xStreamer: Modular Multimedia Streaming

ABSTRACT
Forecasts predict a significant boom in video applications over the internet and dedicated networks. Because of this, video increasingly features as use case in network related projects and research, which requires the creation of new streamers or the extension of current open source solutions, leading to significant redundant work. The current selection of open source streamers is limited, providing an opportunity for the xStreamer which is a flexible and modular streamer avoiding the redundant work. The modularity, inspired by the Click Modular Router project, goes beyond the mere modular programming offered by current solutions and manifests itself in how the user controls the streamer. The user configures the streamer by combining a collection of components in a directed graph. Each component performs basic functions such as reading video frames from a file, packetizing frames into smaller packets or multiplexing video and audio into a transport stream.

Key Words
Streaming, Modular, Multimedia, Open source

1. INTRODUCTION
Video applications are becoming ubiquitous on the present day internet with about 60% of typical users having watched online video before and nearly 20% to do so daily in 2007 [15]. Forecasts such as [8] predict a significant boom in video over the internet and dedicated networks both absolutely and relatively to other applications such as peer-to-peer and browsing. Because of this predicted explosion, video increasingly features as use case in network related projects and research. Many researchers spend considerable effort in creating their own streamer or extending current open source solutions, leading to significant redundant work. In order to avoid this redundant work, the xStreamer intends to be a flexible and modular open source streamer.

1.1 Related Work
The selection of current open source streamers which support both video and audio is limited, with VLC Media Player [4], Darwin Streaming Server [5] and Helix DNA Server [18] being the foremost solutions. VLC Media Player (VLC) distinguishes itself by offering a wide selection of codecs and network protocols through the use of among others the open source libraries FFmpeg [6] and Live555 [9]. Since the xStreamer uses these two libraries as well, it can also handle a good selection of codecs and protocols. Although VLC supports streaming, it remains a media player in the first place. The key strength of Darwin Streaming Server (DSS) is the user-friendly web interface to dynamically control the streams the server offers through the RTSP protocol (Real Time Streaming Protocol [21]). The xStreamer offers the same RTSP protocol but without the dynamical control offered by a web interface. Before DSS can offer content, it requires hinting, a post-process applied to bit streams, while the xStreamer has no such requirement. Darwin Streaming Server and Helix DNA Server are related because both have a commercial enterprise, Apple and RealNetworks respectively, as main contributor. The Helix DNA Server as well offers streams using RTSP and manageable by a web interface. However, it only supports the video codec RealVideo, a proprietary format from its parent RealNetworks, included in binary form. The xStreamer distinguishes itself by providing a modularity that goes beyond the mere modular programming offered by the current open source solutions and that manifests itself in how the user controls the streamer.

1.2 Modularity
The modularity is inspired by the Click Modular Router project [16] and operates by offering components which perform basic functions such as reading video frames from a file, classifying packets based on their frame type or randomly discarding packets with a given probability, etc. The user builds the streamer by combining a collection of these components in a directed graph: the vertices form the components and the directed edges form flows of packets from one component to another. Figure 1a shows as an example the graph of a simple streaming solution: a reader parsing a video file and outputting one packet per frame, a packetizer splitting each packet into a series of packets, each smaller than for example 1500 bytes, a scheduler releasing each packet at the instant corresponding with the timestamp of the packet and finally, a transmitter adding network headers and sending the packet over the network. By means of simple changes to the graph, the streamer performs addi-
tional or similar functions, providing a part of the flexibility of the xStreamer. For example, the streamer in figure 1b performs a similar function to figure 1a, but instead of sending the packets over a single connection, the component classifier splits the flow of packets into three flows based on for example the frame type (I, P or B), with each flow sent over a separate network connection.

Additionally, the xStreamer has components performing the reverse operations of streaming, such as receiving, unpacketizing and writing, allowing it to offer a proxy function which redirects for example the three connections from figure 1b into a single new connection or a capture function which redirects the received packets to a file. Figure 2 shows a configuration performing the reverse operation from figure 1b and providing a proxy as well as a capture function. In the paper [13] the xStreamer is used to stream H.264/AVC (Advanced Video Coding [11]) content over RTP (Real-time Transport Protocol [20]) on the server side and to capture and save this stream on the client side.

Since the scheduler component introduces real time behaviour to the passing flow of packets, without such a component, the graph can function as a video tool, working offline, for example in order to impair a video bit stream with a given packet loss rate as shown figure 3. The program works as fast as possible in this case, processing a stream in a fraction of the actual duration of the bit stream. Video tool possibilities include controlled or random impairment, elementary stream extraction, statistics (average bit rate, packet size, etc.), plots of the bit rate evolution and transcoding (experimental but future work will improve this).

In addition, the scheduler component can introduce simulated time behaviour, allowing the streamer to function as simulator where time increases with arbitrary increments of for example 1 ms. This choice between real time, simulation and offline function, provides the xStreamer with further flexibility.

1.3 Overview

The remainder of this paper is structured as follows. Section 2 provides a detailed description of the architecture concerning the components, connections, configuration and performance. The next section presents an overview of the different categories of components such as readers, packetizers, multiplexers, etc. Section 4 presents in detail a use case which allows distributed transcoding. Continuing, the following section provides a short overview of the features of the xStreamer, including the supported codecs and protocols. The distribution of the xStreamer also contains a simple user interface which section 6 describes. Finally, section 7 briefly discuss how to obtain the distribution and the last section ends with the conclusion.

2. ARCHITECTURE

2.1 Components

Components are the vertices in the directed graph performing basic functions. Section 2.5 provides the details about the actual implementation in the program. Beyond the basics described in the introduction, components can have two additional features.

On the one hand, the framework provides nested components which allow grouping a specific combination of components into a single component. For example, figure 4 shows a configuration testing an experimental network buffer with three identical components block containing nested components. Each component block has the same internal configuration but uses for example different parameters such as the video file. When the nested component out receives a packet, it forwards the packet to the connections leaving the surrounding component block.

On the other hand, components are not mere static ob-
2.2 Connections

Routing the packets from one component to another is determined by a routing label. Each packet is associated with one routing label while each connection is associated with a set of routing labels. When routing a packet, the component sends the packet over each connection which contains the routing label, duplicating the packet if there is more than one matching connection. The routing label "0" functions as wildcard: as label of a packet it forces the component to multicast the packet over each connection leaving the component, while as a label of a connection, it forces the downstream component to send (a copy of) each packet over this connection. In the simple case of only one connection leaving a component, a routing label "0" is sufficient for the connection and therefore is the default label. Internally, components receive packets by the receive method and output packets using the route method which performs the above described routing algorithm. Finally, if a component has no outgoing connections, it silently drops any packet it receives.

For example, figure 5 shows a differentiated impairment configuration. The reader, generating both audio and video frames, puts routing label "100" on video frames and label "200" on audio frames. By setting the labels of the two connections leaving the reader accordingly to "100" and "200", the flow of packets splits into a flow of video frames and a flow of audio frames. The component frame-classifier allows further splitting of the video flow into flows of I, P and B-frames by adding an offset to the label of each packet depending on the type of frame it carries (0 for I-frames, 1 for P-frames and 2 for B-frames) and by setting the labels of the three connections leaving the frame-classifier accordingly to "100", "101" and "102". The flows of I, P and B-frames each pass through a separate discard component using a different loss probability: 2% for B-frames, 0.5% for P-frames and 0.1% for I-frames. The connections leaving each discard component do not split and hence have the default label "0". The flow of audio frames with label "200" directly enters the writer component and does not suffer any impairment. The final effect of this configuration is impairing the video with a different probability depending on the frame type while leaving the audio unaffected. In order to simplify the example, the figure omits the multiplexing required to write both audio and video into a container.

2.3 Scheduling

The receive method triggers components, allowing them to process the incoming packet and output one or more packets during the same function call, causing a cascade of receive function calls in a path of connected components. However, components such as schedulers, buffer the received packet instead of directly routing it again, ending the cascade. Components such as receiver and reader never receive packets from another component because they are always at the top of the directed graph. In order to trigger such components, the framework provides a process method which the program triggers regularly for components marked as scheduled. In this method the component can for example analyse its buffers or parse frames from a file and route the corresponding packets.

The subsequent calls of all scheduled components of the configuration by the process forms one cycle. The duration of each cycle is determined by the crucial parameter named quantum. If the processing of one cycle takes less than this quantum, the process sleeps during the difference, lowering the CPU load. However, if the cycle takes longer than one quantum, for example due to other processes competing for the CPU, the process lags behind and will eventually warn that the system is too slow.

Each scheduled component attempts to process an amount of data corresponding with one quantum, for example if the quantum is 200 ms, the reader component parses 200 ms of video and audio data from the bit stream during one cycle. The quantum presents a trade-off between timing accuracy and CPU load: the lower the quantum, the better the timing accuracy of the streamer which equals one quantum. If the quantum is 0, the process is in active loop, using all CPU.
resources, but capable of producing a very smooth bandwidth on the network or reacting instantaneously to received packets from the network. Section 2.6 about performance presents the trade-off using measured results.

Additionally, the program can work in simulated time, achieving this by changing the function which provides the current time by returning the value of a simulated clock. Each cycle of the process, the simulated clock increases with a value determined by the parameter named simulation. For example, in the configuration, seen in figure 4, the process runs in simulated time using a step of 1 ms to simulate the behaviour of an experimental network buffer.

2.4 Configuration

The directed graph of components and connections which conveys the configuration is inputted into the process as an XML (Extensible Markup Language [7]) based file. Figure 6a shows the code of a simple configuration which extracts an elementary video stream from a video container and saves it to a file and figure 6b shows the corresponding graph. In the configuration file the root element config contains two types of elements: component and route.

On the one hand, the element component has as attributes name, a unique identifier of the component within one configuration, and type whose meaning is straightforward. Two types of nested elements configure the element component: param and gfx. The element param defines a parameter by providing the name, data type and value. For example, the first component in figure 6a is the component ffmpeg-reader which parses a video container and extracts elementary video streams. The parameters of this component include among many others, the parameter filename whose type is a string and has a special value "$1" which means taking the first command line argument specified after the XML filename. Use of command line parameters makes the XML configuration more flexible, as for example the same configuration can be reused for different video containers on the file system. The process ignores the other nested element gfx while the GUI (Graphical User Interface), described in section 6, uses it to position the component. Additionally, the example configures two other components writer and sink. The sink component allows terminating the program when the bit stream is fully processed, figures 3-5 require such a sink component at the end of the directed graph, but omit it in order to keep the example simple.

On the other hand, the element route presents the directed connection between two components. The example includes two routes, one from ffmpeg-reader to writer and one from writer to sink. Since the connections are simple, each connection has the default routing label 0, defined by the attribute xroute of the element.

Additionally, the root element config has three attributes quantum, simulation and seed. The attributes quantum and simulation provide the values of the corresponding parameters described in the previous section 2.3. The attribute seed determines the seed for the random number generator, with the default value of 0 implying initializing it with the current time.

2.5 Implementation

Internally, the xStreamer implements components as classes in C++. Figure 7 shows the class declaration of the base class media_processor common to all components. For each component the program provides a new class inheriting from the base class media_processor. The class media_packet implements packets handled by the components. The class processorManager, given as argument in the constructor of a component, allows for the features such as nesting and dynamic creation of components. As described in the previous sections, components receive packets by the receive method and output packets using the route method. The process triggers at the start the method init once and triggers during each cycle the method process for scheduled components. Currently the program only operates in a single thread, however, future work will allow running each component, if advantageous, in a separate thread.

The plethora of codecs, containers and network protocols forms a too complex whole and the program uses the invaluable aid of 4 open source libraries:

- avformat & avcodecs: allows parsing, decoding and encoding a wide variety of containers and codecs, designed by the open source project FFmpeg [6].
- jrtplib: provides the RTP protocol, both for transmitting and receiving data, developed by Jori Liesenborgs [14].
- live555: provides the RTSP protocol which can set up
class media_processor
{
public:
    media_processor( const string& sName, processorManager* pScope );
    virtual ~media_processor();
protected:
    void route( media_packet* pckt );
    virtual void init();
    virtual void process();
    virtual void receive( media_packet* pckt );
    ...}

Figure 7: Declaration of the class media_processor which implements the base class for each component.

and manage a collection of RTP sessions, designed by Ross Finlayson [9].

- expat: enables parsing of XML based files [1].

Finally, the program is written for UNIX systems but compiles under Windows as well using Cygwin [17]. The source distribution includes the source code of the libraries avformat, avcodec, jrtplib and live555, however, the user can choose to use a version of the library present on his or her system instead.

2.6 Performance

As bit rates increase continuously with the advent of HD (High Definition), streamers must be able to cope with increasingly higher bit rates. The CPU load of the xStreamer depends on the chosen quantum, as explained in the previous section 2.3. Figure 8 shows the average CPU load when streaming in function of the bit rate, given for different quanta and together with the performance of VLC as reference. The test platform is an Athlon 64 3200+ system running Linux and using a gigabit ethernet card. The results show the xStreamer uses slightly more CPU than VLC, in part due to the overhead of the true modularity and can handle streams up to 140 Mbit/s using 50% CPU on the test platform. Additionally, it demonstrates that the CPU load varies somewhat but not significantly between a quantum of 5 and 100 ms and hence, the program uses 5 ms as default quantum.

Table 1 presents the limits of the xStreamer on the same test platform when trying to stream as fast as possible by not using a scheduler component. The program can stream content this way up to 275 Mbit/s using MPEG transport streams (MPEG-TS [12]) and bit rates up to 400 Mbit/s using no multiplexing. When the streams are sent to the loopback interface, the xStreamer can handle even higher bit rates on the test platform.

<table>
<thead>
<tr>
<th>Mode</th>
<th>Maximum bit rate (Mbit/s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>TS over RTP</td>
<td>270</td>
</tr>
<tr>
<td>Raw over RTP</td>
<td>388</td>
</tr>
<tr>
<td>TS over RTP, loopback</td>
<td>377</td>
</tr>
<tr>
<td>Raw over RTP, loopback</td>
<td>617</td>
</tr>
</tbody>
</table>

Table 1: Maximal bit rate manageable by the xStreamer on the test platform with or without multiplexing and sent over either the local network or the loopback interface.

- packetizer: splits or aggregates consecutive frames into packets no larger than the Maximum Transfer Unit (MTU), while unpacketizer provides the reverse operation.

- multiplexer: multiplexes streams emitted by different components into one stream, for example multiplexes video and audio into a single MPEG transport stream.

- scheduler: schedules incoming packets to the desired real time behavior, for example distributing the packets per frame, GOP (Group Of Pictures), fixed window, etc.

- transmitter: provides a sending interface, for example RTP or TCP, to the network

- receiver: provides a receiving interface, for example RTP or TCP, from the network.

- writer: writes all received packets to a specified file. When creating a container, for example an MPEG container, the stream should be accordingly multiplexed beforehand.

3. COMPONENT OVERVIEW

The various components which the program offers fall into different categories, with all components from a category providing a similar basic step such as reading, packetizing, multiplexing, etc. At the end of this paper, table 2 lists a selection of components sorted by category and indicates per component if it is scheduled (see section 2.3). The following gives an overview of the 11 different categories of the program:

- reader: reads and parses frames from a container or elementary stream.
Figure 9: Grid of nodes, each running xStreamer, allowing the real time transcoding of a stream. The server accepts a stream from the network and distributes it over transcoding nodes which process the streams as fast as possible. The proxy combines the transcoded streams into one and streams it over the network to a media player.

- **classifier**: changes the routing label of incoming packets based on some property such as frame type or based on a given probability.
- **analyser**: aggregates data, such as average bit rate, about the stream sent through the component.
- **system**: various components providing basic functions such as dropping packets and terminating the program.
- **high-level**: offers high-level solutions such as streaming without having to construct the underlying graph of components explicitly.

4. **USE CASE: REAL TIME TRANSCODING**

The forecast in [8] predicts an explosion of video not only bandwidth wise, but also across an increasingly wide variety of platforms, requiring the use of transcoding to adapt bit streams to the requirements of the end devices, typically by lowering the resolution in case of a mobile device. When the input consists of high quality streams, for example at 30 Mbit/s, a typical General Purpose Processor (GPP) can no longer transcode the bit stream in real time, requiring the use of a distributed grid of transcoding nodes for real time transcoding. The xStreamer can construct and manage this distributed grid of machines, shown in figure 9 using 3 transcoding nodes, with each machine running a different configuration of the xStreamer. The server accepts a stream from the network and distributes it over a series of transcoding nodes, while each transcoding node transcodes one GOP as fast as possible. Finally, the proxy combines the transcoded streams into one and streams them over the network to a media player. In this example the xStreamer is more than just a streamer, it distributes, transcodes and collects streams.

The server node, shown in figure 10, receives a transport stream over RTP and demultiplexes it, producing two PES-streams (Packetized Elementary Stream [12]), one video (upper half of the figure) and one audio (lower half of the figure). Since the PES-packetization is unsuitable for network transport, its packets are much larger than 1500 bytes, the configuration unpacketizes the PES-stream and repacketizes them

Figure 10: The server node which reads in a transport stream, demultiplexes it in a video and audio stream, sends each GOP of the video stream to the next transcoding node and sends the audio stream directly to the proxy node.

using the generic xstreamer-packetizer, a simple and lightweight packetizer internally used by the xStreamer. On the one hand, for the unpacketized video stream the component GOP-classifier determines the GOP boundaries by parsing the elementary stream headers, which is codec dependent, and detecting the start of a new I-frame. For each new GOP, this component routes the video stream to the next connection ending at an RTP-transmitter which connects over the network to a transcoding node. On the other hand, for the audio stream the server bypasses the transcoding nodes and sends the stream directly to the proxy node.

The configuration of the transcoding node, shown in figure 11, has a much simpler lay-out. The RTP-receiver passes the stream to the xstreamer-unpacketizer which reconstructs the original elementary video stream. The transcoder component accepts this format, internally decodes it to an uncompressed format and re-encodes it to the desired new format, which can be in a different codec, resolution, bit rate, etc. The new elementary stream, with the transcoder component produces, is packetized again and sent to the proxy node by the RTP-transmitter. Because of the distributed grid of three transcoding nodes, each transcoding node has three GOPs the time to transcode one GOP. If the transcoding takes longer, the grid fails and can no longer transcode the stream in real time.

Finally, the proxy node, shown in figure 12, performs the reverse operation of the server node by collecting the streams from the three transcoding nodes and together with the audio stream generating a new transport stream. Both the three video streams and audio stream are unpacketized and additionally the three video streams require merging by a standard multiplexer because the three transcoding nodes do not operate synchronized. The standard multiplexer sim-
Figure 13: Distribution of the bit rate among the 20 combinations of SVC layers, each represented by a cuboid whose height is relative to the fraction of the associated bit rate. The three dimensions are temporal (frame rate), quality and resolution.

Figure 14: Bit rate evolution when scheduling the packets per frame. Peaks in the bit rate are caused by the larger I-frames.

Figure 15: Bit rate evolution when scheduling the packets over a fixed window of 900 ms which causes a sudden change in bit rate every 900 ms.

5. FEATURES

Features of the xStreamer include modularity, efficiency and differentiated streaming as demonstrated in the previous sections and examples.

Furthermore, the program offers SVC (Scalable Video Coding [11]) support, which distinguishes it from the current open source solutions. Figure 13 shows for a particular SVC stream an analysis, generated by the program, of the bit rate distribution among the 20 combinations of the 5 temporal, 2 quality and 2 resolution layers from the encoding. Each cuboid in the figure represents such a combination and its height is relative to the fraction of the associated bit rate. The encoding contains 2 resolutions, CIF (352 × 288, $D_0$) and 4CIF (704 × 576, $D_1$), with the majority of the bit rate concentrated in the largest resolution ($D_1$). Likewise, the encoding contains two quality layers ($Q_0$ and $Q_1$) with the bit rate concentrated in the highest quality layer ($Q_1$).

Additionally, the xStreamer has a wide selection of schedulers, each offering a different smoothing of the bit rate. These schedulers offer for example smoothing per frame, GOP, fixed window, etc. Figures 14, 15 and 16 show the effect of smoothing per frame, over a fixed window (900 ms) and over a sliding window (900 ms) respectively for the same sequence by plotting the instant bit rate at 40 ms intervals. Although the sequence has an average bit rate of 2.4 Mbit/s, the scheduling per frame in figure 14 shows peaks up to 7.5 Mbit/s in the bandwidth, corresponding with the larger I-frames of the H.264 encoding. In figure 15 the effect of the I-frames is smoothed out by using a fixed window of 900 ms. However, sudden changes in bit rate are visible every 900 ms, while in figure 16 these sudden changes are removed by using a sliding window of 900 ms.

The supported codecs, containers and network protocols form a crucial aspect of a streamer. The program can handle any container which the underlying library avformat supports. Nevertheless, the program cannot packetize the extracted streams from these containers if the corresponding packetizer is not supported. The internal generic packetizer of the xStreamer can handle such streams but this packetization is not supported by other programs and players. However, a standardized packetizer is available for any of the codecs listed below. The program supports RTP, the most important protocol for streaming, together with RTSP which manages a collection of RTP connections. In addition, the program supports the standard transport protocols UDP and TCP.

- **codecs:**
  - video: MPEG-1, MPEG-2, MPEG-4, H.264, SVC
  - audio: MPEG-1, MPEG-2, MPEG-4

- **containers:** MPEG-2 PS, MPEG-2 TS, AVI, MOV/MP4, raw
• **protocols**: RTP/UDP, RTSP/RTP/UDP, UDP, TCP

6. **GRAPHICAL USER INTERFACE**

Writing XML files to create the directed graph which configures the program, is not straightforward. Therefore, the xStreamer provides a GUI which lets the user draw, drag and connect components. The GUI simply inputs and outputs XML based configurations, for example the configuration from figure 6a. By adding the element `gfx` in the configuration file, the user interface can store the shape of the directed graph and recreate the graph when loading a configuration. Figure 17 shows the directed graph from figure 5 as seen in the user interface. Written in Tkinter [3], the user interface forms a platform independent solution.

7. **AVAILABILITY**

The xStreamer follows its inspiration, the Click Modular Router project [16], by having an open source license. Since all the included libraries (see section 2.5) are open source under the (Lesser) GNU Public License ((L)GPL [10]), the xStreamer’s open source license is also GPL.

The program is available at [anonymous], a newly launched web site. The distribution contains a detailed practical manual on how to build configurations together with a complete overview of every component and its parameters. Supplemented by the internal view of the program described in this paper, these two documents should provide a thorough introduction.

8. **CONCLUSION**

The goal was to create a flexible, modular and easily extendable streamer in order to avoid redundant work in video
research. The detailed overview of the architecture, using multiple examples and graphs, demonstrates the modularity goes beyond the programming level and determines how a graph of components configures the program. How simple changes in the graph of components create novel configurations together with choice between real time, simulation and offline tool, contribute to a flexible program. The efficiency of the program is similar to its competitors and future extensions such as multithreading can further improve it. The use case about real time transcoding demonstrates the xStreamer is not just a streamer, but can distribute, transform and collect streams across multiple machines. Because of the intensive maintained third-party libraries, the program can support a wide variety of codecs, containers and network protocols.

In addition, the xStreamer participates in the Open Source Software Competition of the conference ACM Multimedia 2009 [2].

9. ACKNOWLEDGEMENTS

10. REFERENCES


<table>
<thead>
<tr>
<th>Category</th>
<th>Component</th>
<th>Scheduled</th>
</tr>
</thead>
<tbody>
<tr>
<td>reader</td>
<td>ffmpeg-reader</td>
<td>yes</td>
</tr>
<tr>
<td></td>
<td>avc-reader</td>
<td>yes</td>
</tr>
<tr>
<td>packetizer</td>
<td>avc-packetizer</td>
<td>no</td>
</tr>
<tr>
<td></td>
<td>mp4-packetizer</td>
<td>no</td>
</tr>
<tr>
<td></td>
<td>mp2-packetizer</td>
<td>no</td>
</tr>
<tr>
<td></td>
<td>pes-packetizer</td>
<td>no</td>
</tr>
<tr>
<td></td>
<td>xstreamer-packetizer</td>
<td>no</td>
</tr>
<tr>
<td></td>
<td>avc-unpacketizer</td>
<td>no</td>
</tr>
<tr>
<td></td>
<td>pes-unpacketizer</td>
<td>no</td>
</tr>
<tr>
<td></td>
<td>xstreamer-unpacketizer</td>
<td>no</td>
</tr>
<tr>
<td>multiplexing</td>
<td>std-multiplexer</td>
<td>yes</td>
</tr>
<tr>
<td></td>
<td>ts-multiplexer</td>
<td>yes</td>
</tr>
<tr>
<td></td>
<td>ts-demultiplexer</td>
<td>yes</td>
</tr>
<tr>
<td>scheduler</td>
<td>basic-scheduler</td>
<td>yes</td>
</tr>
<tr>
<td></td>
<td>frame-scheduler</td>
<td>yes</td>
</tr>
<tr>
<td></td>
<td>window-scheduler</td>
<td>yes</td>
</tr>
<tr>
<td></td>
<td>sliding-window-scheduler</td>
<td>yes</td>
</tr>
<tr>
<td>transmitter</td>
<td>rtp-transmitter</td>
<td>no</td>
</tr>
<tr>
<td></td>
<td>udp-transmitter</td>
<td>no</td>
</tr>
<tr>
<td></td>
<td>tcp-transmitter</td>
<td>no</td>
</tr>
<tr>
<td>receiver</td>
<td>rtp-receiver</td>
<td>yes</td>
</tr>
<tr>
<td></td>
<td>udp-receiver</td>
<td>yes</td>
</tr>
<tr>
<td></td>
<td>tcp-receiver</td>
<td>yes</td>
</tr>
<tr>
<td>writer</td>
<td>writer</td>
<td>no</td>
</tr>
<tr>
<td>classifier</td>
<td>time-classifier</td>
<td>no</td>
</tr>
<tr>
<td></td>
<td>frame-classifier</td>
<td>no</td>
</tr>
<tr>
<td></td>
<td>random-classifier</td>
<td>no</td>
</tr>
<tr>
<td></td>
<td>gilbert-classifier</td>
<td>no</td>
</tr>
<tr>
<td>analyser</td>
<td>statistics</td>
<td>no</td>
</tr>
<tr>
<td></td>
<td>svc-analyser</td>
<td>no</td>
</tr>
<tr>
<td>system</td>
<td>sink</td>
<td>no</td>
</tr>
<tr>
<td></td>
<td>block</td>
<td>no</td>
</tr>
<tr>
<td></td>
<td>in</td>
<td>no</td>
</tr>
<tr>
<td></td>
<td>out</td>
<td>no</td>
</tr>
<tr>
<td>high-level</td>
<td>streamer</td>
<td>yes</td>
</tr>
</tbody>
</table>

Table 2: Selection of components offered by the xStreamer, sorted per category. The last column indicates if the component is scheduled, meaning it is triggered during each cycle (see section 2.3).


