University of Ghent

IBCN

SIRANNON 1.0.0: MODULAR MULTIMEDIA STREAMING

Alexis Rombaut

December 21, 2011

Contents

List of Figures

Chapter 1

Introduction

Sirannon is a flexible and modular media server, client and proxy. It distinguishes itself by providing a modularity that manifests itself in how the user controls and configures the streamer. Each configuration describes graph of components, each handling basic video operations such as reading frames, packetizing frames and transmitting packets. The streamer handles both video and audio, something left out in many experimental streamers. Finally, the program supports variety of protocols such as RTP, RTSP, RTMP and HTTP. Jump to chapter 4 if you immediately want to start installing the sirannon. The latest stable version is always available on `http://sirannon.atlantis.ugent.be`. The latest addition is a brand new GUI featuring graph drawing, XML editing and console execution for multiple configurations using tabs.

Chapter 2

Fact Sheet

2.1 Supported Codecs

- MPEG1 Video & Audio
- MPEG2 Video & Audio
- MPEG4 Video & Audio
- H264 AVC & SVC
- VP6, VP8/WEBM
- Vorbis
- AC3
- AMR-NB, AMR-WB

2.2 Supported Containers

Note, even if the sirannon supports a container, it still needs to support the codecs within the container.

- AVI
- MOV/MP4/F4V
- FLV
- WEBM
- MPEG2 (MPEG2 Program Streams)
- TS (MPEG2 Transport Streams)
- RAW (containing any of the supported codecs)

2.3 Supported Protocols

Sirannon can be both server and client for the following protocols:

- RTSP/RTP/UDP
- RTMP/TCP
- RTMPT/HTTP
- HTTP
- RTP/UDP
- UDP
- TCP
- Apple Live HTTP Streaming

Chapter 3

Media Server

3.1 Universal Server - Universal Client - Protocol translation

The strongest feature is the combination of universal server (RTSP, HTTP, RTMP, RTMPT) and universal client (RTSP, HTTP, RTMPT, RTMPT). This combination gives *Sirannnon* the ability to transcode one protocol to another in real-time, dynamically and for many users. For example a request of the form
`rtmp://mysirannon.com/RTSP-proxy/www.tv-world.net/content/AJaXo93cdW.mov`
in a Flash Player will make it connect to a Sirannon server that will in its turn connect to the fictional site `www.tv-world.net` using RTSP, request the stream and in real-time change to protocol and packetization to sent it to the client using RTMP. The following table provides the supported protocol translations.

To – From	File	RTMP	RTMPT	HTTP	RTSP
File	√	√	√	√	√
RTMP	√	√	√	√	√
RTMPT	√	√	√	√	√
HTTP	√	√	√	√	√
RTSP	√	√	√	√	√

3.1.1 Connecting to the media server

You can run Sirannon as media server (HTTP, RTMP, RTMPT, RTSP) by placing content in the folder "dat/media" and running:
`sirannon dat/xml/media-server-std.xml dat/media`

All requests to the Sirannon media server are of the form:

$$url ::= < protocol > \text{" : //"} < server > \text{"/"} < application > \text{"/"} < request >$$

$$request ::= < server > \text{"/"} < application > \text{"/"} < request > | < file >$$

When using the application FILE or HTTP, the request is a path to a file:

$$rtmp://sirannon.atlantis.ugent.be/FILE/flash/example.flv$$

$$http://sirannon.atlantis.ugent.be/HTTP/mp4/example.mp4$$

When using the Sirannon server as proxy for another protocol the applications are: RTMP-proxy, RTMPT-proxy, RTSP-proxy and HTTP-proxy. In this case request contains the server, application and file to request:

$$http://localhost/RTMP-proxy/vod01.netdna.com/play/vod/demo.flowplayer/metacafe.flv$$

3.1.2 Media Server URL Examples

HTTP

$$"http://" < server > "/" < HTTP|FILE@CONTAINER > "/" < file >$$

$$http://sirannon.atlantis.ugent.be/HTTP/demo.mov$$

$$http://sirannon.atlantis.ugent.be/FILE@FLV/mysequence.mkv$$

Apple Live HTTP

- The short form

$$"http://" < server > "/APPLE/" < file >$$

$$http://sirannon.atlantis.ugent.be/APPLE/demo.mov$$

- The long form

$$"http://" < server > "/M3U/" < server > "/FILE@TS/" < file >$$

$$http://sirannon.atlantis.ugent.be/M3U/sirannon.atlantis.ugent.be/FILE@TS/demo.mov$$

RTMP

$$"rtmp://" < server > "/FILE/" < file >$$

$$rtmp://sirannon.atlantis.ugent.be/FILE/mysequence.mov$$

RTMPT

$$"rtmpt://" < server > "/FILE/" < file >$$

$$rtmpt://sirannon.atlantis.ugent.be/FILE/mysequence.mov$$

RTSP

$$"rtsp://" < server > "/FILE/" < file >$$

$$rtsp://sirannon.atlantis.ugent.be/FILE/mysequence.mov$$

Chapter 4

Installation

Refer to the README in the distribution for the documentation about the compiling sirannon.

Chapter 5

Tutorial

5.1 Introduction

This chapter describes how to construct, using the user interface, a basic streaming solution. Chapter 7 describes several examples of streaming solutions without directly specifying the construction in the user interface. The following will create a basic streamer for streaming a trailer from Apple using RTP. The output from the user interface is an XML configuration file. The next chapter describes how to run the sirannon with this configuration file.

5.2 Exploring the user interface

Launch the user interface by running: `sirannon.py` The user interface will launch and present itself in five tabs: Construct, Draw, XML, Run and Library. This tutorial will focus on the Draw and Run tab. Initially the draw area is empty and should look like in figure 5.1. Many of the functions available in the menu bar such as *save*, *quit*, *undo*, ... are self-explanatory.

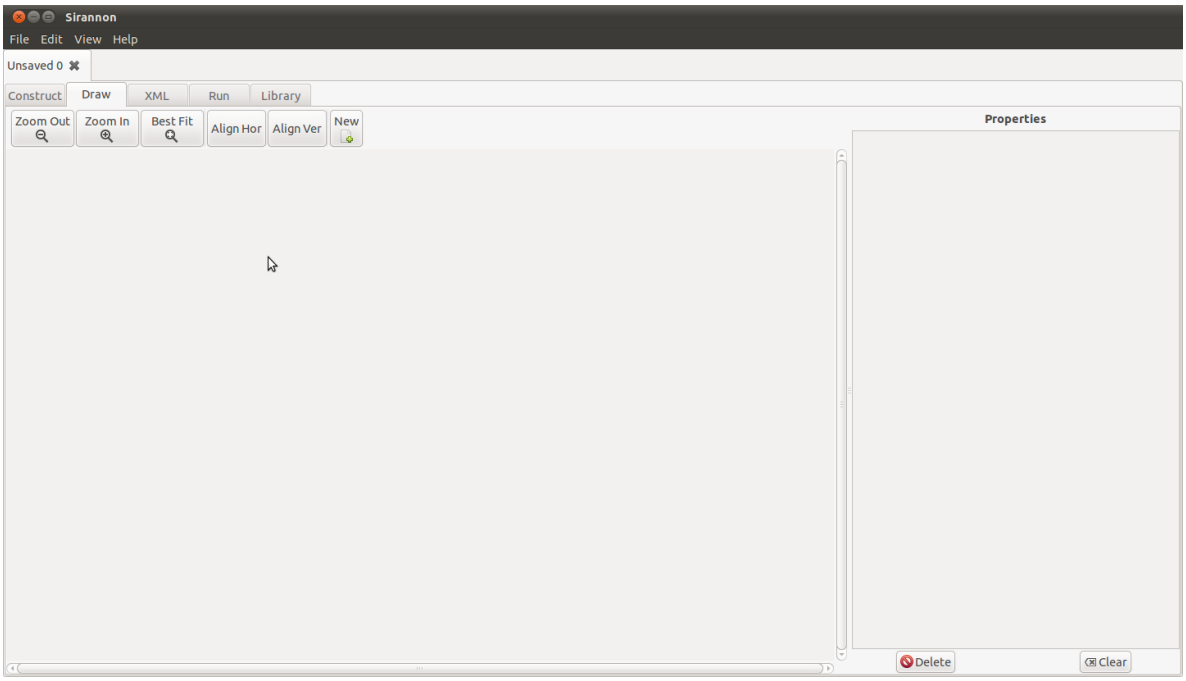

Figure 5.1: the user interface

5.3 Creating your first component

In order to open the trailer, a reader component is needed. The *ffmpeg-reader* component provides access to the Quicktime container. Let us create this component now. Right-click anywhere in the work area or click on the button 'New' in the toolbar. A menu will appear with the different categories of components. Select *reader*, *ffmpeg-reader*. A new component will appear as seen in 5.2. If you left click inside the new component, in the right of the screen an overview of the parameters of the component appears. These parameters are set at sensible defaults, we only need to fill out the *filename* of the trailer. After filling out the *filename*, the result should look as in figure 5.3.

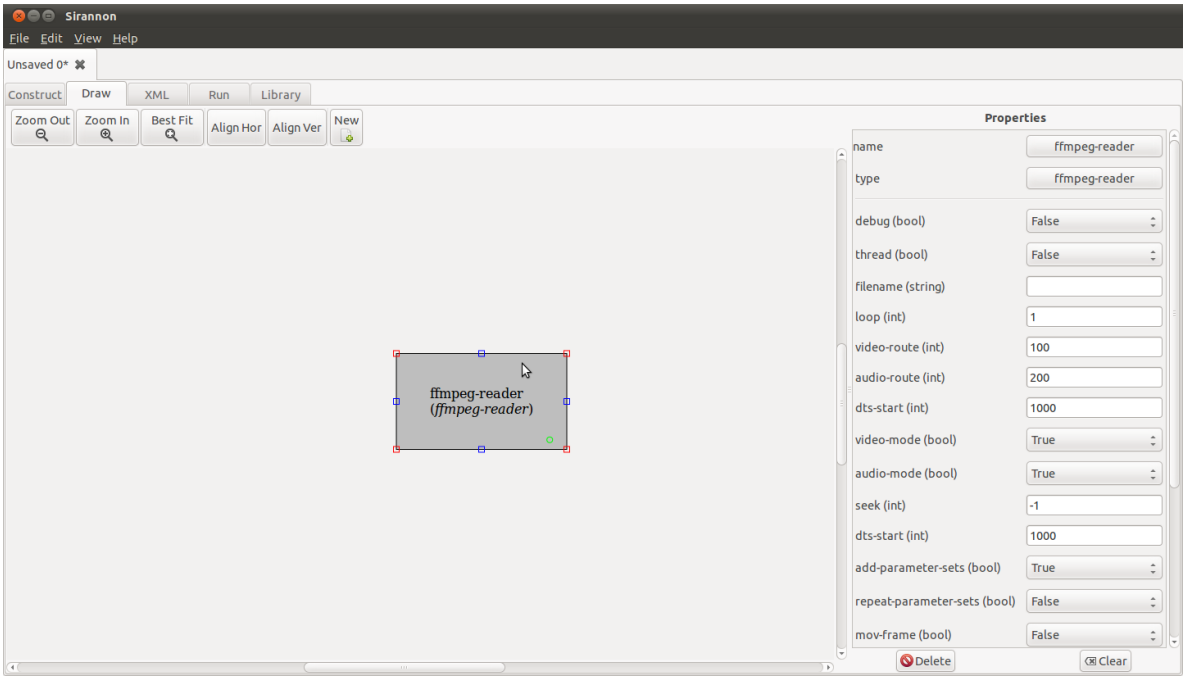

Figure 5.2: creating the first component (a)

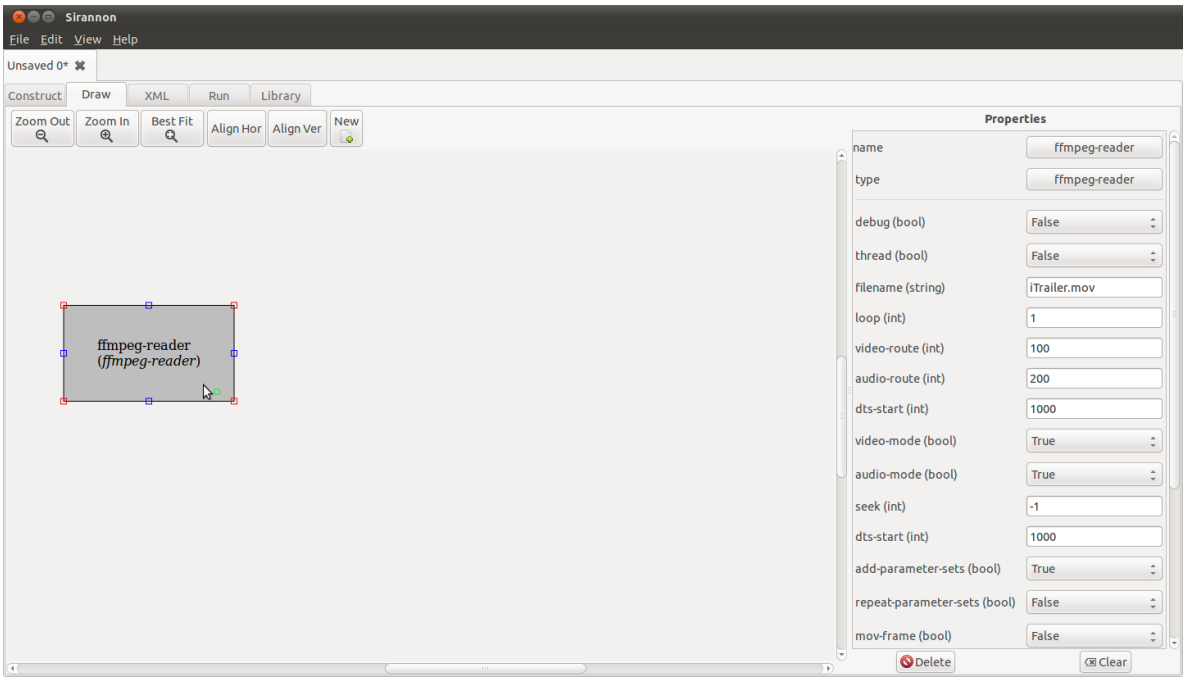

Figure 5.3: creating the first component (b)

5.4 Creating your second component

The component *ffmpeg-reader* creates packets containing video and audio frames. However, such frames are too large to be sent directly on the network: they have to be packetized into smaller packets. Now let us create the second component. Right-click in the empty work area and select *packetizers*, *PES-packetizer* in the menu. Change the name of the component to *PES-packetizer-video* by clicking on the button next to "name" in the properties area in the right of the screen. Drag this new component to a fitting place, as shown in figure 5.4.

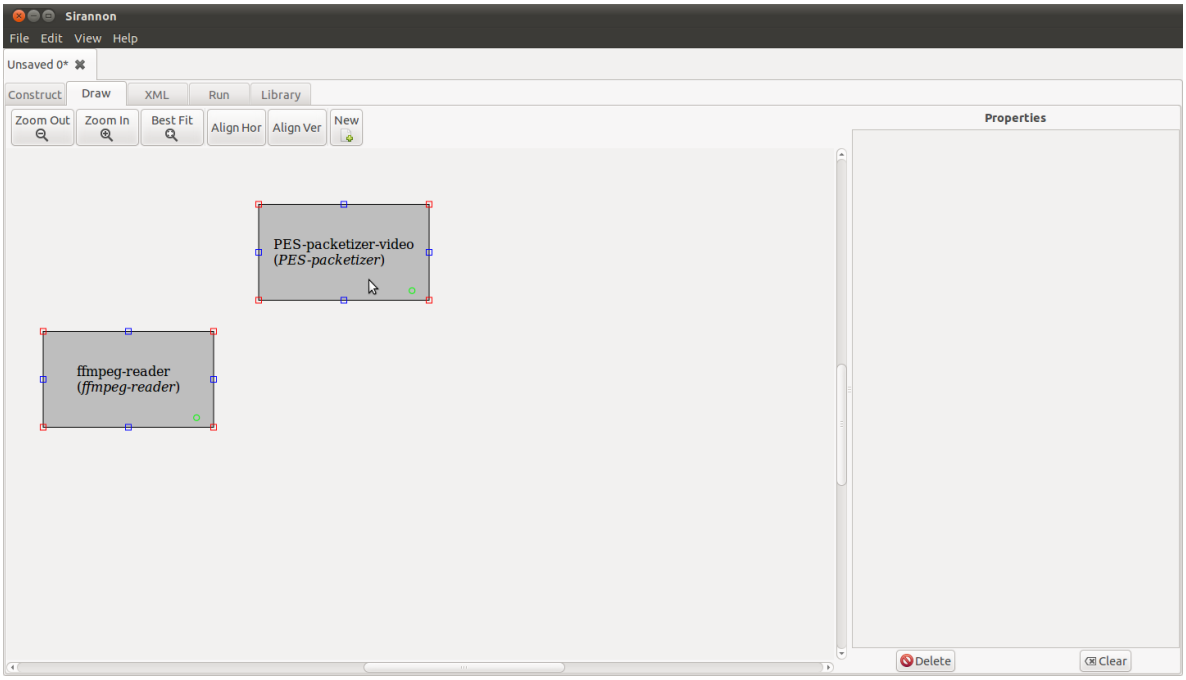

Figure 5.4: creating the second component

5.5 Connecting your components

The two components need to be connected with each other. When you left-drag from inside one of the blue squares of a component, you will start drawing a line. Left-drag from inside the component *ffmeg-reader* and release the mouse inside one of the blue squares of the component *avc-packetizer*. The connection between the two components is now made and the configuration should now look as in figure 5.5.

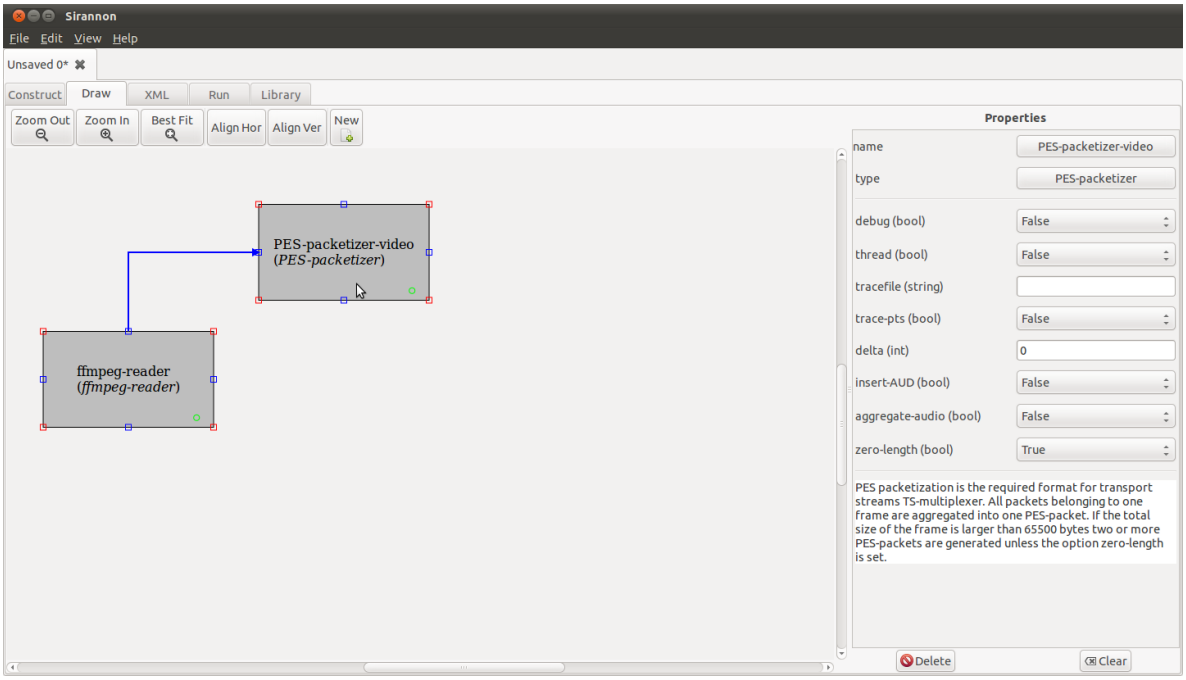

Figure 5.5: connecting the two components

5.6 Creating and connecting your third component

The *PES-packetizer-video* component will only process video frames. We need a similar packetizer for audio frames. Let us create a third component *PES-packetizer-audio*, found as *packetizer*, *PES-packetizer* in the menu. Change the name of the component to *PES-packetizer-audio*. Connect *ffmpeg-reader* with *PES-packetizer-audio*. No parameters have to be changed for this component either. The result should be as in figure 5.6. How can the component *ffmpeg-reader* know on which connection to send packets, since video frames should be sent to *AVC-packetizer* and audio frames to *MP4-packetizer*? In the sirannon each packet is tagged with a number called *xroute*. If you look at the parameters from *ffmpeg-reader*, notice the parameters *video-route* and *audio-route*. Using the default settings, video packets will be tagged with *xroute* 100 and audio packets with *xroute* 200. Left click anywhere on connection between *ffmpeg-reader* and *AVC-packetizer*. In the properties area in right of the screen the parameter xroute appears. Since we want to sent only video packets over connection, fill out the value 100. By default the *xroute* is 0, meaning all packets are accepted on this connection. If a packet can take multiple paths, for example if multiple connections share the same *xroute* value, a separate copy will be sent over each of those connections. Now left-click on the connection between *ffmpeg-reader* and *MP4-packetizer* and fill out the value 200, causing audio packets to be sent over this connection.

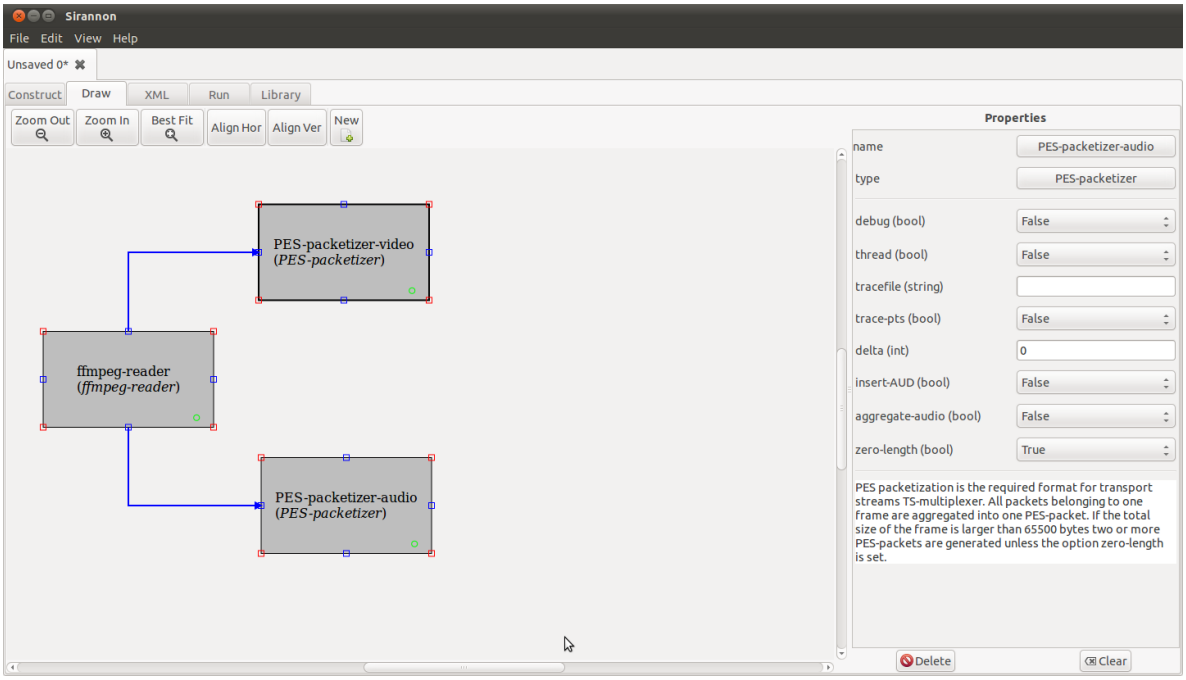

Figure 5.6: creating the third component

5.7 Creating a transport stream

If we want to send the trailer over a single connection we need to multiplex the trailer into an MPEG2 Transport Stream. Create new component *TS-multiplexer* by selecting *multiplexer*, *TS-multiplexer* from the menu. Connect both packetizers with this new component. The result should look in figure 5.7. If you made a mistake, you can always undo using CTRL-Z or selecting undo in the menu.

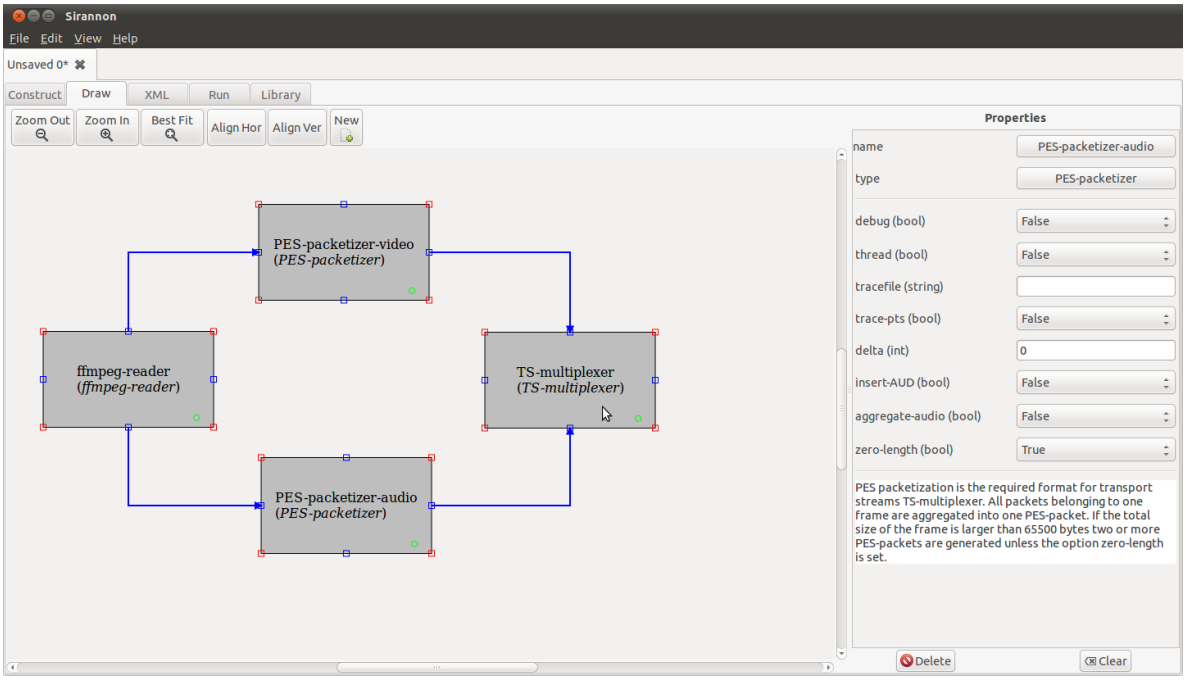

Figure 5.7: multiplexing the packets

5.8 Scheduling the packets

We need a scheduler to add real-time behavior: it stores packets in a buffer and releases them at the correct time. Create a scheduler by selecting *schedulers*, *frame-scheduler* from the menu. Figure 5.8 shows the intended result. If your draw area becomes too small you can always zoom out by clicking the appropriate button in the toolbar.

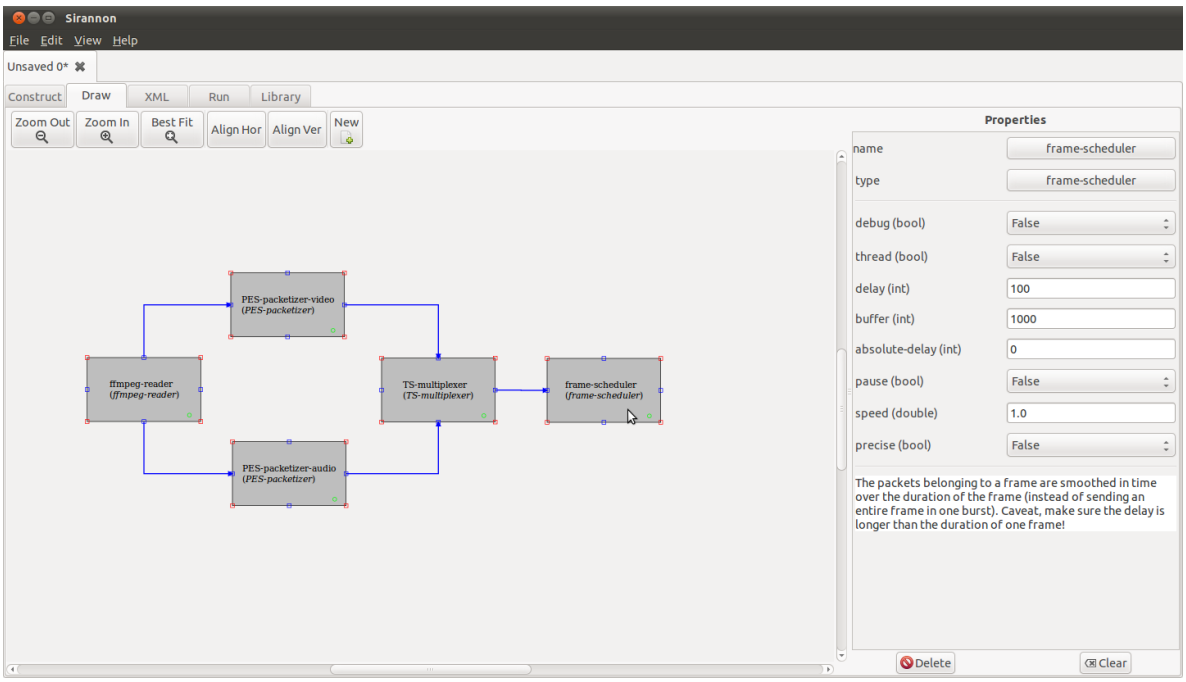

Figure 5.8: scheduling the packets

5.9 Transmitting the packets

The frames are ready now for transmission: they are packetized in sufficiently small packets, multiplexed into an MPEG2 transport stream and scheduled at the correct time. Let us create an RTP-transmitter. Select *transmitter*, *RTP-transmitter* from the component menu and connect the scheduler with the new transmitter. We have to fill out the source port and destination address. In the properties area for the parameter *port* fill out the value 5000 and for the parameter *client* fill out 127.0.0.1 : 1234. Also set the parameter *debug* to *true* for this component. Figure 5.9 shows the result.

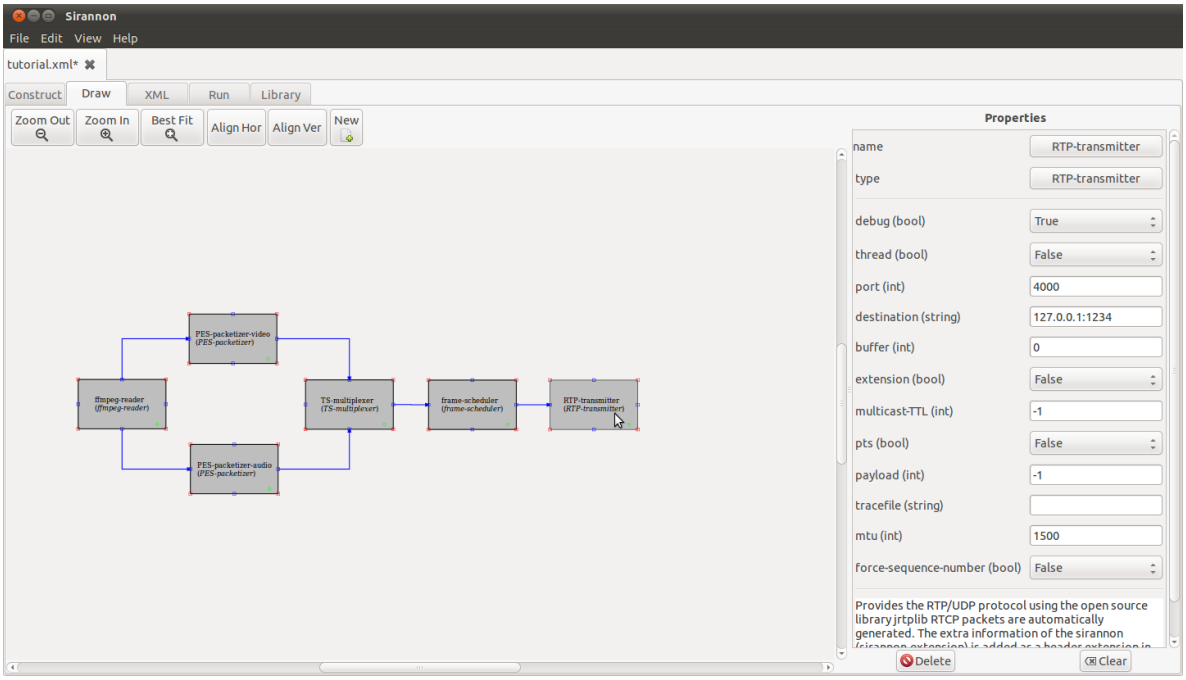

Figure 5.9: transmitting the packets

5.10 Finalizing the chain

In order to close the sirannon after streaming the sequence, we should place a special block called *sink* at the end of the chain. After the last packet of the sequence has passed through this sink, it terminates the program gracefully. Select *system*, *sink* from the component menu and make a connection form the transmitter to this sink. The result should look as in figure 5.10. Zoom out or use best fit if the drawing area is too small.

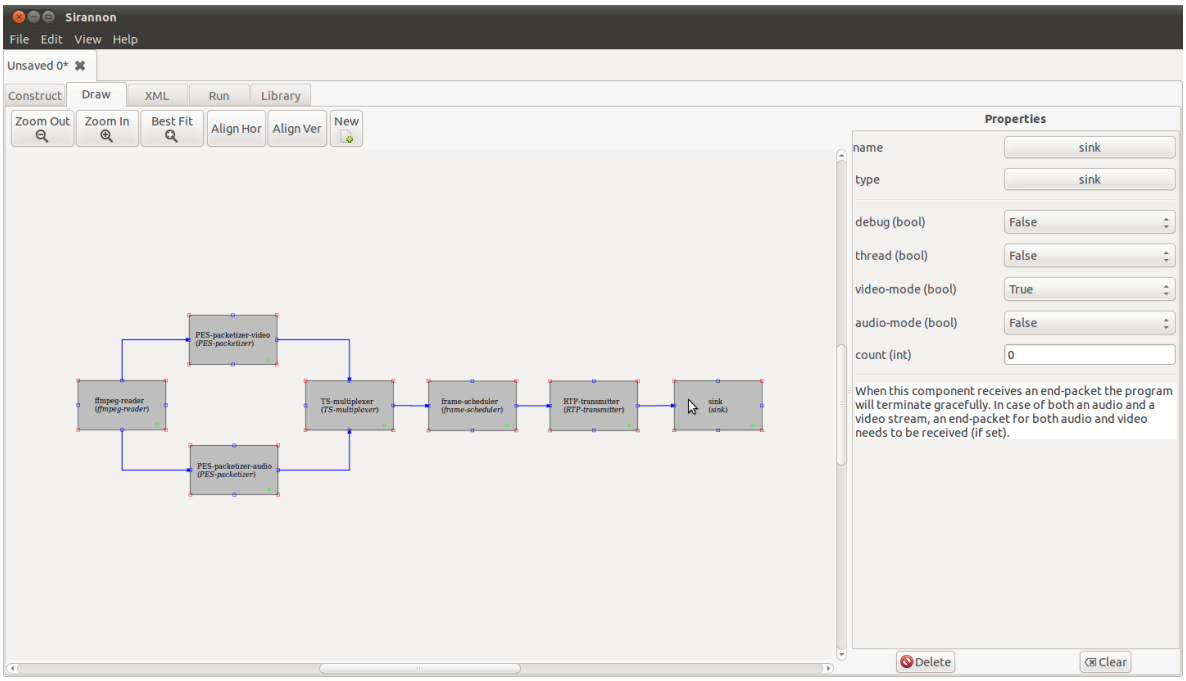

Figure 5.10: finalizing the chain

5.11 Saving and Executing your the configuration

Now that the configuration is complete, we can save it. In the menu bar under *File* use either *Save* or *Save as* or press *CTRL-S*. We can now run sirannon using the tab "Run". A new tab should appear that looks like in figure 5.11. If the GUI finds the binary it should appear as first item of the command line options. Under Unix you should have compiled and installed the sirannon binary if you wish to continue. If you installed the binary in a different location you can always select that location.

We do not need to fill out the configuration, when you press play the GUI will automatically using the current configuration unless you overwrite it.

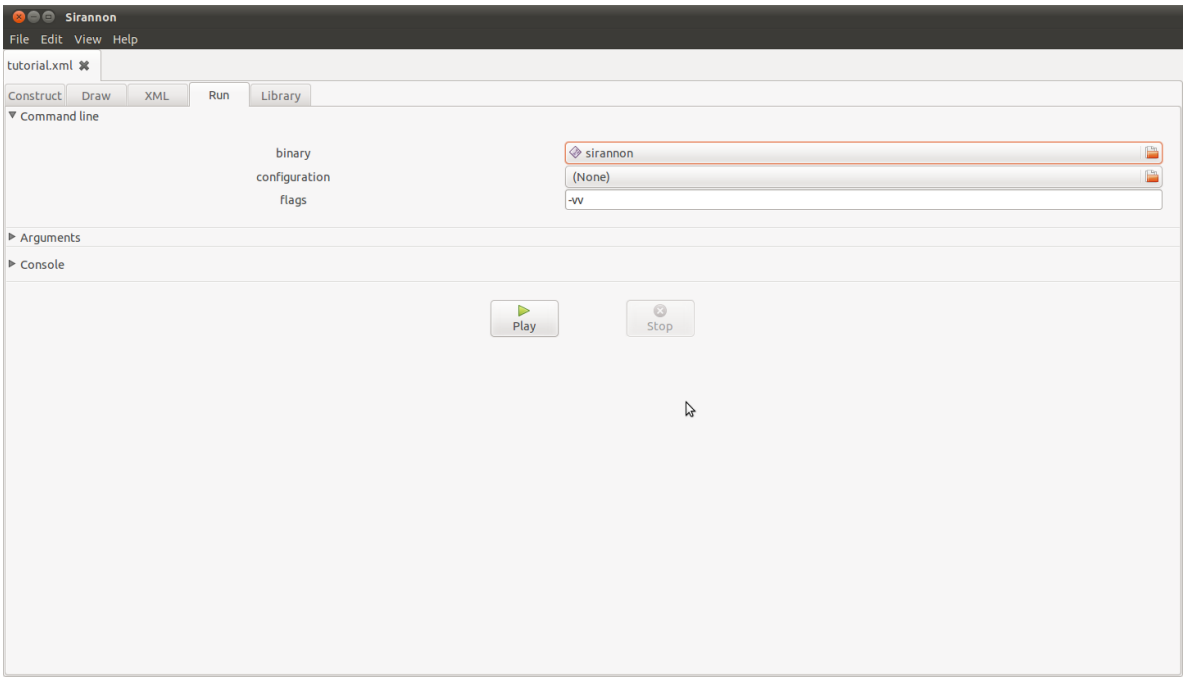

Figure 5.11: the "Run" tab

Press the large *Play* button. The console window expands and shows the output of the sirannon process. If you filled out an incorrect container for the ffmpeg-reader, sirannon should fail with an error like `Unhandled RuntimeError: core.ffmpeg-reader: Could not open file(iTrailer.mov)`. If you do not have a sample video at hand you can always download the demo containers from `http://sirannon.atlantis.ugent.be/files/demo.tar.bz2`. Bear in mind that for MPEG2 Transport Streams only MPEG video and audio codecs are supported. If you created everything correctly output should appear in the console as in figure 5.12.

Once the console is running you can open for example VLC Media Player. Under *Media*, select *Open Network Stream*. Fill out `rtp://@:1234` as URL. VLC should now start playing the video. **CAVEAT:** If your sequence uses H.264/AVC, VLC will not have critical information that was transmitted by streamer before VLC launched. You will need to press *Stop* followed by *Play* to fix it. To prevent this behaviour you can set the parameter *repeat-parameter-sets* to *true* for the component *ffmpeg-reader*.

Now the video should be playing as in figure 5.13. If it is not working, go over the following check list.

- Press *Stop* followed by *Play*
- Is parameter *destination* for *RTP-transmitter* set to *127.0.0.1:1234*?
- Does the console show an error such as `Could not open file` or `Unsupported codec`?

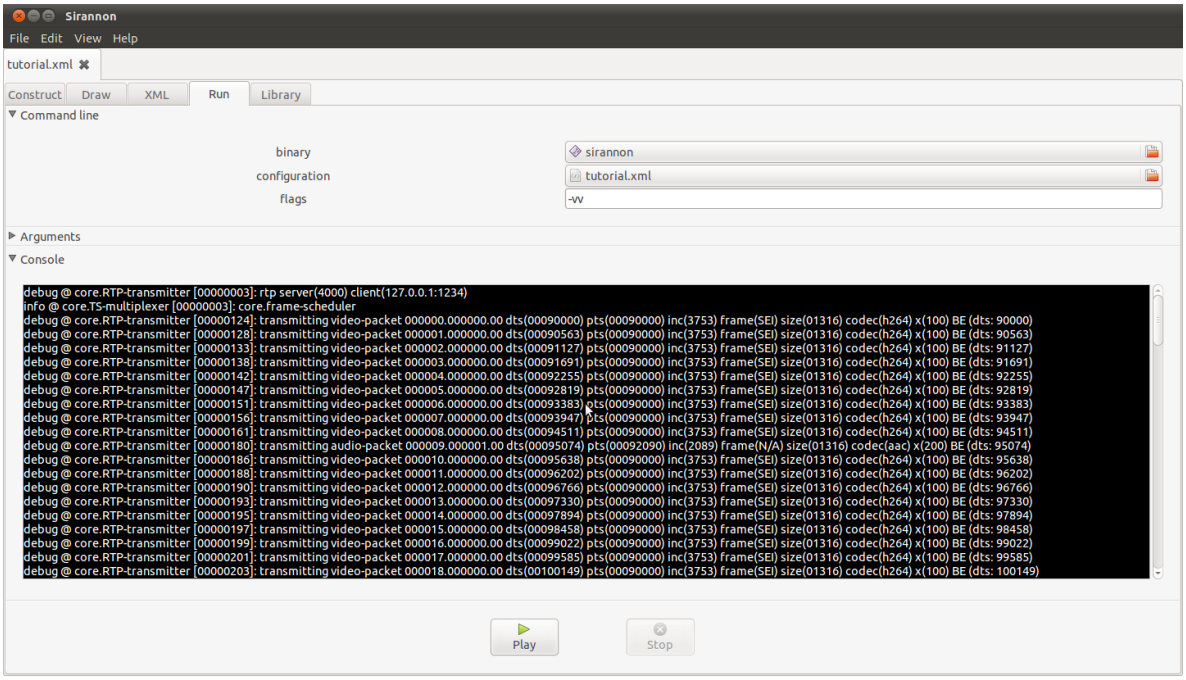

Figure 5.12: the console output

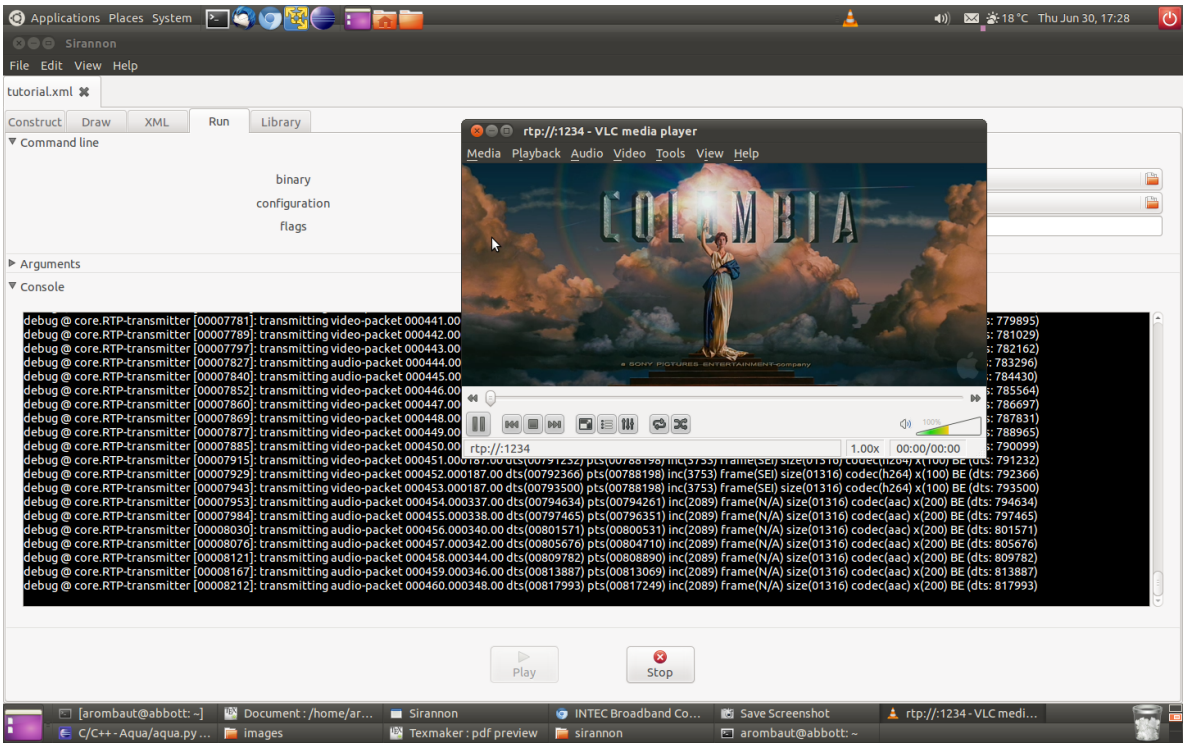

Figure 5.13: VLC receiving and playing the stream

Chapter 6

Execution

6.1 Internal view

We shortly describe here the internal view of the sirannon in order to understand the parameters in the next section. The basic operation of the sirannon is single threaded. The execution consists of a series of cycles with each cycle aiming to process one frame. The real duration of such a cycle, for example 2ms, is often much lower than the duration of one frame, typically 40ms. The process can sleep during the difference, lowering the CPU load considerably.

6.2 Execution parameters

Let us introduce three execution parameters. To modify these parameters in the user interface for a configuration, go in the menu bar to *Settings*, *Settings*. These parameters are specific for each configuration.

1. **quantum**: in milliseconds, defines the minimum time to process one frame. If processing of a frame took less than this quantum, the process sleeps during the difference, lowering the CPU load. If you match this quantum with the duration of each frame, you get a working performance with minimal CPU load. For example a stream with 25 frames per second or 40 ms per frame, can be streamed using less than 1% CPU using a quantum of 40ms. If the quantum is set at 0, the process never sleeps, producing very accurate timing (order 1 ms) at the cost of 100% CPU utilization. default: 0

2. **simulation**: in microseconds, if this value is greater than 0 the sirannon runs in simulated time. Each cycle the simulated time is increased with this value. This has the advantage of providing arbitrary time precision and possibly faster execution at the cost of losing the real time behavior. For example the simulation of streaming an HD/H264 stream using simulation steps of $40000\mu s$, can be run in $2s$ instead of $40s$ in real time. When *simulation* is defined, the parameter *quantum* is ignored. default: 0

3. **seed**: if this value is greater than 0, it provides the seed for the random numbers in the sirannon. If the value is 0, the current time is used as seed. default: 0

6.3 Command line parameters

In order to use the configuration files without having to edit them when using differnt content, component parameters (not execution parameters!) can be entered in the form of $1, $2, $3... The symbol $n will be replaced by the n^{th} command line parameter after the configuration file (see next section). Use the symbol $$ to circumvent this interpretation and represent the character "$".

6.4 Usage

```
sirannon [-cvbh] [-q=NUM[ns|us|ms|s]] [-s=NUM[ns|us|ms|s]] [-r=NUM] FILE [ARG-1] ... [
Run the program with FILE as configuration.

Options:
  -h       Help information
  -b       Build information
  -c       Overview of components
  -v       Verbose, use up to 4 v's to increase the level
  -q=NUM   Quantum in milliseconds
  -s=NUM   Simulation in milliseconds
  -r=NUM   Seed for the random number generator

Arguments:
  ARG-1    Replace any occurrence of "$1" in FILE with ARG-1
  ARG-2    Replace any occurrence of "$2" in FILE with ARG-2
  ARG-NUM  Replace any occurrence of "$NUM" in FILE with ARG-NUM
```

6.4.1 Example 1

```
sirannon -v -q=5 avc-streamer.xml gladiator.avi 127.0.0.1:5000
```

In this example we run the configuration file "*avc-streamer.xml*" with two command line parameters: the video file name and destination address. These command line parameters will replace the symbols $1 and $2 entered in the configuration file. In addition we set the quantum to 5 ms and print debug messages from toggled components.

6.4.2 Example 2

```
sirannon -b -c
```

Prints the build information and the available components. Since no xml file is specified, the program end instantly.

Chapter 7

Examples

The basics of the sirannon were explained in the previous chapters. By using examples, we will demonstrate the many possibilities of the sirannon. In contrast with the tutorial, we will not explain how to construct the configuration in the user interface but we will focus instead on describing the function and structure of different configurations. Sometimes we will refer to specific parameters of components. For a detailed description of the components and a full list of available parameters, refer to the next chapter. The distribution of Sirannon contains for each example the corresponding XML file.

7.1 Example 1: a basic streamer

Let us start with a basic example, even simpler than the tutorial: a basic AVC streamer. Figure 7.1 shows the schematic. It contains the four basic components: a reader, a packetizer, a scheduler and a transmitter, connected in a chain.

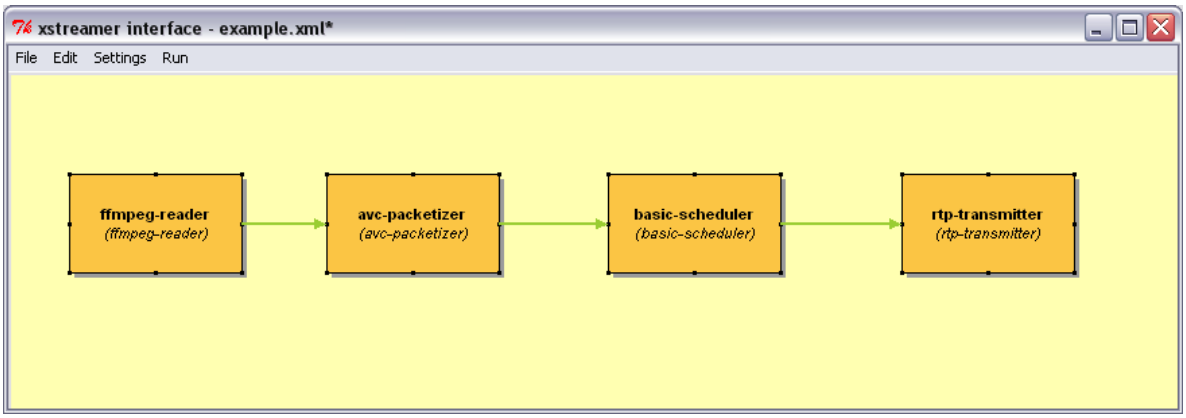

Figure 7.1: a basic streamer

7.2 Example 2: a basic receiver

The stream sent by the basic streamer can be received by a media player, but the sirannon also can act as a receiver. Hence, the sirannon is more than just a streamer. Figure 7.2 shows this receiver. The receiver has four components like the basic streamer, performing the reverse operation: a receiver, a scheduler, an unpacketizer and a writer. Received packets are unpacketized into the original frames and those are written back to a file. The scheduler avoid problems such as UDP reordering or duplication. However, the RTP protocol should circumvent these problems, making the scheduler superfluous in this example.

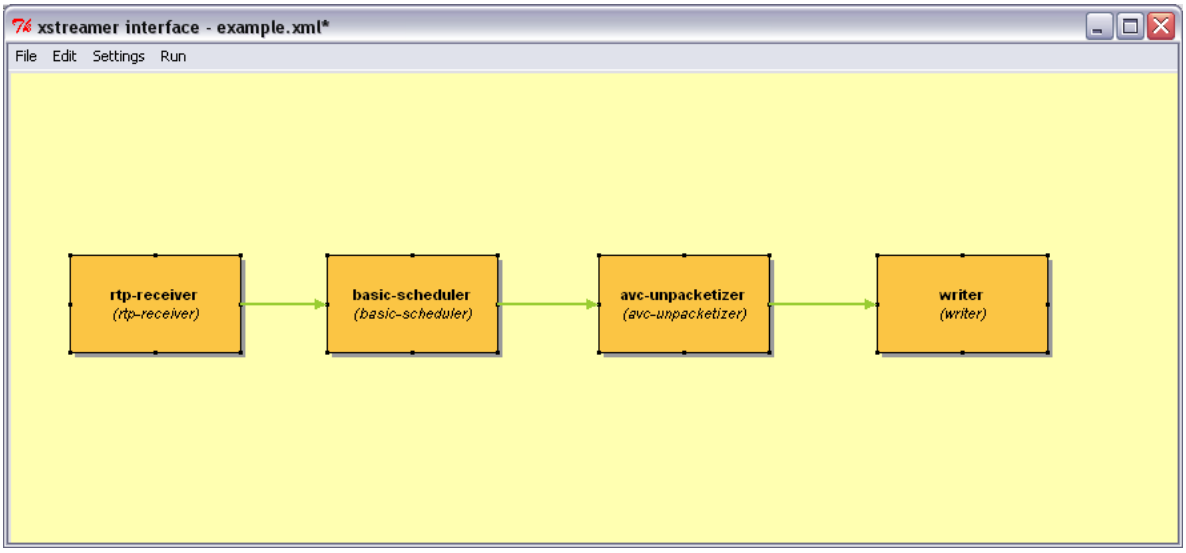

Figure 7.2: a basic receiver

7.3 Example 3: differentiated streaming

This example modifies the structure of the basic streamer from example 1 in order to stream over multiple connections. Figure 7.3 shows the result. Instead of having one *rtp-transmitter*, we now have three. We also add a *frame-classifier* to differentiate the I, P and B frames. The sirannon allows forks in the schematic using a simple routing mechanism based on a label *xroute* per packet. The reader gives each packet an initial *xroute* of 100. A classifier increases the *xroute* with a fixed value for each classification. In this example, *frame-classifier* has three parameters I, P and B with values 1, 2 and 3 respectively, producing packets with *xroutes* 101, 102 and 103. The connections between *frame-classifier* and *rtp-transmitter 1, 2 and 3* use *xroutes* 101, 102 and 103 respectively to split the stream. Section 5.6 also explains this routing system.

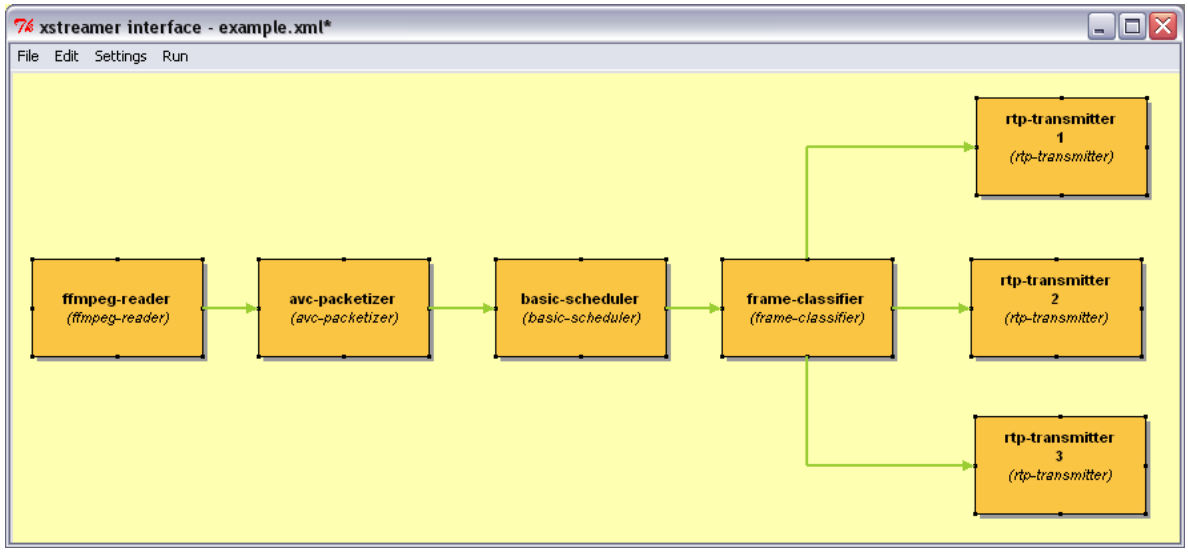

Figure 7.3: a differentiated streamer

7.4 Example 4: proxy

A differentiated stream cannot be played by standard players such as VLC or Quicktime, because it only accepts one connection for video, not three for example. The sirannon can function as a proxy, converting a differentiated stream into a single stream. Such stream can then be received by standard players. Figure 7.4 shows the schematic. The function consists of three types of components: three receivers, a scheduler and a transmitter. The packets from the three *rtp-receivers* pass through the scheduler. In this setting, the scheduler is anything but superfluous since the three connections are unsynchronized. The scheduler restores the original order. It uses additional information included by the transmitters in the RTP header extension since the sequence numbers and time stamps from the RTP connections are insufficient to restore the original order. It also gives the merged stream the correct time behavior, so that the *rtp-transmitter* can send the merged stream. This merged stream is then received by a standard player.

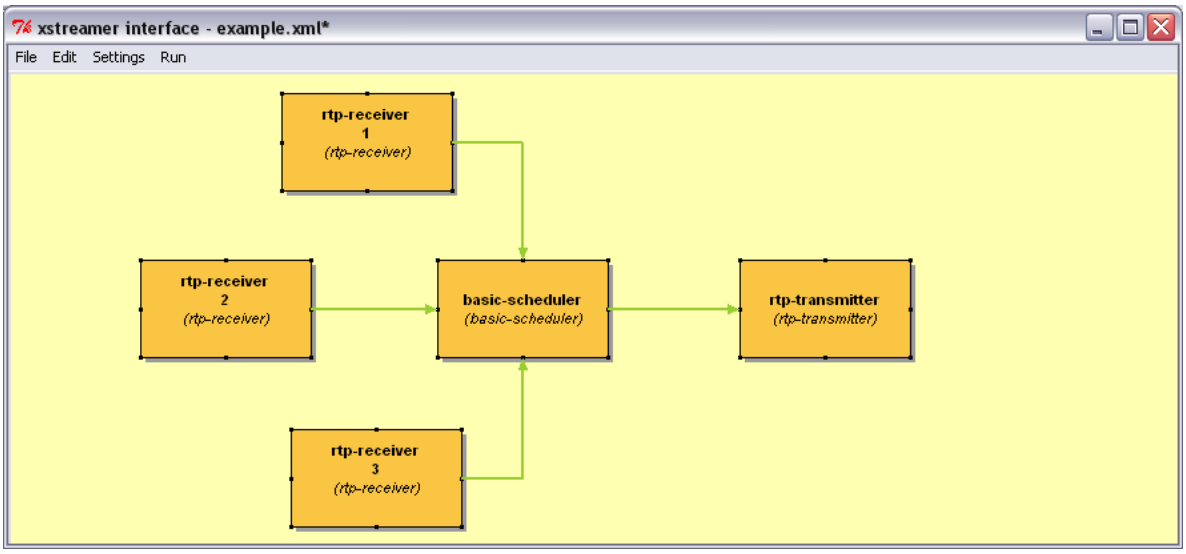

Figure 7.4: a proxy

7.5 Example 5: a packet loss generator

The sirannon can also run as an offline tool. Figure 7.5 shows the schematic to introduce packet loss, with a different percentage for each type of frame. The frames are read and packetized by *avc-reader* and *avc-packetizer*. The *frame-classifier* splits the stream into I, P and B packets. For each of these types there is different component *random-classifier 1, 2* or *3* with its specific packet loss. The damage streams are merged and unpacketized by *avc-unpacketizer* and the resulting frames (some of the original frames are lost) are written by the component *writer*.

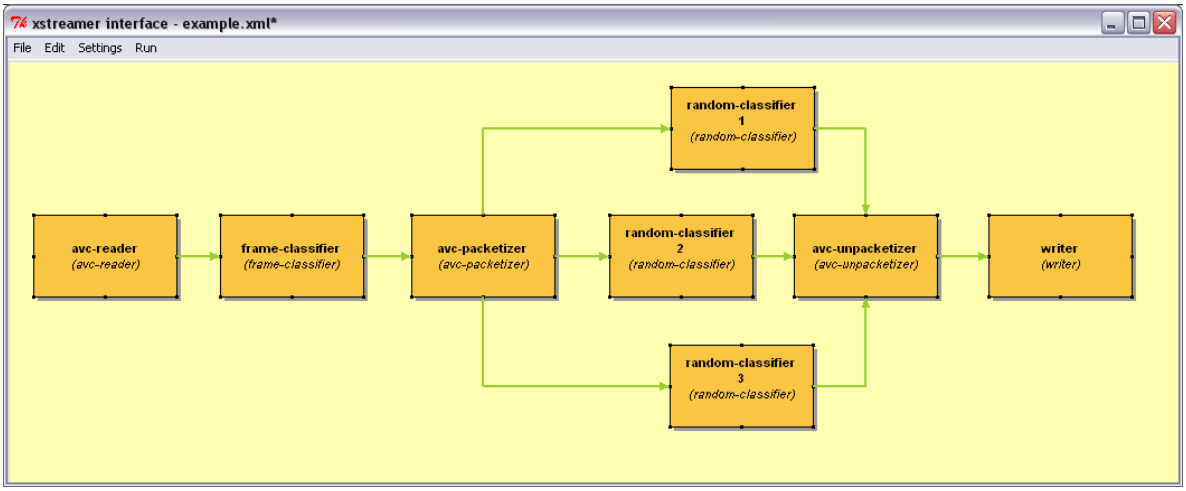

Figure 7.5: a packet loss generator

7.6 Example 6: using transport streams

The sirannon supports MPEG2 transport streams, widely used in digital television. It multiplexes video, audio and meta-data into one stream that we can send over one connection, as opposed to the default RTP mode where each video, audio or meta-data substream has its own RTP session. Figure 7.6 shows the configuration. It does not differ that much from the basic streamer. *ffmpeg-reader* opens a Quicktime file containing both video and audio frames and *xroute* is set at 100 and 200 respectively for video and audio by the reader. Two packetizers create generic PES packets for video and audio respectively. The component *ts-mutliplexer* multiplexes these packets into one transport stream. The two remaining components schedule and transmit the transport stream packets.

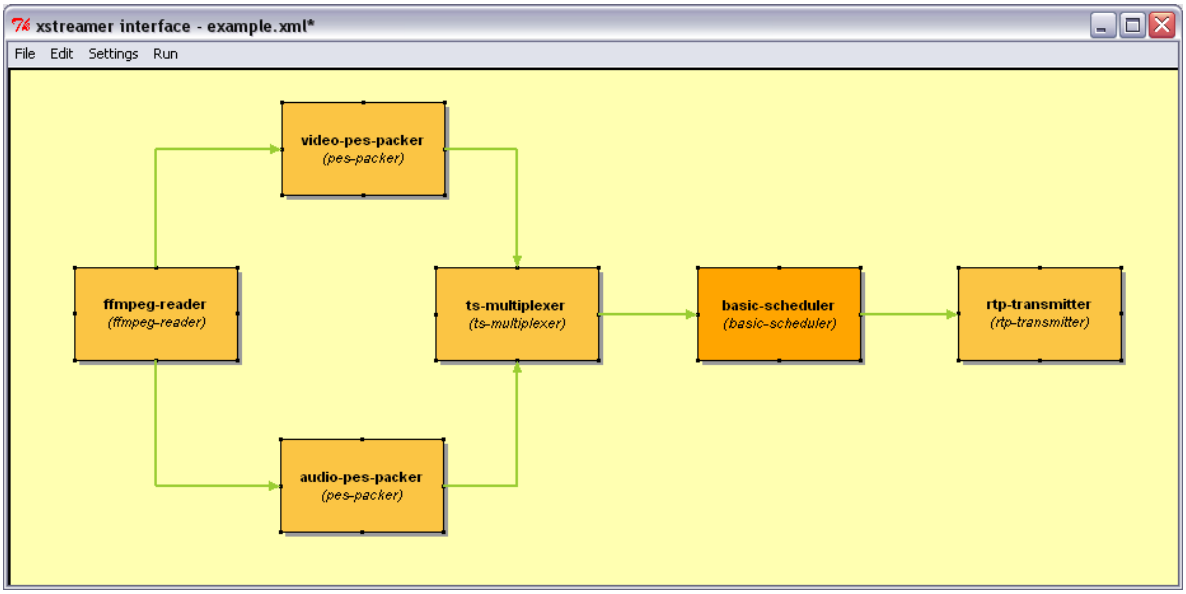

Figure 7.6: using transport streams

7.7 Example 7: using and constructing blocks

Let us create a simulator to test the behavior of a buffer. Block components allows us to group several component into one component, allowing more readable and parameterized configurations. Figure 7.7 shows a *qmatch-buffer* component and three *block* components. Each *block* components uses a configuration file that reads and packetizes a sequence. Figure 7.8 shows this configuration. This looks similar to the basic streamer example but it has at the end of the chain an *out* component which sends packets to the surrounding configuration (figure 7.7). In order to obtains precise timing results, we run the sirannon in simulation mode with a simulation step of 1000 μs.

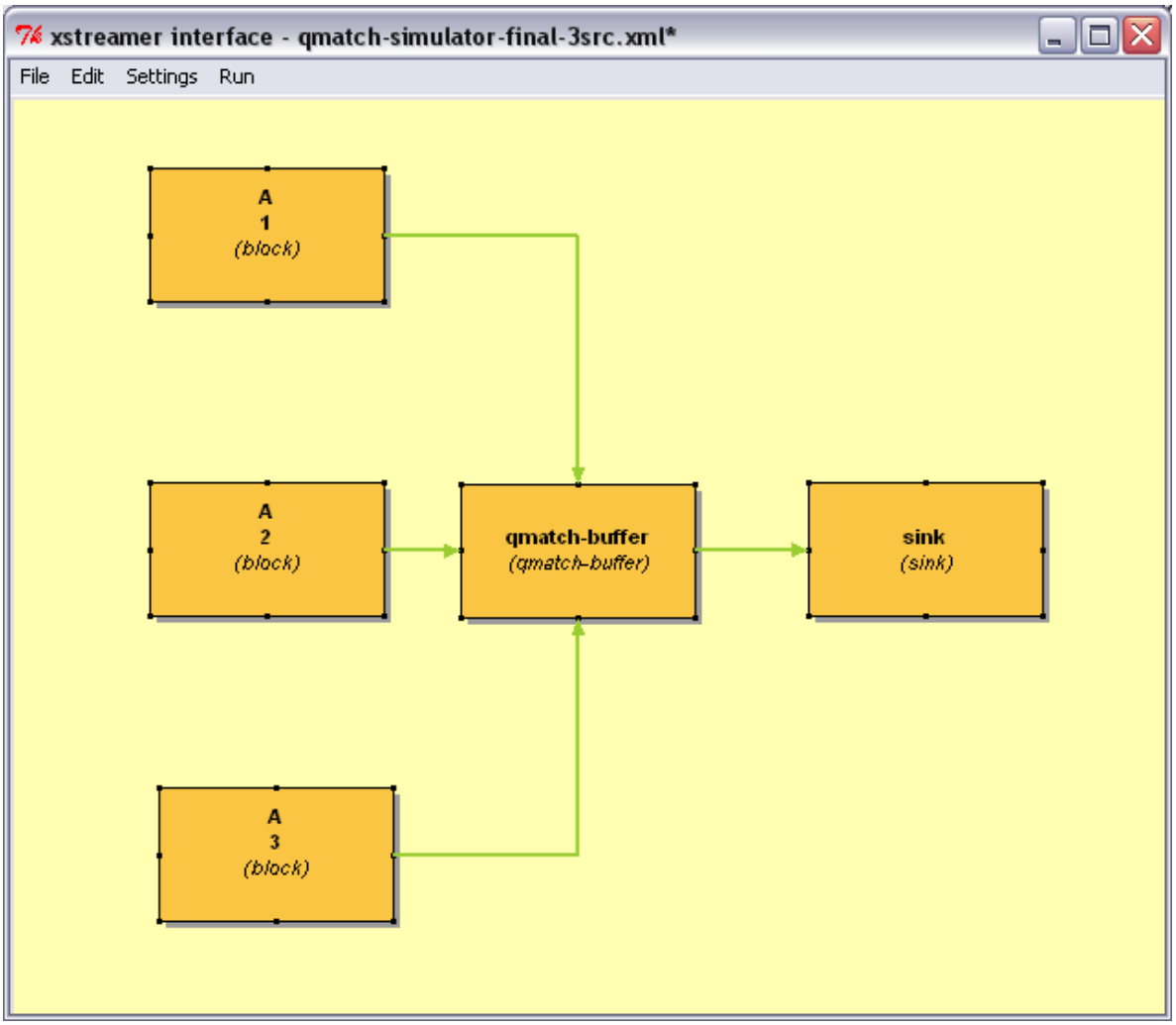

Figure 7.7: simulating a buffer

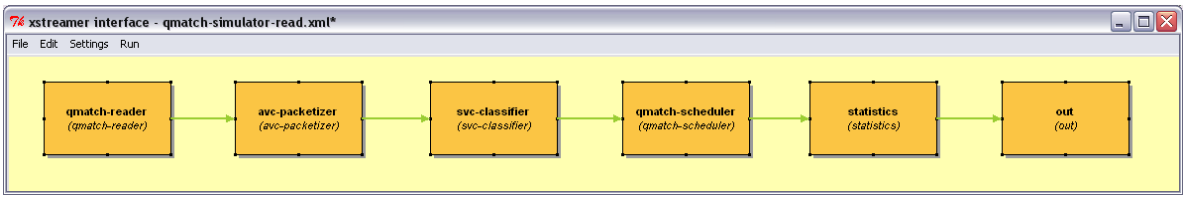

Figure 7.8: streamer contained in the block

7.8 Example 8: massive simulation

Using blocks we can create massive simulators of for example 18 different streams, as shown in figure 7.9.

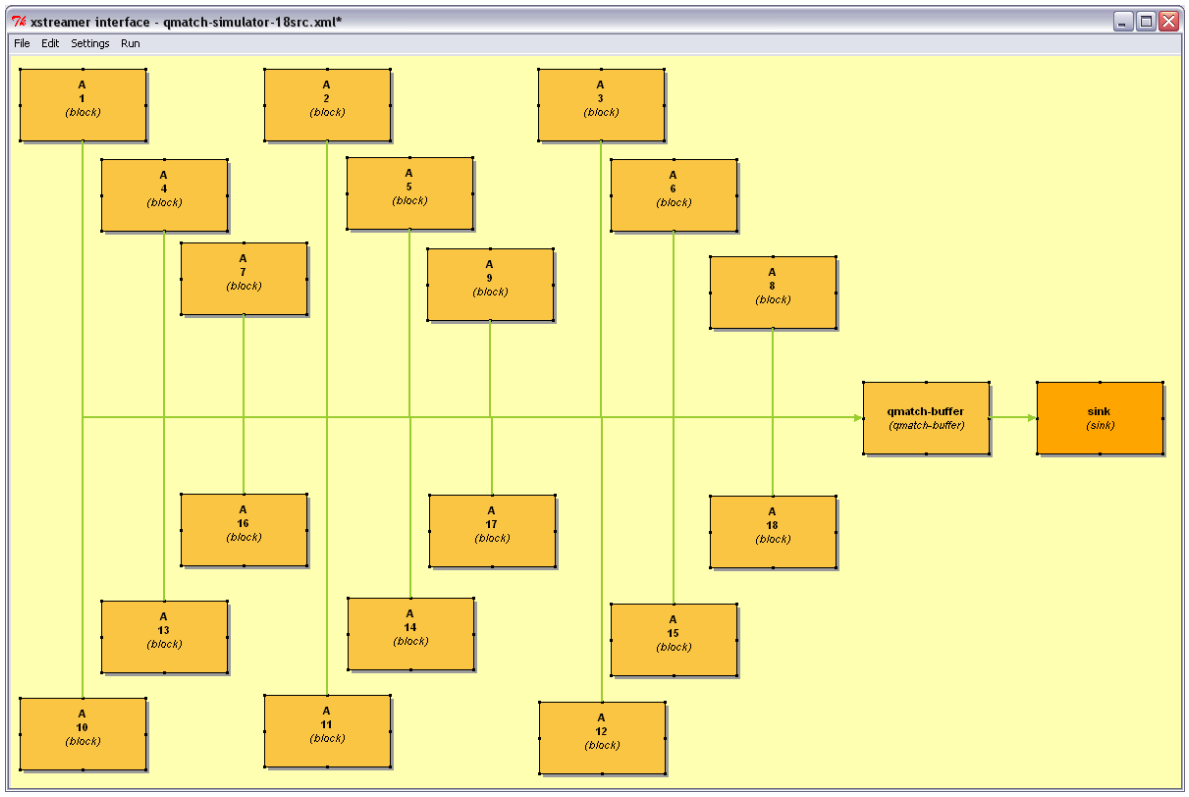

Figure 7.9: a massive simulator

Chapter 8

Extending sirannon

In the source tree `src/Misc/Example.cpp` describes a good commented example about writing your own component. Place any new sources you create in the folder `src/Local` and add the flag `--enable-local` to `configure`. When adding new sources, rerun `configure --enable-local`.

Chapter 9

Components

Components can have two special parameters:

- bool **debug**: if true, the component prints debug information, requires verbose level one or higher (use -v in the command line), default: false
- bool **thread**: if true, run the component in a separate thread, default: false

9.1 classifier

Classifiers add an offset to the xroute of a packet if it meets a certain condition. See section 7.3 for an example. Several classifiers can be chained to obtain a more precision classification.

- Parameters:
  - bool discard: if true, delete the packet instead if the condition is met, default: false
  - string sender-trace: if defined, the path where to log information about the packets entering the classifier, default: ""
  - string receiver-trace: if defined, the path where to log information about the packets exiting the classifier, implies discard is true, default: ""

9.1.1 avc-classifier

This components classifies the many different types of NAL units present in H264, far the beyond the common I, B and P frames.

- Parameters:
  - int I: the offset if the media-packet is an I slice, default: 0
  - int SI: the offset if the media-packet is an SI slice, default: 0
  - int EI: the offset if the media-packet is an EI slice, default: 0
  - int I(B): the offset if the media-packet is an I (data partition B) slice, default: 0
  - int I(C): the offset if the media-packet is an I (data partition C) slice, default: 0
  - int P: the offset if the media-packet is a P slice, default: 0
  - int SP: the offset if the media-packet is an SP slice, default: 0
  - int EP: the offset if the media-packet is an EP slice, default: 0
  - int P(B): the offset if the media-packet is a P (data partition B) slice, default: 0
  - int P(C): the offset if the media-packet is a P (data partition C) slice, default: 0
  - int B: the offset if the media-packet is a B slice, default: 0
  - int EB: the offset if the media-packet is an EB slice, default: 0
  - int B(B): the offset if the media-packet is a B (data partition B) slice, default: 0
  - int B(C): the offset if the media-packet is a B (data partition C) slice, default: 0

- int E: the offset if the media-packet is a prefix NAL, default: 0
- int PPS: the offset if the media-packet is a PPS unit, default: 0
- int SPS: the offset if the media-packet is an SPS unit, default: 0
- int ESPS: the offset if the media-packet is an extended SPS unit, default: 0
- int SEI: the offset if the media-packet is a SEI unit, default: 0
- int default: the offset if the packet does not belong to any of the above, default: 0

9.1.2 count-classifier

This component classifies or discards every nth frame if 'n

- Parameters:
  - int cycle: do not classify every 'cycle * n'th frame, e.g. if cycle is 10, do not classify frames 0, 10, 20, 30, etc., default: 10

9.1.3 fixed-classifier

Classifies the packets, slice or frames based on fixed indices provided by the user. CAVEAT: Indices must rise strict monotonously; Non slice NAL units such as PPS, SPS, SEI, E, are not counted as slices while different SVC layers are also counted as slices. For example an MVC stream with 4 slices and with 2 layers per frame will be counted as 8 slices.

- Parameters:
  - string mode: whether the indices inputted in values or values-file specify packets, slices or frames, accepted values: frame, slice, packet, default: frame
  - string values: if defined, a comma seperated string of indices, indicating which packet, slice or frame to classify counted from 0, default: ""
  - string values-file: if defined, a file containing a new line seperated string of indices, indicating which packet, slice or frame to classify counted from 0, default: ""
  - int xroute: offset added to the xroute of the packet if the packet is classified, default: 1

9.1.4 frame-classifier

Classifies packets based on the type of frame they belong to. The parameter I stands for all sort of I frames (I, IDR, EI, SI...), the same holds for other frame types.

- Parameters:
  - int I: the offset if the media-packet is an I slice or frame, default: 0

- int B: the offset if the media-packet is an B slice or frame, default: 0
- int P: the offset if the media-packet is an P slice or frame, default: 0
- int default: the offset if the frame type is none of the above, default: 0

9.1.5 gilbert-classifier

This component has a random chance of classifying a packet using the Gilbert method.

- Parameters:
  - double alpha: $\in [0, 1]$, probability to transit from the GOOD to the BAD state, default: 0.01
  - double beta: $\in [0, 1]$, probability to transit from the BAD to the GOOD state, default: 0.1
  - double gamma: $\in [0, 1]$, probability to classify a packet in the BAD state, default: 0.75
  - double delta: $\in [0, 1]$, probability to classify a packet in the GOOD state, default: 0.01
  - int xroute: offset if the condition is met, default: 1

9.1.6 i-conceal-classifier

Classifies the packets, slice or frames based on fixed indices provided by the user. CAVEAT: Indices must rise strict monotonously; Non slice NAL units such as PPS, SPS, SEI, E, are not counted as slices while different SVC layers are also counted as slices. For example an MVC stream with 4 slices and with 2 layers per frame will be counted as 8 slices.

- Parameters:
  - string mode: whether the indices inputted in values or values-file specify packets, slices or frames, accepted values: frame, slice, packet, default: frame
  - string values: if defined, a comma seperated string of indices, indicating which packet, slice or frame to classify counted from 0, default: ""
  - string values-file: if defined, a file containing a new line seperated string of indices, indicating which packet, slice or frame to classify counted from 0, default: ""
  - int xroute: offset added to the xroute of the packet if the packet is classified, default: 1

9.1.7 nalu-drop-classifier

Drops specific NAL units from an H.264/AVC encoded video stream. The indices of the NAL units to drop are provided in a tracefile. The first NAL unit of the stream has index 1.

- Parameters:
  - string drop-trace: path to file which contains information on which NAL units to drop. A sample drop-trace file is located in the src/Misc folder., default: ""
  - bool udpmode: set to true in case a UDP receiver is used instead of an RTP receiver. Sequence numbers are retained in UDP mode whereas in case of an RTP receiver and transmitter new sequence numbers are generated., default: false

9.1.8 random-classifier

This component has a random chance of classifying a packet using a uniform distribution.

- Parameters:
  - double P(loss): $\in [0, 1]$, probability to classify the packet, default: 0.01
  - int xroute: offset if the condition is met, default: 0

9.1.9 svc-classifier

This component classifies an SVC stream. The partitioning is defined by a series of (T, D, Q) triplets. For each passing NAL unit this series is reverse iterated. If the NAL unit depends on the n^{th} triplet then it is classified as $layer - n$. When it didn't depend on any layer it is classified as $layer - 0$. The triplets are entered in the form of $'T, D, Q : offset'$, for example $'4, 0, 0 : 2'$, meaning *'add offset 2 to packets of temporal layer (T) 4 or higher'*.

- Parameters:
  - int layer: offset for layer 0, default: 0
  - string layer-1: definintion and offset for layer 1, default: ""
  - string layer-2: definintion and offset for layer 2, default: ""
  - string layer-3: definintion and offset for layer 3, default: ""
  - string layer-4: definintion and offset for layer 4, default: ""
  - string layer-5: definintion and offset for layer 5, default: ""
  - string layer-6: definintion and offset for layer 6, default: ""
  - string layer-7: definintion and offset for layer 7, default: ""

9.1.10 time-classifier

Basic classifier based solely on the real time interval, relative to the first packet. The four time parameters combine to create an interval of the form: $t \in [start, stop[$, when step is defined, it must also hold that:

$$t \in \bigcup -n = 1^{+\infty}[n \cdot step, n \cdot step + delta[$$

.

- Parameters:
  - int start: in ms, the minimum of the range, default: 0
  - int stop: in ms, the maximum of the range, -1 being unlimited, default: -1
  - int step: in ms, the time between two steps, -1 disabling the step effect, default: -1
  - int delta: in ms, the duration of one step, -1 being unlimited, default: -1
  - int xroute: offset if the condition is met, default: 1

9.1.11 timestamp-classifier

Basic classifier based solely on the decoding time stamp (DTS) interval, relative to the first packet.

- Parameters:
  - int start: in DTS clock ($90kHz$), the minimum of the interval, default: 0
  - int stop: in DTS clock ($90kHz$), the maximum of the interval, -1 being unlimited, default: -1
  - int xroute: offset if the condition is met, default: 1

9.2 core

- Parameters:
  - bool debug: if true, print debug info for this component, default: false
  - bool thread: if true, run the component in a seperate thread, default: false

9.3 demultiplexer

9.3.1 FFMPEG-demultiplexer

This component is identical to the component FFMPEG-reader, with the exception that the source is not a file but a stream of chunks from a container.

- Parameters:
  - string format: container format of the input, default: flv
  - int chunk-size: amount of bytes fed to the FFMPEG IOContext each time, default: 32768
  - int loop: the number of times to play the video, -1 being infinite, 0 interpreted as 1, default: 1
  - int videoroute: the xroute that will be assigned to packets containing video, default: 100
  - int audioroute: the xroute that will be assigned to packets containing audio, default: 200
  - int dts-start: in ms, the timestamp that will be added to the DTS & PTS of each frame, default: 1000
  - bool video-mode: if true, video will be read from the container, if false video will be ignored, default: true
  - bool audio-mode: if true, audio will be read from the container, if false audio will be ignored, default: true
  - int seek: in ms, the timestamp to jump to into the stream, -1 implying no seek, default: -1
  - int dts-start: in ms, specifies the value of the timestamp of the first frame, default: 1000
  - bool add-parameter-sets: H.264/AVC only, if true, extract the parameter sets from the container and insert them into the stream, default: true
  - bool repeat-parameter-sets: H.264/AVC only, if true, repeat the parameter sets before each IDR frame, default: false
  - bool mov-frame: MOV/MP4/F4V container only, if true, keep frames in the format of the container, as opposed to annex-B H.264/AVC streams with start codes before each NAL unit, default: false
  - bool skip-SEI: if true, remove SEI NALUs from the stream, default: false
  - bool skip-AUD: if true, remove AUD NALUs from the stream, default: false

9.3.2 TS-demultiplexer

Unmultiplexes an MPEG Transport Stream into the original streams each consisting of series of PES-packets. It performs the reverse operation of ts-multiplexer. The MPEG Transport Stream can be as large as entire multi-channel stream.

- Parameters:
  - int channel: the selected channel to extract, -1 being all channels, default: -1
  - int video-route: the xroute that will be assigned to packets containing video, default: 100
  - int audio-route: the xroute that will be assigned to packets containing audio, default: 100

9.4 high-level

9.4.1 TS-segmenter

This component ingest a video container and generates a series of transport streams at varies bit rates

- Parameters:
  - string media: path of the folder in which to search for requested files, default: dat/media
  - string filename: the path of the container to stream (relative to the folder specified by media), default: ""
  - string target: string indicating the target display (iphone, ipad, youtube), default: ""
  - int duration: in seconds the maximum duration of one segment, default: 10
  - int bitrate-0: in kbps, the 1st bit rate set point, if 0, do not transcode and only segment and do not check for more bitrates, default: 0
  - int width-0: the 1st width set point, default: 0
  - int height-0: the 1st height set point, default: 0
  - int bitrate-1: in kbps, the 2nd bit rate set point, if 0, do not transcode and only segment and do not check for more bitrates, default: 0
  - int width-1: the 2nd width set point, default: 0
  - int height-1: the 2nd height set point, default: 0
  - int bitrate-2: in kbps, the 3rd bit rate set point, if 0, do not transcode and only segment and do not check for more bitrates, default: 0
  - int width-2: the 3rd width set point, default: 0
  - int height-2: the 3rd height set point, default: 0

9.4.2 streamer

This component provides a streaming solution without having to construct a scheme of components. Its many parameters reflect the options of the hidden underlying components (delay, port, destination...).

- Parameters:
  - string filename: the path of the container to stream, default: ""
  - string mode: what content to read from the container (video, audio, default), default: default
  - int loop: the number of times to play the stream, -1 being infinite, 0 interpreted as 1, default: 1

- int seek: the time index to seek to into the container, default: 0
- bool aggregate: option of some packetizers, default: false
- int multiplexer-delay: delay of the multiplexer, used for transport streams or RTMP, default: 0
- string scheduler: which sort of scheduler to use (basic, frame, gop, window), default: basic
- int scheduler-buffer: in ms, the buffer size in time of the scheduler, default: 1000
- int scheduler-delay: in ms, the delay of the scheduler, default: 0
- double loss: if larger than 0, randomly lose packets with the give chance, default: 0.0
- string transmitter: type of the transmitter(rtp, udp, tcp), default: rtp
- int port: source port of the transmitter, default: 4000
- string destination: destination of the transmitter, default: 127.0.0.1:5000
- bool ts-mode: if true, multiplex the streams into a transport stream, default: false
- bool RTMP-mode: if true, muliplex the streams into RTMP chunk streams as output from this component, default: false
- int RTMP-streamID: streamID used for each RTMP chunk stream, default: 1
- int RTMP-video-chunkID: RTMP chunk stream ID for video, default: 7
- int RTMP-audio-chunkID: RTMP chunk stream ID for audio, default: 6
- int RTMP-chunk-size: chunk size for the RTMP chunk streams, default: 4096

9.5 media-client

Media Clients connect to servers and retrieve a stream using a particular protocol. The component outputs frames just as would a a Reader component. In this way streams can be requested and captured (using for example *FFMPEG-writer* or *writer*) or resent using a different protocol (for example converting an RTMP stream to a HTTP stream). Proxy applications for Media Servers (RTMP-proxy, RTMPT-proxy, RTSP-proxy, HTTP-proxy) use this feature.

- Parameters:
  - string url: the url to retrieve, default: ""
  - bool auto-play: if true, instantly play the stream, default: true

9.5.1 HTTP-capture

Requests and receives a stream via HTTP. Keeps the container and generates chunks from it.

- Parameters:
  - int chunk-size: in bytes, the size of the chunks from the container to generate, default: 65536

9.5.2 HTTP-client

Requests and receives a stream via HTTP. Demultiplexes the container and generates frames.

- Parameters:
  - string format: the format of the container, if not defined, guess the format based on the extension in the URL, default: ""
  - bool M3U: if true, the URL links to an extended M3U file used by Apple Live HTTP Streaming. The URLs contained in the M3U response will be contacted in turn., default: false

9.5.3 RTMP-client

Requests and receives a stream via RTMP. Generates audio and video frames.

- Parameters:
  - bool mov-frame: if true, keep the frames in MOV/MP4 structure for AVC and AAC, default: false
  - int chunk-size: in bytes, the size of the chunks from the container to generate, default: 65536

9.5.4 RTMPT-client

Requests and receives a stream via RTMPT. Generates audio and video frames.

- Parameters:
  - int polling: in ms, the time interval between two successive HTTP POSTs during play, default: 1000

9.5.5 RTSP-client

Requests and receives a stream via RTSP. Generates audio and video frames.

9.6 media-server

Media Servers are complex components, listening to a specific TCP port for incoming connections and dynamically creating a session as nested component for each new connection. URLs in the request should be of the form: `<protocol>://<server-address>/<application>/`
For example `rtmpt://myserver.com/FILE/flash/example.flv` with server(`myserver.com`), application(`FILE`), media(`flash/example.flv`). Each session creates a nested component of the type *application* given in the URL. Typical applications are FILE, HTTP, RTMP-proxy, RTMPT-proxy, RTSP-proxy, HTTP-proxy.

- Parameters:
  - string interface: if defined, listen to incoming connections on this specific interface, default: ""
  - string media: path of the folder in which to search for requested files, default: dat/media

9.6.1 HTTP-server

Listens for incoming HTTP connections. Does not support pausing or seeking. In the URL use application HTTP for file based transfer of requested file. Use the application FILE@FORMAT (eg. FILE@FLV, FILE@WEBM) to change the container type. Example URLs: `http://myserver.com/HTTP/demo.mov`, `http://192.168.1.3/FILE@FLV/demo.mov`

- Parameters:
  - int port: listen to this port for incoming connections, default: 80
  - bool cache: if true, cache generated HLS segments, default: false
  - bool cached: if true, used cached segments and playlists when available, default: false
  - int segment: in seconds, the duration of one HLS segment, default: 10
  - int high: in kbit/s, the bitrate of the high quality HLS encoding, default: 1000
  - int medium: in kbit/s, the bit rate of the medium quality HLS encoding, default: 500
  - int low: in kbits/s, the bit rate of low quality HLS encoding, default: 200

9.6.2 RTMP-server

Listens for incoming RTMP connections. Supports pausing and seeking. Performs a cryptographical handshake to enable Flash Player to play H.264/AVC and MPEG4-Audio.

- Parameters:
  - int port: listen to this port for incoming connections, default: 1935

9.6.3 RTMPT-server

Listens for incoming RTMPT connections. This is the tunneled version of RTMP.

- Parameters:
  - int port: listen to this port for incoming connections, default: 80

9.6.4 RTSP-server

Listens for RTSP connections. Supports pausing, but not seeking.

- Parameters:
  - int port: listen to this port for incoming connections, default: 554

9.7 miscellaneous

9.7.1 GOP-splitter

Cyclically classifies each GOP by increasing the original xroute, so that xroute cycles through [route,route+split[

- Parameters:
  - int split: in how many parts to split the stream, default: 1
  - bool sync: if true, drop all frames until the first PPS/SPS/IDR packet, default: false

9.7.2 PCAP-writer

Captures packets using a filter and saves those to a file. Uses libpcap for capturing. Needs to have permission to access the interfaces (eg. sudo).
Availability: Unix

- Parameters:
  - string interface: the name of device on which to capture, default: eth0
  - string filename: the path of the file where to write the captured packets, default: ""
  - string filter: if defined, overwrite the built-in filter, default: ""
  - int port: the UDP port on which to listen for traffic (when using the built-in filter), default: 1234
  - int bitrate: in Mbps, estimated bitrate of stream which determines the underlying buffer size. When unsure of the bitrate, use a royal upper limit. If the debug reports dropped packets, increase this value., default: 20

9.7.3 YUV-display

Creates a rudimentary video player with no user controls save 'ESC' (which forcibly terminates the video). This component displays immediately any YUV frame it receives. Sirannon must be compiled with the option –with-libSDL to use this component.
WARNING: this component is in a beta version and might not function properly.

- Parameters:
  - int width: in pixels, if defined, the width of the video display, otherwise obtain the information from the MediaPackets containing the YUV frames (pPckt-¿desc-¿width), default: -1
  - int height: in pixels, if defined, the width of the video display, otherwise obtain the information from the MediaPackets containing the YUV frames (pPckt-¿desc-¿height), default: -1
  - bool full-screen: if true, display the video full screen, default: false

9.7.4 example

Example component demonstrating the basic API of a functional component

- Parameters:
  - int test: input variable, default: 5
  - bool foo: an example flag, default: false

9.7.5 fake-reader

Generates random YUV frames with specified dimensions and/or targetted bitrate.

- Parameters:
  - int width: the width of the fake frame, default: 1920
  - int height: the height of the fake frame, default: 1080
  - int bits-per-pixel: the number of bits per pixel of the fake frame, default: 12
  - double fps: the number of frames per second, default: 25.
  - int bitrate: if defined, overwrite the bitrate implied by the combination of fps and frame dimensions, default: -1
  - int duration: in ms, if defined the duration of the sequence, otherwise generate frames indefinately, default: -1
  - int mtu: if ¿ 0, divide each frame into packets of maximum mtu size, default: -1

9.7.6 frame-analyzer

This component parses elementary MPEG-1, MPEG-2, MPEG-4 or H.264/AVC streams to determine the frame type.

9.7.7 gigabit-transmitter

- Parameters:
  - string destination: the destination, default: 10.10.0.1:1234
  - int port: the source port, default: 4000
  - int mtu: the size of each generated packet (including headers), default: 1450
  - int bitrate: in megabits per seconds, bitrate of the generated stream, default: 100
  - int fps: number of frames per second, default: 25

9.7.8 live-reader

This component finds a component a runtime and creates a route from that component to this. It also buffers the received packets untill the component is scheduled. Using this technique a live captured stream can be tapped into and routed to this component.

- Parameters:
  - string url: if defined, create a route from the component out which uses this url, default: ""

9.7.9 restamp

This components buffers as many packets as the reordering delay and release them we correctly generated DTSs inferred from PTS only streams. This component is only useful when the DTS is corrupted or undefined. CAVEAT: Two packets with the same PTS will cause a RuntuimeError.

- Parameters:
  - int delay: the maximal reordering delay in packets, default: 2

9.7.10 statistics

This component generates at regular intervals information about the passing stream

- Parameters:
  - int interval: in ms, the amount of time between two reports, -1 disables the reports, default: 5000
  - string log: if defined, the path where to log information about the passing packets, default: ""
  - string label: if defined, label used for each entry in the log, this is needed when sending multiple statistics output to one single file, default: ""
  - bool append: if true, append data to the log, if false, overwrite the log, default: true
  - int overhead: the amount of header overhead from network headers to add to the packet size, default: 0
  - bool draw: if true, when an end-packet is received, draw a graph of the bandwidth, requires that *log* is defined, default: false

9.7.11 svc-analyzer

This component constructs an inter layer bandwidth comparison at the end of a passing svc stream. A 3D graph is constructed with for each layer a cube scaled with the relative weight of the layer in the total bandwidth. The graph is generated as a maple worksheet (`www.maplesoft.com`).

- Parameters:
  - string filename: the path of the maple worksheet with extension .mws, default: ""
  - int dimension: in how many dimensions to scale the cube (1, 2 or 3), default: 1

9.7.12 time-splitter

Cyclically classifies each time interval by increasing the original xroute, so that xroute cycles through [route,route+split[

- Parameters:
  - int split: in how many parts to split the stream, default: 1
  - int interval: in ms, the duration of one part, default: 10000
  - bool key: if true, split only if the frame is a keyframe, if false, omit this condition, default: false

9.8 multiplexer

Multiplexers buffer MediaPackets coming from different sources and multiplex those based on the DTS of each MediaPacket. Unmultiplexers perform the reverse operation and restore the original streams.

- Parameters:
  - int delay: in ms, the minimal amount of data present for each stream of the multiplex before the next packet is released, default: 1000
  - int streams: the number of different streams required before multiplexing, -1 omits this requirement, default: -1

9.8.1 FFMPEG-multiplexer

Joins packets from different source into a container (no file!) supported by FFMPEG (eg. FLV, WEBM, MP4). The container is released in a stream of MediaPackets containing chunks of the container.

- Parameters:
  - int chunk-size: in bytes, the maximum size of each chunk, default: 262144
  - string format: format of the container given as file extension (eg. flv, webm, mov, mp4, avi), default: flv
  - bool streamed: if true, disables seeking backwards into the generated chunks, default: false

9.8.2 TS-multiplexer

Multiplexes audio and video into an MPEG Transport Stream (TS). CAVEAT: If the video or audio come from different readers, you must set the parameter streams or initial-delay, or the component will throw an error.

- Parameters:
  - int mtu: in bytes, the maximum size of an aggregated packet, typically for an iMTU of 1500 it means 7 Transport Stream packets (1370 bytes), default: 1500
  - bool aggregate: if true, several transport stream packets will be aggregated until their size would exceed the iMTU, default: true
  - int shape: in ms, the amount of media to multiplex each pass, this value overwrites the value of delay, default: 200
  - int pcr-delay: in ms, how much the PCR is in advance of the DTS, the pcr-delay must be lower than the initial DTS of the reader component, default: 400

- int mux-rate: in kbps, if -1, no fixed multiplexing rate, if 0, the component will guess the mux rate based on the bitrate of the input, if ¿ 0, use this value as mux-rate, default: -1
- bool continuous-timestamps: if true, use the PCR clock as timestamp for the transport stream, if false, use the timestamps of the PCR stream as timestamps for the transport stream, default: true
- bool interleave: if false, all transport stream packets belonging to one frame from a PID are consecutive, if true, they are interleaved with transport stream packets from other PIDs (e.g. audio), default: true
- bool psi-on-key: if true, generate an extra PSI triplet (SDT, PAT and PMT) before the start of every key frame of the PCR stream, default: false

9.8.3 std-multiplexer

Joins packets from different sources by only ensuring that the DTS and other timing information rise monotonely. This is a requirement for many protocols and containers.

9.8.4 unit-multiplexer

Joins packets from different sources by only ensuring that the unitnumber rises monotonely.

9.9 packetizer

- Parameters:
  - string tracefile: if defined, the path of trace where to log information about the split packets, default: ""
  - bool trace-pts: if true, write the PTS instead of the DTS in the log file, default: false

9.9.1 AC3-packetizer

Packetizes AC3 audio frames into packets suitable for RTP as defined in (draft) RFC 4184.

- Parameters:
  - int mtu: in bytes, the maximum size of a network packet, default: 1500
  - bool draft: if true, fill out the header according to the draft version of RFC 4184; this is the version supported by live555/vlc/mplayer, default: true

9.9.2 AMR-packetizer

AMR and AMR-WB packets are aggregated into packets of (near) MTU size according to RFC 3267
WARNING: this component is in a beta version and might not function properly.

- Parameters:
  - int mtu: in bytes, the maximum size of a network packet, default: 1500
  - int maxptime: in milliseconds, if ¿= 0, the maximum duration of an aggregate frame, if ¡ 0, omit this requirement, default: 200

9.9.3 AVC-packetizer

Packetizes H264 frames into packets suitable for the network as defined in RFC 3984.

- Parameters:
  - int iMTU: in bytes, the maximum size of a network packet, default: 1500
  - bool aggregate: if true, aggregate small packets into one network packet, default: false

9.9.4 MP2-packetizer

Packetizes MPEG1&2 audio and video frames into packets suitable for the network as defined in RFC 2250.

- Parameters:
  - int mtu: in bytes, the maximum size of a network packet, default: 1500

9.9.5 MP4-packetizer

Packetizes MPEG4 audio and video frames into packets suitable for the network as defined in RFC 3640.

- Parameters:
  - int mtu: in bytes, the maximum size of a network packet, default: 1500
  - bool aggregate: if true, aggregate small packets into one network packet, default: false

9.9.6 PES-packetizer

PES packetization is the required format for transport streams TS-multiplexer. All packets belonging to one frame are aggregated into one PES-packet. If the total size of the frame is larger than 65500 bytes two or more PES-packets are generated unless the option zero-length is set.

- Parameters:
  - int delta: in ms, the amount of time to add to DTS/PTS to make sure is substractable by a PCR, default: 0
  - bool insert-AUD: if true, add AUDs before each H.264 frame, default: false
  - int audio-per-PES: number of audio frames per PES-packet, default: 1
  - bool zero-length: video only, if true, set 0 as PES-packet-length and generate a single PES-packet per frame, default: true

9.9.7 RTMP-packetizer

This packetizer produces an RTMP chunk stream from the ingested MediaPackets. Such a stream can directly be sent to a Flash Media Server. Unpacketization of this stream is done by the component RMTP-client internally.

- Parameters:
  - int chunk-ID: the identifier for this RTMP chunk stream, default: 5
  - int stream-ID: the identifier to which global stream this RTMP chunk stream belongs, default: 1
  - int chunk-size: in bytes, the maximum size of each RTMP chunk, default: 128

9.9.8 default-packetizer

For each new stream creates at runtime a fitting packetizer depending on the codec or the parameter type.

- Parameters:
  - string type: if defined, force every packetizer to be of this type, useful for forcing PES-packetizer for example, default: ""

9.9.9 sirannon-packetizer

This simple and generic packetizer can handle any content. However, this packetization is internal to Sirannon and is not recognized by other players. Use sirannon-unpacketizer to unpacketize this stream again.

- Parameters:
  - int mtu: in bytes, the maximum size of a network packet, default: 1500

9.10 reader

Readers form the primary source of data in the sirannon. When the file reaches its end, a reader either closes the file, generating an end-packet or loops creating a reset-packet. A reader checks if no buffers downstream (typically a scheduler buffer) are full before processing the next frame.

- Parameters:
  - string filename: the path of the file to open, default: ""
  - int loop: the number of times to play the video, -1 being infinite, 0 interpreted as 1, default: 1
  - int video-route: the xroute that will be assigned to packets containing video, default: 100
  - int audio-route: the xroute that will be assigned to packets containing audio, default: 200
  - int dts-start: in ms, the timestamp that will be added to the DTS & PTS of each frame, default: 1000

9.10.1 avc-reader

Reads in a raw H264 video file. Each generated *MediaPacket* contains one H.264 NAL-unit, possibly generating multiple *MediaPackets* per frame.

- Parameters:
  - bool skip-SEI: if true, ignore the SEI NAL-units, default: false
  - bool skip-AUD: if true, ignore the AUD NAL-units, default: false
  - bool mov-frame: if true, convert NAL-units into MP4/MOV/F4V frames, default: false

9.10.2 basic-reader

Reads in any container in large chunks. Its main use is the HTTP transmission in chunks of a container.

- Parameters:
  - int chunk-size: in bytes, the maximum size of a chunk, default: 64000
  - int length: in ms, if ¿ 0, provides the reader with the duration of the file so it can guess the DTS, default: 0

9.10.3 ffmpeg-reader

Reads in a wide variety of containers supported by ffmpeg. Audio and video are put into different MediaPackets. Per cycle, the reader processes one video frame (if present) and associated audio, possibly generating multiple packets. A separate end- or reset-packet is generated for audio & video. Note, ffmpeg-reader can also process audio only files.

- Parameters:
  - bool video-mode: if true, video will be read from the container, if false video will be ignored, default: true
  - bool audio-mode: if true, audio will be read from the container, if false audio will be ignored, default: true
  - int min-ts: in ms, the timestamp to jump to into the stream, -1 implying no seek, default: -1
  - int max-ts: in ms, the maximum timestamp to read from the file, -1 implying no limit, default: -1
  - int dts-start: in ms, specifies the value of the timestamp of the first frame, default: 1000
  - bool add-parameter-sets: H.264/AVC only, if true, extract the parameter sets from the container and insert them into the stream, default: true
  - bool repeat-parameter-sets: H.264/AVC only, if true, repeat the parameter sets before each IDR frame, default: false
  - bool mov-frame: MOV/MP4/F4V container only, if true, keep frames in the format of the container, as opposed to annex-B H.264/AVC streams with start codes before each NAL unit, default: false
  - bool skip-SEI: if true, remove SEI NALUs from the stream, default: false
  - bool skip-AUD: if true, remove AUD NALUs from the stream, default: false
  - bool fix-PTS: if true, parse H.264 frames to extract the POC from which to calculate the correct PTS, default: false

9.11 receiver

Receivers provide the interface from the network for the following protocols: UDP, TCP and RTP/UDP.

- Parameters:
  - int port: reception port, default: 5000
  - int video-route: the xroute that will be assigned to packets containing video, default: 100
  - int audio-route: the xroute that will be assigned to packets containing audio, default: 200
  - int buffer: if ¿0, the size of the protocol buffer, your OS must still accept this value, check 'UDP Buffer Sizing' in Google for more information, default: 0
  - bool extension: if true, the additional sirannon header is parsed from the packet. Caveat, if the header was not present the stream will be corrupted except for RTP. Conversely, if the header was present and this value is false, the stream will be corrupted except for RTP, default: false
  - bool multicast: if true, join a multicast address (not for TCP), default: false
  - string multicast-server: multicast address (not for TCP), default: ""

9.11.1 RTP-receiver

Receives RTP streams using the open source library jrtplib. RTCP packets are automatically generated. The additional header is parsed from the RTP header extension if present.

- Parameters:
  - string tracefile: if defined, the path of the trace where to log information about the received packets, default: ""
  - int buffer: in bytes, the size of the underlyinhg UDP buffer, increase this value when receiving high bitrate streams, make sure yours OS accepts such large values (see `http://www.29west.com/docs/THPM/udp-buffer-sizing.html`), default: 8388608
  - string hash-file: if defined, the path of a file in which the content of a header extension with ID(EXT-HASH: 0xB) is written, default: ""

9.11.2 TCP-receiver

Provides a non-blocking TCP socket from the network.

- Parameters:
  - bool connect: if true, connect to the server, if false, listen for an incoming connection, default: false

9.11.3 UDP-receiver

Provides a non-blocking UDP socket from the network.

9.12 scheduler

Schedulers introduce real-time behavior to the stream (readers generate packets at an arbitrary speed). It also introduces correct real-time behavior to streams coming from several receivers. It buffers all incoming packets and releases them at the rate set by the packet's decoding time stamp (DTS). Some schedulers introduce shifts to these time stamps order to obtain for example a smoother bandwidth usage.

- Parameters:
  - int delay: in ms, the scheduler starts working delay ms after the first packet, default: 100
  - int buffer: in ms, the size in time of the buffer, default: 1000
  - int absolute-delay: in ms, start scheduling absolule-delay ms after the start of Sirannon, 0 disables this, default: 0
  - bool pause: if true, pause scheduling until the function play is called, default: false
  - double speed: the factor at which to schedule slower or faster than real-time, default: 1.0
  - bool precise: if true, run the scheduler in a seperate thread with a quantum of 1 ms for precise timings, default: false

9.12.1 basic-scheduler

Simplest form in which the packets are sent at the time solely indicated by the DTS. This leads to a burst of packets for each frame since all the packets have the same DTS.

9.12.2 frame-scheduler

The packets belonging to a frame are smoothed in time over the duration of the frame (instead of sending an entire frame in one burst). Caveat, make sure the delay is longer than the duration of one frame!

9.12.3 gop-scheduler

The packets belonging to a GOP are smoothed in time over the entire duration of that GOP. Caveat, make sure the delay is longer than the duration of one GOP!

9.12.4 svc-scheduler

The packets belonging to one temporal pyramid are smoothed in time over the entire duration of that pyramid. Caveat, make sure the delay is longer than the duration of one pyramid!

9.12.5 window-scheduler

The packets are smoothed in time over a fixed non-sliding window. Caveat, make sure the delay is longer than the duration of one window!

- Parameters:
  - int window: in ms, the size of the fixed window, default: 900

9.13 system

9.13.1 block

This component takes Sirannon configuration and loads this configuration within itself. It allows grouping of components into one component that can be used in many configurations. Packets are sent to or received from the surrounding scope of the block using the *in* component for input and *out* component for output.

- Parameters:
  - string config: path of the sirannon configuration file, default: ""
  - string param1: if defined, command line parameter 1 for the configuration file, default: ""
  - string param2: if defined, command line parameter 2 for the configuration file, default: ""
  - string param3: if defined, command line parameter 3 for the configuration file, default: ""
  - string param4: if defined, command line parameter 4 for the configuration file, default: ""
  - string param5: if defined, command line parameter 5 for the configuration file, default: ""
  - string param6: if defined, command line parameter 6 for the configuration file, default: ""
  - string param7: if defined, command line parameter 7 for the configuration file, default: ""
  - string param8: if defined, command line parameter 8 for the configuration file, default: ""
  - string param9: if defined, command line parameter 9 for the configuration file, default: ""
  - string param10: if defined, command line parameter 10 for the configuration file, default: ""
  - string param11: if defined, command line parameter 11 for the configuration file, default: ""
  - string param12: if defined, command line parameter 12 for the configuration file, default: ""
  - string param13: if defined, command line parameter 13 for the configuration file, default: ""
  - string param14: if defined, command line parameter 14 for the configuration file, default: ""
  - string param15: if defined, command line parameter 15 for the configuration file, default: ""

9.13.2 discard

Any received packet will be deleted.

9.13.3 dummy

This component does absolutely nothing!

9.13.4 in

This component finds a component a runtime and creates a route from that component to this. Using this technique a live captured stream can be tapped into and routed to this component.

- Parameters:
  - string url: if defined, create a route from the component 'out' which uses this url, default: ""

9.13.5 out

This component declares a url to be associated with this component and routes to components subscribing to this url

- Parameters:
  - string url: if defined, links this component with this url, when a new component 'in' is made, default: ""

9.13.6 sink

When this component receives an end-packet the program will terminate gracefully. In case of both an audio and a video stream, an end-packet for both audio and video needs to be received (if set).

- Parameters:
  - bool video-mode: if true, a video end-packet must be received, default: true
  - bool audio-mode: if true, an audio end-packet must be received, default: false
  - int count: if larger than 0, the number of end-packets which must be received, default: 0

9.13.7 time-out

If this component does not receive any packet within a given interval after the last packet, forcibly terminate the program or generate an end-packet.

- Parameters:
  - int time-out: in ms, the maximum interval in which no packets are received, default: 1000
  - bool kill: if true, terminate the program, if false, generate an end-packet, default: true

9.14 transformer

These components transform the received frames for example by transcoding or changing the header format

9.14.1 ffmpeg-decoder

This components decodes a video sequence and generates a stream of YUV packets

- Parameters:
  - bool reset-on-reset: if true, reset the decoder when a reset-packet is received, default: true
  - bool frame-copy: if true, use frame copy as basic error concealment, default: false

9.14.2 frame-transformer

Handles the mess caused by the MP4 container that strips and merges MP4A and H.264 frames, while TSs and RTP keep the frames in the original format. Reconstructs the header based on the meta data.

- Parameters:
  - bool ES: if true, convert the packets to ES format, if false, convert to MP4 format, default: true

9.14.3 transcoder-audio

Decodes received packets and reencodes them using the specified settings. This component runs the transcoding in a seperate thread and may consume all CPU.

- Parameters:
  - string output-codec: the target codec, default: mp4a
  - int bitrate: the target bitrate, -1 implies maintaining the same bitrate, default: -1
  - string target: if defined, use specific encoding settings for this target, values: iphone, ipad, youtube, default: ""
  - int route: xroute assigned to the transcoded audio frames, default: 200

9.14.4 transcoder-video

Decodes received packets and reencodes them using the specified settings. This component runs the transcoding in a seperate thread and may consume all CPU. The component works best effort and can not garantee realtime transcoding.

- Parameters:
  - string output-codec: the target codec, default: h264
  - int bitrate: the target bitrate, -1 implies maintaining the same bitrate, default: -1
  - int width: the target width, 0 implies maintaining the same width, default: 0
  - int height: the target height, 0 implies maintaining the same height, default: 0
  - int framerate: the new frame rate which must be less or equal to the current frame rate, -1 implies maintaning the same frame rate, default: -1
  - bool mov-frame: if true, generate H.264 in MOV/MP4 frames, default: false
  - string target: if defined, use specific encoding settings for this target, values: iphone, ipad, youtube, default: ""

9.15 transmitter

Transmitters provide the interface to the network for the following protocols: UDP, TCP and RTP/UDP. These components send the packets without any delay or buffering.

- Parameters:
  - int port: source port, default: 4000
  - string destination: the IP address of the receiver, default: 127.0.0.1:5000
  - int buffer: if ¿0, the size of the protocol buffer, your OS must still accept this value, check 'UDP Buffer Sizing' in Google for more information, default: 0
  - bool extension: if true, add an additional header with sirannon frame numbers to the packet, although making it incompatible with a standard player (except for RTP), default: false
  - int multicast-TTL: the TTL when sending to a multicast destination (not for TCP), -1 disables this, default: -1

9.15.1 RTP-transmitter

Provides the RTP/UDP protocol using the open source library jrtplib RTCP packets are automatically generated. The extra information of the sirannon (sirannon-extension) is added as a header extension in RTP packets and this keeps it compatible with a standard player.

- Parameters:
  - bool pts: if true, the streamer uses the PTS of a packet instead of the DTS as time stamp in the RTP header. This can solve the problem where VLC ometimes interprets the RTP timestamp as a PTS instead of a DTS, default: false
  - int payload: payload type (PT) of the RTP packets, -1 means leaving the decision to the component, default: -1
  - string tracefile: if defined, the path of the trace where to log information about the sent packets, default: ""
  - int mtu: in bytes, the maximum packet size accepted by the RTP session, default: 1500
  - bool force-sequence-number: if true, force the RTP sequence number to follow the unitnumber, hence if you remove packets beforehand, the sequence number will also have gaps, if false, use the default RTP implementation, default: false

9.15.2 TCP-transmitter

Provides a non-blocking TCP socket to the network.

- Parameters:
  - bool connect: if true, connect to the server, if false, listen for an incoming connection, default: true

9.15.3 UDP-transmitter

Provides a non-blocking UDP socket to the network.

9.16 unpacketizer

9.16.1 AMR-unpacketizer

Unpacketizes an AMR-NB/-WB stream packed according to RFC 3267.
WARNING: this component is in a beta version and might not function properly.

9.16.2 AVC-unpacketizer

The fragmented or aggregated frames are transformed again into their original NAL units, as defined in RFC 3984. It performs the reverse operation of the AVC-packetizer.

- Parameters:
  - bool startcodes: if true, add startcodes to the unpacked NALs, default: true
  - bool strict-annex-b: if false, all startcodes will be 4 bytes, default: true

9.16.3 MP2-unpacketizer

Unpacketizes a stream fragmented with MP2 packetization for RTP (RFC 2250) by for example (MP2-packetizer).

9.16.4 MP4-unpacketizer

Unpacketizes a stream fragmented with MP4 packetization for RTP (RFC 3640) by for example MP4-packetizer.

9.16.5 PES-unpacketizer

One or more PES-packets belonging to one frame are split again into the original parts. It performs the reverse operation of the PES-packetizer.

9.16.6 sirannon-unpacketizer

Unpacketizes a stream fragmented by the internal packetizer of Sirannon (sirannon-packetizer).

- Parameters:
  - bool recover-frame: if true, unpacked parts of a damage frame instead of discarding the entire frame, default: false
  - bool error-on-loss: if true, throw an exception when packet loss occurs, default: false

9.17 writer

9.17.1 basic-writer

Writes the content of each received packet to a file and deletes the packet. When an end-packet is received, the file is closed.

- Parameters:
  - string filename: the path of the file where to write to, default: ""
  - bool flush: if true, flush the IO buffer after each write, default: false
  - bool fragmented: if true, after each reset, close the container and open a new container with a name of the form e.g. demo-0.avi, demo-1.avi, demo-2.avi, etc., default: false
  - bool complete: if true, if the file is not closed when entering the destructor, delete the file, default: false

9.17.2 ffmpeg-writer

Joins packets from different source into a container format supported by FFMPEG (eg. FLV, WEBM, MP4). The container is written to a file.

- Parameters:
  - string filename: the path of the file where to write to, default: ""
  - int initial-delay: in ms, the minimal delay between the first received packet and first multiplexed packet, default: 0
  - int delay: in ms, the minimal amount of data present for each stream of the multiplex before the next packet is written, default: 1000
  - int streams: the number of different streams required before multiplexing, -1 omits this requirement, default: -1
  - string format: if defined, overrules the format determined by the extension in the filename, default: ""
  - bool fragmented: if true, after each reset, close the container and open a new container with a name of the form e.g. demo-0.avi, demo-1.avi, demo-2.avi, etc., default: false