oZone Documentation

Release 0.1

Pliable Pixels

May 08, 2017

Contents

1 Architecture **3**
1.1 Introduction 3
1.2 Key Architecture Principles 3
1.3 Application Lifeycle 5

2 Installation **11**
2.1 Installation of oZone libraries and examples 11
2.2 Dlib optimizations 12
2.3 Building Documentation 12
2.4 Using oZone libraries in your own app 13

3 Examples **15**

4 API **17**

5 FAQ **19**

6 Contributing **21**

oZone is a powerful framework to develop innovative solutions around video surveillance. At its core, it offers powerful components that implement important functions (such as reading video feeds from multiple sources, performing motion/face/people detection, event recording and more), allowing easy daisy-chaining of components with each other. It also allows developers to create their own components to implement innovative solutions on top of the provided base primitives.

oZone is light enough to be embedded inside a camera and scalable enough to be used as a base for a cloud based NVR.

CHAPTER 1

Architecture

Introduction

oZone is a powerful, yet simple to use framework for developers looking to create their own NVR system.

The next few sections will introduce key architectural principles of oZone.

Key Architecture Principles

Frames

oZone centers around the concept of passing Frames between Components. A 'Frame' is really just an abstract concept. Common types of frames may be:

- A **video frame** - a special type of frame that contains one frame of a video stream
- An **audio frame** - a special type of frame that contains one frame of audio from a video/audio stream
- A **data frame** - may contain any kind of data
- A **notification frame** - this is really a type of data frame, but its important to bring this out as it serves a specific purpose - an ability of one component to notify another component (if it is interested) of an event of interest.

Key Concepts Summary

- Each Component serves a specific purpose
- Components can register with each other
- Components pass "Frames" around to registered components
- Each Component runs in its own thread

Components

Now that we understand the core data structure of inter-component-communication, lets understand what "Components" are.

Components are cohesive objects that serve specific functions. Specific to the purpose of oZone, examples of components are:

- A component that can read audio/video frames from a camera
- A component that can perform motion detection from the 'frames' received from the component above (see how chaining works?)
- A component that can record motion frames to disk when a defined threshold of movement is detected? (Example, only store frames that involve people moving around, not your cats or dogs. Again, see how we keep chaining components?)

Types of Components

Given that components are a critical part of oZone, lets talk about the types of components you can use.

Components Types Summary

- A Provider generates frames (typically audio/video)
- A Consumer consumes these frames for some purpose
- A Process both consumes frames from upstream components and generate frames for downstream components

- Providers
- Consumers
- Processors
- Listeners
- Controllers

A **Provider** is a type of component that "generates" frames. A perfect example of a Provider is AVInput, which is able to connect to a source like `/dev/video0` for a local webcam, `/path/to/file/fulldayrecording.mp4` for a recorded video, or, `rtsp://myliveurl` for a RTSP camera. It can connect to *any* such input source and produces audio and/or video frame, completely abstracting the nature of the source for other components down the chain.

A **Consumer** is a type of component that "consumes" frames. Unlike a Provider, it doesn't generate any frames, so there is no point "registering" for frames with a consumer. A good example of a Consumer is EventRecorder, that writes motion events to disk.

Note: It's not totally true that consumers don't generate frames. An exception is that it can generate notification frames - example, when you want to notify a downstream component that a new event is about to be written to disk (maybe you want to update your UI)

A **Processor** is really a hybrid between a Provider and a Consumer. A Processor accepts frames and generates frames. Can you think of an example for this? MotionDetector is a good example! It typically 'registers' with a Provider, analyzes the frames and outputs then overlayed with motion information for further downstream processing. Or

take for example, the uber awesome MatrixVideo processor which accepts frames from N components and creates a configurable NxM matrix of frames stitched together and outputs it as a single frame for downstream display!

Finally, **Listeners** and **Controllers** are somewhat specialized in its purpose. A Listener listens for data. A Listener connects to a Controller that controls what needs to be done when the listener receives data. For example, HttpController is a controller that can be attached to a listener like so:

```
HttpController httpController( "watch", 9292 );
httpController.addStream("watchcam1",cam1);
```

This bit of code would allow for browsers to connect to port 9292 and render the output of a camera feed as MJPEG, automagically.

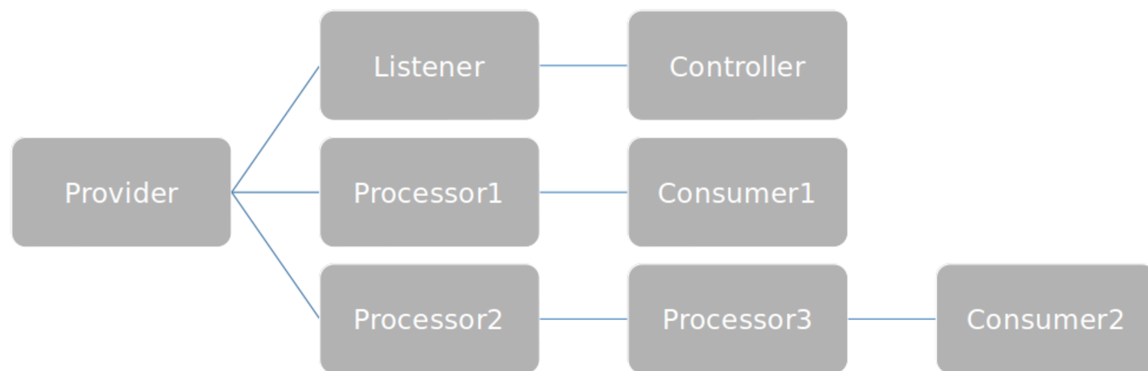

Fig. 1.1: An abstract view of application linking components

Application Lifeycle

This chapter will describe the application lifecycle of the oZone framework, from init->run->termination.

Summary

At a conceptual level, initializing the oZone framework involves:

- Initializing the debug/logging subsystem
- Initializing various audio/video handlers to manage streams/decoding/encoding
- Instantiating an `Application` object to manage application lifecycle
- Instantiating various components per your need
- Registering various components with each other to establish a workflow
- Adding all components to the `Application` object so they can be started
- Invoke the `Application` object's `run()` method

The `Application` object can be thought of as the master object that keeps track of all the components. When components are instantiated, they register with the `Application` object by invoking its `addThread()` method. This essentially adds the object to the `Application` queue.

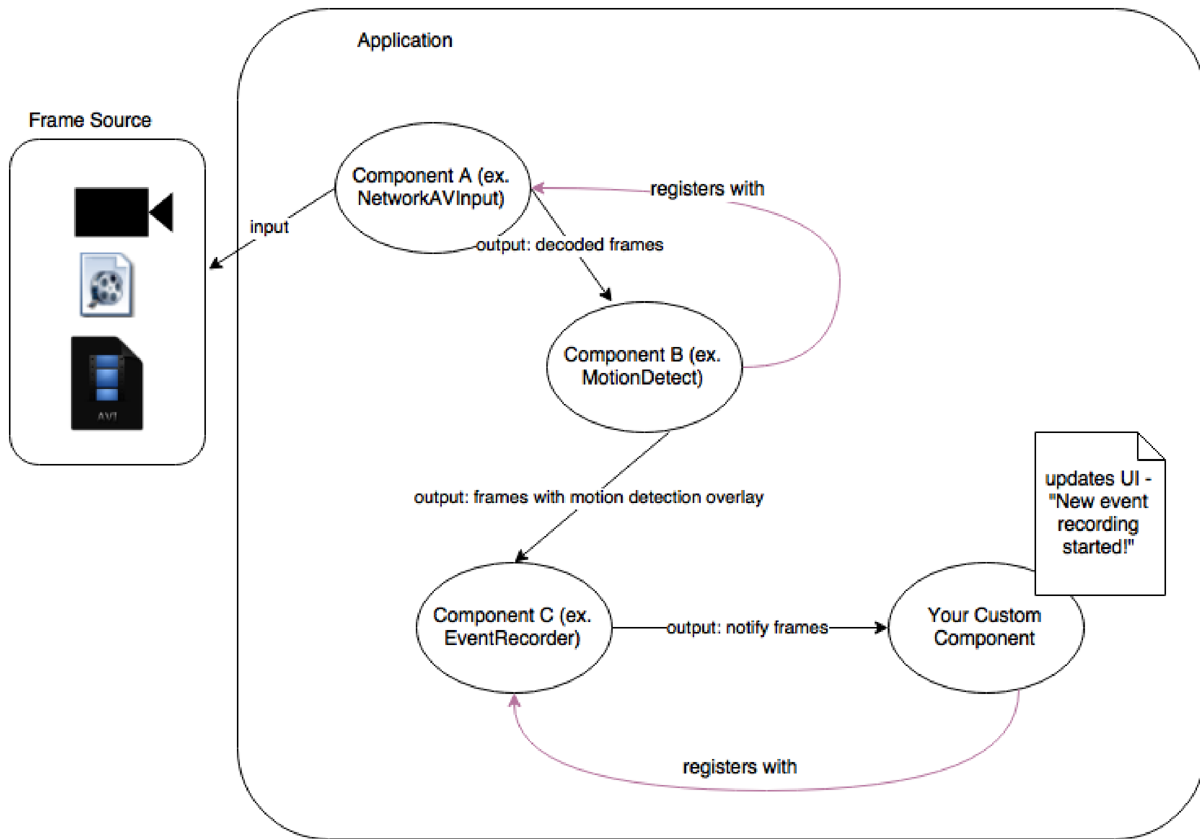

Fig. 1.2: An application specific instance of chaining components

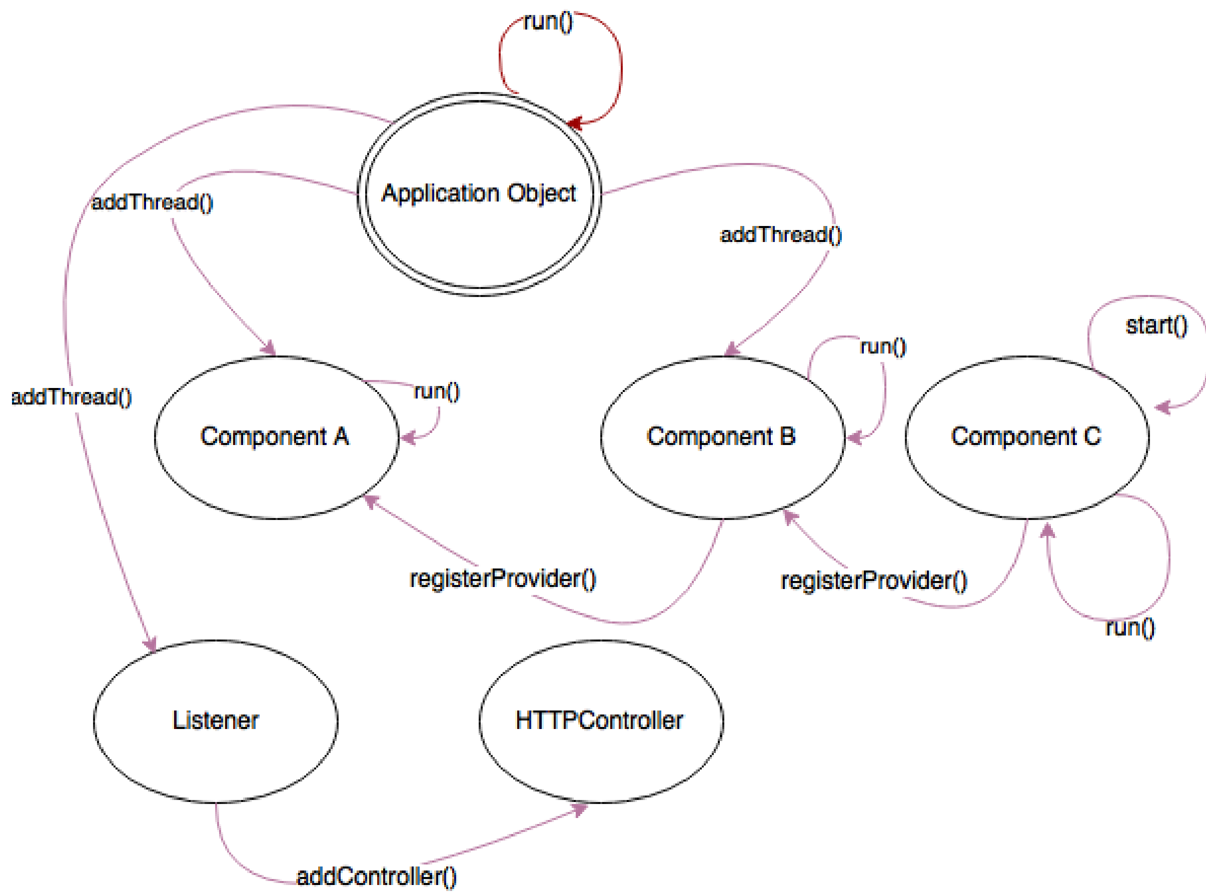

Fig. 1.3: A high level view of the application lifecycle

When you invoke the `Application run()` method, it iterates through the list of components and invokes the `start()` methof of each object, which essentially launches a thread for each component. Following this, it invokes the `run()` method of each component, which is what is expected to be the entry point of each component's functionality.

Note that the `Application` object is just a convenience. You can easily invoke the `start()` method of each component yourelf. The latter approach is typically useful when you dynamically create new components and remove them after you start the application.

The `Listerner` and `HTTPController` components are used when you need to convert the frames of a component into a browser viewable version. `Listerner` along with `HTTPController` are an easy way to create MJPEG images which you can display on the browser with a simple `<img src="">` tag.

Summary of oZone framework lifecycle

- Initialize debug and AV subsystems
- Create components
- connect them via `registerProvider()`
- add each component to Application object via `addThread()`
- To render frames to browsers, instantiate Listerner and HttpController object
- Invoke `app.run()`

More details

Application Life Cycle Manager

The `Application` object is really very simple. It's implementation simply keeps a list of components. Components are added to the `Application` object by invoking its *addThread()* method which simply pushes a pointer to the component into its internal queue.

Once you connect all the components to the `Application` object, and you invoke the `run()` method, all it really does is iterate through the list and invoke the *run()* method of each component *in a new thread* (yes, each component **must** have a run method) and then waits for them to terminate.

Component Chaining

We also talked about how components could chain to each other to create workflows. Chaining is achieved by invoking a `registerProvider` method of a component.

Here is a live example:

```
1  Application app;
2
3  AVInput input( "input", "http://kxhcm10/nphMotionJpeg?Resolution=640x480&
   ↪Quality=Standard");
4  app.addThread( &input );
5
6  FaceDetector detector( "detector" );
7  detector.registerProvider( input );
8  app.addThread( &detector );
9
```

```
10  VideoParms videoParms( 320, 240 );
11  AudioParms audioParms;
12  MovieFileOutput movie( detector.cname(), "/transfer", "mp4", 60, videoParms,␣
    ↪audioParms );
13  output2.registerProvider( detector );
14  app.addThread( &movie );
15
16  app.run();
```

Explanation:

- line 1 - create Application object (app)
- line 3-4: create a provider component (input) that reads video feeds from a URL and add it to the application object
- line 6 - create a FaceDetector processor component (detector)
- line 7 - register the provider component of line 3 (input) to be the frame provider for this new face detector component
- line 8 - also add this facedetector component to the master Application object (app)
- line 10-12: instantiate a consumer component that will create video files (movie)
- line 13: register the provider component of movie to be the facedetector component
- line 14: add this output component to the master Application object (app)

Note: What just happened?

- input will read frames from that URL
- detector will attempt to detect faces in the frames input provides above
- movie will attempt to create video files which will essentially be the same frames generated by input, but overlayed with face detection markers detector creates

---

- line 16: launch all the threads and have fun!

CHAPTER 2

Installation

The examples below are for a typical Ubuntu/Debian system.

Installation of oZone libraries and examples

oZone is a portable solution with a very easy installation process. This example assumes you want all the ozone libraries (including dependencies) to be installed at ~/ozoneroot. This is a great way to isolate your install from other libraries you may already have installed.

There are two parts, a one time process and then building just the ozone library repeatedly (if you are making changes to the examples or core code)

One time setup:

```
# -------------------install dependencies------------------------

sudo apt-get update
sudo apt-get install git cmake nasm libjpeg-dev libssl-dev
sudo apt-get install libatlas-base-dev libfontconfig1-dev lib4l-dev

# ---------------------clone codebase----------------------------

git clone https://github.com/ozonesecurity/ozonebase
cd ozonebase
git submodule update --init --recursive

# --------------- build & install --------------------------------
export INSTALLDIR=~/ozoneroot/ # change this to whatever you want
./ozone-build.sh
```

Note: if you face compilation issues with ffmpeg not finding fontconfig or other package files, you need to search for libv4l2.pc, fontconfig.pc files and copy then to the lib/pkgconfig directory of your INSTALL_DIR path

Once the one time setup is done, you don't need to keep doing it (building external dependencies take a long time) For subsequent changes, you can keep doing these steps:

```
# ---- Optional: For ad-hoc in-source re-building----------------
cd server
cmake -DCMAKE_INSTALL_PREFIX=$INSTALLDIR -DOZ_EXAMPLES=ON -DCMAKE_INCLUDE_PATH=
↪$INSTALLDIR/include
make
make install

# ----- Optional: build nvrcli - a starter NVR example ----------
cd server
edit src/examples/CMakeLists.txt and uncomment lines 14 and 27 (add_executable for␣
↪nvrcli and target_link_libraries for nvrcli

make
```

That's all!

Dlib optimizations

If your processor supports AVX instructions, `(cat /proc/cpuinfo | grep avx)` then add `-mavx` in `server/CMakeLists.txt` to `CMAKE-C_FLAGS_RELEASE` and `CMAKE_CXX_FLAGS_RELEASE` and rebuild. Note, please check before you add it, otherwise your code may core dump.

Building Documentation

oZone documentation has two parts:

- The API document that uses Doxygen
- The User Guide which is developed using Sphinx

API docs

To build the APIs all you need is `Doxygen` and simply run `doxygen` inside `ozonebase/server`. This will generate HTML documents. oZone uses `dot` to genarate API class and relationship graphs, so you should also install dot, which is typically part of the `graphviz` package.

User docs

You need sphinx and dependencies for generating your own user guide. The user guide source files are located in `ozonebase/docs/server/guide`

```
# Install dependencies
sudo apt-get install python-sphinx
sudo apt-get install python-pip
pip install sphinx_rtd_theme
```

And then all you need to do is `make html` inside `ozonebase/docs/server/guide` and it generates beautiful documents inside the `build` directory.

Using oZone libraries in your own app

Take a look at nvrcli's Makefile here and modify it for your needs.

CHAPTER 3

Examples

CHAPTER 4

API

The APIs are automatically generated from the source code and generated via Doxygen. Make sure you have read the *Architecture* before you dive into the API.

The APIs are located >HERE< and will be frequently re-generated as we add more functionality. So please feel free to revisit every once in a while.

CHAPTER 5

FAQ

Work in progress.

You have questions? We have answers! Time is what is missing for now.

CHAPTER 6

Contributing

Contributions to oZonebase does not grant any rights to the contributor for any renumeration. Any contributions made by 3rd parties will automatically be dual licensed as follows:

License 1: Free for personal use, under GPLv3

License 2: Non GPL, for commercial use. Please send us an email for licensing for commercial terms. Ozone is intended for developers and OEMs/ISVs building their own security solutions

Architecture Understanding the oZone architecture is an important part of developing your own NVR app.

Installation How to download and install oZone

Examples Walks you through creating a live example using code snippets with annotation.

API Interface document for oZone

FAQ Frequently Asked Questions

Contributing How to contribute to oZone.