                                    branch 2
```

Frames are fed to a ForkFrameFilter that copies the stream into two branches.

At branch 1, there is an on/off gate. When the gate is on, the frames are passed further on to the verbose InfoFrameFilter.

At branch 2, frames are written to a file

The filtergraph can be implemented in python like this:

```
# branch 1
info_filter      =InfoFrameFilter("info_filter")
gate_filter      =GateFrameFilter("gate_filter",info_filter)

# branch 2
file_filter      =FileFrameFilter("file_filter")

# main branch
live_out_filter  =ForkFrameFilter("live_out_filter",gate_filter,file_filter)
livethread       =LiveThread("livethread")
```

Like in the previous example, when constructing programmatically the framefilter chain, we start from the outer leafs of the tree (in this case, from “info_filter”, etc.) and move from the outer edge towards the main branch.

Let’s run it like this:

```
# close the gate before streaming
gate_filter.unSet()

livethread.startCall()
livethread.registerStreamCall(ctx)
livethread.playStreamCall(ctx)
time.sleep(5)

print("start writing to disk")
file_filter.activate("stream.mkv")
time.sleep(5)

print("let's get verbose")
gate_filter.set()
time.sleep(2)

print("close file and exit")
file_filter.deActivate()

livethread.stopCall()

print("bye")
```

Here we first close the gate, so no information about the frames is printed to the terminal. We write the stream to the disk by calling “activate” method of the FileFrameFilter. After 5 secs. we turn on the gate and start getting information about the frames into the screen. Finally we close the file by calling “deActivate”.

You can play the resulting “stream.mkv” with your favorite media player.

Note: Valkka is *not* a mediaplayer that understands thousands of codecs and container formats. Emphasis is on an internally consistent (for that a single or a few codec/container formats are enough, i.e. what we write we can also read) video management library. At the moment only H264 video is accepted. Container format is matroska (mkv).

FrameFilter reference

API level 1 considered in this lesson, is nothing but cpp code wrapped to python.

To see all available FrameFilters, refer to the [cpp documentation](#).

In the cpp docs, only a small part of the member methods are wrapped and visible from python (these are marked with the “pyapi” tag)

Note: FrameFilter chains are nothing but callback cascades - they will block the execution of LiveThread when executing code. At some moment, the callback chain should terminate and continue in another, independent thread. This will be discussed in the next lesson.

Controlling verbosity

If libValkka dumps too much to your terminal, you can shut it off by using `loglevel_silent`.

Verbosity can be controlled like this:

```
from valkka.api2.logging import setValkkaLogLevel, setLogLevel_livelogger, loglevel_
↳silent, loglevel_normal

setValkkaLogLevel(loglevel_silent) # set all loggers to silent
setLogLevel_livelogger(loglevel_normal) # set an individual loggers
```

5.3.2 Lesson 2 : Decoding

Download lesson [here]

Let's consider the following filtergraph:

```
Streaming part          | Decoding part
                        |
(LiveThread:livethread) -->> (AVThread:avthread) --> {InfoFrameFilter:info_filter}
```

Like in the previous lessons, we are reading frames from an IP camera. Instead of churning them through a series of filters, we pass them to another, independently running thread that performs decoding (AVThread).

Let's list all the symbols used until now and the corresponding objects:

Symbol	Base class	Explanation
()	Thread	An independently running thread
>>		Crossover between two threads
{}	FrameFilter	A framefilter

That's all you need to create complex filtergraphs with Valkka.

We start as usual, by constructing the filterchain from end-to-beginning:

```
# decoding part
info_filter      =InfoFrameFilter("info_filter")
avthread         =AVThread("avthread",info_filter)
```

AVThread is a thread that does all the heavy-lifting of decoding from H264 to bitmap images. It decodes on the CPU. If you fancy to use a *hardware accelerator*, then you could substitute *AVThread* with *VAAPIThread* instead.

Next, we need a framefilter to feed the frames into AVThread. This framefilter is requested from the AVThread itself:

```
# streaming part
av_in_filter     =avthread.getFrameFilter()
livethread       =LiveThread("livethread")
```

Finally, proceed as before: pass *av_in_filter* as a parameter to the connection context, start threads, etc.

```
ctx =LiveConnectionContext(LiveConnectionType_rtsp, "rtsp://admin:nordic12345@192.168.
↳1.41", 1, av_in_filter)
```

Start threads. Starting the threads should be done in end-to-beginning order (in the same order we constructed the filterchain).

```
avthread.startCall()
livethread.startCall()

# start decoding
```

(continues on next page)

(continued from previous page)

```
avthread.decodingOnCall()

livethread.registerStreamCall(ctx)
livethread.playStreamCall(ctx)
time.sleep(5)

# stop decoding
# avthread.decodingOffCall()
```

Stop threads. Stop threads in beginning-to-end order (i.e., following the filtergraph from left to right).

```
livethread.stopCall()
avthread.stopCall()

print("bye")
```

You will see output like this:

```
InfoFrameFilter: info_filter start dump>>
InfoFrameFilter: FRAME    : <AVBitmapFrame: timestamp=1525870759898 subsession_index=0_
↳slot=1 / h=1080; w=1920; l=(1920,960,960); f=12>
InfoFrameFilter: PAYLOAD  : [47 47 47 47 47 47 47 47 47 47 ]
InfoFrameFilter: timediff: -22
InfoFrameFilter: info_filter <<end dump
InfoFrameFilter: info_filter start dump>>
InfoFrameFilter: FRAME    : <AVBitmapFrame: timestamp=1525870759938 subsession_index=0_
↳slot=1 / h=1080; w=1920; l=(1920,960,960); f=12>
InfoFrameFilter: PAYLOAD  : [47 47 47 47 47 47 47 47 47 47 ]
InfoFrameFilter: timediff: -11
InfoFrameFilter: info_filter <<end dump
...
...
```

So, instead of H264 packets, we have decoded bitmap frames here.

In the next lesson, we'll dump them on the screen.

When using the API to pass frames between threads, that's all you need to know for now.

“Under the hood”, however, things are a bit more complex.

The framefilter requested from AVThread writes into AVThread's internal *FrameFifo*. This is a first-in-first-out queue where a copy of the incoming frame is placed. Frames are copied into pre-reserved frames, taken from a pre-reserved stack. Both the fifo and the stack are thread-safe and mutex-protected. The user has the possibility to define the size of the stack when instantiating AVThread.

For more details, see the [cpp documentation](#) and especially the [FrameFifo](#) class.

However, all these gory details are not a concern for the API user at this stage. :)

Note: There are several *FrameFifo* and *Thread* classes in Valkka. See the [inheritance diagram](#). Only a small subset of the methods should be called by the API user. These typically end with the word “Call” (and are marked with the “<pyapi>” tag).

5.3.3 Lesson 3 : Streaming to the X-window system

One camera to one window

Download lesson [\[here\]](#)

Let's consider the following filtergraph with streaming, decoding and presentation:

```
Streaming part
(LiveThread:livethread)---+
                        |
Decoding part           |
(AVThread:avthread) <-----+
|
|      Presentation part
+--->> (OpenGLThread:glthread)
```

Compared to the previous lesson, we're continuing the filterchain from AVThread to OpenGLThread. OpenGLThread is responsible for sending the frames to designated x windows.

Note: OpenGLThread uses OpenGL texture streaming. YUV interpolation to RGB is done on the GPU, using the shader language.

Start constructing the filterchain from end-to-beginning:

```
# presentation part
glthread      =OpenGLThread ("glthread")
gl_in_filter  =glthread.getFrameFilter()
```

We requested a framefilter from the OpenGLThread. It is passed to the AVThread:

```
# decoding part
avthread      =AVThread("avthread",gl_in_filter)
av_in_filter  =avthread.getFrameFilter()

# streaming part
livethread    =LiveThread("livethread")
```

Define the connection to the IP camera as usual, with **slot number "1"**:

```
# ctx =LiveConnectionContext (LiveConnectionType_rtsp, "rtsp://admin:nordic12345@192.
↪168.1.41", 1, av_in_filter)
ctx =LiveConnectionContext (LiveConnectionType_rtsp, "rtsp://admin:12345@192.168.0.157
↪", 1, av_in_filter)
```

Start all threads, start decoding, and register the live stream. Starting the threads should be done in end-to-beginning order (in the same order we constructed the filterchain).

```
glthread.startCall()
avthread.startCall()
livethread.startCall()

# start decoding
avthread.decodingOnCall()

livethread.registerStreamCall(ctx)
livethread.playStreamCall(ctx)
```

Now comes the new bit. First, we create a new X window on the screen:


```
window_id =glthread.createWindow()
```

We could also use the window id of an existing X window.

Next, we create a new “render group” to the OpenGLThread. Render group is a place where we can render bitmaps - in this case it’s just the X window.

```
glthread.newRenderGroupCall(window_id)
```

We still need a “render context”. Render context is a mapping from a frame source (in this case, the IP camera) to a certain render group (X window) on the screen:

```
context_id=glthread.newRenderContextCall(1,window_id,0) # slot, render group, z
```

The first argument to newRenderContextCall is the **slot number**. We defined the slot number for the IP camera when we used the [LiveConnectionContext](#).

Now, each time a frame with slot number “1” arrives to OpenGLThread it will be rendered to render group “window_id”.

Stream for a while, and finally, close all threads:

```
time.sleep(10)

glthread.delRenderContextCall(context_id)
glthread.delRenderGroupCall(window_id)

# stop decoding
avthread.decodingOffCall()
```

Close threads. Stop threads in beginning-to-end order (i.e., following the filtergraph from left to right).

```
livethread.stopCall()
avthread.stopCall()
glthread.stopCall()

print("bye")
```

So, all nice and simple with the API.

However, here it is important to understand what’s going on “under-the-hood”. Similar to AVThread, OpenGLThread manages a stack of YUV bitmap frames. These are pre-reserved on the GPU (for details, see the [OpenGLFrameFifo](#) class in the cpp documentation).

The number of pre-reserved frames you need, depends on the buffering time used to queue the frames.

You can adjust the number of pre-reserved frames for different resolutions and the buffering time like this:

```
gl_ctx =OpenGLFrameFifoContext()
gl_ctx.n_720p    =20
gl_ctx.n_1080p   =20
gl_ctx.n_1440p   =20
gl_ctx.n_4K      =20

glthread =OpenGLThread("glthread", gl_ctx, 300)
```

Here we have reserved 20 frames for each available resolution. A buffering time of 300 milliseconds is used.

For example, if you are going to use two 720p cameras, each at 20 fps, with 300 millisecond buffering time, then you should reserve

```
2 * 20 fps * 0.3 sec = 12 frames
```

for 720p. If this math is too hard for you, just reserve several hundred frames for each frame resolution (or until you run out of GPU memory). :)

If you're extremely ambitious libValkka user who wants to use that brand-new 8K running at 80 frames per second, then read [this](#) first.

One camera to several windows

Download lesson [[here](#)]

Streaming the same camera to several X windows is trivial; we just need to add more render groups (aka x windows) and render contexes (mappings):

```
id_list=[]

for i in range(10):
    window_id =glthread.createWindow()
    glthread.newRenderGroupCall(window_id)
    context_id=glthread.newRenderContextCall(1,window_id,0)
    id_list.append((context_id,window_id)) # save context and window ids

time.sleep(10)

for ids in id_list:
    glthread.delRenderContextCall(ids[0])
    glthread.delRenderGroupCall(ids[1])
```

Presenting the same stream in several windows is a typical situation in video surveillance applications, where one would like to have the same stream be shown simultaneously in various “views”

Keep in mind that here we have connected to the IP camera only once - and that the H264 stream has been decoded only once.

Note: When streaming video (from multiple sources) to multiple windows, OpenGL rendering synchronization to vertical refresh (“vsync”) should be disabled, as it will limit your total framerate to the refresh rate of your monitor (i.e. to around 50 frames per second). On MESA based X.org drivers (intel, nouveau, etc.), this can be achieved from command line with “export vblank_mode=0”. With nvidia proprietary drivers, use the nvidia-settings program. You can test if vsync is disabled with the “glxgears” command (in package “mesa-utils”). Glxgears should report 1000+ frames per second with vsync disabled.

Decoding multiple streams

Download lesson [[here](#)]

Let's consider decoding the H264 streams from multiple RTSP cameras. For that, we'll be needing several decoding AVThreads. Let's take another look at the filtergraph:

```
Streaming part
(LiveThread:livethread) ---+
                             |
Decoding part                |   [This part of the filtergraph should be replicated]
(AVThread:avthread) <-----+
```

(continues on next page)

(continued from previous page)

```
|
|      Presentation part
+--->> (OpenGLThread:glthread)
```

LiveThread and OpenGLThread can deal with several simultaneous media streams, while for decoding, we need one thread per decoder. Take a look at the [library architecture page](#)

It's a good idea to encapsulate the decoding part into its own class. This class takes as an input, the framefilter where it writes the decoded frames and as an input, the stream rtsp address:

```
class LiveStream:

    def __init__(self, gl_in_filter, address, slot):
        self.gl_in_filter =gl_in_filter

        self.address      =address
        self.slot          =slot

        # decoding part
        self.avthread      =AVThread("avthread", self.gl_in_filter)
        self.av_in_filter   =self.avthread.getFrameFilter()

        # define connection to camera
        self.ctx =LiveConnectionContext(LiveConnectionType_rtsp, self.address, self.slot,
        ↪self.av_in_filter)

        self.avthread.startCall()
        self.avthread.decodingOnCall()

    def close(self):
        self.avthread.decodingOffCall()
        self.avthread.stopCall()
```

Construct the filtergraph from end-to-beginning:

```
# presentation part
glthread      =OpenGLThread ("glthread")
gl_in_filter   =glthread.getFrameFilter()

# streaming part
livethread     =LiveThread("livethread")

# start threads
glthread.startCall()
livethread.startCall()
```

Instantiate LiveStreams. This will also start the AVThreads. Frames from the first camera are tagged with slot number 1, while frames from the second camera are tagged with slot number 2:

```
stream1 = LiveStream(gl_in_filter, "rtsp://admin:nordic12345@192.168.1.41", 1) # slot_
↪1
stream2 = LiveStream(gl_in_filter, "rtsp://admin:nordic12345@192.168.1.42", 2) # slot_
↪2
```

Register streams to LiveThread and start playing them:

```

livethread.registerStreamCall(stream1.ctx)
livethread.playStreamCall(stream1.ctx)

livethread.registerStreamCall(stream2.ctx)
livethread.playStreamCall(stream2.ctx)

```

Create x windows, and map slot numbers to certain x windows:

```

# stream1 uses slot 1
window_id1 =glthread.createWindow()
glthread.newRenderGroupCall(window_id1)
context_id1 =glthread.newRenderContextCall(1, window_id1, 0)

# stream2 uses slot 2
window_id2 =glthread.createWindow()
glthread.newRenderGroupCall(window_id2)
context_id2 =glthread.newRenderContextCall(2, window_id2, 0)

```

Render video for a while, stop threads and exit:

```

time.sleep(10)

glthread.delRenderContextCall(context_id1)
glthread.delRenderGroupCall(window_id1)

glthread.delRenderContextCall(context_id2)
glthread.delRenderGroupCall(window_id2)

# Stop threads in beginning-to-end order
livethread.stopCall()
stream1.close()
stream2.close()
glthread.stopCall()

print("bye")

```

There are many ways to organize threads, render contexes (slot to x window mappings) and complex filtergraphs into classes. It's all quite flexible and left for the API user.

One could even opt for an architecture, where there is a LiveThread and OpenGLThread for each individual stream (however, this is not recommended).

The level 2 API provides ready-made filtergraph classes for different purposes (similar to class LiveStream constructed here).

5.3.4 Lesson 4 : Receiving Frames at Python

Here we start with two separate python programs: (1) a server that reads RTSP cameras and writes RGB frames into shared memory and (2) a client that reads those RGB frames from memory. For the client program, two versions are provided, the API level 2 being the most compact one.

Such scheme is only for demo/tutorial purposes. Normally you would start both the server and client from within the same python program. We give an example of that as well.

Server side

Download server side [\[here\]](#)

By now, we have learned how to receive, decode and send streams to the x window system. In this chapter, we do all that, but at the same time, also send copies of the decoded frames to another python process.

The filtergraph looks like this:

```
(LiveThread:livethread) -----+ main branch, streaming
                                |
{ForkFrameFilter: fork_filter} <---- (AVThread:avthread) << --+ main branch, decoding
                        |
                    branch 1 +--> (OpenGLThread:glthread)
                        |
                    branch 2 +--> {IntervalFrameFilter: interval_filter} --> {SwScaleFrameFilter:
↪sws_filter} --> {RGBSharedMemFrameFilter: shmem_filter}
```

We are using the ForkFrameFilter to branch the decoded stream into two branches. Branch 1 goes to screen, while branch 2 does a lot of new stuff.

In branch 2, IntervalFrameFilter passes a frame through on regular intervals. In our case we are going to use an interval of 1 second, i.e. even if your camera is sending 25 fps, at the other side of IntervalFrameFilter we'll be observing only 1 fps.

SwScaleFrameFilter does YUV => RGB interpolation on the CPU. The final, interpolated RGB frame is passed to the posix shared memory with the RGBSharedMemFrameFilter. From there it can be read by another python process.

(Remember that branch 1 does YUV => RGB interpolation as well, but on the GPU (and at 25 fps rate))

To summarize, branch 1 interpolates once a second a frame to RGB and passes it to shared memory. The size of the frame can be adjusted.

Let's start the construction of the filtergraph by defining some parameters. Frames are passed to SwScaleFrameFilter at 1000 millisecond intervals. The image dimensions of the frame passed into shared memory, will be one quarter of a full-hd frame:

```
# define yuv=>rgb interpolation interval
image_interval=1000 # YUV => RGB interpolation to the small size is done each 1000
↪milliseconds and passed on to the shmem ringbuffer

# define rgb image dimensions
width  =1920//4
height =1080//4
```

RGBSharedMemFrameFilter needs also a unique name and the size of the shared memory ring-buffer:

```
# posix shared memory
shmem_name  ="lesson_4" # This identifies posix shared memory - must be unique
shmem_buffers =10 # Size of the shmem ringbuffer
```

Next, we construct the filterchain as usual, from end-to-beginning:

```
# branch 1
glthread      =OpenGLThread("glthread")
gl_in_filter  =glthread.getFrameFilter()

# branch 2
shmem_filter  =RGBShmemFrameFilter(shmem_name, shmem_buffers, width, height)
```

(continues on next page)

(continued from previous page)

```
# shmem_filter    =BriefInfoFrameFilter("shmem") # a nice way for debugging to see of_
↳you are actually getting any frames here ..
sws_filter        =SwScaleFrameFilter("sws_filter", width, height, shmem_filter)
interval_filter   =TimeIntervalFrameFilter("interval_filter", image_interval, sws_
↳filter)

# fork
fork_filter        =ForkFrameFilter("fork_filter", gl_in_filter, interval_filter)

# main branch
avthread          =AVThread("avthread",fork_filter)
av_in_filter       =avthread.getFrameFilter()
livethread        =LiveThread("livethread")
```

Define connection to camera: frames from 192.168.1.41 are written to live_out_filter and tagged with slot number 1:

```
# ctx =LiveConnectionContext(LiveConnectionType_rtsp, "rtsp://admin:nordic12345@192.
↳168.1.41", 1, av_in_filter)
ctx =LiveConnectionContext(LiveConnectionType_rtsp, "rtsp://admin:123456@192.168.0.134
↳", 1, av_in_filter)
```

Start processes, stream for 60 seconds and exit:

```
glthread.startCall()
avthread.startCall()
livethread.startCall()

# start decoding
avthread.decodingOnCall()

livethread.registerStreamCall(ctx)
livethread.playStreamCall(ctx)

# create an X-window
window_id =glthread.createWindow()
glthread.newRenderGroupCall(window_id)

# maps stream with slot 1 to window "window_id"
context_id=glthread.newRenderContextCall(1,window_id,0)

time.sleep(60)

glthread.delRenderContextCall(context_id)
glthread.delRenderGroupCall(window_id)

# stop decoding
avthread.decodingOffCall()

# stop threads
livethread.stopCall()
avthread.stopCall()
glthread.stopCall()

print("bye")
```

Note: In the previous lessons, all streaming has taken place at the cpp level. Here we are starting to use posix shared

memory and semaphores in order to share frames between python processes, with the ultimate goal to share them with machine vision processes. However, if you need very high-resolution and high fps solutions, you might want to implement the sharing of frames and your machine vision routines directly at the cpp level.

Client side: API level 2

Download client side API level 2 [\[here\]](#)

This is a *separate* python program for reading the frames that are written by Valkka to the shared memory.

The parameters used both in the server side (above) and on the client side (below) **must be exactly the same** and the client program should be started *after* the server program (and while the server is running). Otherwise undefined behaviour will occur.

The used shmem_name(s) should be same in both server and client, but different for another server/client pair.

```
from valkka.api2 import ShmemRGBClient

width = 1920//4
height = 1080//4

# This identifies posix shared memory - must be same as in the server side
shmem_name = "lesson_4"
shmem_buffers = 10                # Size of the shmem ringbuffer

client = ShmemRGBClient(
    name=shmem_name,
    n_ringbuffer=shmem_buffers,
    width=width,
    height=height,
    mstimeout=1000,                # client timeouts if nothing has been received in 1000_
    ↪milliseconds
    verbose=False
)
```

The *mstimeout* defines the semaphore timeout in milliseconds, i.e. the time when the client returns even if no frame was received:

```
while True:
    index, meta = client.pullFrame()
    if (index == None):
        print("timeout")
    else:
        data = client.shmem_list[index][0:meta.size]
        data = data.reshape((meta.height, meta.width, 3))
        print("got data: ", data.shape)
```

The *client.shmem_list* is a list of numpy arrays, while *isize* defines the extent of data in the array. This example simply prints out the first ten bytes of the RGB image.

Client side: openCV

Download client side openCV example [\[here\]](#)

OpenCV is a popular machine vision library. We modify the previous example to make it work with openCV like this:

```
import cv2
from valkka.api2 import ShmemRGBClient

width = 1920//4
height = 1080//4

# This identifies posix shared memory - must be same as in the server side
shmem_name = "lesson_4"
shmem_buffers = 10          # Size of the shmem ringbuffer

client = ShmemRGBClient(
    name=shmem_name,
    n_ringbuffer=shmem_buffers,
    width=width,
    height=height,
    mtimeout=1000,          # client timeouts if nothing has been received in 1000_
    ↪milliseconds
    verbose=False
)

while True:
    index, meta = client.pullFrame()
    if (index == None):
        print("timeout")
        continue
    data = client.shmem_list[index][0:meta.size]
    print("data   : ", data[0:min(10, meta.size)])
    print("width  : ", meta.width)
    print("height : ", meta.height)
    print("slot   : ", meta.slot)
    print("time   : ", meta.mtimestamp)
    img = data.reshape((meta.height, meta.width, 3))
    # img2 = imutils.resize(img, width=500)
    img = cv2.GaussianBlur(img, (21, 21), 0)
    print("got frame", img.shape)
    ## depending on how you have installed your openCV, this might not work:
    ## (use at your own risk)
    #cv2.imshow("valkka_opencv_demo", img)
    #cv2.waitKey(1)
```

After receiving the RGB frame, some gaussian blur is applied to the image. Then it is visualized using openCV's own "high-gui" infrastructure. If everything went ok, you should see a blurred image of your video once in a second.

Start this script *after* starting the server side script (server side must also be running).

Client side: API level 1

Download client side example [\[here\]](#)

API level 2 provides extra wrapping. Let's see what goes on at the lowest level (plain, cpp wrapped python code).

```
from valkka.core import *

width = 1920//4
height = 1080//4
```

(continues on next page)

(continued from previous page)

```
shmem_name = "lesson_4" # This identifies posix shared memory - must be unique
shmem_buffers = 10      # Size of the shmem ringbuffer
```

The wrapped cpp class is *SharedMemRingBufferRGB* (at the server side, RGBShmemFrameFilter is using SharedMemRingBufferRGB):

```
shmem = SharedMemRingBufferRGB(shmem_name, shmem_buffers, width, height,
                               1000, False) # shmem id, buffers, w, h, timeout,
↪False=this is a client
```

Next, get handles to the shared memory as numpy arrays:

```
shmem_list = []
for i in range(shmem_buffers):
    # getNumpyShmem defined in the swig interface file
    shmem_list.append(getNumpyShmem(shmem, i))
print("got element i=", i)
```

Finally, start reading frames.

shmem.clientPullPy() returns a tuple with the shared memory ringbuffer index and metadata.

```
while True:
    tup = shmem.clientPullPy()
    index = tup[0]
    if index < 0:
        print("timeout")
        continue
    isize      = tup[1]
    width      = tup[2]
    height     = tup[3]
    slot       = tup[4]
    mtimestamp = tup[5]
    print("got index, size =", index, isize)
    ar = shmem_list[index][0:isize] # this is just a numpy array
    ar = ar.reshape((height, width, 3)) # this is your rgb image
    print("payload      =", ar.shape)
```

C++ documentation for Valkka shared memory classes be found [here](#).

Server + Client

Download server + client example [[here](#)]

Here we have a complete example running both server & client within the same python file.

You could wrap the client part further into a python thread, releasing your main python process to, say, run a GUI.

Yet another possibility is to run the server and client in separate multiprocesses. In this case one must be extra careful to spawn the multiprocesses *before* instantiating any libValkka objects, since libValkka relies heavily on multithreading (this is the well-known “fork must go before threading” problem).

These problems have been addressed/resolved more deeply in the valkka-live video surveillance client.

But let’s turn back to the complete server + client example

```
import time
from valkka.core import *
from valkka.api2 import ShmemRGBClient
```

The filtergraph, once again:

```
(LiveThread:livethread) -----+ main branch, streaming
                                |
{ForkFrameFilter: fork_filter} <---- (AVThread:avthread) << --+ main branch, decoding
                        |
                    branch 1 +--> (OpenGLThread:glthread)
                        |
                    branch 2 +--> {IntervalFrameFilter: interval_filter} --> {SwScaleFrameFilter:
↪sws_filter} --> {RGBSharedMemFrameFilter: shmem_filter}
```

```
# define yuv=>rgb interpolation interval
image_interval=1000 # YUV => RGB interpolation to the small size is done each 1000
↪milliseconds and passed on to the shmem ringbuffer

# define rgb image dimensions
width  =1920//4
height =1080//4
```

RGBSharedMemFrameFilter needs unique name and the size of the shared memory ring-buffer:

```
# posix shared memory
shmem_name      ="lesson_4"          # This identifies posix shared memory - must be unique
shmem_buffers   =10                  # Size of the shmem ringbuffer
```

Next, we construct the filterchain as usual, from end-to-beginning:

```
# branch 1
glthread        =OpenGLThread("glthread")
gl_in_filter    =glthread.getFrameFilter()

# branch 2
shmem_filter    =RGBShmemFrameFilter(shmem_name, shmem_buffers, width, height)
# shmem_filter  =BriefInfoFrameFilter("shmem") # a nice way for debugging to see of
↪you are actually getting any frames here ..
sws_filter      =SwScaleFrameFilter("sws_filter", width, height, shmem_filter)
interval_filter =TimeIntervalFrameFilter("interval_filter", image_interval, sws_
↪filter)

# fork
fork_filter     =ForkFrameFilter("fork_filter", gl_in_filter, interval_filter)

# main branch
avthread        =AVThread("avthread", fork_filter)
av_in_filter    =avthread.getFrameFilter()
livethread      =LiveThread("livethread")
```

Define connection to camera: frames from the IP camera are written to live_out_filter and tagged with slot number 1:

```
# ctx =LiveConnectionContext(LiveConnectionType_rtsp, "rtsp://admin:nordic12345@192.
↪168.1.41", 1, av_in_filter)
ctx =LiveConnectionContext(LiveConnectionType_rtsp, "rtsp://admin:123456@192.168.0.134
↪", 1, av_in_filter)
```

Start threads:

```
glthread.startCall()
avthread.startCall()
livethread.startCall()

# start decoding
avthread.decodingOnCall()

livethread.registerStreamCall(ctx)

# create an X-window
window_id = glthread.createWindow()
glthread.newRenderGroupCall(window_id)

# maps stream with slot 1 to window "window_id"
context_id = glthread.newRenderContextCall(1, window_id, 0)
```

Ok, the server is alive and running. Let's do the client part for receiving frames.

```
client = ShmemRGBClient(
    name=shmem_name,
    n_ringbuffer=shmem_buffers,
    width=width,
    height=height,
    mstimeout=1000,          # client timeouts if nothing has been received in 1000_
    ↪milliseconds
    verbose=False
)
```

The client is ready to go. Before starting to receive frames, start playing the RTSP camera

```
livethread.playStreamCall(ctx)
```

Read 10 frames & exit

```
print("client starting")
cc = 0
while True:
    index, meta = client.pullFrame()
    if (index == None):
        print("timeout")
    else:
        data = client.shmem_list[index][0:meta.size]
        data = data.reshape((meta.height, meta.width, 3))
        print("got data: ", data.shape)
        cc += 1
    if cc >= 10:
        break
print("stopping..")
```

Clear the server

```
glthread.delRenderContextCall(context_id)
glthread.delRenderGroupCall(window_id)

# stop decoding
```

(continues on next page)

(continued from previous page)

```
avthread.decodingOffCall()

# stop threads
livethread.stopCall()
avthread.stopCall()
glthread.stopCall()

time.sleep(1)

print("bye")
```

Receiving frag-MP4 at Python

Download frag-MP4 example [\[here\]](#)

Fragmented MP4 (frag-MP4) is a container format suitable for live streaming and playing the video in most web browsers. For more information about this, see [here](#).

With libValkka you can mux your IP camera's H264 stream on-the-fly into frag-MP4 and then push it into cloud, using Python3 only.

This is similar what we have just done for the RGB bitmap frames. Now, instead of RGB24 frames, we receive frag-MP4 to the python side.

And, of course, we could do all the following things simultaneously: decode, show on screen, push RGB24 frames for video analysis, push frag-MP4 to your browser, etc. However, for clarity, here we just show the video on screen & receive frag-MP4 frames in our python process.

```
import time
from valkka.core import *
from valkka.api2 import FragMP4ShmemClient
```

The filtergraph for simultaneous video viewing and frag-MP4 muxing looks like this:

```
(LiveThread:livethread) -->-----+ main branch (forks_
↳into two)                          |
                                     |
(OpenGLThread:glthread) <-----(AVThread:avthread) << -----+ decoding branch
                                     |
+-----<-----+ mux branch
|
+--> {FragMP4MuxFrameFilter:fragmp4muxer} -->
↳{FragMP4ShmemFrameFilter:fragmp4shmem}
```

```
shmem_buffers = 10 # 10 element in the ring-buffer
shmem_name = "lesson_4_c" # unique name identifying the shared memory
cellsize = 1024*1024*3 # max size for each MP4 fragment
timeout = 1000 # in ms

# decoding branch
glthread = OpenGLThread("glthread")
gl_in_filter = glthread.getFrameFilter()
avthread = AVThread("avthread", gl_in_filter)
av_in_filter = avthread.getFrameFilter()

# mux branch
```

(continues on next page)

(continued from previous page)

```

shmem_filter    =FragMP4ShmemFrameFilter(shmem_name, shmem_buffers, cellsize)
mux_filter      =FragMP4MuxFrameFilter("fragmp4muxer", shmem_filter)
mux_filter.activate() # don't forget!

# fork
fork_filter     =ForkFrameFilter("fork_filter", av_in_filter, mux_filter)

# main branch
livethread      =LiveThread("livethread")

```

Define connection to camera: frames from the IP camera are written to live_out_filter and tagged with slot number 1:

```

# ctx =LiveConnectionContext(LiveConnectionType_rtsp, "rtsp://admin:nordic12345@192.168.1.41", 1, fork_filter)
ctx =LiveConnectionContext(LiveConnectionType_rtsp, "rtsp://admin:123456@192.168.0.134", 1, fork_filter)

```

Start threads:

```

glthread.startCall()
avthread.startCall()
livethread.startCall()

# start decoding
avthread.decodingOnCall()

livethread.registerStreamCall(ctx)

# create an X-window
window_id =glthread.createWindow()
glthread.newRenderGroupCall(window_id)

# maps stream with slot 1 to window "window_id"
context_id=glthread.newRenderContextCall(1,window_id,0)

```

Ok, the server is alive and running. Let's do the client part for receiving frames.

```

client = FragMP4ShmemClient(
    name=shmem_name,
    n_ringbuffer=shmem_buffers,
    n_size=cellsize,
    mstimeout=timeout,
    verbose=False
)

```

The client is ready to go. Before starting to receive frames, start playing the RTSP camera

```

livethread.playStreamCall(ctx)

```

Read 10 frames & exit

```

print("client starting")
cc = 0
while True:
    index, meta = client.pullFrame()
    if (index == None):

```

(continues on next page)

(continued from previous page)

```

        print("timeout")
    else:
        data = client.shmem_list[index][0:meta.size]
        print("got", meta.name.decode("utf-8"), "of size", meta.size)
        cc += 1
    if cc >= 100:
        break

print("stopping..")

mux_filter.deActivate() # don't forget!

```

Clear the server

```

glthread.delRenderContextCall(context_id)
glthread.delRenderGroupCall(window_id)

# stop decoding
avthread.decodingOffCall()

# stop threads
livethread.stopCall()
avthread.stopCall()
glthread.stopCall()

print("bye")

```

Advanced topics

By now you have learned how to pass frames from the libValkka infrastructure into python.

When creating more serious solutions, you can use a single python program to span multiprocesses (using Python's multiprocessing module) into servers and clients.

In these cases you must remember to span all multiprocesses in the very beginning of your code and then arrange an interprocess communication between them, so that the multiprocesses will instantiate the server and client in the correct order.

You can also create shared memory servers, where you can feed frames from the python side (vs. at the cpp side)

LibValkka shared memory server and client also features a posix file-descriptor API. It is convenient in cases, where a single process is listening simultaneously to several shared memory servers, and you want to do the i/o efficiently: you can use python's "select" module to do efficient "multiplexing" of pulling frames from several shmem clients.

For example, the Valkka Live program takes advantage of these features. It performs the following joggling of the frames through the shared memory:

1. Several shared memory servers, each one sending video from one camera.
2. Several client processes, each one receiving video from a shared memory server. Each client process establish it's own shared memory server for further sharing of the frames.
3. A master process that listens to multiple clients at the same time.

Number (1) works at the cpp side. (2) Is a separate multiprocess running OpenCV-based analysis. (3) Is a common Yolo object detector for all the clients.

For an example of a more serious multiprocessing project, please take a look [here](#).

5.3.5 Lesson 5 : Transmitting stream

Sending multicast

Download lesson [\[here\]](#)

In this lesson, we are receiving frames from an IP camera using LiveThread and recast those frames to a multicast address using another LiveThread. The filterchain looks like this:

```
(LiveThread:livethread) --> {InfoFrameFilter:info_filter} -->>
↳ (LiveThread:livethread2)
```

Let's start by importing Valkka:

```
import time
from valkka.core import *
```

Live555's default output packet buffer size might be too small, so let's make it bigger before instantiating any LiveThreads:

```
setLiveOutPacketBuffermaxSize(95000)
```

Construct the filtergraph from end-to-beginning:

```
livethread2 = LiveThread("livethread2")
live_in_filter = livethread2.getFrameFilter()
info_filter = InfoFrameFilter("info_filter", live_in_filter)
livethread = LiveThread("livethread")
```

Start threads

```
livethread2.startCall()
livethread.startCall()
```

Define stream source: incoming frames from IP camera 192.168.1.41 are tagged with slot number "2" and they are written to "info_filter":

```
ctx = LiveConnectionContext(LiveConnectionType_rtsp, "rtsp://admin:nordic12345@192.
↳ 168.1.41", 2, info_filter)
```

Define stream sink: all outgoing frames with slot number "2" are sent to port 50000:

```
out_ctx = LiveOutboundContext(LiveConnectionType_sdp, "224.1.168.91", 2, 50000)
```

Start playing:

```
livethread2.registerOutboundCall(out_ctx)
livethread.registerStreamCall(ctx)
livethread.playStreamCall(ctx)
```

Stream and recast to multicast for a while:

```
time.sleep(60)

livethread.stopStreamCall(ctx)
livethread.deregisterStreamCall(ctx)
livethread2.deregisterOutboundCall(out_ctx)
```

Stop threads in beginning-to-end order

```
livethread. stopCall();
livethread2.stopCall();

print("bye")
```

To receive the multicast stream, you need this file, save it as “multicast.sdp”:

```
v=0
o=- 0 0 IN IP4 127.0.0.1
s=No Name
c=IN IP4 224.1.168.91
t=0 0
a=tool:libavformat 56.36.100
m=video 50000 RTP/AVP 96
a=rtpmap:96 H264/90000
a=fmtp:96 packetization-mode=1
a=control:streamid=0
```

Then you can test that the stream is multicasted (while running the python script) with:

```
ffplay multicast.sdp
```

(feel free to launch this command several times simultaneously)

Note: Receiving and recasting the stream can also be done using a single LiveThread only. This is left as an exercise.

Using the RTSP server

Download lesson [\[here\]](#)

In this lesson, we establish an on-demand RTSP server at the localhost.

Stream is read from an IP camera and then re-streamed (shared) to a local RTSP server that serves at port 8554. While this snippet is running, you can test the RTSP server with:

```
ffplay rtsp://127.0.0.1:8554/stream1
```

Let’s start by importing Valkka:

```
import time
from valkka.core import *
```

Live555’s default output packet buffer size might be too small, so let’s make it bigger before instantiating any LiveThreads:

```
setLiveOutPacketBuffermaxSize(95000)
```

Construct the filtergraph from end-to-beginning:

```
livethread2 = LiveThread("livethread2")
live_in_filter = livethread2.getFrameFilter()
info_filter = InfoFrameFilter("info_filter", live_in_filter)
livethread = LiveThread("livethread")
```


Before starting the threads, establish an RTSP server on livethread2 at port 8554:

```
livethread2.setRTSPServer(8554);
```

Start threads

```
livethread2.startCall()
livethread.startCall()
```

Define stream source: incoming frames from IP camera 192.168.1.41 are tagged with slot number “2” and they are written to “info_filter”:

```
ctx = LiveConnectionContext(LiveConnectionType_rtsp, "rtsp://admin:nordic12345@192.168.1.41", 2, info_filter)
```

Define stream sink: all outgoing frames with slot number “2” are sent to the RTSP server, with substream id “stream1”:

```
out_ctx = LiveOutboundContext(LiveConnectionType_rtsp, "stream1", 2, 0)
```

Start playing:

```
livethread2.registerOutboundCall(out_ctx)
livethread.registerStreamCall(ctx)
livethread.playStreamCall(ctx)
```

Stream and recast to the RTSP server for a while:

```
time.sleep(60)

livethread.stopStreamCall(ctx)
livethread.deregisterStreamCall(ctx)
livethread2.deregisterOutboundCall(out_ctx)
```

Stop threads in beginning-to-end order

```
livethread.stopCall();
livethread2.stopCall();

print("bye")
```

5.3.6 Lesson 6 : Writing / reading stream

Download lesson [\[here\]](#)

In this lesson, we are (a) writing from a live stream to a file and (b) reading the file, decoding the stream and presenting it on the screen. The filtergraph goes like this:

```
*** (a) writing ***

(LiveThread:livethread) --> {FileFrameFilter:file_filter}

*** (b) reading ***

Reading part
(FileThread:filethread) -----+
```

(continues on next page)

(continued from previous page)

```

                                     |
Decoding part                       |
(AVThread:avthread) << -----+
    |
    |      Presentation part
+--->> (OpenGLThread:glthread)

```

Note that live and file streams are treated on an equal basis and with a similar filtergraph. We could also send the file over the net as a multicast stream.

Let's start by importing Valkka:

```

import time
from valkka.core import *

debug_log_all()

```

Writing is done by piping the stream into a FileFrameFilter:

```

file_filter  =FileFrameFilter("file_filter")
livethread   =LiveThread("livethread")

```

For reading, decoding and presenting, we construct the filtergraph as usual, from end-to-beginning:

```

# presentation part
glthread      =OpenGLThread ("glthread")
gl_in_filter  =glthread.getFrameFilter()

```

For file streams, the execution should block for frame bursts, so we request a blocking input FrameFilter from the AVThread:

```

avthread      =AVThread("avthread",gl_in_filter)
av_in_filter  =avthread.getBlockingFrameFilter()

# reading part
filethread    =FileThread("filethread")

```

Starting LiveThread will stream the frames to FileFrameFilter:

```

livethread .startCall()

ctx          =LiveConnectionContext(LiveConnectionType_rtsp, "rtsp://
↳admin:nordic12345@192.168.1.41", 1, file_filter)
# stream from 192.168.1.41, tag frames with slot number 1 and write to file_filter

livethread .registerStreamCall(ctx)
livethread .playStreamCall(ctx)

```

In order to start writing to disk, FileFrameFilter's "activate" method must be called with the filename. Only matroska (.mkv) files are supported:

```

print("writing to file during 30 secs")
file_filter.activate("kokkelis.mkv")

# stream for 30 secs
time.sleep(30)

```

(continues on next page)

(continued from previous page)

```
# close the file
file_filter.deActivate()

# stop livethread
livethread.stopCall()
```

File “kokkelis.mkv” has been created. Next, let’s setup stream decoding, presenting, etc. as usual and read the file:

```
print("reading file")
glthread.startCall()
filethread.startCall()
avthread.startCall()

# start decoding
avthread.decodingOnCall()

# create an x-window
window_id = glthread.createWindow()
glthread.newRenderGroupCall(window_id)

# maps stream with slot 1 to window "window_id"
context_id = glthread.newRenderContextCall(1, window_id, 0)
```

Open the file by sending it a call with the FileContext (file_ctx) identifying the file stream:

```
print("open file")
file_ctx = FileContext("kokkelis.mkv", 1, av_in_filter) # read from file "kokkelis.mkv
→", tag frames with slot number 1 and write to av_in_filter
filethread.openFileStreamCall(file_ctx)
```

Playing, seeking and stopping is done as follows:

```
print("play file")
filethread.playFileStreamCall(file_ctx)

# play the file for 10 secs
time.sleep(10)

# let's seek to seekpoint 2 seconds
print("seeking")
file_ctx.seektime_ = 2000
filethread.seekFileStreamCall(file_ctx)

# pause for 3 secs
print("pausing")
filethread.stopFileStreamCall(file_ctx)
time.sleep(3)

# continue playing for 5 secs
print("play again")
filethread.playFileStreamCall(file_ctx)
time.sleep(5)
```

Finally, exit:

```

glthread.delRenderContextCall(context_id)
glthread.delRenderGroupCall(window_id)

# exit
avthread .stopCall()
filethread.stopCall()
glthread .stopCall()

print("bye")

```

5.3.7 Lesson 7 : Decode, save, visualize, analyze and re-transmit

Download lesson [\[here\]](#)

In this example, we do simultaneously a lot of stuff, namely, save the stream to disk, decode it to bitmap, visualize it in two different x windows, pass the decoded frames to an OpenCV analyzer and re-transmit the stream to a multicast address.

Only a single connection to the IP camera is required and the stream is decoded only once.

The filtergraph looks like this:

```

main branch
(LiveThread:livethread) --> {ForkFrameFilter3: fork_filter}
                                |
                                branch 1 <---+
                                |
                                branch 2 <---+
                                |
                                branch 3 <---+

branch 1 : recast
-->> (LiveThread:livethread2_1)

branch 2 : save to disk
--> (FileFrameFilter:file_filter_2)

branch 3 : decode
-->> {AVThread:avthread_3} -----+
                                |
                                {ForkFrameFilter: fork_filter_3} <---+
                                |
                                branch 3.1 +--->> (OpenGLThread:glthread_3_1) --> to two x-windows
                                |
                                branch 3.2 +----> {IntervalFrameFilter: interval_filter_3_2} -->
->{SwScaleFrameFilter: sws_filter_3_2} --> {RGBSharedMemFrameFilter: shmem_filter_3_2}

```

There is a new naming convention: the names of filters, threads and fifos are tagged with “_branch_sub-branch”.

Programming the filtergraph tree is started as usual, from the outer leaves, moving towards the main branch:

```

# *** branch 1 ***
livethread2_1 =LiveThread("livethread2_1")
live2_in_filter =livethread2_1.getFrameFilter()

# *** branch 2 ***
file_filter_2 =FileFrameFilter("file_filter_2")

```

(continues on next page)

(continued from previous page)

```

# *** branch 3.1 ***
glthread_3_1      =OpenGLThread("glthread")
gl_in_filter_3_1  =glthread_3_1.getFrameFilter()

# *** branch 3.2 ***
image_interval=1000 # YUV => RGB interpolation to the small size is done each 1000_
↳milliseconds and passed on to the shmem ringbuffer
width  =1920//4      # CPU YUV => RGB interpolation
height =1080//4      # CPU YUV => RGB interpolation
shmem_name      ="lesson_4"      # This identifies posix shared memory - must be unique
shmem_buffers   =10              # Size of the shmem ringbuffer

shmem_filter_3_2  =RGBShmemFrameFilter(shmem_name, shmem_buffers, width, height)
sws_filter_3_2    =SwScaleFrameFilter("sws_filter", width, height, shmem_filter_3_2)
interval_filter_3_2 =TimeIntervalFrameFilter("interval_filter", image_interval, sws_
↳filter_3_2)

# *** branch 3 ***
fork_filter_3     =ForkFrameFilter("fork_3",gl_in_filter_3_1,interval_filter_3_2)
avthread_3        =AVThread("avthread_3",fork_filter_3)
av3_in_filter     =avthread_3.getFrameFilter()

# *** main branch ***
livethread        =LiveThread("livethread_1")
fork_filter        =ForkFrameFilter3("fork_filter",live2_in_filter,file_filter_2,av3_in_
↳filter)

```

The full code can be downloaded from [\[here\]](#).

The OpenCV client program for reading shared memory can be found from [\[lesson 4\]](#).

Testing the shared multicast stream was explained in [\[lesson 5\]](#).

5.3.8 Lesson 8: API level 2

General aspects

API level 2 tutorial codes are available at:

```

cd valkka_examples/api_level_2/tutorial
python3 lesson_8_a.py

```

So, by now you have learned how to construct complex filtergraphs with framefilters and threads, and how to encapsulate parts of the filtergraphs into separate classes in [lesson 3](#).

API level 2 does just that. It encapsulates some common cases into compact classes, starts the decoding threads for you, and provides easily accessible end-points (for the posix shared memory interface, etc.) for the API user.

The API level 2 filterchains, threads and shared memory processes can be imported with

```

from valkka.api2 import ...

```

API level 2 provides also extra wrapping for the thread classes (LiveThread, OpenGLThread, etc.). The underlying API level 1 instances can be accessed like this:

```
from valkka.api2 import LiveThread

livethread=LiveThread("live_thread")
livethread.core # this is the API level 1 instance, i.e. valkka.valkka_core.LiveThread
```

Keep in mind never to do a full import simultaneously from API levels one and two, i.e.

```
# NEVER DO THIS!
from valkka.valkka_core import *
from valkka.api2 import *
```

as the threads (LiveThread, OpenGLThread, etc.) have identical names.

The *PyQT testsuite* serves also as API level 2 reference.

A simple example

Download lesson [\[here\]](#)

First, import API level 2:

```
import time
from valkka.api2 import LiveThread, OpenGLThread
from valkka.api2 import BasicFilterchain
```

Instantiating the API level 2 LiveThread starts running the underlying cpp thread:

```
livethread=LiveThread( # starts live stream services (using live555)
    name      ="live_thread",
    verbose=False,
    affinity=-1
)
```

Same goes for OpenGLThread:

```
openglthread=OpenGLThread(
    name      ="glthread",
    n_720p    =20, # reserve stacks of YUV video frames for various resolutions
    n_1080p   =20,
    n_1440p   =0,
    n_4K      =0,
    verbose   =False,
    msbuftime=100,
    affinity=-1
)
```

The filterchain and decoder (AVThread) are encapsulated into a single class. Instantiating starts the AVThread (decoding is off by default):

```
chain=BasicFilterchain( # decoding and branching the stream happens here
    livethread  =livethread,
    openglthread=openglthread,
    address     ="rtsp://admin:nordic12345@192.168.1.41",
    slot        =1,
    affinity    =-1,
    verbose     =False,
```

(continues on next page)

(continued from previous page)

```
msreconnect =10000 # if no frames in ten seconds, try to reconnect
)
```

BasicFilterchain takes as an argument the LiveThread and OpenGLThread instances. It creates the relevant connections between the threads.

Next, create an x-window, map stream to the screen, and start decoding:

```
# create a window
win_id =openglthread.createWindow()

# create a stream-to-window mapping
token =openglthread.connect(slot=1,window_id=win_id) # present frames with slot_
↪number 1 at window win_id

# start decoding
chain.decodingOn()
# stream for 20 secs
time.sleep(20)
```

Close threads in beginning-to-end order

```
livethread.close()
chain.close()
openglthread.close()
print("bye")
```

5.3.9 Lesson 9 : Drawing Bounding Boxes

Here we stream video to a single X-window just like in tutorial example 3, but we also draw some bounding boxes on the video.

Download lesson [\[here\]](#)

First, business as usual (and like in tutorial example 3)

```
import time
from valkka.core import *
glthread =OpenGLThread ("glthread")
gl_in_filter =glthread.getFrameFilter()

avthread =AVThread("avthread",gl_in_filter)
av_in_filter =avthread.getFrameFilter()

livethread =LiveThread("livethread")

ctx =LiveConnectionContext(LiveConnectionType_rtsp, "rtsp://admin:nordic12345@192.168.
↪1.41", 1, av_in_filter)

glthread.startCall()
avthread.startCall()
livethread.startCall()

avthread.decodingOnCall()

livethread.registerStreamCall(ctx)
```

(continues on next page)

(continued from previous page)

```
livethread.playStreamCall(ctx)

window_id =glthread.createWindow()

glthread.newRenderGroupCall(window_id)

context_id=glthread.newRenderContextCall(1,window_id,0) # slot, render group, z

time.sleep(1)
```

Let's add a bounding box, overlaying the video. Parameters for bounding box (left, bottom) -> (right, top) are given in the order left, right, top, bottom.

Coordinates are relative coordinates from 0 to 1.

```
bbox=(0.25, 0.75, 0.75, 0.25) # left, right, top, bottom

glthread.addRectangleCall(context_id, bbox[0], bbox[1], bbox[2], bbox[3])
```

You could add more bounding boxes with consecutive calls to **glthread.addRectangleCall**

Let's play video for 10 seconds

```
time.sleep(10)
```

Finally, clear the bounding boxes and exit

```
glthread.clearObjectsCall(context_id)

glthread.delRenderContextCall(context_id)
glthread.delRenderGroupCall(window_id)

avthread.decodingOffCall()

livethread.stopCall()
avthread.stopCall()
glthread.stopCall()

print("bye")
```

5.3.10 Lesson 10 : USB Cameras

Valkka has experimental support for H264 streaming USB Cameras. To see if your camera supports H264 streaming, use the following command:

```
v4l2-ctl --list-formats -d /dev/video2
```

Information about your cameras can be found also under this directory structure:

```
/sys/class/video4linux/
```

The only difference to handling IP cameras is that a different thread (*USBDeviceThread*) is used to stream the video.

Download lesson [\[here\]](#)


```
import time
from valkka.core import *
glthread      =OpenGLThread ("glthread")
gl_in_filter   =glthread.getFrameFilter()

avthread       =AVThread("avthread",gl_in_filter)
av_in_filter    =avthread.getFrameFilter()
```

USBDeviceThread reads and multiplexes all USB cameras

```
usbthread      =USBDeviceThread("usbthread")
```

Define the usb camera (/dev/video2) and where it is going to be streamed (to av_in_filter with slot number 1):

```
ctx = USBCameraConnectionContext("/dev/video2", 1, av_in_filter)
# The default resolution is 720p
# If you want to set the width and height yourself, uncomment the following line
# ctx.width = 1920; ctx.height = 1080;

glthread.startCall()
avthread.startCall()
usbthread.startCall()

avthread.decodingOnCall()

window_id =glthread.createWindow()

glthread.newRenderGroupCall(window_id)

context_id=glthread.newRenderContextCall(1,window_id,0) # slot, render group, z

time.sleep(1)

usbthread.playCameraStreamCall(ctx);
```

Stream for a minute. Patience. At least the HD Pro Webcam C920 does not send keyframes too often ..

```
time.sleep(60)

usbthread.stopCameraStreamCall(ctx);

glthread.delRenderContextCall(context_id)
glthread.delRenderGroupCall(window_id)

avthread.decodingOffCall()

usbthread.stopCall()
avthread.stopCall()
glthread.stopCall()

print("bye")
```

5.3.11 Lesson 11 : ValkkaFS

As you learned from earlier lessons, you can redirect video streams to matroska (.mkv) video files.

Here we'll be streaming video to the custom ValkkaFS filesystem.

ValkkaFS dumps video to a dedicated file, or to an entire partition or disk. Arriving H264 frames are written in their arriving time order, into the same (large) file that is organized in blocks. For more details, consult the [ValkkaFS section](#) and the `cpp` documentation.

Here we provide several examples for writing to and reading from ValkkaFS. These include importing video from ValkkaFS to matroska, and caching frames from ValkkaFS and passing them downstream at play speed.

In a typical VMS application, writing and reading run concurrently: writing thread dumps frames continuously to the disk, while reading thread is evoked only at user's request.

Writing

Let's start by dumping video from IP cameras into ValkkaFS.

Download lesson [\[here\]](#)

This will be our filtergraph:

```
(LiveThread:livethread) -->> (ValkkaFSWriterThread:writethread)
```

Let's import valkka level 1 API, and ValkkaSingleFS from level 2 API:

```
import time
from valkka.core import *
from valkka.fs import ValkkaSingleFS
from valkka.api2 import loglevel_debug, loglevel_normal, loglevel_crazy
```

Let's set our IP camera's address:

```
rtsp_address="rtsp://admin:12345@192.168.0.157"
```

If you want to see the filesystem writing each frame, enable these debugging loggers:

```
#setLogLevel_filelogger(loglevel_crazy)
#setLogLevel_valkkafslogger(loglevel_crazy)
```

There are two flavors of ValkkaFS under the `valkka.fs` namespace, namely `ValkkaSingleFS` and `ValkkaMultiFS`. In the former, there is one file per one camera/stream, while in the latter you can dump several streams into the same file.

The `ValkkaSingleFS` instance handles the metadata of the filesystem. Let's create a new filesystem and save the metadata into directory `/tmp/testvalkkafs`

```
valkkafs = ValkkaSingleFS.newFromDirectory(
    dirname = "/tmp/testvalkkafs",
    blocksize = (2048//8)*1024, # note division by 8: 2048 KiloBITS
    n_blocks = 10,
    verbose = True)
```

Here one block holds 2048 KBits (256 KBytes) of data. Let suppose that your camera streams 1024KBits per second (kbps): now a block will be finished every 2 seconds.

If you now set your IP camera to key-frame every one second, you will have two key frames per each block, which is a necessary condition for efficient seeking using the filesystem.

The total size of the device file where frames are streamed, will be (256kB * 10) 2560 kB.

You could also skip the parameter `n_blocks` and instead define the device file size directly with `device_size = 2560*1024`.

Now the directory has the following files:

blockfile	Table of block timestamps. Used for seeking, etc.
dumpfile	Frames are streamed into this file (the "device file")
valkkafs.json	Metadata: block size, current block, etc.

Next, we create and start (1) the thread responsible for writing the frames into ValkkaFS and (2) LiveThread that is reading the cameras:

```
writerthread = ValkkaFSWriterThread("writer", valkkafs.core)
livethread = LiveThread("livethread")
```

All cameras write to the same FrameFilter, handled by the writing thread:

```
file_input_framefilter = writerthread.getFrameFilter()
```

Read camera and designate it with slot number 1

```
ctx = LiveConnectionContext(LiveConnectionType_rtsp, rtsp_address, 1, file_input_
↪framefilter)
```

Next, start threads

```
writerthread.startCall()
livethread.startCall()
```

Frames with slot number 1 are identified in the filesystem with id number 925412 (which we just invented):

```
writerthread.setSlotIdCall(1, 925412)

livethread.registerStreamCall(ctx)
livethread.playStreamCall(ctx)
```

Idle for some secs while the threads run in the background

```
print("recording!")
time.sleep(3)
```

At this moment, let's take a look at the blocktable

```
a=valkkafs.getBlockTable()
print(a)
if a.max() <= 0:
    print("Not a single block finished so far..")
    valkkafs.core.updateTable(disk_write=True)
    print("Check blocktable again")
    a=valkkafs.getBlockTable()
    if a.max() <= 0:
        print("Not a single frame so far..")
    print(a)
```

Frames in a certain block are saved definitely into the book-keeping only once a block is finished.

In the code above, we force a block write even if the block has not filled up.

Let's continue & let the threads do their stuff for some more time

```
print("recording some more")
time.sleep(30)

livethread.stopCall()
writerthread.stopCall()
```

Let's take a look at the blocktable again:

```
print(a)

print("bye")
```

Reading 1

In these following two examples, we request frames from ValkkaFS

Download lesson [\[here\]](#)

Same imports as before:

```
import time, sys
from valkka.core import *
from valkka.fs import ValkkaSingleFS
```

Load ValkkaFS metadata:

```
valkkafs = ValkkaSingleFS.loadFromDirectory(dirname="/tmp/testvalkkafs")
```

Let's take a look at the blocktable:

```
a = valkkafs.getBlockTable()
print(a)
```

Construct the filterchain: write from the reader thread into the verbose InfoFrameFilter

```
out_filter = InfoFrameFilter("reader_out_filter")
readerthread = ValkkaFSReaderThread("reader", valkkafs.core, out_filter)
```

Start the reader thread:

```
readerthread.startCall()
```

Frames with id number 925412 are mapped into slot 1:

```
readerthread.setSlotIdCall(1, 925412)
```

Request blocks 0, 1 from the reader thread. Information of frames from these blocks are dumped on the terminal

```
readerthread.pullBlocksPyCall([0,1])
time.sleep(1)
```

Exit the thread:

```
readerthread.stopCall()
print("bye")
```

Reading 2

Download lesson [\[here\]](#)

```
import time, sys
from valkka.core import *
from valkka.fs import ValkkaSingleFS
```

Load ValkkaFS metadata:

```
valkkafs = ValkkaSingleFS.loadFromDirectory(dirname="/tmp/testvalkkafs")
```

Let's take a look at the blocktable:

```
a = valkkafs.getBlockTable()
print(a)
```

Instantiate ValkkaFSTool that allows us to peek into the written data

```
tool = ValkkaFSTool(valkkafs.core)
```

Contents of individual blocks can now be inspected like this:

```
tool.dumpBlock(0)
tool.dumpBlock(1)
```

You'll get output like this:

```
----- Block number : 0 -----
[925412] <BasicFrame: timestamp=1543314164986 subsession_index=0 slot=0 / payload_
↪size=29 / H264: slice_type=7> *      0 0 0 1 103 100 0 31 172 17 22 160 80 5 186 16_
↪0 1 25 64
[925412] <BasicFrame: timestamp=1543314164986 subsession_index=0 slot=0 / payload_
↪size=8 / H264: slice_type=8>      0 0 0 1 104 238 56 176
[925412] <BasicFrame: timestamp=1543314165135 subsession_index=0 slot=0 / payload_
↪size=32 / H264: slice_type=7> *      0 0 0 1 103 100 0 31 172 17 22 160 80 5 186 16_
↪0 1 25 64
[925412] <BasicFrame: timestamp=1543314165135 subsession_index=0 slot=0 / payload_
↪size=8 / H264: slice_type=8>      0 0 0 1 104 238 56 176
[925412] <BasicFrame: timestamp=1543314165135 subsession_index=0 slot=0 / payload_
↪size=19460 / H264: slice_type=5>      0 0 0 1 101 136 128 8 0 1 191 180 142 114 29_
↪255 192 79 52 19
[925412] <BasicFrame: timestamp=1543314165215 subsession_index=0 slot=0 / payload_
↪size=32 / H264: slice_type=7> *      0 0 0 1 103 100 0 31 172 17 22 160 80 5 186 16_
↪0 1 25 64
[925412] <BasicFrame: timestamp=1543314165215 subsession_index=0 slot=0 / payload_
↪size=8 / H264: slice_type=8>      0 0 0 1 104 238 56 176
[925412] <BasicFrame: timestamp=1543314165215 subsession_index=0 slot=0 / payload_
↪size=19408 / H264: slice_type=5>      0 0 0 1 101 136 128 8 0 1 191 180 142 114 29_
↪255 193 80 200 71
[925412] <BasicFrame: timestamp=1543314165335 subsession_index=0 slot=0 / payload_
↪size=4928 / H264: slice_type=1>      0 0 0 1 65 154 0 64 2 19 127 208 117 223 181 129_
↪22 206 32 84
...
```

Frame id number is indicated in the first column. Asterix (*) marks the seek points. In the final rows, first few bytes of the actual payload are shown.

Let's see the min and max time of frames written in this ValkkaFS

```
(t0, t1) = valkkafs.getTimeRange()
print("Min and Max time in milliseconds:", t0, t1)
```

These are milliseconds, so to get *struct_time* object we need to do this:

```
print("Min time:", time.gmtime(t0/1000))
print("Max time:", time.gmtime(t1/1000))
```

Block numbers corresponding to a certain time range can be searched like this:

```
req = (t0, t1)
block_indices = valkkafs.getInd(req)
print("Block indices =", block_indices)
```

Now you could pass to indices to the *ValkkaFSReaderThread* method **pullBlocksPyCall** to recover all frames from that time interval.

Another usefull method is *getIndNeigh*. It returns blocks from the neighborhood of a certain target time:

```
req = (t1+t0)//2
block_indices = valkkafs.getIndNeigh(n=2, time=req)
print("Block indices =", block_indices)
```

That will return the target block plus two blocks surrounding it.

You would call this method when a user requests a seek to a certain time and you want to be sure that there are enough frames surrounding that time instant

Matroska export

Let's start by recalling *the very first lesson*. There we saw how *LiveThread* sends **Setup Frames** at streaming initialization. Setup frames are used all over the libValkka infrastructure, to carry information about the video stream, to signal the stream start and to initialize decoders, muxers, etc.

On the other hand, *ValkkaFSReaderThread* is designed to be a simple beast: it does not have any notion of stream initialization. It simply provides frames on a per-block basis.

We must use a special *FrameFilter* called **InitStreamFrameFilter**, in order to add the Setup Frames into the stream.

Download lesson [here]

Same imports as before:

```
import time, sys
from valkka.core import *
from valkka.api2 import loglevel_debug, loglevel_normal
from valkka.fs import ValkkaSingleFS

setLogLevel_filelogger(loglevel_debug)
```

Load ValkkaFS metadata:

```
valkkafs = ValkkaSingleFS.loadFromDirectory(dirname="/tmp/testvalkkafs")
```

Let's take a look at the blocktable:

```
a = valkkafs.getBlockTable()
print(a)
```

Next, construct the filterchain. It looks like this:

```

main branch
(ValkkaFSWriterThread:readerthread) --> {ForkFrameFilterN:fork_filter} ---+
                                                                    |
                                                                    +--> branch_1
↪1

branch 1 : {PassSlotFrameFilter:slot_filter} --> {InitStreamFrameFilter:init_stream} -
↪-> {FileFrameFilter:file_filter} --> output file

```

Here we have introduced yet another **FrameFilter** that performs forking. An arbitrary number of terminals can be connected to **ForkFrameFilterN**. Terminals can be connected and disconnected also while threads are running.

The **PassSlotFrameFilter** passes frames with a certain slot number as we want frames only from a single stream to the final output file.

```

# main branch
fork_filter = ForkFrameFilterN("fork")
readerthread = ValkkaFSReaderThread("reader", valkkafs.core, fork_filter)

# branch 1
file_filter = FileFrameFilter("file_filter")
init_stream = InitStreamFrameFilter("init_filter", file_filter)
slot_filter = PassSlotFrameFilter("", 1, init_stream)

# connect branch 1
fork_filter.connect("info", slot_filter)

# activate file write
file_filter.activate("kokkelis.mkv")

```

Start the reader thread:

```
readerthread.startCall()
```

Frames with id number 925412 are mapped into slot 1:

```
readerthread.setSlotIdCall(1, 925412)
```

Request blocks 0-4 from the reader thread. Information of frames from these blocks are dumped on the terminal

```
readerthread.pullBlocksPyCall([0,1,3,4])
time.sleep(1)
```

Exit the thread:

```
readerthread.stopCall()
print("bye")
```

Playing frames

As you learned in the previous examples of this section, **ValkkaFSReader** pushes frames downstream in “bursts”, several blocks worth of frames in a single shot.

However, we also need something that passes recorded frames downstream (say, for visualization and/or for transmission) at “play speed” (say, at that 25 fps).

This is achieved with **FrameCacherThread**, which caches, seeks and passes frames downstream at play speed.

In detail, ValkkaFSReaderThread passes frames to FrameCacherThread which caches them into memory. After this, *seek*, *play* and *stop* can be requested from FrameCacherThread, which then passes the frames downstream from a seek point and at the original play speed at which the frames were recorded into ValkkaFS.

FrameCacherThread can be given special python callback functions that are being called when the min and max time of cached frames changes and when frame presentation time goes forward.

FrameCacherThread is very similar to other threads that send stream (like LiveThread), so it also handles the sending of Setup Frames downstream correctly.

Download lesson [\[here\]](#)

Same imports as before:

```
import time, sys
from valkka.core import *
from valkka.api2 import loglevel_debug, loglevel_normal
from valkka.fs import ValkkaSingleFS

setLogLevel_filelogger(loglevel_debug)
```

Load ValkkaFS metadata:

```
valkkafs = ValkkaSingleFS.loadFromDirectory(dirname="/tmp/testvalkkafs")
```

Let's take a look at the blocktable:

```
a = valkkafs.getBlockTable()
print(a)
```

Filterchain is going to look like this:

```
(ValkkaFSReaderThread:readerthread) --> (FileCacheThread:cachertthread) -->
↳{InfoFrameFilter:out_filter}
                                     |
                                     $
                                     setPyCallback : [int] current mstime, freq: 500 ms
                                     setPyCallback2 : [tuple] (min mtimestamp, max_
↳mtimestamp)
```

As you can see, where have introduced new notation here.

\$ designates callbacks that are used by FileCacheThread. It's up to you to define the python code in these callbacks. The callbacks are registered by using the *setPyCallback* and *setPyCallback2* methods.

Next, we proceed in constructing the filterchain in end-to-beginning order.

ValkkaFSReaderThread will write all it's frames into FileCacheThread's input FrameFilter.

```
out_filter = InfoFrameFilter("out_filter") # will be registered later with_
↳cachertthread
cachertthread = FileCacheThread("cachert")
readerthread = ValkkaFSReaderThread("reader", valkkafs.core, cachertthread.
↳getFrameFilter()) # ValkkaFSReaderThread => FileCacheThread
```

Next, define callbacks for FileCacheThread

Define a global variable: a tuple that holds the min and max millisecond timestamps of cached frames:


```
current_time_limits = None
```

This following function will be called frequently by FileCacheThread to inform us about the current millisecond timestamp:

```
def current_time_callback(mstime: int):
    global current_time_limits
    try:
        print("current time", mstime)
        if current_time_limits is None:
            return

        if mstime >= current_time_limits[1]:
            print("current time over cached time limits!")
            # cacherthread.rewindCall() # TODO
            # # or alternatively, handle the situation as you please
            # # .. for example, request more blocks:
            # readerthread.pullBlocksPyCall(your_list_of_blocks)
            pass

    except Exception as e:
        print("current_time_callback failed with ", str(e))
        return
```

The next callback is evoked when FileCacheThread receives new frames for caching. It informs us about the minimum and maximum millisecond timestamps:

```
def time_limits_callback(times: tuple):
    global current_time_limits
    try:
        print("new time limits", times)
        current_time_limits = times

    except Exception as e:
        print("time_limits_callback failed with ", str(e))
        return
```

The callbacks should be kept ASAP (as-simple-as-possible) and return immediately. You also might want them to send a Qt signal in your GUI application.

Typically, they should use only the following methods of the libValkka API:

```
valkka.fs.ValkkaSingleFS
    .getBlockTable
    .getTimeRange

valkka.core.ValkkaFSReaderThread
    .pullBlocksPyCall
```

Register the callbacks into the FileCacheThread

```
calerthread.setPyCallback(current_time_callback)
calerthread.setPyCallback2(time_limits_callback)
```

Start the threads

```
calerthread.startCall()
readerthread.startCall()
```

Frames saved with id number 925412 to ValkkaFS are mapped into slot number 1:

```
readerthread.setSlotIdCall(1, 925412)
```

FileCacheThread will write frames with slot number 1 into InfoFrameFilter:

```
ctx = FileStreamContext(1, out_filter)
cacherthread.registerStreamCall(ctx)
```

Request blocks 0-4 from the reader thread. The frames will be cached by FileCacheThread.

```
readerthread.pullBlocksPyCall([0,1,3,4])
```

Before frames can be played, a seek must be performed to set a time reference.

```
mstimestamp = int(a[1,0]) # take the first timestamp from the blocktable. Use int()
↳to convert to normal python integer.
print("seeking to", mstimestamp)
cacherthread.seekStreamsCall(mstimestamp)
```

It's up to the API user to assure that the used mstimestamp is within the correct limits (i.e. requested blocks).

Next, let the stream play for 10 seconds

```
cacherthread.playStreamsCall()
time.sleep(10)
```

Stop threads

```
cacherthread.stopCall()
readerthread.stopCall()

print("bye")
```

ValkkaFSManager

In the previous example, two callback functions which define the application's behaviour with respect to recorded and cached frames were used.

How you define the callback functions, depends completely on your application, say, if you're creating an application that does playback of recorded stream, you might want to request new blocks at your *current_time_callback*, once the time goes over limits of currently cached frames.

We are starting to get some idea on the challenges that arise when doing *simultaneous reading, writing, caching and playing* of a large number of (non-continuous) video streams. For a more discussions on this, please see the [ValkkaFS section](#).

To make things easier, `valkka.fs` namespace has a special class `ValkkaFSManager` that handles the simultaneous & synchronous writing and playing of multiple video streams.

For an example on how to use `ValkkaFSManager`, please see `test_studio_6.py` at the [PyQt testsuite](#).

6.1 Single thread

By default, libValkka uses only a single core per decoder (the decoding threads can also be bound to a certain core - see [the testsuite](#) for more details).

This is a good idea if you have a *large number of light streams*. What is exactly a *light stream* depends on your linux box, but let's assume here that it is a 1080p video running approx. at 20 frames per second.

6.2 Multithread

If you need to use a single *heavy stream*, then you might want to dedicate several cores in decoding a single stream. A *heavy stream* could be example that 4000x3000 4K camera of yours running at 60 frames per second (!)

However, before using such a beast, you must ask yourself, do you really need something like that?

The biggest screen you'll ever be viewing the video from, is probably 1080p, while a framerate of 15 fps is good for the human eye. Modern convoluted neural networks (yolo, for example), are using typically a resolution of ~ 600x600 pixels and analyzing max. 30 frames per seconds. And we still haven't talked about clogging up your LAN.

If you really, really have to use several threads per decoder, modify tutorial's [lesson 2](#) like this:

```
avthread = AVThread("avthread",info_filter)
avthread.setNumberOfThreads(4)
```

That will dedicate four cores to the decoder. Remember to call *setNumberOfThreads* before starting the AVThread instance.

6.3 GPU Accelerated

Hardware (hw) accelerated decoders are available in libValkka. For more details, please see [here](#). However, before using them, you should ask yourself if you really need them. Maybe it is better to save all GPU muscle for deep learning inference instead?

Video hw acceleration libraries are typically closed-source implementations, and the underlying “black-box” can be poorly implemented and suffer from memory leaks. Read for example [this thread](#). Slowly accumulating memleaks are poison for live video streaming applications which are supposed to stream continuously for days, weeks and even forever.

Sometimes the proprietary libraries may also restrict how many simultaneous hw video decoders you can have, while there are no such restrictions on CPU decoding.

So, if you have a linux box, dedicated solely for streaming and with decent CPU(s), don’t be over-obsessed with hw decoding.

6.4 Queueing frames

Typically, when decoding H264 video, handling the intra-frame takes much more time than decoding the consecutive B- and P-frames. This is very pronounced for *heavy streams* (see above).

Because of that the intra frame will arrive late for the presentation, while the consecutive frames arrive in a burst.

This problem can be solved with buffering. Modify tutorial’s [lesson 3](#) like this:

```
from valkka.core import *

glthread = OpenGLThread ("glthread")

gl_ctx = core.OpenGLFrameFifoContext ()
gl_ctx.n_720p = 0
gl_ctx.n_1080p = 0
gl_ctx.n_1440p = 0
gl_ctx.n_4K = 40

glthread = OpenGLThread("glthread", gl_ctx, 500)
```

That will reserve 40 4K frames for queueing and presentation of video, while the buffering time is 500 milliseconds.

For level 2 API, it would look like this:

```
from valkka.api2 import *

glthread = OpenGLThread(
    name="glthread",
    n_720p = 0,
    n_1080p = 0,
    n_1440p = 0,
    n_4K = 40,
    msbuftime = 500
)
```

Remember also that for certain class of frames (720p, 1080p, etc.):

```
number of pre-reserved frames >= total framerate x buffering time
```

For testing, you should use the [test_studio_1.py](#) program. See also [this](#) lesson of the tutorial.

Buffering solves many other issues as well. If you don't get any image and the terminal screaming that "there are no more frames", then just enhance the buffering.

Integrating with Qt and multiprocessing

7.1 Qt integration

Valkka can be used with any GUI framework, say, with GTK or Qt. Here we have an emphasis on Qt, but the general guidelines discussed here, apply to any other GUI framework as well. Concrete examples are provided only for Qt.

At the GUI's main window constructor:

1. Start your python multiprocesses if you have them (typically used for machine vision analysis)
2. Instantiate filtergraphs (from dedicated filtergraph classes, like we did in [tutorial](#))
3. Start all libValkka threads (LiveThread, OpenGLThread, etc.)
4. Start a QThread listening to your python multiprocesses (1), in order to translate messages from multiprocesses to Qt signals.

Finally:

5. Start your GUI framework's execution loop
6. At main window close event, close all threads, filterchains and multiprocesses

Examples of all this can be found in [the PyQt testsuite](#) together with several filtergraph classes.

7.2 Drawing video into a widget

X-windows, i.e. “widgets” in the Qt slang, can be created at the Qt side and passed to Valkka. Alternatively, x-windows can be created at the Valkka side and passed to Qt as “foreign widgets”.

As you learned in the tutorial, we use the X-window window ids like this:

```
context_id=glthread.newRenderContextCall(1,window_id,0)
```

That creates a mapping: all frames with slot number “1” are directed to an X-window with a window id “window_id” (the last number “0” is the z-stacking and is not currently used).

We can use the window id of an existing Qt widget “some_widget” like this:

```
window_id=int(some_widget.winId())
```

There is a stripped-down example of this in

```
valkka_examples/api_level_1/qt/  
    single_stream_rtsp.py
```

You can also let Valkka create the X-window (with correct visual parameters, no XSignals, etc.) and embed that X-window into Qt. This can be done with:

```
foreign_window = QtGui.QWindow.fromWinId(win_id)
foreign_widget = QtWidgets.QWidget.createWindowContainer(foreign_window, parent=parent)
```

where “win_id” is the window_id returned by Valkka, “parent” is the parent widget of the widget we’re creating here and “foreign_widget” is the resulting widget we’re going to use in Qt.

However, “foreign_widget” created this way does not catch mouse gestures. This can be solved by placing a “dummy” QWidget on top of the “foreign_widget” (using a layout). An example of this can be found in

7.3 Python multiprocessing

In *lesson 4* of the tutorial, we launched a separate python interpreter running a client program that was using decoded and shared frames.

That approach works for Qt programs as well, but it is more convenient to use multiprocesses constructed with Python3's `multiprocessing` library.

Using python multiprocesses with Qt complicates things a bit: we need a way to map messages from the multiprocessing into signals at the main Qt program. This can be done by communicating with the python multiprocessing via pipes and converting the pipe messages into incoming and outgoing Qt signals.

Let's state that graphically:

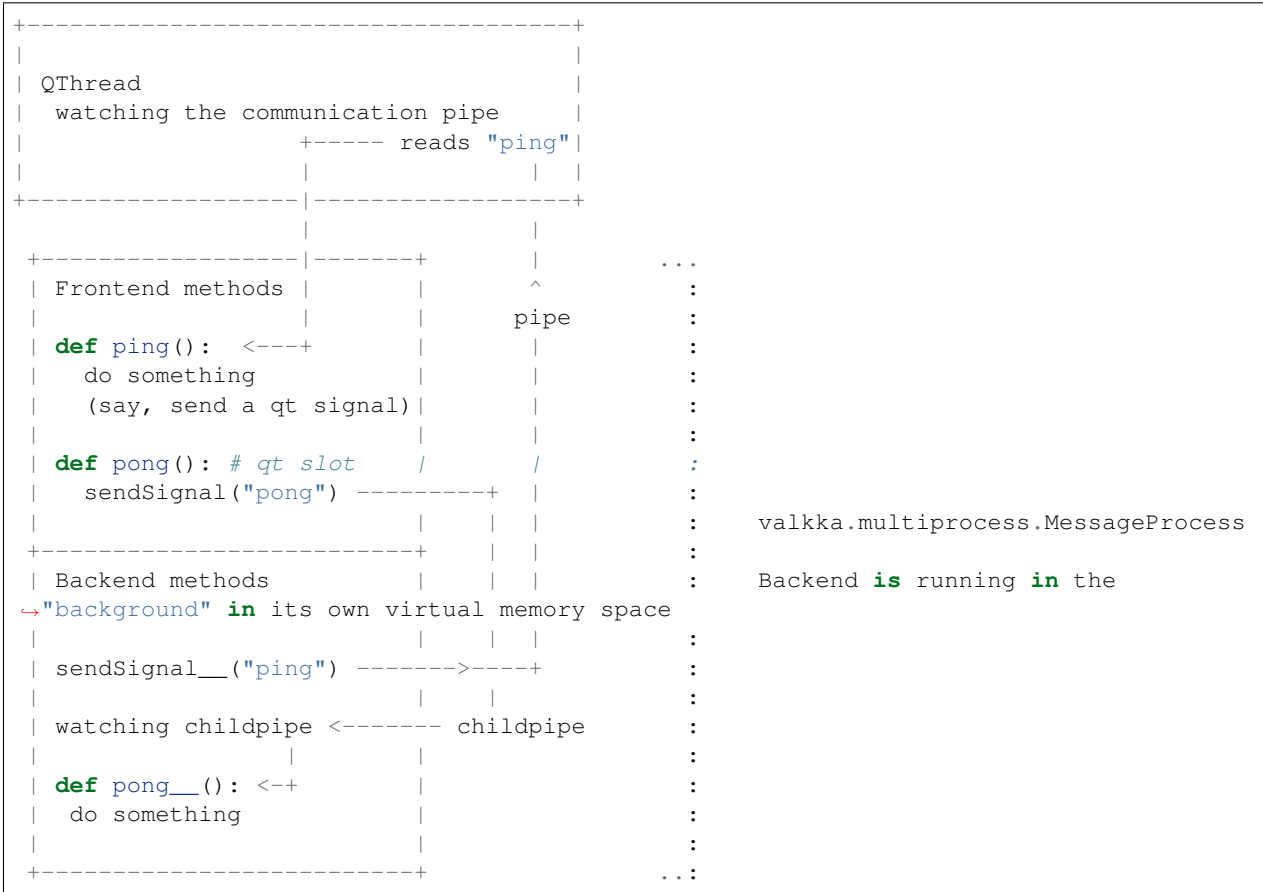
```

Qt main loop running with signals and slots
|
+--- QThread receiving/sending signals --- writing/reading communication pipes
|
+-----+-----+-----+
|
multiprocess_1    multiprocess_2
python multiprocesses doing
and writing/reading their
==> subclass from valkka.

```


Note that we only need a single QThread to control several multiprocesses.

Let's dig deeper into our strategy for interprocess communication with the Qt signal/slot system:



The class `valkka.multiprocess.MessageProcess` provides a model class that has been derived from python's `multiprocessing.Process` class. In `MessageProcess`, the class has both "frontend" and "backend" methods.

The `MessageProcess` class comes with the main `libValkka` package, but you can also install it separately.

It is documented in detail in [valkka-multiprocess package documentation](#).

We highly recommend that you read that documentation as it is important to understand what you are doing here - what is running in the "background" and what in your main python (Qt) process as including `libValkka` threads and `QThreads` into the same mix can easily result in the classical "fork-combined-with-threading" pitfall, leading to a leaky-crashy program.

Please refer also to *the PyQt testsuite* how to do things correctly.

A stand-alone python multiprocessing/Qt sample program is provided here (without any `libValkka` components):

```

valkka_examples/api_level_2/qt/

multiprocessing_demo.py

```

Try it to see the magic of python multiprocessing connected with the Qt signal/slot system.

Finally, for creating a `libValkka` Qt application having a frontend `QThread`, that controls `OpenCV` process(es), take a look at

```
valkka_examples/api_level_2/qt/  
  
    test_studio_detector.py
```

And follow the code therein. You will find these classes:

- *MovementDetectorProcess* : multiprocessing with Qt signals and OpenCV
- *QHandlerThread* : the frontend QThread

A more full-blown multiprocessing orchestration example can be found as in [this python package](#).

7.4 C++ API

There is no obligation to use Valkka from python - the API is usable from cpp as well: all python libValkka threads and filters are just swig-wrapped cpp code.

If programming in Qt with C++ is your thing, then you can just forget all that multiprocessing considered here and use cpp threads instead.

Say, you can use Valkka's FrameFifo and Thread infrastructure to create threads that read frames and feed them to an OpenCV analyzer (written in cpp).

You can also communicate from your custom cpp thread to the python side. A python program using an example cpp thread (*TestThread*) which communicates with PyQt signals and slots can be found here:

```
valkka_examples/api_level_2/qt/  
  
    cpp_thread_demo.py
```

See also the documentation for the cpp source code of [TestThread](#)

(If you just have monitors connected to a single graphics card, no need to be here)

8.1 Introduction

Consider the following setup:

- You have 2 graphic cards (GPUs)
- Each card is connected to 4 monitors
- You want to build a video wall with 8 monitors

The two graphic cards are physically separate entities with their own resources, so it actually makes sense to keep them that way in the software side as well.

For creating such a separation, Linux offers a robust, well-tested and ancient (!) solution: **multiple X-Screens**.

Let's state the example case graphically:

```

      +----> monitor 1-1  |
      |                   |
GPU1  --+----> monitor 1-2  |   X-Screen 0
      |                   |   Spanning four monitors
      +----> monitor 1-3  |
      |                   |
      +----> monitor 1-4  |

      +----> monitor 2-1  |
      |                   |
GPU2  --+----> monitor 2-2  |   X-Screen 1
      |                   |   Spanning four monitors
      +----> monitor 2-3  |
      |                   |
      +----> monitor 2-4  |

```

The advantage of this setup is, that the different GPUs don't have to communicate or cross-over data between them. In OpenGL, they do not have to share contexes. The disadvantage is that one can't move a program window from GPU1 to GPU2, just the mouse pointer.

On the contrary, if you form a "macro" desktop (with a single X-Screen), spanning all 8 monitors, prepare yourself for performance bottlenecks. A nice demo is to run "glxgears" and observe what happens to the framerate when you move the glxgears window from one GPU to another. For a deeper discussion on the subject, see for example [this page](#).

Unfortunately, many Linux desktop environments (KDE for example) have deprecated their ability to handle several X-Screens: do check out [this](#) rather frustrating discussion thread / bug report on the subject.

8.2 Our approach

As you learned from the tutorials and from the PyQt testsuite, Valkka uses a dedicated thread (OpenGLThread) to pre-reserve resources from the GPU and to communicate with it.

In a multi-gpu case, one simply launches an OpenGLThread for each GPU: OpenGLThread takes as a parameter a string defining the connection to the X-server (e.g. ":0.0", ":0.1", .. ":0.n", where n is the GPU number).

It is up to the API user to send the decoded frames to the correct OpenGLThread (and GPU). A simple example, where all decoded frames are sent to all GPUs in the system can be found in

```
valkka_examples/api_level_2/qt/
test_studio_3.py
```

8.3 Configuration

We've been succesful in setting up multi-gpu systems with the following setup:

- Use identical Nvidia graphic cards
- Use the NVidia proprietary driver
- With the *nvidia-settings* applet, configure your system as follows:
 - Do **not** use Xinerama
 - Configure each graphic card as a separate X-screen
 - Use "relative" not "absolute" positioning of the screens and monitors
- Use the Xcfe desktop/window manager instead of Kwin/KDE
 - Can be installed with *sudo apt-get install xubuntu-desktop*
 - Turn off window-manager composition: *Settings Manager -> Window Manager Tweaks -> Compositor -> uncheck Enable Display Compositor*
- Use Pyside2 or PyQt5 version 5.11.2 or greater (you probably have to install with *pip3 install*)

Finally, test the configuration with the PyQt testsuite's "test_studio_3.py"

8.4 Note on \$DISPLAY

The environmental variable *DISPLAY* defines the “display” (aka X-server) and “screen” (logical screen that can span multiple monitors) with the following format

`[host]:display[.screen]`

where *[]* is something optional (see also in [here](#)).

9.1 VMS Architecture

When creating a video management system capable of recording and playing simultaneously a large number of streams, here are *some* of the problems one encounters:

- Recorded video streams are played back in sync, i.e. their timestamps are matched together
- There can be large temporal chunks of video missing from any of the streams (i.e. the video stream is not continuous)
- Video is recorded simultaneously while it is being played, to/from the same file
- Only a finite amount of frames can be stored/buffered into memory for playback
- Book-keeping of the key-frames and rewinding from these key-frames to target time instants as per user's requests

In order to solve this (surprisingly nasty) problem, we have developed several objects and thread classes. Here is an overview of them:

- `core.ValkkaFS2` book-keeping of the frames
- `core.ValkkaFSReaderThread` reads frames from a file
- `core.ValkkaFSWriterThread` writes frames to a file
- `core.FileCacherThread` caches frames into memory and passes them down the filterchain
- Both `core.ValkkaFSReaderThread` and `core.ValkkaFSWriterThread` read/manipulate the book-keeping entity (`core.ValkkaFS2`) simultaneously

`core.ValkkaFS2`, `core.ValkkaFSReaderThread` and `core.ValkkaFSWriterThread` can be used for simple dumping & reading streams to/from disk. Please see [the tutorial](#).

For more complex solution, i.e. the mentioned simultaneous reading, writing & caching, the following filterchain (a) is used:

```
(core.ValkkaFSWriterThread) --> FILE --> (core.ValkkaFSReaderThread) --> (core.
↪FileCacherThread)
      slot-to-id      id-to-slot
                                     |
                                     |
                                     (DecoderThread) <-----+
                                     +
```

Typically, when recorded frames are played, the following takes place:

- Blocks of frames are requested from `core.ValkkaFSReaderThread`. From there they flow to `core.FileCacherThread`
- When play is requested from `core.FileCacherThread` it passes frames to the decoder at play speed

To make matter simpler for the API user the filterchain in (a) is further encapsulated into an `fs.FSGroup` object.

Several `FSGroup` objects are further encapsulated into a `fs.ValkkaFSManager` object, the final hierarchical object encapsulation looking like this:

```
fs.ValkkaFSManager
  fs.FSGroup
    fs.ValkkaSingleFS
      core.ValkkaFS2
      core.ValkkaFSWriterThread
      core.ValkkaFSReaderThread
      core.FileCacherThread
  fs.FSGroup
    fs.ValkkaSingleFS
      core.ValkkaFS2
      core.ValkkaFS2
      core.ValkkaFSWriterThread
  ...
```

`ValkkaFSManager` being the “end-point”, from where a user can request synchronized playing and seeking for a number of streams. `ValkkaFSManager` would be typically connected to a GUI component for interactive playback.

Please refer to the [PyQt testsuite](#) on how to use `FSGroup` and `ValkkaFSManager`.

9.2 Filesystem

`ValkkaSingleFS` is a simple filesystem for storing streaming media data. Video frames (H264) are organized in “blocks” and written into a “dumpfile”. The dumpfile can be a regular file or a dedicated disk partition.

Dumpfile is *pre-reserved*, which makes the life easier for the underlying filesystem and avoids fragmentation (in contrast to creating a huge amount of small, timestamped video segment files).

The size of a single block (S) and the number of blocks (N) are predefined by the user. The total disk space for a recording is then $N \cdot S$ bytes.

Once the last block is written, writing is “wrapped” and resumed from block number 1. This way the oldest recorded stream is overwritten automatically.

Per each `ValkkaFS`, a directory is created with the following files:

```
directory/
  blockfile      # book-keeping of the frames
  dumpfile       # recorded H264 stream
  valkkafs.json  # metadata
```


`blockfile` is simple binary file that encapsulates a table with N (number of blocks) rows and two columns. Each column represents a millisecond timestamp:

```
mstime1      mstime2
...          ...
```

where `mstime1` indicates the first *key-frame* available in a block, while `mstime2` indicates the last frame available in that block.

`valkkafs.json` saves data about the current written block, number of blocks and the block size.

For efficient recording and playback with ValkkaFS (or with *any* VMS system), consider this:

- For efficient seeking, set your camera to emit **one key-frame per second** (or even two)
- Be aware of the bitrate of your camera and adjust the blocksize in ValkkaFS to that: ideally you'd want **1-2 key frames per block**

Consult *the tutorial* for more details.

9.3 Multiple Streams per File

You can also dump multiple streams into a single ValkkaFS. The variant for this is `valkka.fs.ValkkaMultiFS`.

This requires that all cameras have the same bitrate and key-frame interval!

The advantage of this approach is, that all frames from all your cameras are streamed continuously into the same (large) file or a dedicated block device, minimizing the wear and tear on your device if you are using a hdd.

The architecture is identical to `ValkkaSingleFS`, with a very small modification to the `blockfile` format: `mstime1` presents now the *last* key-frame among all keyframes of all the streams.

WARNING: writing multiple streams to the same file / block device is at very experimental stage and not well tested

9.4 Using an entire partition

WARNING: this makes sense only if you are using ValkkaMultiFS, i.e. streaming several cameras into a same ValkkaFS

An entire hard-drive/partition can be dedicated to ValkkaFS. In the following example, we assume that your external hard-disk appears under `/dev/sdb`

To grant access to a linux user to read and write block devices directly, use the following command:

```
sudo usermod -a -G disk username
```

After that you still need to logout and login again.

Now you can verify that block devices can be read and written as regular files. Try this command:

```
head -n 10 /dev/sdb
```

to read the first ten bytes of that external hard-drive.

ValkkaFS uses devices with **GPT partition tables**, having **Linux swap partitions**, located on **block devices**.

Why such a scheme? We'll be writing over that partition, so we just wan't to be sure it's not a normal user filesystem. :)

The next thing we need, is to create a Linux swap partition on that external (or internal) hard disk. The recommended tool for this is *gparted*.

Start *gparted* with:

```
sudo gparted /dev/sdb
```

Once in *gparted*, choose *device => create partition table*. Choose *gpt* partition table and press *apply*. Next choose *partition*, and there, choose *linux swap*.

Let's see how it worked out, so type

```
sudo fdisk -l
```

You should get something like this:

Device	Start	End	Sectors	Size	Type
/dev/sdb1	2048	976773134	976771087	465,8G	Linux swap

To get the exact size in bytes, type:

```
blockdev --getsize64 /dev/sdb1
```

So, in this case we'd be dedicating an external USB drive of 465 GB for recording streaming video.

To identify disks, Valkka uses uuid partition identification. The uuid can be found with

```
blkid /dev/sdb1
```

Suppose you get:

```
/dev/sdb1: UUID="db572185-2ac1-4ef5-b8af-c2763e639a67" TYPE="swap" PARTUUID="37c591e3-
↪b33b-4548-a1eb-81add9da8a58"
```

Then "37c591e3-b33b-4548-a1eb-81add9da8a58" is what you are looking for.

In this example case, you would instantiate the ValkkaFS like this:

```
valkkafs = ValkkaMultiFS.newFromDirectory(
    dirname="/home/sampsa/tmp/testvalkkafs",
    partition_uuid="37c591e3-b33b-4548-a1eb-81add9da8a58",
    blocksize=YOUR_BLOCKSIZE,
    device_size=1024*1024*1024*465) # 465 GB
```

CHAPTER 10

Cloud Streaming

Here we describe how to stream live, low latency video from your IP cameras to cloud and how to visualize the live video in a web browser.

A good video container format for this is “fragmented MP4” (frag-MP4 aka FMP4). This is basically the same format you have in those .mp4 files of yours, but it is fragmented into smaller chunks (aka “boxes”), so that it can be sent, chunk-by-chunk, for low-latency live video streaming over your LAN or WAN.

LibValkka is able produce the fragmented MP4 “boxes” for you, while you can read them one-by-one, in your python code. How to do this, please refer to the [tutorial](#).

After obtaining the MP4 boxes, just use your imagination. You can send them over the internet using websockets, [gRRP](#), or any protocol of your choice. You can also dump them into an .mp4 file, and that file is understood by all media clients (just remember to cache and write the ftyp and moov packets in the beginning of the file). For creating a pipelines like that, please take a look [here](#).

To play live video in your browser, use [Media Source Extensions \(MSEs\)](#). Receive the MP4 boxes through a websocket and push them into the MSE API to achieve low-latency live video.

This is a true cross-platform solution, that works in Linux, Windows and desktop Mac iOS.

As of September 2020, iPhone iOS is still lacking the MSEs, so that is the only device where this approach doesn’t work. In that case, you should use dynamically generated [HLS playlists](#), while in this approach it is again convenient to use frag-MP4.

For more information about the frag-MP4 structure, see [this stack overflow post](#) and [this github repository](#).

MP4 is described extensively in [this document](#).

Remember that not all codec + container format combinations are supported by the major browser. Most typical combination for video is H264 + MP4. For a list of supported codec + container formats, please see [this link](#). Note that H265 support is still lacking behind.

For some more references on the subject, see in [here](#) and [here](#).

CHAPTER 11

OnVif & Discovery

(Your short primer to SOAP and OnVif)

11.1 Intro

OnVif is a remote-control protocol for manipulating IP cameras, developed by [Axis](#).

You can use it to PTZ (pan-tilt-zoom) the camera, for setting camera's credentials and resolution, and for almost anything else you can imagine.

OnVif is based on [SOAP](#), i.e. on sending rather complex XML messages between your client computer and the IP camera. The messages (remote protocol calls), the responses and the parameters, are defined by **WSDL files**, which (when visualized nicely) look like [this](#).

11.2 Python OnVif

In Python, the main bottleneck was in finding a decent open source SOAP library that would do the trick. Recently, things have got better with the arrival of [Zeep](#).

Before Zeep existed, people used [Suds](#), which has become a bit obsolete by now. A library called [python-onvif](#) was based on Suds.

That python-onvif module has since then been forked and modded [to work with Zeep](#).

However, we don't need any of that, since it's a better idea to

use Zeep directly

with minimum extra code bloat on top of it.

So, use Zeep as your SOAP client, give it the WSDL file and that's about it.

11.3 OnVif with Zeep

Rather than giving you an obscure OnVif client implementation, you'll learn to do this by yourself using Zeep. Let's begin with:

```
pip3 install zeep
```

You also need this table to get started:

WSDL Declaration	Camera http sub address	WsdL file	Subclass
http://www.onvif.org/ver10/device/wsdL	device_service	de- vicemgmt.wsdL	DeviceManage- ment
http://www.onvif.org/ver10/device/wsdL	Media	media.wsdL	Media
http://www.onvif.org/ver10/events/wsdL	Events	events.wsdL	Events
http://www.onvif.org/ver20/ptz/wsdL	PTZ	ptz.wsdL	PTZ
http://www.onvif.org/ver20/imaging/wsdL	Imaging	imaging.wsdL	Imaging
http://www.onvif.org/ver10/deviceIO/wsdL	DeviceIO	deviceio.wsdL	DeviceIO
http://www.onvif.org/ver20/analytics/wsdL	Analytics	analytics.wsdL	Analytics

Here is an example on how to create your own class for an OnVif device service, based on the class OnVif:

```
from valkka.onvif import OnVif, getWSDLPath

# (1) create your own class:

class DeviceManagement(OnVif):
    namespace = "http://www.onvif.org/ver10/device/wsdL"
    wsdl_file = getWSDLPath("devicemgmt.wsdL")
    sub_xaddr = "device_service"
    port      = "DeviceBinding"

# (2) instantiate your class:

device_service = DeviceManagement(
    ip      = "192.168.0.24",
    port    = 80,
    user    = "admin",
    password = "12345"
)
```

(the implementation of the base class OnVif is only a few lines long)

The things you need for (1) subclassing an OnVif service are:

- The remote control protocol is declared / visualized in the link at the first column. Go to <http://www.onvif.org/ver10/device/wsdL> to see the detailed specifications.
- In that specification, we see that the WSDL “port” is DeviceBinding.
- Each SOAP remote control protocol comes with a certain namespace. This is the same as that address in the first column, so we set namespace to <http://www.onvif.org/ver10/device/wsdL>.

- We use a local modified version of the wsdl file. This can be found in the third column, i.e. set `wsdl_file` to `devicemgmt.wsdl` (these files come included in `libValkka`).
- Camera's local http subaddress `sub_xaddr` is `device_service` (the second column of the table)

When you (2) instantiate the class into the `device_service` object, you just give the camera's local IP address and credentials

11.4 Service classes

You can create your own OnVif subclass as described above.

However, we have done some of the work for you. Take a look at the column "Subclass" in the table, and you'll find them:

```
from valkka.onvif import Media

media_service = Media(
    ip          = "192.168.0.24",
    port        = 80,
    user        = "admin",
    password    = "12345"
)
```

11.5 Example call

Let's try a remote protocol call.

If you look at that specification in <http://www.onvif.org/ver10/device/wsdl>, there is a remote protocol call name `GetCapabilities`. Let's call it:

```
cap = device_service.ws_client.GetCapabilities()
print(cap)
```

We can also pass a variable to that `GetCapabilities` call.

Variables are nested objects, that must be constructed separately. Like this:

```
factory = device_service.zeeq_client.type_factory("http://www.onvif.org/ver10/schema")
category = factory.CapabilityCategory("Device")
cap = device_service.ws_client.GetCapabilities(category)
print(cap)
```

The namespace <http://www.onvif.org/ver10/schema> declares all basic variables used by the wsdl files. We also provide a short-cut command for that:

```
category = device_service.getVariable("Device")
```

That's about it. Now you are able to remote control your camera.

One extra bonus: to open the specifications directly with Firefox, try this

```
device_service.openSpecs()
```

11.6 Notes

When specifying durations with Zeep, you must use the `isodate` module, like this:

```
import isodate
timeout = isodate.Duration(seconds = 2)
```

Now that variable `timeout` can be used with `OnVif` calls

11.7 Discovery

In `libValkka`, cameras can be discovered like this:

```
from valkka.discovery import runWSDiscovery, runARPSan
ips = runWSDiscovery()
ips2 = runARPSan(exclude_list = ips) # run this if you want to use arp-scan
```

If you want `arp-scan` to work, you must permit normal users to run the executable, with:

```
sudo apt-get install arp-scan
sudo chmod u+s /usr/sbin/arp-scan
```


12.1 Pitfalls

Valkka has been designed for massive video streaming. If your linux box, running a Valkka-based program, starts to choke up and you get frame jittering, stuttering / video freezes, and typically, this:

```
OpenGLThread: handleFifo: DISCARDING late frame ...
```

You should consider the following issues:

0. Are you using correct Valkka version?

Use the latest version. When running the PyQt testsuite, remember to run *quicktest.py* to see if your installation is consistent.

1. Are you using sub-standard cameras?

Nowadays the image quality is impressive in all stock IP cameras, however, the rtsp server and/or timestamping of the cameras can be buggy (there might be problems when maintaining several connections to the same camera, or when re-connecting several times to the same camera).

Before blaming us, *generate the same situation with a reference program*, say with ffplay, and see if it works or not. The *PyQt testsuite* offers nice tools for benchmarking against ffplay and vlc.

2. Is your PC powerful enough to decode simultaneously 4+ full-hd videos?

Test against a reference program (ffplay). Launch KSysGuard to monitor your processor usage. Read also *this*.

3. Have you told Valkka to reserve enough bitmap frames on the GPU? Is your buffering time too large?

The issue of pre-reserved frames and buffering time has been discussed *here* and in the *PyQt testsuite section*.

4. Disable OpenGL rendering synchronization to vertical refresh (“vsync”).

On MESA based X.org drivers (intel, nouveau, etc.), this can be achieved from command line with “export vblank_mode=0”. With nvidia proprietary drivers, use the *nvidia-settings* program.

Test if vsync is disabled with the “glxgears” command. It should report 1000+ frames per second with vsync disabled.

5. Disable OpenGL composition.

In a KDE based system, go to *System Settings => Display and Monitor => Compositor* and uncheck “Enable compositor on startup”. After that, you still have to restart your X-server (i.e. do logout and login). CTRL-ALT-F12 might also work. In Xfce based desktop, do *Settings Manager -> Window Manager Tweaks -> Compositor -> uncheck Enable Display Compositor*.

Alternatively, you can use this command:

```
dbus org.kde.KWin /Compositor suspend
```

6. Is your IP camera’s time set correctly?

Valkka tries hard to correct the timestamps of arriving frames, but if the timestamps are “almost” right (i.e. off by a second or so), there is no way to know if the frames are stamped incorrectly or if they really arrive late..!

So, either set your IP camera’s clock really off (say, 5+ mins off) or exactly to the correct time. In the latter case, you might want to sync both your IP camera and PC to the same NTP server.

7. Are you really using a gigabit nic?

How’s your network interface hardware? Using an old usb dongle? Check your nic’s capacity with this command:

```
cat /sys/class/net/<device_name>/speed
```

It should say at least “100”, preferably “1000”

12.2 Bottlenecks

Once you ramp up the number of streams, you might start to experience some *real* performance issues. Some typical problems include:

8. Your LAN and/or the LiveThread process sending frames in bursts

- Frames arrive late, and all in once. You should increase the buffering time OpenGLThread. See [here](#).
- This is very common problem when streaming over Wifi
- If you observe broken frames, most likely your network interface is not keeping up. What is the bandwidth of your network and nic ? (as we just discussed. See also “System tuning” below)

9. The AVThread(s) performing the decoding and uploading to GPU are taking too long

- This is to be expected if all your CPU(s) are screaming 100%, so check that.

10. OpenGLThread is stalling

If you have tried everything suggested on this page so far, but you’re still getting

```
OpenGLThread: handleFifo: DISCARDING late frame ...
```

Then the problem is in the last part of the pipeline: the OpenGLThread.

It performs image interpolation and various other OpenGL calls, that, depending on your driver’s OpenGL implementation, might be efficient or not. Generally, NVidia seems to perform better than Intel.

If you compile libValkka from source, there are many available debug options that can be enabled in *run_cmake.bash*. A particularly useful ones are *profile_timing* and *opengl_timing*. Enabling these debug switch allows you to trace the culprit for frame dropping to slow network, slow decoding or the OpenGL part.

Some common fixes (that are frequently used in commercial video surveillance applications) include:

- Configure your cameras to a lower frame rate (say, 10 fps): unfortunately this sucks.
- Tell AVThread to decode only keyframes: choppy video.
- The mainstream/substream scheme:
 - If you have, say, 20 small-sized video streams in your grid, it is an exaggeration to decode full-HD video for each one of the streams.
 - For small windows, you should switch to using a substream provided by your IP camera. A resolution of, say, half of HD-ready might be enough.
 - Decode and present the full-HD mainstream only when there are video windows that are large enough

Valkka provides (or will provide) API methods and FrameFilter(s) to implement each one of these strategies.

11. Packet drop

Maybe might have saturated your NIC (see also above (7))? Check in your camera's web-interface/config the bandwidth it is using. Typical values are (which you can adjust): 2048 kbps (~2 mbps), 4096kbps (~4 mbps), etc. so do the math.

You can try to use TCP streaming instead of the default UDP streaming - see FAQ (a) below (valkka-live tip: you can choose TCP streaming in the camera configuration)

12.3 System tuning

Adding the following lines into */etc/sysctl.conf*

```
vm.swappiness = 1
net.core.wmem_max=2097152
net.core.rmem_max=2097152
```

And running

```
sudo sysctl -p
```

Turns off swap and sets maximum allowed read/write socket sizes to 2 MB.

Receiving socket size can be adjusted for each live connection with the associated *LiveConnectionContext* (see the tutorial).

12.4 FAQ

(a) *How can I stream over internet, instead of just LAN?*

By default, stream is transported through UDP sockets. When streaming over internet, most of the ports are closed due to firewalls, etc., so you have to stream through the same TCP port that is used for the RTSP negotiation (typically port 554).

Modify your *LiveConnectionContext* like this:

```
ctx = LiveConnectionContext(LiveConnectionType_rtsp, "rtsp://admin:nordic12345@192.168.
↪1.41", 1, live_out_filter)
ctx.request_tcp = True
```

(for more information, see [here](#))

(b) *Could not load the Qt platform plugin "xcb"*

If you get this error:

```
qt.qpa.plugin: Could not load the Qt platform plugin "xcb" in "" even though it was
found.
```

Then you are *not* running valkka-live directly in a desktop, but from remote etc. connection (or in docker, etc. “head-less” environment).

It has really nothing to do with libValkka or valkka-live. In fact, *none* of your Qt and KDE-based desktop programs would work at all. Check with this command:

```
echo $XDG_SESSION_TYPE
```

and make sure that it reports the value *x11*.

If the error persists, your desktop environment might have missing or broken Qt/KDE dependencies. Install the whole KDE and Qt stack with:

```
sudo apt-get install kate
```

(this pulls a minimal KDE + Qt installation as dependencies of the Kate editor)

If this error *still* persists and is reported by python’s cv2 module, you have a broken cv2 version, so uninstall cv2 with:

```
pip3 uninstall opencv-python
sudo pip3 uninstall opencv-python # just in case!
```

And install your linux distro’s default opencv instead with:

```
sudo apt-get install python3-opencv
```

CHAPTER 13

Debugging

segfaults, memleaks, etc.

LibValkka is rigorously “valgrinded” to remove any memory leaks at the cpp level. However, combining cpp and python (with swig) and throwing into the mix multithreading, multiprocessing and sharing memory between processes, can (and will) give surprises.

1. Check that you are not pulling frames from the same shared-memory channel using more than one client

2. Run Python + libValkka using gdb

First, install python3 debugging symbols:

```
sudo apt-get install gdb python3-dbg
```

Then, create a custom build of libValkka with debug symbols enabled.

Finally, run your application’s entry point with:

```
gdb --args python3 python_program.py  
run
```

See backtrace with

```
bt
```

If the trace point into `Objects/obmalloc.c`, then the cpp extensions have messed up python object reference counting. See also [here](#)

3. Clear semaphores and shared memory every now and then by removing these files

```
/dev/shm/*valkka*
```

4. Follow python process memory consumption

Use the [setproctitle python module](#) to name your python multiprocesses. This way you can find them easily using standard linux monitoring tools, such as `htop` and `smem`.

Setting the name of the process should, of course, happen after the multiprocessing fork.

Install smem and htop:

```
sudo apt-get install smem htop
```

After that, run for example the script `memwatch.bash` in the `aux/` directory. Or just launch `htop`. In `htop`, remember to go to setup => display options and enable “Hide userland process threads” to make the output more readable.

Valkka-live, for example, names all multiprocesses adequately, so you can easily see if a process is leaking memory.

5. Prefer PyQt5 over PySide2

You have the option of using PyQt5 instead of PySide2. The former is significantly more stable and handles the tricky cpp Qt vs. Python reference counting correctly. Especially if you get that thing mention in (2), consider switching to PyQt5.

CHAPTER 14

Repository Index

Repository	License	About	Namespaces
valkka-core	LGPL	Core cpp code, api level 1 & 2 python code	valkka.core valkka.api2 valkka.discovery valkka.fs valkka.multiprocess valkka.onvif
valkka-examples	MIT	Python tests & demo programs. Qt demo programs. See here	
valkka-multiprocess	MIT	valkka.multiprocess is “mirrored” here from valkka.core, if you’re just interested in the multiprocessing code	valkka.multiprocess
valkka-live	AGPL	A demo video surveillance program using libValkka	valkka.live valkka.mvision
92			Chapter 14. Repository Index
valkka-cpp-examples	MIT	Create your own Valkka cpp extension module	

If you need an LGPL license for **valkka-live**, please contact us.

CHAPTER 15

Licence & Copyright

(C) Copyright 2017 - 2020 Sampsa Riikonen and Valkka Security Ltd.

The core component libValkka is licensed under the [LGPL](#) license

CHAPTER 16

Authors

[Sampsariikonen](mailto:sampsariikonen_at_iki.fi) (sampsariikonen_at_iki.fi)

Tips, instructions, etc. for compiling libValkka, Qt & Yolo on out-of-the-ordinary hardware

17.1 General

When compiling and generating yourself python binary packages these commands come handy:

```
pip3 wheel --wheel-dir=YOUR_DIRECTORY -r requirements.txt
pip3 install --no-index --find-links=YOUR_DIRECTORY -r requirements.txt
```

The first one downloads binary whl packages, defined in requirements.txt, from pypi.org to directory YOUR_DIRECTORY.

Next, put your manually compiled packages into YOUR_DIRECTORY

After that, launch the second command: it installs packages, defined in requirements.txt from YOUR_DIRECTORY.

References:

- https://pip.readthedocs.io/en/stable/user_guide/#installing-from-wheels

17.2 OpenCV & OpenCV contrib

Normally you might install OpenCV & its python bindings just with

```
pip3 install --user --upgrade opencv-python opencv-contrib-python
```

The “contrib” module includes the “non-free” part (with patented algorithms etc.) of OpenCV library. However, most of the time this won’t work either, since the libraries have been compiled with non-free algorithms disabled.

There’s no other way here than to compile this by yourself. You need to install (at least):

```
sudo apt-get install build-essential cmake git libgtk2.0-dev pkg-config libavcodec-
↳dev libavformat-dev libswscale-dev libv4l-dev python-dev python-numpy libtbb2_
↳libtbb-dev libjpeg-dev libpng-dev libtiff-dev libjasper-dev libdc1394-22-dev_
↳libxvidcore-dev libx264-dev
```

Check out [opencv](#) & [opencv-contrib](#) from github. Add build directory and therein a file named run_cmake.bash. Your directory structure should look like this:

```
opencv/
  build/
    run_cmake.bash
opencv-contrib/
```

run_cmake.bash looks like this:

```
#!/bin/bash
cmake -D WITH_CUDA=OFF \
      -D OPENCV_EXTRA_MODULES_PATH=../../opencv_contrib/modules \
      -D OPENCV_ENABLE_NONFREE=ON \
      -D WITH_GSTREAMER=ON \
      -D WITH_LIBV4L=ON \
      -D BUILD_opencv_python2=OFF \
      -D BUILD_opencv_python3=ON \
      -D CPACK_BINARY_DEB=ON \
      -D BUILD_TESTS=OFF \
      -D BUILD_PERF_TESTS=OFF \
      -D BUILD_EXAMPLES=OFF \
      -D CMAKE_BUILD_TYPE=RELEASE \
      -D CMAKE_INSTALL_PREFIX=/usr/local \
      ..
```

While at build directory, do

```
./run_cmake.bash
make -j 4
```

There's a bug in the opencv build system, so we have to employ a [trick](#) before building the debian packages: comment out this line

```
# set (CPACK_DEBIAN_PACKAGE_SHLIBDEPS "TRUE")
```

from “CPackConfig.cmake”. After that you should be able to run

```
make package
```

Before installing all deb packages from the directory with

```
sudo dpkg -i *.deb
```

remember to remove any pip-installed opencv and opencv contrib modules

17.3 Jetson Nano

17.3.1 Qt Python Bindings

There are two flavors of Qt Python bindings, namely, PyQt and PySide2. Here we deal with the latter. If you have information on PyQt on JetsonNano, please do send us an email.

PySide2 Qt python bindings are not available for all architectures simply from pypi using `pip3 install` command. This is the case for Jetson Nano. So we have to compile ourselves.

Install clang, build tools, Qt module clang header files, etc:

```
sudo apt-get install git build-essential cmake libclang-dev qt5-default qtscript5-dev
↳ libssl-dev qttools5-dev qttools5-dev-tools qtmultimedia5-dev libqt5svg5-dev
↳ libqt5webkit5-dev libstdl2-dev libasound2 libxmu-dev libxi-dev freeglut3-dev
↳ libasound2-dev libjack-jackd2-dev libxrandr-dev libqt5xmlpatterns5-dev
↳ libqt5xmlpatterns5 libqt5xmlpatterns5-dev qtdeclarative5-private-dev qtbase5-
↳ private-dev qttools5-private-dev qtwebengine5-private-dev
```

Git clone PySide2 python bindings source code:

```
git clone git://code.qt.io/pyside/pyside-setup.git
cd pyside_setup
```

PySide2 python bindings must be compatible with your system's Qt version. Find out the version with:

```
qmake --version
```

For ubuntu 18 LTS for example, the version is 5.9.5, so:

```
git checkout 5.9
```

Next, edit this file:

```
sources/pyside2/PySide2/QtGui/CMakeLists.txt
```

Comment out (using #), these two lines:

```
${QtGui_GEN_DIR}/qopengltimeonitor_wrapper.cpp
${QtGui_GEN_DIR}/qopengltimerquery_wrapper.cpp
```

Finally, compile the bindings with:

```
python3 setup.py build
```

That might take up to 8 hrs, so see a movie using your favorite streaming service. :)

That compiles python bindings for all Qt features, so it could be a good idea to comment out more wrappers in that `CMakeLists.txt`

After that, you can create a distributable package by:

```
python3 setup.py --only-package bdist_wheel
```

The package appears in directory `dist/` and is installable with `pip3 install --user packagename.whl`

References:

- <https://github.com/PySide/pyside2/wiki/Dependencies>

- https://wiki.qt.io/Qt_for_Python
- Pyside's `setup.py` : read the comments within the first lines
- <https://bugreports.qt.io/browse/PYSIDE-568>
- search