VideoLAN Server developer documentation

Cyril Deguet

asmax@via.ecp.fr

Tristan Leteurtre

tristan.leteurtre@videolan.org

VideoLAN Server developer documentation

by Cyril Deguet

by Tristan Leteurtre

Published $Id: vls-devel.sgml 655 2004-03-09 15:13:45Z sam $

Copyright © 2002 VideoLAN

This documentation is distributed under the FDL.

Table of Contents

1. VLS overview 1
- 1.1. VLS design 1
- 1.2. General architecture 1
- 1.3. Common classes overview 2

2. VLS framework 5
- 2.1. Overview 5
- 2.2. Threads 5
- 2.3. Modules 6

3. How is a stream broadcasted ? 8
- 3.1. What happens when a stream is started ? 8
- 3.2. How is a TS packet sent ? 8

4. The Converter and the Reader 10
- 4.1. The Converter 10
- 4.2. Interfaces 11
  - 4.2.1. Reader-Converter 11
  - 4.2.2. Converter SyncFifo 11
- 4.3. The Reader 12

5. The TsStreamer 13
- 5.1. What's a PCR ? 13
- 5.2. Using or not PCRs 13
- 5.3. How vls handles PCRs 14
- 5.4. The TsStreamer itself 15

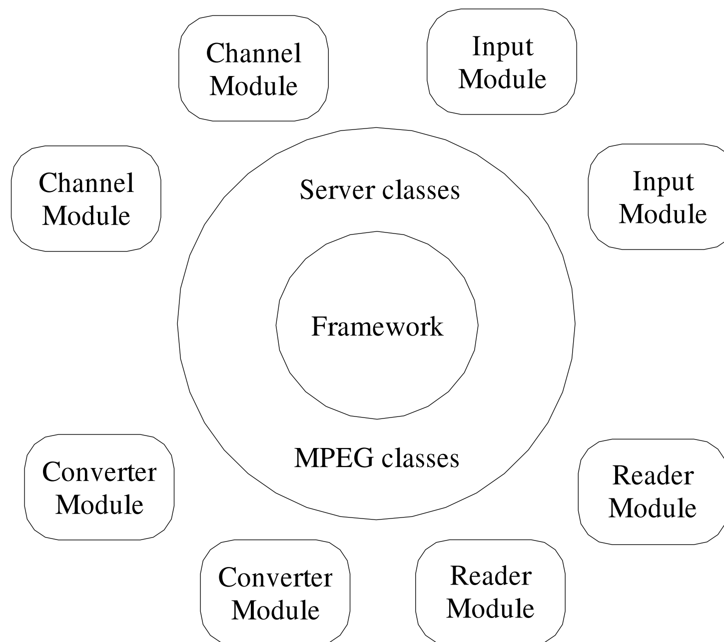
6. The Output Module 16
- 6.1. The Module itself 16
- 6.2. The internal Fifo 16

Chapter 1. VLS overview

1.1. VLS design

The VideoLAN Server is programmed in C++, with an entirely "hand-made" framework. It means that VLS does not (and will not) use Standard Template Library classes such as iostreams or vectors. The internal VLS framework is supposed to be complete enough, so nothing should have to be written any more (except maybe for porting VLS to other operating systems).

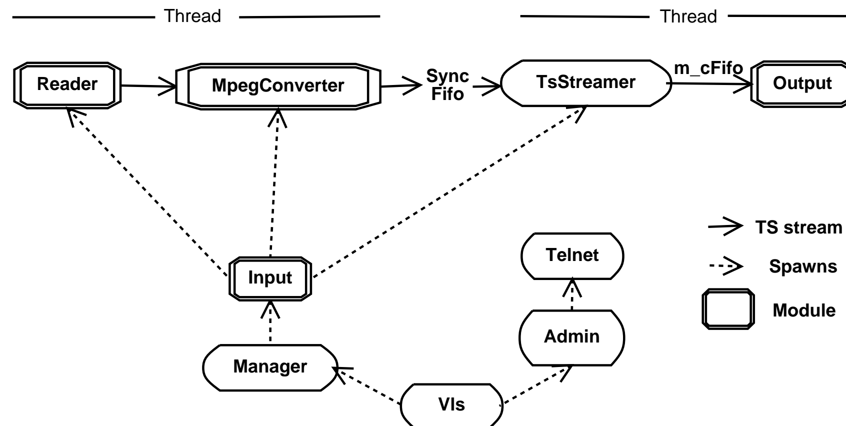
VLS is made up of three parts: a framework (located in `src/core`), common classes (in `src/server` and `src/mpeg`) and optional classes called *modules* (in `src/modules`). Modules are actually classes inherited from common classes. Some classes that are considered as common classes at the moment may become modules in the future.



Since version 0.3.1, modules can be either *built-in* (i.e. statically linked) or *plug-in* (i.e. dynamically loaded). Whether a module is built-in or plug-in is defined in `configure.in`.

1.2. General architecture

The general architecture of the VLS is shows on the diagram below :



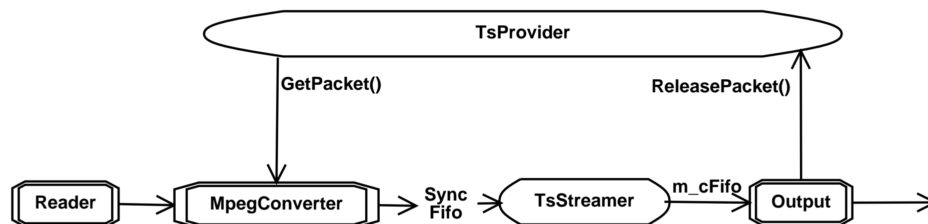
We can isolate 3 main parts : the source thread (Reader and MpegConverter), the consumer thread (TsStreamer and the Output) and the Management part (Input, Admin and Manager).

The Management part spawns all the needed parts and modules, when the Vls is strarted or when a new stream is started.

The source thread reads the data through the reader as fast as possible to fill up the SyncFifo.

Then, the consumer thread gets the packets from the SyncFifo and stream them. The timing of this streaming is managed by the TsStreamer, which give them the Output module. The m_cFifo of TS_IN_ETHER packet length is used to gather several TS Packets before sending them together in one UDP packet (for the NetOutput module).

To improve the efficiency and avoid too many memory allocations, a pool of TS Packet is allocated once : it is the TsProvider. That means that at the beginning of the chain, a new TsPacket is called by the Converter (TsProvider->GetPacket()), filled up with bytes from the reader, goes through the TsStreamer and the Output. After being send, the TsPacket is reset available in the TsProvider (TsProvider->ReleasePacket()).

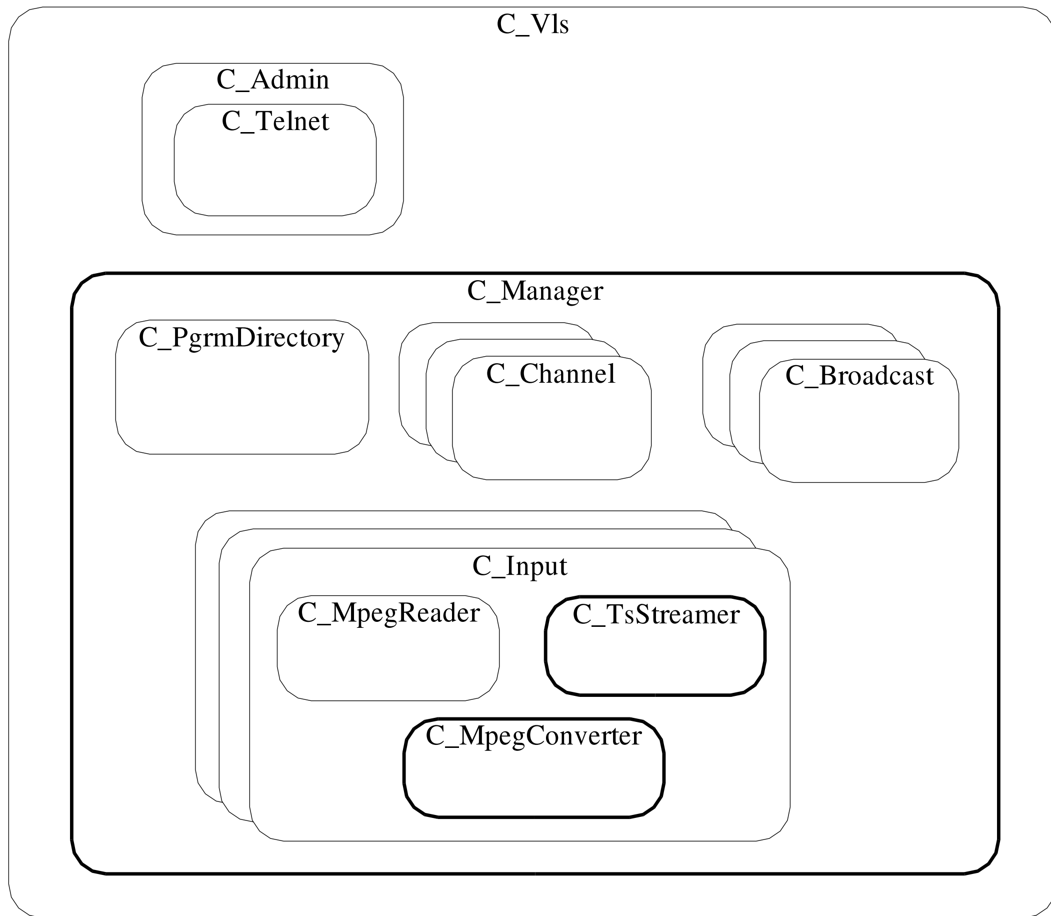


The size of the TsProvider is calculated to be able to handle about 3 seconds of stream.

1.3. Common classes overview

Here is an overview of some common classes present in VLS (be careful: a class drawn within another

one is a *member* of that class, not a child) :



VLS class overview. Classes drawn with a bold border are run in separate threads.

C_Vls

It is the main class of the VideoLAN Server. Its role is to load modules, launch the Admin and the Manager, and forward messages between them.

C_Admin

It is the main administration class, whose role is to control the various administration interfaces, and to parse and handle the commands sent to these interfaces.

C_Telnet

This is the telnet administration interface, the only one supported at the moment.

C_Manager

The Manager is one of the most important classes of VLS. It launches the various inputs and channels according to the configuration file `vls.cfg`, contains a list of all the available programs

and a list of the currently broadcasted programs. It is this class which interprets the commands sent to the Admin, and can tell inputs to start, stop, suspend or resume streams.

C_Channel

This is the parent class of all the channel (i.e. output) modules. Currently implemented channels are network and file.

C_PgrmDirectory

It is the list of all the programs available in the various inputs.

C_Broadcast

A "broadcast" is defined by an input/program/channel combination. It represents a stream that is currently broadcasted by VLS.

C_Input

This is the parent class of all the input modules. An input basically gets a MPEG stream from a source (thanks to a "reader") and sends it to a converter. Each input contains a list of the available programs it can provide.

C_MpegReader

An MpegReader reads data from a source (file, DVD, ...) and delivers a continuous MPEG (PS or TS) stream.

C_MpegConverter

An MpegConverter converts the stream provided by the Reader into a valid MPEG-TS stream.

C_TsStreamer

The Streamer reads data from a Converter and sends it to a Channel at the right speed.

Chapter 2. VLS framework

2.1. Overview

The VLS framework is a set of low-level classes used by every other classes. Currently, it can handle:

- Buffers (`src/core/buffers.*`)
- Configuration files (`src/core/settings.*`)
- Exceptions (`src/core/exception.*`)
- Files (`src/core/file.*`)
- Hash tables (`src/core/hashtable.*`)
- Dynamic libraries (`src/core/library.*`)
- IO streams (`src/core/stream.*`)
- Lists (`src/core/list.*`)
- Logging (`src/core/log.*`)
- Modules (`src/core/module.*`)
- Parsing (`src/core/parsers.*`)
- Session handling (`src/core/network.*`)
- Serialization (`src/core/serialization.*`)
- Sockets (`src/core/socket.*`)
- Stacks (`src/core/stack.*`)
- Threads (`src/core/thread.*`)
- Vectors (`src/core/vector.*`)

2.2. Threads

A class that must be run in a new thread has to extend the C_Thread abstract class. Pure virtual functions that must be implemented by such a class are:

`void InitWork()`

This method is called before the thread is created.

`void DoWork()`

It is called just after the thread is created. The main loop of the class should be put here.

`void StopWork()`

This method is called when the thread is to be stopped or canceled.

`void CleanWork()`

This method is called after the shutdown of the thread.

To create the thread, the derived class just has to call the `Create()` method. To stop the thread, just call `Stop()`.

2.3. Modules

Whenever a class is optional, it should be implemented as a module. A module can be:

- a built-in module: it means the module is linked statically, though classes are of course instanciated dynamically.
- a plug-in module: in this case, the classes are linked in a shared library, and loaded dynamically at runtime.

Here are the differents steps required to add a new module:

1. Declare a new module type. Indeed, modules of the same type are derived from the same class, called a "virtual module". For instance, DvdMpegReader and FileMpegReader are both derived from the module type "MpegReader". To declare a class as a virtual module, use the DECLARE_VIRTUAL_MODULE macro:

   ```
   // in classtype.h

   C_ClassType
   {
    public:
     C_ClassType(C_Module *pModule, argType val)
     // Class declaration
   };

   DECLARE_VIRTUAL_MODULE(ClassType, "strType", argType);
   ```

   The virtual module constructor must have two arguments: C_Module *pModule and *argType* val, where *argType* can be anything (defined in the DECLARE_VIRTUAL_MODULE macro). The virtual module destructor have to call `pModule->Unref()`. *strType* is the name given to the module type.

2. Declare the module itself. Like virtual modules, use the DECLARE_MODULE macro after the class declaration:

   ```
   // in classname.h

   C_ClassNameClassType : public ClassType
   {
    public:
   ```

```
  C_ClassNameClassType(C_Module *pModule, argType val)
  // Class declaration
};

DECLARE_MODULE(ClassName, ClassType, "strType", argType);
```

3. Add some declarations for libraries and built-in modules:

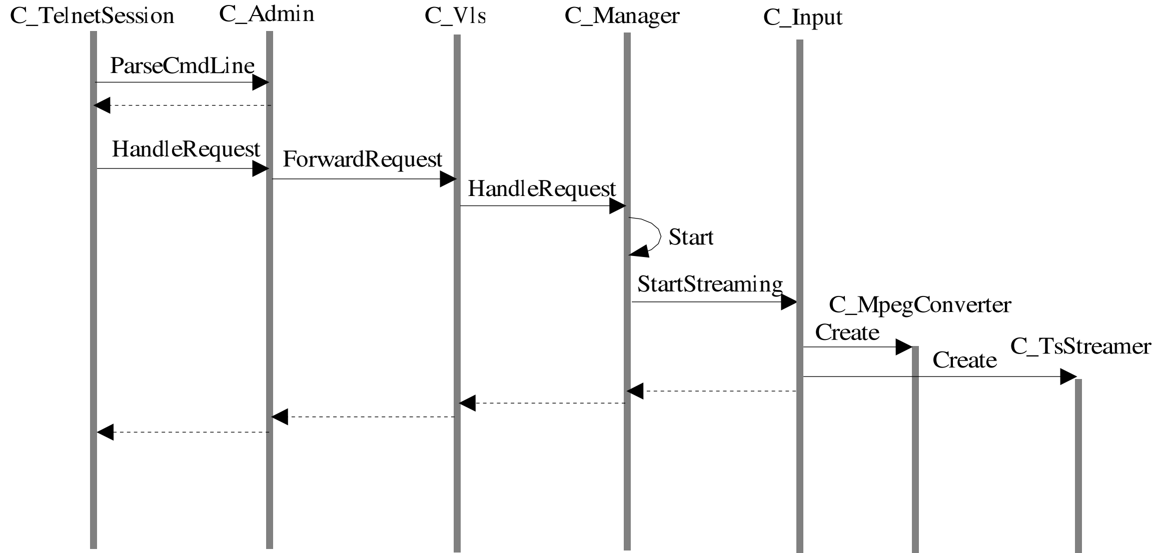
```
// in classname.cpp

//------------------------------------------------------------------------------
// Library declaration
//------------------------------------------------------------------------------
#ifdef PIC
GENERATE_LIB_ARGS(C_ClassNameClassTypeModule, handle);
#endif


//------------------------------------------------------------------------------
// Builtin declaration
//------------------------------------------------------------------------------
#ifndef PIC
C_Module* NewBuiltin_classnameclasstype(handle hLog)
{
  return new C_ClassNameClassTypeModule(hLog);
}
#endif
```

4. Add the module in `configure.in`. For a built-in module, add the file name in the line BUILTINS=... For a plug-in module, it is more complicated; look what is done for existing plug-ins.

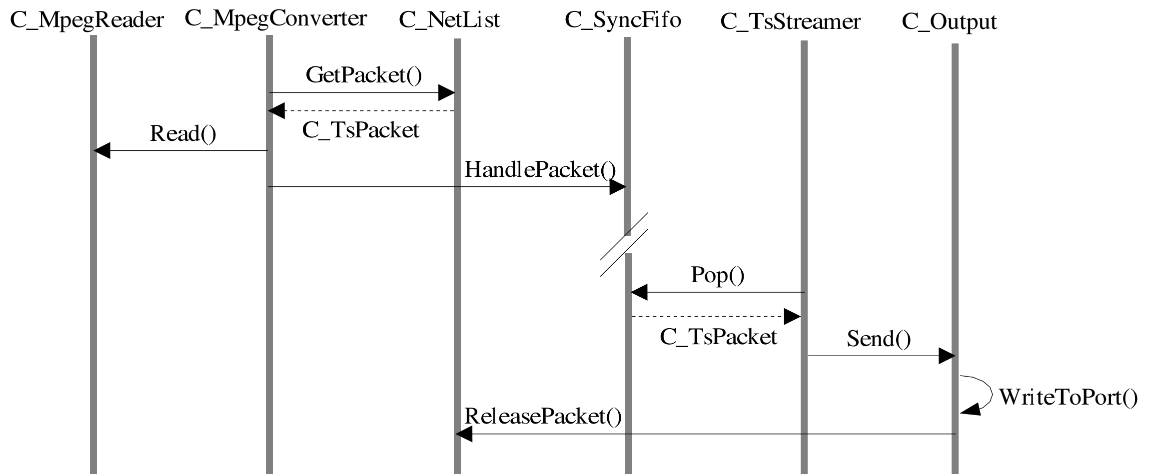
Chapter 3. How is a stream broadcasted ?

3.1. What happens when a stream is started ?

Here is a simplified sequence diagram to show what happens when a "start" command is sent through the telnet interface:



3.2. How is a TS packet sent ?



The Packet Handler (C_SyncFifo) is shared between two threads: in one thread the Converter pushes TS packets in the fifo, and in another thread the Streamer pops them.

Chapter 4. The Converter and the Reader

The role of the couple (Reader, Converter) is to get raw data from several medium (dvd, file, video encoder, satellite card), and to convert it into valid TS packets. I choose to gather them in the same section since they belong to the same thread. Refer to *General architecture* to see the position of converter between other parts of vls.

4.1. The Converter

The Converter - or MpegConverter - is a module that is responsible for providing TS packets. Today, the modules implemented are ps2ts and ts2ts, which functions talk by themselves.

As any modules in the VideoLAN server, the main class C_MpegConverter is implemented in the main program : src/mpeg/converter.cpp . It is in general better for a first overview to have a look at the .h defs (src/mpeg/converter.h) :

```
class C_MpegConverter : public C_Thread
{
public:
  (...)
  void Resume();
  void Suspend();
  void ShortPause();

protected:
  virtual void InitWork();
  virtual void StopWork();
  virtual void CleanWork();

  (...)
};
```

The C_MpegConverter inherits from the C_Thread, since it is the motor of the source thread (Reader & Converter).

This class implements the basic functions of the converter, which will be common to all the Converter modules (ps2ts, ts2ts). Resume(), Suspend(), ShortPause() do all the common stuff to Init/Close the Reader, and deal with the m_cCondition stuff for Suspending/Resuming the streaming (see further).

But what is interesting are the virtual functions. In the present case, they are declared as 'virtual' since there are re-implementation of the basic InitWork(), StopWork() and CleanWork() of the C_Thread class. But there are also virtuals as every Converter module will also re-implement them : that will contitute the essence of a module. All the differences between the modules will take place in thoses functions, and that's why MpegConverter modules have been created.

As the C_MpegConverter inherits from C_Thread, the thread has to be created. This is made in the Input modules (for example in the Localinput.cpp, OnStartStreaming) :

```
 // Create the converter
 C_MpegConverterModule* pConverterModule = (C_MpegConverterModule*)
```

```
                                        C_Application::GetModuleManager()
                                                ->GetModule("mpegconverter",
                                                            strConverterType);
(...)
 pConverter = pConverterModule->NewMpegConverter(cConfig);
  // Create the Thread
 pConverter->Create();
```

That's why we can say that the Input spawns the Converter (as well as other parts of the TS chain : Reader, Streamer).

To suspend a stream, we use the thread structure : we block the condition m_cCondition.

4.2. Interfaces

The two interfaces are Reader and the SyncFifo (to the TsStreamer).

4.2.1. Reader-Converter

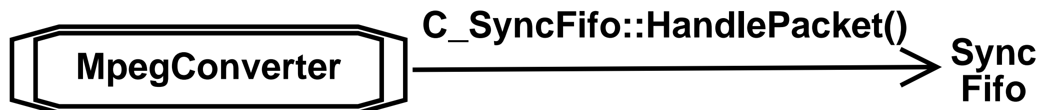
The member m_pReader is a reference to the MpegReader module used in the streaming chain. This value is initialized during the creation of the Converter, in the appropriate Input Module (have a look in C_LocalInput::OnStartStreaming for example).

So, to get data, the converter makes direct calls to the Reader :

```
 m_pReader->Read();
 m_pReader->Seek();
```

4.2.2. Converter SyncFifo

The SyncFifo is called in the converter m_pHandler. m_pHandler is declared in converter.h as an interface : I_TsPacketHandler, which is the mother class of C_SyncFifo.



We call it SyncFifo because it is a fifo which is thread-safe, and allows synchro between source thread and consumer thread. Moreover, the I_TsPacketHandler used is declared as C_SyncFifo in the Input modules :

```
C_SyncFifo* pBuffer;
(...)
cConfig.m_pHandler = pBuffer;
```

So, the packet is send to the SyncFifo using the command :

```
m_pHandler->HandlePacket(pPacket);
```

If for some reason the packet has to be trashed (pre-parsing, end of stream, read error....), then the call to m_pHandler->HandlePacket() is not made. However, one should not forget to set it back available in the TsProvider though the command :

```
m_pTsProvider->ReleasePacket(pPacket);
```

4.3. The Reader

There is not much to say more than reading the code... All the interest locates in the implementation of Read() and Seek().

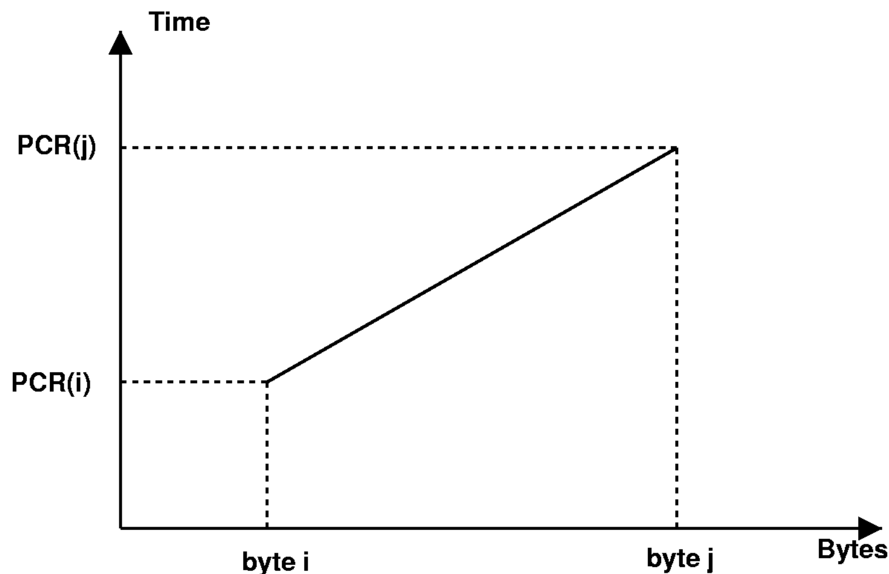
Chapter 5. The TsStreamer

The Streamer, part of that we called the "consumer thread", takes the data from the SyncFifo, makes some timing synchro and sends it to the output. Refer to *General architecture* to see the position of TsStreamer between other parts of vls.

The TsStreamer is declared and implemented in server/tsstreamer.* . The main purpose of TsStreamer is to deal with timing management, especially with Program Clock References.

5.1. What's a PCR ?

Program Clock Reference are 33 bits time values, which are in the adaptation_field of a TS packet header (see 13818-1 specifications). There are not in each TS packet, the specs say that there should be at least one PCR every 0.1 second. The PCR located at a given byte i is the time at which the byte i has to be processed (sent in the case of a streamer). Obviously, there can not be a PCR for each byte of a TS stream, so we make a linear calculation between two PCRs to send the data regularly along the time. So, the bitrate between two PCR shall be constant.



That is why PCR are the core of streaming : they say when datas have to be sent over the network. In the case of file stored on and hard disk, this System Layer (TS in our case) provide timing information. That is why, for the moment, it is not possible to stream any file format. For example, in the case of mpeg4, most files does not contain streaming time stamps since they don't have a system layer. It would be possible to re-generate them knowing the frame-rate, and going deeper in the stream layer to get each frame, remixing the data and sending the at the right date.

5.2. Using or not PCRs

First of all, the streamer can (or not) take into account PCRs. This is specified when instanciating a new

TsStreaming : the constructor has a field `bool m_bUsePcr` :

```
C_TsStreamer(handle hLog, C_Broadcast* pBroadcast,
             C_NetList* pTsProvider, C_SyncFifo* pBuffer,
             C_EventHandler* pEventHandler,
             bool m_bOwnProvider, bool m_bUsePcr);
```

In the case of a stored stream (file or Dvd), the use of PCR is essential since we do not have any other timing info. That's why, in `modules/localinput/localinput.cpp`, C_TsStreamer is instantiated with :

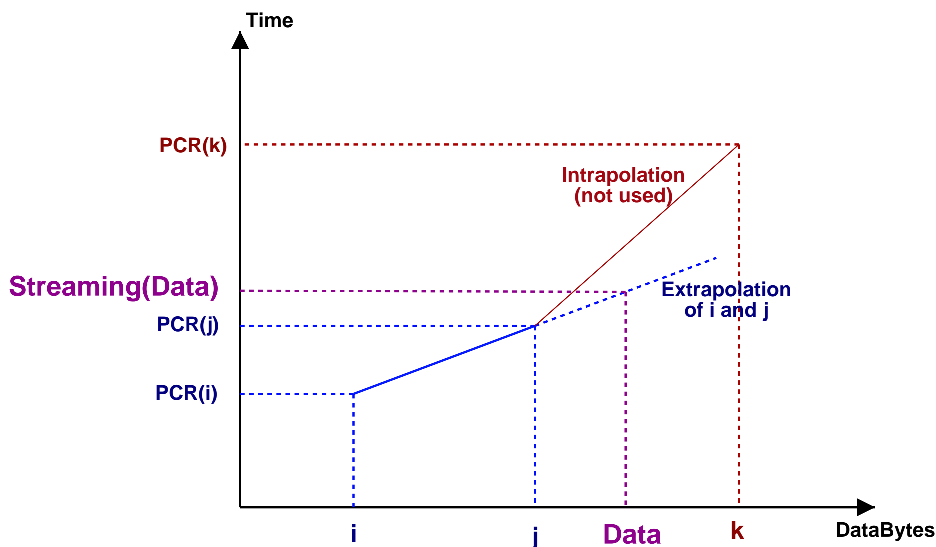
```
C_TsStreamer* pStreamer = new C_TsStreamer(m_hLog, pBroadcast,
                                           pTsProvider, pBuffer,
                                           m_pEventHandler, true, true);
```

Don't forget that the localinput input deals with Dvd and files, and - like other input modules - spawns all the streaming chain.

In the case of satellite or live-video encoding, the data comes at the correct rate (since it is live and supposed to be well broadcasted...). So the TsStreamer doesn't need to handle the PCRs, it just transmits the packets as they come. That's why the value `m_bUsePcr` is set to *false*.

5.3. How vls handles PCRs

We said before that between two PCRs, a streamer should make a linear interpolation. However, as the streamer does not contain any buffer, it is not possible to know what the next PCR will be, so it is impossible to calculate the interpolated date at which a byte has to be sent.



Instead, vls uses extrapolation : when the PCR(j) is received, the slope and offset are updated to correspond to the segment [i,j]. This is done in `modules/server/tsstreamer.cpp` :

```
inline void C_TsStreamer::AdjustClock(C_TsPacket* pPacket)
{
```

```
(...)
  m_dSlope = ((double)iPCRTime - m_iLastTime) / (s64)m_uiByteRead;
(...)
  m_iLastTime = iPCRTime;
(...)
```

Then, when holding the data D, the streamer computes the linear extrapolation of the sending date Streaming(Data) using the previous values `m_dSlope` and `m_iLastTime` of segment [i,j] :

```
inline void C_TsStreamer::WaitSendDate()
{
  s64 iSendDate = m_iLastTime + m_iDeltaClock + (s64)m_dSlope*m_uiByteRead;
  (...)
}
```

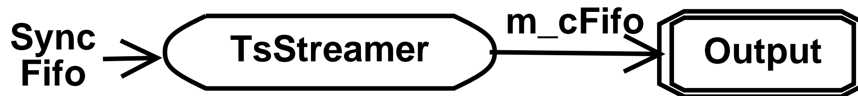
Of course, this method is not exactly accurate, since just before k, extrapolated_date(k) and PCR(k) do not match : there is a tiny jitter. However, extrapolation avoids an extra buffer, and the jitter may be negligible in regards to the amount of data transmited.

At last, one must be aware that PCR are expressed in a 27 MHz clock, so when mixing PCR and system clock (in seconds or milliseconds), conversions have to be done.

5.4. The TsStreamer itself

The main job is made in `void C_TsStreamer::DoWork()`. First, a packet is taken from the SyncFifo :

```
C_TsPacket* pPacket = m_pBuffer->Pop();
```



After all the timing calculations described before, each TS packet is sent to the output :

```
Output->Send(pPacket, (...) );
```

Chapter 6. The Output Module

The Output Module takes the data from the TsStreamer, gather the TS packets together and add RTP header when needed, and then writes the datas to a file of to the network. Refer to *General architecture* to see the position of Output between other parts of vls.

6.1. The Module itself

As other modules of vls, a core file `server/output.*` describes the definition of the output, and its behavior against others parts of vls. The modules `NetOutput` and `FileOutput` differs from their implementation of the virtual function `WriteToPort()`.

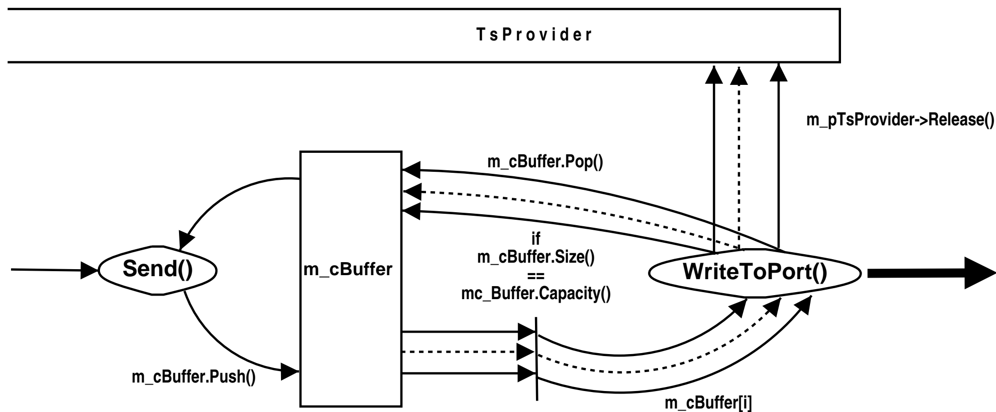
The Output is created by the Channel Module. In `modules/filechannel.cpp` :

```
m_pOutput = new C_FileOutput(strFilename, bAppend);
```

and in `modules/netchannel.cpp`:

```
m_pOutput = new C_Net4Output(m_strName);
```

6.2. The internal Fifo



Since the output may have to gather packets, it needs a Fifo of TsPackets. Its name is `m_cBuffer()`. The capacity (ie the maximum size) of m_cBuffer() is specified in the Output constructor :

```
C_Output::C_Output(unsigned int iBuffSize) : m_cTsBuff(iBuffSize)
```

In the case of FileOutput, the vls can write the packet when he wants, to we don't care about the number of packets written. `iBuffSize` can be 1, or 7 if we want to store 7 TsPackets in an RTP packet when using RTP.

In the case of NetOutput, this is more important since this will influence the size of each UDP packets. The maximum amount is 7. Indeed, the maximum payload size for an UDP packet is 1472 Bytes. Each TsPacket is 188 Bytes, and 1472/188 = 7.

Moreover, when using RTP output, 12 extra header bytes are added. But it's still ok since :

```
7 * 188 + 12 = 1328 < 1472.
```

By default, the TS_IN_ETHER value defined in `server/output.h` is set to 7. You may change that depending on how your network hardware can handle big UDP packets.