Real-time Streaming with Millisecond Granularity

Simon Ofner
Fraunhofer FKIE
Friedrich-Ebert-Allee 144
D-53113 Bonn, Germany
Email: ofner@cs.uni-bonn.de

Matthias Frank
University of Bonn
Institute of Computer Science 4 / Laser & Light Lab
Friedrich-Ebert-Allee 144, D-53113 Bonn, Germany
Email: matthew@cs.uni-bonn.de
http://lll.net.cs.uni-bonn.de/

***Abstract*—We investigate the conditions for streaming real-time laser data with commercial off-the-shelf hard- and software. We design scheduling algorithms to work around the identified limitations. We run end-to-end tests of our resulting software to show that we can achieve streams with millisecond granularity.**

***Index Terms*—Local area networks; UDP; Multimedia & real-time streaming; Scheduling Algorithms; End-to-end analysis**

I. Introduction

Laser shows are downright spectacular. They are highly entertaining, and it's for good reason that they are part of many a rock concert or electronic music show. This paper will take a peek behind the scenes. We'll combine real-time network communication with live laser show data.

We're going to research, design, and evaluate software methods to stream real-time laser data over local networks, at a minimum data rate of 100 kHz. This will necessitate sending data with millisecond granularity. We want to achieve this on commercial off-the-shelf (COTS) computers rather than specialized machines or peripherals.

A. Context & Motivation

There is an established infrastructure for computers to control laser projectors. The International Laser Display Association (ILDA) [8] publishes standards for use in the field. The systems consist of several components, notably media playback software on one end, one or more laser projectors on the other end, and supporting infrastructure in between [3].

The ILDA Standard Projector uses analog input signals. With playback software generally generating digital data, the supporting infrastructure converts that digital data to an analog signal. That signal is then routed through an analog cable to the projector, which ultimately displays the show [3].

Moving from an analog connection to network-based digital transmissions can be advantageous. Digital transmissions can take advantage of existing network infrastructure and solutions. They're cheap, less susceptible to interference, and decouple the playback computer and the laser projector. Cable length would also be less of an issue [2], [3].

Our work takes place in the Laser & Light Lab (LLL) at the Institute of Computer Science 4 at the University of Bonn [6]. We took advantage of the StageMate ISP [5], an FPGA-based digital-to-analog converter (DAC), which alleviates the transition process from analog to digital. It can be attached to an ILDA Standard Projector and features an Ethernet interface for input [3]. Input laser data must be provided via real-time streams of UDP (User Datagram Protocol) datagrams.

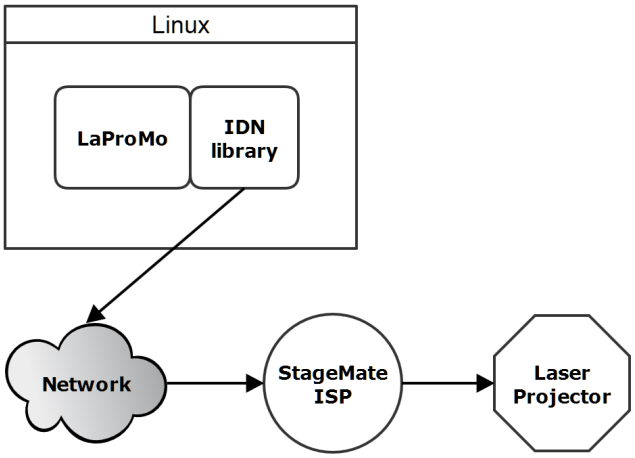

Fig. 1. LLL sample setup with IDN library and StageMate ISP

Figure 1 illustrates a sample setup with the StageMate. A Linux computer is running LaProMo, which in turn is feeding laser data to the IDN library. LaProMo is the short form of "Laser-Projection-Module", an application built and evolved in the LLL. IDN is short for ILDA Digital Network. Note that the laser projector is decoupled from the computer.

The goal of our larger project is to provide a working and stress-tested IDN library with documented programming interfaces and network protocols. It could then serve as a reference implementation in the standardization process of a digital laser projector interface. However, documenting the full project is out of scope of this paper.

Instead, we will focus on the following key challenges.

B. Challenges

The first challenge is for our software to run on COTS computers. We do not want to require special hardware or special operating systems. Additionally, we want our software to be cross-platform, running on both Linux and Windows.

Further challenges arise from the tight scheduling requirements. The DAC requires a real-time stream of UDP datagrams. At a target projection rate of 100 kHz and 100 laser data samples per UDP datagram[1], we need to send one such

[1]Dictated by typical maximum transmission unit (MTU) values

978-1-4673-6770-7/15/$31.00 ©2015 IEEE

datagram each 1000 µs.

We therefore need to investigate how well suited our target platforms are for this type of streaming. We will then need to design a scheduler capable of working around any limitations or idiosyncrasies we will have identified. Once the theoretical design is done, we will need to assess its quality in practice.

II. Building Blocks

Sleep functions are a simple and portable way to schedule work. We have provided basic research of this topic in a technical report [10]. We tested a suitable selection with regard to our need to schedule UDP datagrams in 1000 µs intervals. We tested on three different hardware configurations per platform.

On Linux, there was little to no difference between the individual sleep functions. We saw small amounts of jitter and a priori unpredictable but run-time constant oversleeping of the desired sleep time, up to 100 µs off target.

On Windows, the internal timer operates at $\frac{1\,\mathrm{s}}{64} = 15\,625\,\mu\mathrm{s}$ [11] by default. Our initial results were clustered around that value, far off the desired 1000 µs and thus clearly unacceptable.

The Windows API `timeBeginPeriod` [9] lowers the timer resolution down to 1000 µs. With this enabled, our results were generally comparable to Linux. However, oversleeping still occurred with more variance than on Linux.

Thus we could not sleep deterministically on either platform. We also saw enough jitter in the results that our scheduler had to be self-correcting and account for occasional hiccups in the sleep time.

Finally, while busy loop variants performed well on both platforms, we chose to only consider them a fallback option. Once our scheduler is integrated into a playback application, we may no longer be able to monopolize the CPU.

The full details of our measurement setup and of our results can be found in the long version of this paper on our website.

III. Scheduler

After several iterations we settled on a scheduler algorithm that we believe is both simple and powerful. Algorithm 1 shows its essence in pseudo code. The main control structure is a virtually endless loop. Its basic purpose is to call a work item once per `tick_length`. In our 100 kHz-scenario, `tick_length` is 1000 µs.

For our specific purposes, the work item is the function `schedule_packets_for`. Once it returns, the scheduler calculates for how long it should sleep to wake up at the next `tick_length` interval.

In case of `sleep_for` oversleeping, the next loop iterations make up the difference. The same happens if the work item were to occasionally take longer than `tick_length`. On average, the work item will be called the correct number of times in an average interval of `tick_length`.

Algorithm 2 presents our work item `schedule_packets_for` in pseudo code. As parameter, it receives the time until the next scheduler tick. The algorithm's job is to send the receiver enough data until that time. The function

Algorithm 1 scheduler main loop

```
1:  procedure SCHEDULER_LOOP(tick_length)
2:      next ← clock::now().
3:
4:      while !stop do
5:          schedule_packets_for(tick_length). ▷ cf. Algorithm 2
6:
7:          next ← next + tick_length.
8:          current ← clock::now().
9:          remainder ← next − current.
10:
11:         if remainder > 0 then
12:             sleep_for(remainder).
13:         end if
14:     end while
15: end procedure
```

Algorithm 2 packet scheduler

```
1:  procedure SCHEDULE_PACKETS_FOR(tick_length)
2:                              ▷ data_left set to 0 at show start
3:      while data_left < tick_length do
4:          data_sent ← send_next_packet().
5:          data_left ← data_left + data_sent.
6:      end while
7:
8:      data_left ← data_left − tick_length.
9:  end procedure
```

`send_next_packet` sends one UDP datagram or "IDN packet" and returns the time the data in that packet represents. Its implementation is not shown here, as it is not relevant to the discussion at hand.

Once a datagram is sent, `schedule_packets_for` remembers in `data_left` if more data than necessary was sent for the next time it is called. In our 100 kHz-scenario, `data_sent` will always be 1000 µs and therefore lines up with `tick_length`, but generally this algorithm supports any sensible configuration of parameters.

IV. End-to-end Testing

To put our scheduler to the test, we ran a slew of end-to-end tests. We wanted to find out if there were any deviations from our designs and expectations in practice. It is entirely possible that our COTS infrastructure proves to be a bottleneck at 1000 µs scheduling granularity - especially since we're already pushing towards the boundaries on Windows (cf. section II).

A. Tools and Terminology

To aid in the testing process, we wrote a custom utility called `idn_server`. It served as an emulated endpoint for our UDP data stream. It has sufficient understanding of the laser data network protocol to extract various data and metadata from each datagram.

To parse the data, we wrote `idn_parser` as an offline analysis tool. Among its features is the calculation of inter-packet arrival times. The difference of the arrival times of two adjacent packets make up the *packet offset*. By keeping track of all packet offsets in a data stream, `idn_parser` can derive a number of statistics. For example, it can generate histograms of the packet offsets, shown in this paper, which help analyzing the sender's scheduling behavior.

It can also compute the *average* packet offset and the *expected* packet offset. Each datagram includes the amount of projection display time its payload represents. Thus the tool can compute the packet offset the datagrams *should* have arrived with. For example, for a stream with a constant rate of 200 kHz, the metadata would indicate 500 µs of display time per datagram, which would therefore be the expected packet offset. The difference between the observed average packet offset and the expected packet offset reveals whether the sender's scheduler was consistently too slow or too fast.

In our 100 kHz-scenario, the expected packet offset is fixed at 1000 µs.

B. Methods

Each end-to-end test consisted of one host sending laser data with a playback application and another host receiving the data stream. The two computers were connected by a 1 Gbit/s Ethernet link and an appropriate switch.

To keep our systems close to COTS conditions, we did not change the default configuration settings of the operating systems and kept the number of installed applications low. As the only exception, we changed power management options, to avoid hard disks or monitors turning on or off during our test runs.

To further minimize the amount of spurious activity, we did not run any applications other than the measurement tools. We made sure not to produce additional disk or network I/O. We did not interact with the machines while the tests were running. We allowed each machine a "warm-up" phase of 15 minutes to make sure any automatic start-up applications or services had run to completion.

On the sender side, we chose different configurations depending on the platform as described in their respective sections. We made sure that the input data from the playback software, our internal scheduler, and the output data streams were synced at 100 kHz. We thus tried to minimize the likelihood that deviations from a 100 kHz pattern in the result data were caused by our measurement setup.

In each case, we were sending a single data stream to our receiver for about 60 seconds. Each test run consisted of a combination of five individual runs. If necessary, the data was normalized to 60 seconds.

As receiving host, we chose a `Core 2 Duo E7500` with `Ubuntu 14.04 LTS`. A single instance of `idn_server` was running at all times.

The results are primarily shown in a series of histograms. They show the distribution of packet offsets received by `idn_server`. Note that the y-axis in each histogram is scaled logarithmically. This is necessary to discern some of the finer detail, especially on Windows. Each bucket on the x-axis is 100 µs wide. The x-axis is clipped at 17 000 µs, because any data points beyond that were not discernible in the plots. We kept the axis scales the same for each plot.

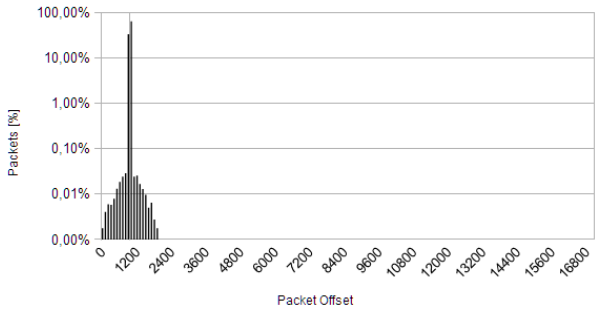

Fig. 2. End-to-end test on Linux

The histograms also feature a vertical line at the 1000 µs mark on the x-axis, which is the expected packet offset our scheduler is aiming for, given the 100 kHz setup.

C. Linux

Since we found Linux to have the best technical foundation for our scheduler in section II, we started our measurement series on this platform. We used a clone of the receiver as sending host and `IDN_lasetest` as the playback application generating its built-in test pattern. `IDN_lasetest` is a Linux tool adapted from `easylasetest` [1]. We modified it to use our scheduler and lifted its internal cap on playback speed to allow for 100 kHz (and above) streams.

Figure 2 shows the results of our first run. With 99.78% of all packet offsets falling in the 900 µs and 1000 µs buckets, we call this test a resounding success.

In our results we find no signs that the operating system, the CPU, the network infrastructure, or indeed any external factors are impeding the performance of our software. It is doing its work as intended. This is a step towards confidence that any deviation from this performance is due to factors external to our design.

D. Windows

For Windows, we used a `Core 2 Duo T9600` with `Windows 7` as the sending host and `MediaLas M-III` as the playback application with its included `Halloween` show. `MediaLas M-III` is a commercial Windows application [4] with a plugin architecture. We integrated our scheduler into that architecture.

In figure 3 we see `timeBeginPeriod` in action. With 84.70% of the offsets falling in the 900 µs and 1000 µs buckets, it is notably worse than what we saw on Linux. However, a value of 84.70% still represents the vast majority falling into an acceptable range.

Contrast this with disabling the `timeBeginPeriod` optimization for figure 4. Here we clearly see Windows' default

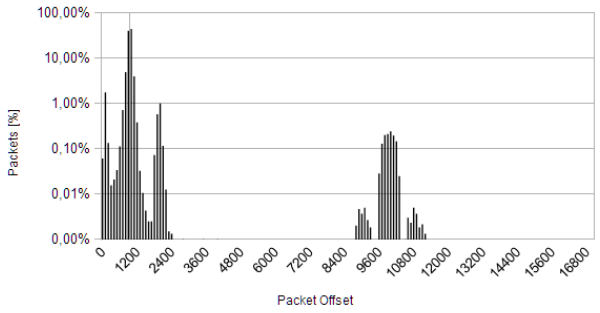

Fig. 3. End-to-end test on Windows with `timeBeginPeriod` enabled

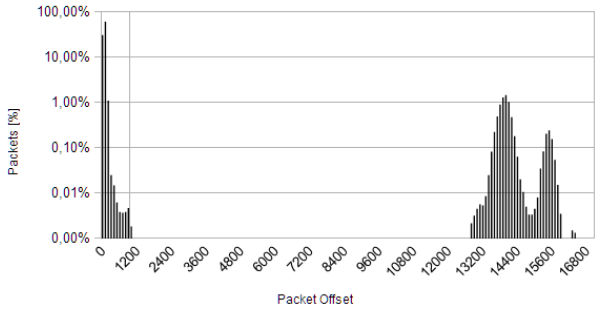

Fig. 4. End-to-end test on Windows with `timeBeginPeriod` disabled

scheduler resolution of $15.625\,\mu s$ dominating the results. Almost no offsets fall in the region of $1000\,\mu s$. Instead, most offsets are very small, indicating that the scheduler spends the vast majority of its time catching up to missed scheduler ticks, sending multiple packets in bursts with little to no delay.

In contrast to the Linux results, it is evident that our Windows infrastructure significantly impacts our performance. It is not entirely clear how much our toolchain or playback software contribute to this. But considering that an obscure yet simple Windows API had such tremendous influence on the results, it stands to reason that the default Windows configuration is at least partially at fault.

E. Findings

In each case on either platform, the average packet offset was less than $1\,\mu s$ off the target $1000\,\mu s$. Our scheduler successfully kept up with the demand of streaming at $100\,kHz$ on average, even in face of the challenges on Windows.

The results on Linux were excellent and require no further analysis at this time. On Windows, the default timer configuration showed undesirable behavior. In contrast, the results with `timeBeginPeriod` enabled were more acceptable.

V. Conclusion

We have successfully built a scheduler capable of producing real-time UDP laser data streams – even on COTS hard- and software. From its low-level building blocks to its end-to-end behavior we have covered a wide range of aspects of design and implementation.

At the lowest level, we discussed sleep functions on Windows and Linux. We then presented a scheduler design to work around their peculiarities. Despite its simple algorithms, it proved to be quite powerful even when embedded into larger laser show playback applications.

In our end-to-end tests, the Linux setup performed excellently. No factors external to our implementation impeded our performance. The default Windows configuration showed clearly undesirable behavior. The results were much improved with a simple tweak, though still inferior to Linux.

Thus, comparing all three sets of results, it seems evident that the configuration of the Windows operating system plays a significant role in real-time scheduling performance.

There may exist further tweaks to improve the performance on Windows. We did not follow this lead any further as we were focusing on off-the-shelf behavior, but it may be a worthwhile research endeavor nonetheless.

In the context of our larger project we looked at more design, implementation and performance issues and performed many more measurements that could not fit this paper. We discussed and demonstrated some of these at a technical seminar at the 2014 ILDA Conference [7] and have included others in the long version of this paper on our website.

Acknowledgment

The authors would like to thank Dirk Apitz from Dexlogic Karlsruhe for providing early IDN protocol descriptions, StageMate ISP hardware, and excellent technical assistance.

References

[1] M. Elektronik, *easylasetest*, Spaichingen, Germany, http://www.jmlaser.com/ (last accessed: 2015-07-27).

[2] M. Frank, *An Experimental Architecture of IP-based Network Control of Laser Show Projection Systems*, IEEE LCN 2011 demo pub., Bonn, October 2011, http://www.ieeelcn.org/prior/LCN36/lcn36demos.html (last accessed: 2015-07-27).

[3] ——, *Demonstration of Bandwidth Demand and Jitter Properties of a Software Sender/Scheduler for the (proposed) ILDA Digital Network*, IEEE LCN 2014 demo publication, Edmonton, September 2014, http://www.ieeelcn.org/prior/LCN39/Program_demos.html (last accessed: 2015-07-27).

[4] M. E. GmbH, *M-III Lasershow Software*, Balingen, Germany, http://www.medialas-showlaser.de/m3.html (last accessed: 2015-07-27).

[5] D. Hardware and S. Solutions, *StageMate ISP*, Karlsruhe, Germany, http://dexlogic.com/work/4108-isp/StageMate-ISP-en.html (last accessed: 2015-07-27).

[6] *Laser & Light Lab (LLL)*, Institute of Computer Science 4, University of Bonn, http://lll.net.cs.uni-bonn.de (last accessed: 2015-07-27).

[7] *2014 ILDA Conference*, International Laser Display Association, Las Vegas, USA, http://www.laserist.org/c2014-day3a.htm#IDN (last accessed: 2015-07-27).

[8] *ILDA*, International Laser Display Association, Portland, USA, http://www.laserist.org/ (last accessed: 2015-07-27).

[9] MSDN, *timeBeginPeriod*, Microsoft, http://msdn.microsoft.com/en-us/library/dd757624(v=vs.85).aspx (last accessed: 2015-07-27).

[10] S. Ofner, *Application-level timing and scheduling techniques on Unix and Windows*, Seminar report, LLL, University of Bonn, June 2013.

[11] M. Russinovich, *Inside the Windows Vista Kernel: Part 1*, Microsoft Technet Magazine, http://technet.microsoft.com/en-us/magazine/2007.02.vistakernel.aspx (last accessed: 2015-75-27).