An Open-source Software Toolkit for Professional Media over IP (ST 2110 and more)

Ievgen Kostiukevych
Willem Vermost
European Broadcasting Union
Geneva, Switzerland
kostiukevych@ebu.ch
vermost@ebu.ch

Pedro Ferreira
Bisect
Porto, Portugal
pedro@bisect.pt

Abstract - *As the broadcast industry is on the eve of replacing SDI with IP for critical live applications, new ways of measuring, monitoring and fault detection need to be explored. This paper breaks down certain technical aspects of a key document out of the SMPTE ST 2110 suite, the ST 2110-21. It also explains the challenges of building a software-based generic toolkit to measure some basic characteristics. A practical example of an open source software-based ST 2110 implementation is provided. Its functionalities to decode and generate streams, measure and validate stream parameters according to ST 2110-21 requirements are explained.*

INTRODUCTION

The publishing of the SMPTE ST 2110 suite of standards signals the beginning of a new era in TV broadcasting and production industries. It will have major influence and will dictate new requirements to both the TV/Radio and IT worlds.

Traditionally we tend to separate broadcast engineering and IT/network engineering. SMPTE ST 2110 takes the best from both worlds and offers us exciting new possibilities in flexibility, automation, virtualization, etc.

However, as everything new comes with a price, the new SMPTE standard requires a specific skill set and careful planning of aspects we were not aware of before.

The standard itself contains certain models and calculations that are not so easy to grasp at a glance.

During extensive analysis of the SMPTE ST 2110 suite of standards, and ST 2110-21 in particular, the authors came to the conclusion that certain solutions are required in order to properly examine and validate ST 2110 streams.

Initial internal tools realized in the form of Python scripts started to evolve into a software platform that enables precise validation, decoding and generation of media streams, compatible with ST 2110.

The purpose of this paper is to help the reader to understand the traffic shaping models offered in the ST 2110-21 standard, evaluate the challenges that one needs to overcome when developing, or selecting a solution in compliance with the SMPTE ST 2110 suite. In order to do so, the paper describes a toolset developed within the EBU LIST (Live IP Software Toolkit) project.

The EBU LIST toolkit is currently being developed as open-source software written in C++.

This paper offers an overview of solutions currently implemented and gives an overview of a direction of future project development.

By using the theoretical and practical approaches described in this paper, a reader can easily apply the toolkit and techniques to understand and analyze the traffic shaping principles set out in SMPTE ST 2110-21.

TRAFFIC SHAPING AND DELIVERY TIMING

Why is there a traffic model?

The traffic model proposed in SMPTE ST 2110-21 [1] specifies a timing model for senders and receivers of video RTP streams. The reason to constrain the traffic shape or packet delay variation is twofold: PDV could lead to increased latency and, even worse, dropped packets.

In a perfect streamed media world, all packets arrive equally spaced at the receiver side. In other words, the average bandwidth is equal to the bandwidth measured on a microscopic scale (see Figure 1 below). In reality senders do not create perfect streams and network operations can introduce packet delay variation.

The implementation of a sender plays an important role. The closer your sender operates to the physical hardware, the easier it is to create perfectly shaped traffic (e.g. FPGA based implementations). If the sender is just a piece of software installed on an operating system, it is abstracted from the driver of the network card and therefore has less control over the traffic shape. The operating system also has to handle and prioritize other tasks besides the video application. In a virtualized system, this control is even further diluted. The network components can introduce latency due to multiple issues, the serialization and deserialization of packets, for example.

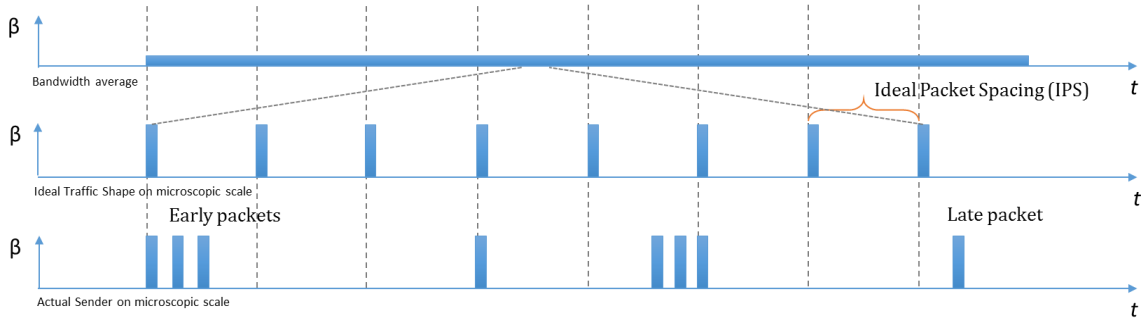

FIGURE 1: BANDWIDTH USAGE ON MICROSCOPIC SCALE.

The result of increased latency is easy to grasp. Some packets arrive early and stack closely, and some other (late) packets have a lot more space between them. In order to recreate the video stream, one needs to wait for these late packets. Techniques to cope with these imperfections are based upon the use of buffering and so-called watermark levels. These are widely known and implemented in the IT industry. However, latency is something we don't want to perceive in our live media production facility.

For the application of IP networks to professional media it must be realised that multiple streams are the norm. The system needs to be capable of handling many streams at the same time, and be capable of synchronizing them when needed.

WHAT DOES THE MODEL DESCRIBE?

I. Leaky buckets

The model mainly describes two virtual leaky buckets. These buckets are defined by the number of packets they can store and the speed at which the packets leave the bucket (drain rate).

The first leaky bucket is called the "network compatibility model". It is located at the output of the sender, prior to any network-induced delivery impairments. (see Figure 2 below). This model measures the packet delay variation introduced by a sender. If the number of packets exceeds the C_{MAX} value at a given point in time (C_{INST}), the test fails. The packets drain out of the leaky bucket at a rate of $1/T_{DRAIN}$ (See Figure 2)

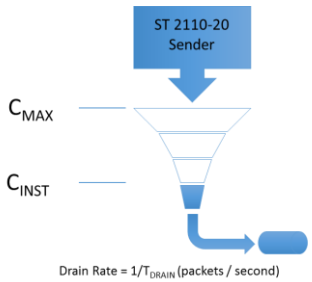

FIGURE 2: C_{MAX} LEAKY BUCKET.

The "virtual receiver buffer model" looks similar to the "network compatibility model", except for the fact that its drain rate is based upon a receiver packet read schedule. This schedule describes the timely fashion the receiver reads the packets out of the receiving buffer, potentially synchronized to the video transmission datum (T_{VD}) (See Figure 3).

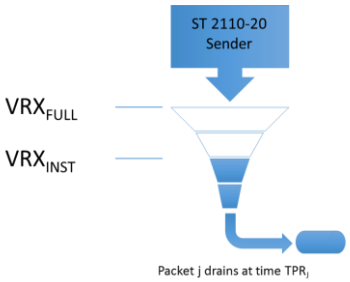

FIGURE 3: VRX LEAKY BUCKET.

II. Packet Read Schedules

There are two different read schedules described in the SMPTE ST 2110-21 standard.

The linear packet read schedule defines a sequence of packets which are equally spaced throughout the frame period T_{FRAME}. From a network perspective this schedule is optimal. It consumes the least possible bandwidth and is easy to measure.

The gapped packet read schedule is modelled as SDI, meaning that data packets are sent equally spaced throughout the active field or frame interval. During the vertical blanking of the SDI signal, no packets are sent. This creates a gap between the last packet of the previous frame or field and the first packet of the next one. From a network perspective the schedule uses a bit more bandwidth on a microscopic scale (see Figure 4 below).

The RTP header has a field labeled as Marker bit. For progressive scan video, the marker bit shall be set to 1 to denote when this RTP packet is the last packet carrying video essence data for a video frame. For interlaced video, the marker bit shall be set to 1 to denote when this RTP packet is the last packet carrying video essence data for a video field. The marker bit shall be set to 0 for all other packets [2].

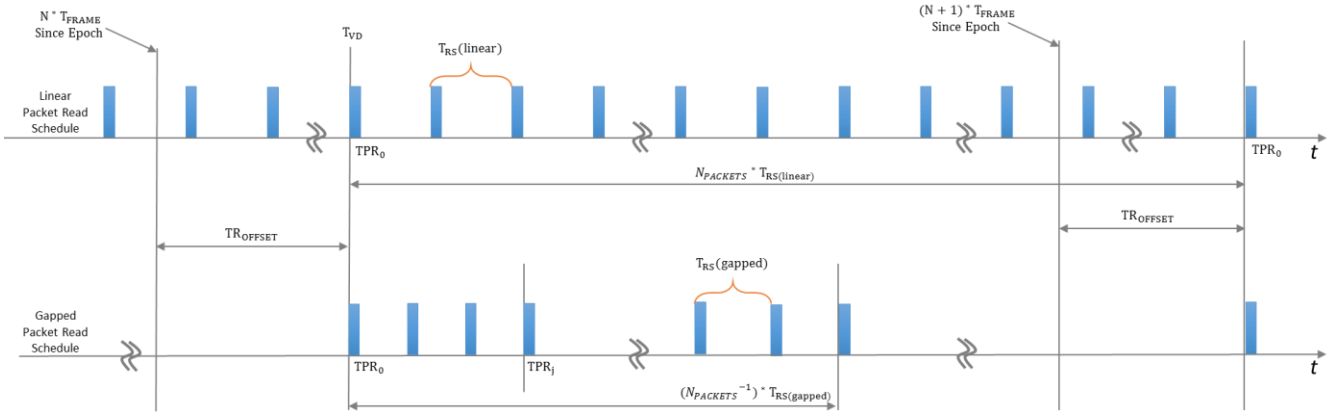

FIGURE 4: LINEAR VS. GAPPED PACKET READ SCHEDULES.

III. Senders

The standard defines narrow (N) and wide (W) senders. Both types constrain the maximum values of the leaky bucket buffer models relative to the used video format. Since we have a linear packet read schedule, there is a well-defined Narrow Linear (NL) sender. The table shows the calculated C_{MAX} and VRX_{FULL} values valid for 720p60, 1080i25 and 1080p50 (see Table 1).

Narrow senders will typically be FPGA-based implementations. This is an extremely tight value and probably impossible to achieve in a purely software, non-hardware assisted, product. To accommodate the introduction of software-based senders, as this is the future according to the JT-NM roadmap for open interoperability, the wide sender was added to the specifications.

Type	C_{MAX}	VRX_{FULL}
N	4	8
NL	4	8
W	16	720

TABLE 1: CALCULATED C_{MAX} AND VRX_{FULL} VALUES.

IV. Receivers

Practical receivers ought to accommodate, for a reasonable amount of accumulated packet arrival jitter and delay, over and above the specification in the traffic profile. Three receivers are defined: Narrow Synchronous (N), Wide Synchronous (W) and Asynchronous Receivers (A).

The narrow receiver could be equipped with a shallow receiving buffer as it should not be capable of receiving type NL and type W senders. This is probably the case for many FPGA based implementations. The following matrix reveals what receiver type is compatible with what sender type (see Table 2).

Senders \ Receivers	N	W	A
N	yes	yes	yes
NL	no	yes	yes
W	no	yes	yes

TABLE 2: SENDER – RECEIVER COMPATIBILITY MATRIX.

HOW TO MEASURE PACKET PACING

To measure whether a stream is compliant or not with the network compatibility model one needs to calculate C_{PEAK}, the maximum value a given stream produces. If $C_{PEAK} > C_{MAX}$ the given RTP video stream is considered as not compliant with the specifications defined in the standard. It gives us an indication of how well or badly the sender behaves on the network. Therefore, two aspects need to be deduced for the RTP video stream: the actual time the packet is sent out of the sender and the drain rate. The following description of the method shows some Python code snippets to clarify the explanation.

The complete and fully working Python script is available as an open source project on GitHub [3].

```
python cfull_analysis.py -c
[CAPTURE_FILE.cap] -g [MULTICAST_IP] -p
[UDP_PORT]
```

I. The packet timestamp

In order to retrieve a usable / sensible packet timestamp, the device capturing the stream needs to be a high-precision capturing device (nanosecond precision). It should be capturing the stream as close to the sender as possible. When analyzing a stream capture using Wireshark, this packet timestamp can be found on the frame level, labelled as epoch time.

```
packettime = Decimal(pkt.sniff_timestamp)
```

II. The Drain Rate

The drain rate isn't as easy to observe as the packet timestamp; it needs to be calculated (1).

$$T_{DRAIN} = (T_{FRAME} / N_{PACKETS}) * (1 / \beta) \quad (1)$$

Where:

- $\mathbf{T_{FRAME}}$ is the time period between consecutive frames of video.
- $\mathbf{N_{PACKETS}}$ is the number of packets per frame of video.
- $\boldsymbol{\beta}$ is the scaling factor = 1.1

III. The number of RTP packets per frame, $N_{PACKETS}$

Without diving into further details of the actual packet, a simplistic version would be to look for a flagged RTP marker bit and to store the RTP sequence number of this packet. Next, scan for the following flagged RTP marker bit, store this packets' sequence number and subtract it from the previously stored RTP sequence number. That is $N_{PACKETS}$.

The RTP sequence number rolls over to zero rather frequently as it is a 16 bit number. The modulo operator is your best friend in coping with this challenge. A careful reader might notice that this might be the amount of packets for a frame or a field. In the case of a field, this result should be multiplied by 2.

```
def frame_len(capture):
   # To calculate Npackets, you need to count the
number of packets between two rtp.markers== 1 flags.
   # This is as easy as looking at 2 rtp.markers == 1
packets and subtracting their rtp.sequence numbers.
   # The exception that will occur is that the packet
sequence number rotates. Modulo is then your friend!

   first_frame = None
   for pkt in capture:
       if pkt.rtp.marker == '1':
           if not first_frame:
               first_frame = int(pkt.rtp.seq)
           else:
               return (int(pkt.rtp.seq) - first_frame)
% 65536
   return None
```

IV. The frame time period, TFRAME

To calculate the framerate ($1/T_{FRAME}$) of a given capture, one needs to look at three consequent RTP timestamp values. The frame periods (difference between 90 kHz timestamps) might not appear constant. For example 60/1.001 Hz frame periods effectively alternate between increments of 1501 and 1502 ticks of the 90 kHz clock.

```
def frame_rate(capture):
   rtp_timestamp = []
   for pkt in capture:
       if pkt.rtp.marker == '1':
           if len(rtp_timestamp) < 3:

           rtp_timestamp.append(int(pkt.rtp.timestamp))
           else:
               frame_rate_c = Decimal(RTP_CLOCK /
                   (( (rtp_timestamp[2] -
                   rtp_timestamp[1])
                   % RTP_TIMESTAMP_BIT_DEPTH +
                      (rtp_timestamp[1] -
                     rtp_timestamp[0])
                   % RTP_TIMESTAMP_BIT_DEPTH) / 2))
               return frame_rate_c
   return None
```

As the reader may have noticed, this again might be the result for a frame or a field of video. In the case of a field, the result should be multiplied by 2. The good news is that these two values are compensated for in the equation (2) for T_{DRAIN}.

$$T_{DRAIN} = (2*T_{FRAME} / 2*N_{PACKETS}) * (1 / \beta) \qquad (2)$$

We actually don't need to know whether we are dealing with fields or frames of video to calculate C_{PEAK}.

V. The algorithm to calculate C_{PEAK}

We record the initial time of the first packet of the RTP video stream. This is the first packet that drops into the virtual leaky bucket (C_{INST} = 1). As the following packet arrives, the packet time of the previous packet will be subtracted from the packet time of the current packet and the result divided by T_{DRAIN}. The integer result of the previous calculation gives us the number of packets that are drained between the previous and the current packet (packets_drained).

$C_{INST} = C_{INST}$ - *packets_drained*
If $C_{PEAK} < C_{INST}$, then: $C_{PEAK} = C_{INST}$

EBU LIST - LIVE IP SOFTWARE TOOLKIT

I. Purpose

The major goal of the LIST project is the development of a set of open source software tools to validate, play and generate media compatible with ST 2110.
Regarding validation, LIST is able to read network traffic captured as files by a high-precision hardware device and verify the conformance of the packets' headers, payload and the timings.

In terms of media playback, LIST plays audio, video and data contained in capture files, allowing operators to check the contents manually. If it is run on capable enough hardware, LIST can play the streams from the network in real-time.

LIST is also able to generate signals that can serve as a reference for testing sink devices. This behaviour is akin to an SDI signal generator.

Additionally, and since PTP conformance is such a critical factor for the quality of Live IP implementations, LIST provides a framework for analyzing PTP packets.

II. Why develop a new library?

There are several open source projects that are able to play and generate RTP-based media streams. Among others, a few notable examples are FFmpeg [4] and GStreamer [5], both widely used in broadcasting environments. The availability of these projects raises the question of why to develop a new toolkit rather than collaborating on the extension of any of those open source offerings.

There are several reasons for us having decided to develop our own library. Firstly, the fact that one of the major goals of the project is education. We want to help

people learn how to implement support for those protocols and we want them to be the focal point of the library. Extending other implementations would certainly be valuable but anyone reading the source code would probably miss the forest for the trees. The amount of "glue" code needed and the additional boilerplate would make the learning task more difficult.

We did not, however, dismiss the importance of open source projects. In fact, we provide sample applications to show how to integrate them with LIST, leveraging, for instance, their ability to decode most media formats, allowing users to stream media decoded by, e.g. FFmpeg via LIST. Plus, LIST uses open source projects to provide horizontal functionality, for instance, memory management, logging or text formatting.

Another fundamental reason for deciding to develop our own implementation of the network and protocol handling code was efficiency. Processing Live IP media pushes the requirements of general purpose hardware to its limits. Most of the open source projects, although they provide RTP processing and generation, are geared towards applications with much less stringent requirements and usually for formats and bit rates used for distribution, as opposed to uncompressed, full resolution audio and video. We elaborate further on this later on in this paper.

III. Platforms and Languages

LIST aims at helping to develop software for both server and desktop. Hence the goal was to support Linux, Microsoft Windows and Apple macOS. However, the code should be easy to port to other platforms, if required.

Given the performance requirements, the options for programming languages ranged from the established C and C++, to newer offering such as Go [6], Rust [7] or D [8].

Despite some advantages any of the latter could offer, we immediately dismissed them for several reasons, chief amongst them being the fact that they are not as well known by the target audience as either C or C++. Additionally, despite their efficiency when compared to other languages, they are still not on a par with either C or C++.

Therefore, the choice was between C and C++. Many open source projects are developed in C and claim to be faster than an equivalent C++ implementation. However, evidence shows that, given a modern enough compiler, C++ is at least as fast as C. Additionally, C++ is much more expressive than C, providing higher level constructs, which allowed us to create a simpler, cleaner and effective design. Finally, there is a wealth of excellent C++ libraries that we could use to build upon, such as boost [9], Microsoft's C++ REST SDK [10], etc.

Given the evolution of the C++ language in recent years [11], we decided to go to the bleeding edge and base the implementation on C++17, which gives us the ability to express the code more clearly without losing efficiency. Despite the fact that it is very recent, there is excellent support in the major compilers [12], [13], [14], [15] which gave us enough confidence to aim for it.

IV. Structure

LIST is divided into four major parts: Libraries, Unit Tests, Demo Applications, and End-User Applications. It is complemented by third-party libraries and built using an open source build system generator (see Figure 5).

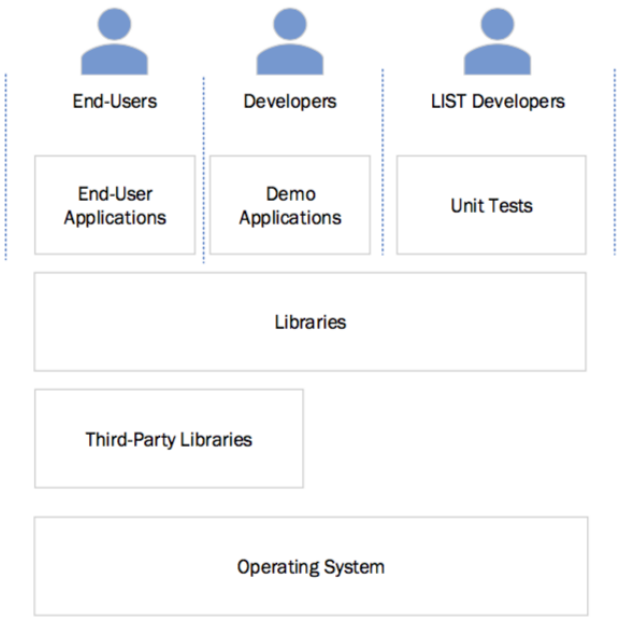

FIGURE 5: LIST BLOCK DIAGRAM.

V. Libraries

The libraries are the modules that provide the core functionality of the toolkit. This functionality includes, for instance, receiving packets from the network or reading them from a capture file, processing the RTP headers and reassembling the video frames.

The services provided by the libraries are exposed via C++ header files and are available for integration into applications by linking these with the libraries.

As mentioned above, LIST leverages third-party libraries for horizontal, non-core, functionality.

VI. Unit Tests

The unit tests verify that the behaviour of the libraries matches the specification. These include black-box tests, which test the libraries' Application Programming Interfaces (APIs), as well as white-box tests, which verify that the internals of the libraries behave correctly. LIST used the Catch2 framework [16] to assist in the development of unit tests.

VII. Demo Applications

Demo applications are small programs, with a command line interface and limited functionality. Their purpose is not to be useful tools on their own (although they may be) but rather to explain how parts of the library should be used by other developers.

VIII. End-User Applications

End-User applications are the "user facing" aspect of LIST. They are built on top of the LIST libraries and are full-blown applications, including all the interface aspects required for actual operational use.

The applications include, for instance, an application for analyzing and visualizing live and previously captured network data; a "signal generator" that is able to play captured data as ST 2110 streams; an application for receiving and displaying live streams.

IX. Third-party Libraries

LIST uses several third-party libraries, among which boost, Microsoft's C++ REST SDK [10], BIMO [17], spdlog [18] and {fmt} [19].

X. Build System

C++ software needs to be compiled and, unfortunately, the compilation process varies widely across platforms and compilers. In order to minimize this complexity, LIST uses CMake [20] to generate native make files or IDE projects, as well as Conan [21] to automatically install required dependencies. This makes the whole process of building LIST extremely simple and robust.

XI. Performance

The major challenge we found during the development of LIST was how to maximize efficiency. In fact, dealing with streams of several gigabits per second, even with powerful hardware, is not an easy task.

LIST performs very well, due to several factors:

- **Memory Management.** Memory copying and dynamic memory allocation have always been important reasons for systems to underperform. LIST addresses these problems by reusing buffers in a way that minimizes both copying and dynamic allocations, using BIMO's ability to share memory blocks intelligently.
- **Sequential Memory Access.** Another factor that causes applications to underperform is poor caching. Current processors have main memory access latencies that are several orders of magnitude higher than when data is located in the processor cache. LIST tries to maximize predictable memory access, using data structures and algorithms that are cache-friendly.
- **Parallelization and Functional-style Programming.** LIST strives to use a pure functional style, almost completely eradicating side-effects from its processing pipelines. This style lends itself to simpler parallelization, while also enhancing readability, testability and correctness.
- **Efficient Network I/O.** LIST has a special purpose Platform Adaptation Layer for network access. This layer implements an asynchronous, OS Kernel-friendly I/O model, minimizing memory copying and kernel to user mode switching and leveraging the most recent OS APIs for asynchronous I/O.

REFERENCES

[1] *ST 2110-21:2017 - SMPTE Standard - Professional Media Over Managed IP Networks: Traffic Shaping and Delivery Timing for Video*, http://ieeexplore.ieee.org/document/8165971/

[2] *Internet Engineering Task Force (IETF) RFC 3550 RTP: A Transport Protocol for Real-Time Applications*, https://www.ietf.org/rfc/rfc3550.txt

[3] *EBU ST 2110 analyzer*, https://github.com/ebu/smpte2110-analyzer

[4] *FFmpeg*, https://www.ffmpeg.org/

[5] *GStreamer*, https://gstreamer.freedesktop.org/

[6] *The Go Programming Language*, https://golang.org/

[7] *The Rust Programming Language*, https://www.rust-lang.org/

[8] *The D Programming Language*, https://dlang.org/

[9] *boost*, http://www.boost.org/

[10] *The C++ REST SDK*, https://github.com/Microsoft/cpprestsdk

[11] *ISO, C++ Recent Milestones*, https://isocpp.org/std/status

[12] *C++ Standards Support in GCC*, https://gcc.gnu.org/projects/cxx-status.html

[13] *C++ Support in Clang*, https://clang.llvm.org/cxx_status.html

[14] *Microsoft Visual C++ - Support For C++11/14/17 Features*, https://msdn.microsoft.com/en-us/library/hh567368.aspx

[15] *C++17 Features Supported by Intel® C++ Compiler*, https://software.intel.com/en-us/articles/c17-features-supported-by-intel-c-compiler

[16] *Catch2*, https://github.com/catchorg/Catch2

[17] *BISECT Media Core Library*, https://github.com/pedro-alves-ferreira/bimo

[18] *spdlog*, https://github.com/gabime/spdlog

[19] *{fmt}*, http://fmtlib.net/latest/index.html

[20] *CMake*, https://cmake.org/

[21] *Conan C/C++ package manager*, https://www.conan.io/