How can we leverage networks and distributed systems to reimagine real time audio applications?

Using networks such as the Internet to do real time processing on audio streams has obvious limitations due to latency and package loss. Audio processing requires special hardware designed for the task. In the past, much of the computing industry has revolved around the prediction by Moore's law, which specified that roughly every 18 months the computational power of our technology increases by a factor of 2. Although we have reached a post-Moore-law era, were our innovation is stagnated by the physical limitations reaching as far as the size of a single atom. The transistors used to make a CPUs will not continue to shrink in size, but the effectiveness of the hardware will continue to improve through software innovation and especially the use of networks and distributed systems.

The bandwidth of networks has not stagnated in growth like the shrinkage of transistors has, cable networks commonly found in homes can be expanded to 10Gb/s while optical hardware can go even beyond that. The remaining limiting factor of using networks is the latency introduced by the various protocol layers built on top of the hardware.

Bela is a modern and innovative approach to reimagining embedded audio hardware; it can run patches made with PureData but it's true strength comes from writing custom C++ software with the libraries it provides. In this paper I discuss why C++ was the clear choice for enabling Bela and equivalent devices to be able to achieve what it can, but I continue to question whether the choice of language is not ultimately a barrier in making audio programming more accessible to the greater developer and artist community.

In this thought experiment I bring forward an analysis of current technology and make the suggestion that a language such as Elixir/Erlang would have its own costs but ultimately expand the accessibility of this niche in software engineering. My hypothesis is that in the past, ease of use was sacrificed for speed but that this sacrifice is no longer necessary since even languages that operate in soft real time can be leveraged in ways to enable real time audio processing.

One of the greatest barriers in real time audio processing on ordinary computers is the operating system specific scheduling implementations of the processor. The OS scheduler is part of the Kernel and is the software that decides which running program gets CPU time. When the scheduler switches between two running programs it is called an interruption, the fewer tasks an operating system must deal with the fewer interruptions a single process will receive. A process is assigned a priority when it is launched, depending on the scheduling implementation this means that the CPU will be less likely to interrupt the process. Two common reasons for interrupting a process that cannot be avoided are; Disk I/O and Garbage Collection.

Disk Input and Output are the procedures that copy data between RAM and the Hard disk. Garbage Collection is the process of discarding unused memory from RAM. Depending on the implementation of the programming language there have been many approaches to minimize the need for these two procedures. In C, the programmer is responsible for initiating GC, in Java GC occurs automatically whenever it sees fit, Rust has been able to remove the need of garbage collection by introducing a new system for variable scoping.

Programming audio applications was an endeavor previously reserved for large teams of experienced developers embedded in corporations due to an enormous amount of complexity in real time signal processing. These barriers have started to be torn down through software simplifications and abstractions such as MaxMSP and PureData. Audio applications such as these are created using low level and compiled languages like C and its relative C++ and abstract away the complexities inherent to them.

Visual programming, as it is possible in PD and Max, is not for everyone and has its own shortcomings. I believe that language oriented programming and domain specific languages are a step towards innovation away from the overuse of frameworks and libraries in C based languages. Audio specific Libraries and Frameworks for C++ work great as long as they offer convenience but in my experience they ultimately create barriers of maintainability and readability in software long term that macro based solutions like the one I am recommending do not.

What are the features in a language that we are looking for?

- Macros
  - Abstraction possible not only using function but also macros
- Native Implemented Functions (NIFs)
  - The ability to write native Rust or C++ and integrate it into another language
- Scheduling Control
  - Ability to prioritize some processes over others.
- Distributed Architecture
  - Ability for different nodes to communicate seamlessly

The candidate: Erlang.

Erlang is a compiled language that runs on the BEAM virtual machine, and has its own implementations of a scheduler and memory management. The BEAM virtual machine similar in its nature to how the JVM runs Java, has been ported to all mayor platforms including MacOS, Windows and a variety of Linux distributions. I will focus on the implementation on top of MacOS and Linux because Windows has some barriers unique to only itself.

Elixir is a macro-based programming language built on top of Erlang, is fully backwards compatible with it and can therefore be used interchangeably; Elixir is easier to read and reason about due to immutability, compiles to Erlang, which is compiled to BEAM byte code.

Elixir allows for NIFs to be used wherever the programmer needs hard real time processing.

The BEAM will create one OS process for each core the computer has, these process typically run at a normal priority but can be instructed to have higher priority. Internally, the BEAM has its own notion of processes that are not interchangeable with operating system processes. These internal processes are assigned by the BEAM to one of the schedulers. Erlang processes have three levels of priority; low, normal and high. In general the BEAM prefers to have the entire system move slowly than have a single process steal CPU time from all the others, therefore all processes run at normal priority. So long as there is a process with high priority the BEAM will not schedule a process with normal or low.

Elixir is build around the notion of Nodes, where a node is one instance of the BEAM. Each node can connect to other nodes over the network and communicate via remote procedure calls. Erlangs RPC implementation uses a TCP.

Using a C++ application to drive a stereo works great, expanding it to serve 2000 speakers such as the AudiMax at the TU Berlin requires a extensive hardware investment to have a single computer serve so many speakers. Porting a C++ program to a distributed system requires intensive redesign, whereas if the underlying architecture was already written in Elixir, migrating it to a distributed infrastructure is exactly what Erlang was built to do. Instead of a single computer with many audio interfaces, each speaker could be connected to its own embedded device, the remaining problem is the latency introduced by the network.

Network Transport Layer

HTTP and HTTPS, known for serving the World Wide Web, are application protocols implemented on top of TCP, a transport layer protocol. The internet as the public knows and uses this unless there is video and audio streaming involved. There are key differences in the two mayor transport layer protocols, UDP and TCP, that are important to understand in order to reason about distributed audio applications.

- TCP is bidirectional, and Fair
- UDP is monodirectional and Unfair

W “fairness” has to do with networking technology? It is a subset of the larger problem commonly known as Net Neutrality. The topology of the internet is quite complex but can be broken down into three layers;

1. A strongly connected local area network (LAN)
2. A weakly connected wide area network (WAN)
3. An almost-strongly connected network of Internet Service Providers (ISP’s)

The layer that causes the most latency is layer 3. In order to enable the fastest speed possible an Internet Service Provider cannot deep-package-scan every packet that goes through their system, therefore they can only discriminate on information that they can read the in package header, which only contains information such as Source IP, Destination IP and **transport protocol**.

In this arbitrary choice of network topology, it was unofficially agreed upon that it would not be the ISP’s responsibility to throttle certain traffic from another but that the implementations of TCP built into every operating system and software library would do this for them. The process of establishing a TCP connection between two computers in a network is called the TCP Teardown algorithm.

The TCP teardown algorithm is implemented to solve two distinct problems inherent to the architecture of the Internet Protocol (IP). First, any single package has no guarantee to ever be delivered. Second, a package sent to an IP address that does not exists is indistinguishable from a package that was lost along the way. The TCP teardown will therefore send a handshake package first, wait for acknowledgement, and acknowledge the receiving of the acknowledgement.

After the TCP handshake, the sender will send 1 data package and wait for acknowledgment, then send 2 packages and wait for acknowledgement for each, then 4, 8, and 16 etc. The exact numbers chosen often differ from one implementation to the next but usually follow an exponential curve. If a package is not acknowledged, this can have two reasons; either the data package never arrived, or it was delivered and the acknowledgement package was lost. In either case; the unacknowledged package will be resent, and the algorithm will take an exponential step **backwards** and send less packages in the next batch.

This solves the obvious problem of when you want to send a 1 GB file, and the first 100 packages never arrive, there is no point in clogging the network with the rest. The above therefore describes a fair TCP implementation, but by modifying the underlying algorithm it is entirely possible to create an aggressive (or unfair) implementation and has been done many times over. This aggressive implementation is unfair because when one side loses connection for example, the network will handle more traffic than it needs resulting in more latency for everyone else.

UDP is different from TCP in many ways, there is no initial handshake, received packages are never acknowledged and package that are lost are ignored and never retried. This makes UDP an aggressive and “unfair” protocol but desperately necessary to enable services such as audio and video streaming. A dropped frame or corrupted audio sample is a necessary sacrifice to keep the entire system running.

ISP have widely accepted this state of the matter to enable streaming platform like YouTube, Netflix, Amazon Prime. But ISP still monitor and detect the uses of aggressive TCP clients and are known for throttling those people’s connection as punishment. This although being a disputed position, does not stop us from using aggressive TCP implementation on our own and private local area networks, so long as this traffic does not congest the larger internet architecture.

Ultimately, to enable our distributed audio application we can therefore have the best of both; limited package loss due to the use of TCP and the benefit of low latency due to the rewriting of the TCP teardown algorithm to be more aggressive like UDP.

By swapping out the TCP client implementation used by Erlangs Node interface a much faster yet reliable communication is possible, this comes at the costs of regular network traffic being slowed down.

Conclusion

…

Annotated Bibliography

Martin P. Ward. Language oriented programming. *Software Concepts and Tools*, 15:147–161, April 1994.

Language oriented programming is not a new topic, since Martin P. Ward was able to accurately describe in 1994 what many developers today are recognizing to be a good approach to organize software. Mature software is based on reliability, but changes to a software's behavior are frequent and as business objectives change, so will the programs running. Developers must maintain software for many reasons, and any design guidelines that standardize this process are an advantage.

Culpepper, Ryan, et al. *From Macros to DSLs: The Evolution of Racket**. 2019.

Racket is a programming language developed and maintained by university professors many of whom teach at Northeastern Universities' Khoury College of Computer Science, Racket is similar to Elixir is some ways e.g. the abstract syntax tree is valid Elixir/Racket code. This common foundation powers the macro system in Racket and is a benefit in Elixir as well.

"TCP vs UDP." *Diffen*, www.diffen.com/difference/TCP_vs_UDP.

TCP and UDP have very specific use cases, TCP is used when data needs to be reliably sent from one computer to another, TCP guarantees that even if packages are lost along the way, in the end the data is successfully transmitted unless the connection is interrupted. This sacrifice of speed for reliability is something not built into UDP.

"Erlang -- Nifs". *Erlang.Org*, 2020, http://erlang.org/doc/tutorial/nif.html. Accessed 20 Apr 2020.

Native Integrated Functions or NIFS allow a developer to embed foreign languages into Erlang. NIFs come with certain draw backs, e.g. if a fatal error occurs while an erlang process is executing a NIF the entire BEAM virtual machine will crash.

"Distribunomicon | Learn You Some Erlang For Great Good!". *Learnyousomeerlang.Com*, 2020, https://learnyousomeerlang.com/distribunomicon. Accessed 20 Apr 2020.

Erlang is a programming language developed by the telephone line company Ericson for their switch boards. Reliability is built into the language for that reason, but so is a native remote procedure implementation that allows one Erlang Node to communicate with another. With this architecture, it is possible to design a system with redundant servers that can serve customers without downtime even if a single node fails. Nodes operate in a strongly connected fashion; If node A connects to B, and then C connects to B, A is automatically connected to C also.

"TCP Connection Establish And Terminate - Tutorial". *Tutorial*, 2020, https://www.vskills.in/certification/tutorial/information-technology/basic-network-support-professional/tcp-connection-establish-and-terminate/.

The TCP teardown algorithm differs from one implementation to the next but common structures exist and are used by major software companies. Creating fair TCP clients has many advantages to the network at the cost of latency for the individual.

Scraps:

Case Studies:

- Using C++ on Bela
  - Bela is a tool for developers to create embedded audio systems. This is the perfect starting point to introduce the basics of real time audio processing.
- Creating a modern audio application in Elixir
  - Elixir is built on Erlang and is a very fancy new language. Its syntax is much easier to grasp than that of C++ and creating abstractions over patters will make audio software more maintainable.

More Ideas

Using high level programming languages for digital signal processing has advantages due to easier syntax, safer garbage collection and integration opportunities. The main disadvantage is speed, due to the soft real time nature of modern languages.

To create new technologies for audio processing we must innovate the current technology.

Difference between soft real time and hard real time in programming.

Is Max MSP a programming environment? Is it more like an IDE?

Experiment: Creating an Audio environment in Erlang / Elixir. How can we circumvent the shortcomings of soft real time programming but leverage the power of the erlang scheduler?

What is latency?

Latency is the time delay between encoding and decoding an analog signal to digital. All things USB (sound card) add another 5ms delay.

What is the control plane?

The control plane is the functionality that reads user input and manipulates the audio signal in real time to the information provided by the user.

How do computers typically process signals, especially audio signals. What are the parallels to images / photoshop?

Computers process signals in *blocks,* a block consists of a number of samples.

What possibilities does the internet open up for signal processing. In 2015 the average internet bandwidth in the US exceeded 50mb/s, which is plenty for an HD stereo audio stream. What kind of possibilities does this open up for possible