PIPES: A NETWORKED RAPID DEVELOPMENT PROTOCOL FOR SOUND APPLICATIONS

Paolo Marrone
Orastron Srl, Sessa Cilento, Italy
University of Udine, Udine, Italy
paolo.marrone@orastron.com

Stefano D'Angelo
Orastron Srl
Sessa Cilento, Italy
stefano.dangelo@orastron.com

Federico Fontana
University of Udine
Udine, Italy
federico.fontana@uniud.it

ABSTRACT

The development of audio Digital Signal Processing (DSP) algorithms typically requires iterative design, analysis, and testing, possibly on different target platforms, furthermore often asking for resets or restarts of execution environments between iterations. Manually performing deployment, setup, and output data collection can quickly become intolerably time-consuming. Therefore, we propose a new, experimental, open-ended, and automatable protocol to separate the coding, building, and deployment tasks onto different network nodes. The proposed protocol is mostly based on widespread technology and designed to be easy to implement and integrate with existing software infrastructure. Its flexibility has been validated through a proof-of-concept implementation. Despite being still in its infancy, it already shows potential in allowing faster and more comfortable development workflows.

1. INTRODUCTION

A primary focus within the audio software industry is the development of algorithms that create and manipulate digital audio streams. These algorithms serve as fundamental building blocks of end-user products like digital synthesizers and effect processors.

Audio DSP engineers typically rely on various prototyping tools to iteratively design and analyse sound processing algorithms, assessing their features through quantitative measures, graphical plots, and auditory evaluations. These tools encompass a spectrum of programming languages and frameworks, ranging from numerical simulation environments like MATLAB and Julia to general-purpose languages such as Python and JavaScript, as well as domain-specific tools like Csound[1], Pure Data[2], Max/gen[3], Faust[4], SuperCollider[5], and Ciaramella[6]. The latter ones often allow immediate execution, based on interpretation or just-in-time compilation.

Following the prototyping phase, DSP algorithms undergo implementation and integration into final products. This is typically carried out in high-performance compiled general-purpose languages such as C and C++, although their underlying computational models are not particularly well-suited for DSP applications [1]. Indeed, the compilers of some domain-specific languages can act as source-to-source translators that produce C/C++ code (e.g., Faust [2] and Ciaramella [3]), thus facilitating the implementation process.

Further development effort goes into adapting software to different formats and platforms, such as audio plugins implementing standardized Application Programming Interfaces (APIs) like VST[7], Audio Unit[8], LV2[9], AAX[10], and CLAP[11], stand-alone audio applications running on desktop and mobile platforms, or embedded software to be executed on top of lightweight real-time operating systems or directly on bare metal.

Sometimes these algorithms are also executed in a network context, as for example inside a host such as SuperCollider that is based on a client-server architecture. In this model, the client hosts the control and/or (part of) the audio source code, while the server actually runs the audio processing algorithms. This setup is particularly suitable for interactive programming [4, 5] and live coding [6, 7], enabling runtime code changes without interrupting the audio processing. Moreover, client-server architecture facilitates distributed execution across networked nodes.

1.1. Problem and proposal

Our objective is to develop an open protocol for rapid algorithm development, testing, and deployment across multiple targets, potentially simultaneously. This protocol must support various programming languages and enable networking, empowering developers to utilize lightweight clients.

Since runtime performance is of utmost importance in audio DSP algorithms, we are bound to the transmission and execution of native binary code for each target platform. At the same time, we do not want to hinder a comfortable development workflow. For example, when testing a new project iteration, the developer should not need to restart the execution environment, which could be a full digital audio workstation (DAW) if developing a plugin, when testing a new iteration of the algorithm. These two requirements appear to partially conflict, therefore a good part of the proposed framework is dedicated to overcoming this specific issue.

Ours is not the first attempt at mitigating this very problem. For example, CMajor provides a "JIT engine and hot-reloader"[12] and Faust researchers implemented FaustLive [8] which allows to

[1] https://csound.com/
[2] https://puredata.info/
[3] https://docs.cycling74.com/max8/vignettes/gen_topic
[4] https://faust.grame.fr/
[5] https://supercollider.github.io/
[6] https://ciaramella.dev/

[7] https://steinbergmedia.github.io/vst3_dev_portal/
[8] https://developer.apple.com/documentation/audiounit
[9] https://lv2plug.in/
[10] https://developer.avid.com/aax/
[11] https://cleveraudio.org/
[12] https://cmajor.dev/

modify Faust code at runtime without interruption. However these approaches are strictly tied to specific tools and miss to provide a general solution and an open framework. Meanwhile, in the realm of networking, there is currently no open standard for remote audio software deployment. While tools like SuperCollider offer some related features, likewise these also remain specific to their environments.

In this work we propose a protocol, named PIPES, designed to overcome these issues. Central in PIPES is the separation of concerns w.r.t. coding, building, and deployment. The system comprises three distinct network node types, and a PIPES network is defined as a set of such node instances, with at least one per type. Every node is specialized in a specific and independent task, thus resulting in Orchestrator, Compiler, and Player nodes. In essence, Orchestrator nodes transmit source code to Compiler nodes, receiving back compiled binaries, and deploy such binaries to Player nodes for execution. Communication between nodes is standardized and based on the HTTP protocol.

This architecture brings significant benefits. Firstly, it enables interactive programming at a DSP level with full native performance through a relatively simple form of Dynamic Software Updating (DSU) [9, 10]. Also, its simplicity and use of established standards and infrastructure make it relatively simple to implement. Then, it also allows for Orchestrator nodes, which effectively acts as the main interface for developers, to be lightweight, hence relieving them from the burden of carrying compilation software; indeed, in a hypothetical scenario, a developer could utilize a graphical audio environment on a tablet device, compile on a remote server, and deploy to a number of network-attached embedded systems. Finally, it adds a modest amount of complexity to development ecosystems as sole cost for faster development workflows.

For this to become possible, audio DSP algorithms must be wrapped in modules that adhere to a unified API. We have designed Yet Another Audio API (YAAAPI). Such an API is deliberately simple, shielding developers from platform-specific intricacies, and easy to interface with existing tools, thus also representing an easy output target for domain-specific language compilers.

As a proof of concept, we implemented PIPES targeting Linux VST3 for x86-64 and arm64 architectures. Additionally, we took the Zampogna compiler for the Ciaramella audio programming language [3] and upgraded it to produce YAAAPI modules.

This paper is organized as follows. Section 2 delineates the basics of PIPES, describing its architecture and components;. Section 3 exposes the current proof-of-concept implementation and, especially, the DSU technique used. Section 4 asses the current state of the protocol, its strengths and weaknesses, and the potential future directions of development. Finally, Section 5 concludes this work.

2. PIPES

PIPES is a distributed model for rapid audio application development, compilation, and deployment. In PIPES, separation of roles is fundamental: three different node types are defined and each one is specialized for an independent task, as detailed in Section 2.1. A combination of such nodes, with at least one node per typology, constitutes a PIPES network, an example of which is shown in Figure 1. The communication protocol and logic are discussed in Section 2.2. Source code and binary modules which are exchanged among nodes must implement YAAAPI, as described in

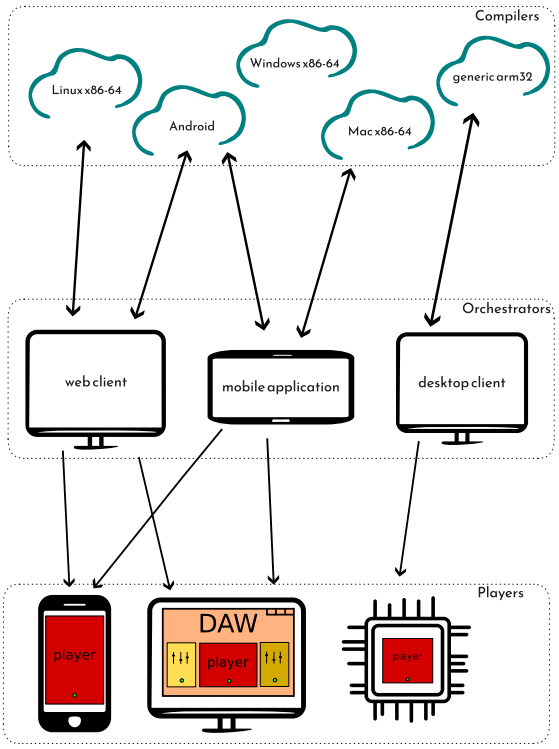

Figure 1: PIPES network example. Compiler nodes, at the top, are tailored to specific targets; Orchestrator nodes, in the middle, are end-developer applications; finally, Player nodes, at the bottom, can be stand-alone applications, plugins running inside host environments, or firmware.

Section 2.3.

2.1. Nodes

2.1.1. Orchestrator

The **Orchestrator** node works as the main interface for developers, and facilitates the coding of audio modules. It does not directly compile or execute algorithms, conversely it sends user-supplied source code to Compiler nodes, receives the resulting executable binaries back, and then forwards them to Player nodes for execution.

It is the only node that initiates communication with other nodes, therefore also the only one that needs to be actually aware of the PIPES network configuration, i.e. it knows the IP addresses and ports of the other nodes. Other than that, it is just required to receive user input and exchange data over HTTP. Thus, Orchestrator nodes can be really lightweight and simple to implement, potentially ranging from simple command-line scripts to full-blown Integrated Development Environments (IDEs), and could reasonably run on a variety of different devices or live in a web page.

No specific provision is made regarding the source code language, as long as Compiler nodes are able to convert the supplied code to YAAAPI modules. In effect, this means that any language, including domain-specific ones, can be easily supported provided

that a Compiler node handles it. On the other hand, this arrangement also accommodates the use of source-to-source transpilers in the Orchestrator nodes if needed or wanted, which might better fit the case of certain domain-specific languages.

2.1.2. Compiler

The **Compiler** node, a heavyweight computational unit, compiles source code received from Orchestrators. It can be specialized for a particular architecture, operating system, and programming language, and produces binary modules that are meant to be loaded by Player nodes. Consequently, the output must be in a shared library format, such as ELF[13], Mach-O[14] Windows PE[15], or WebAssembly[16], and must expose a symbol table containing YAAAPI function and data symbols. This allows YAAAPI modules to be written in any language that can be compiled to a shared library, besides the API being specified in C language.

In practice, Compiler nodes typically run on local or remote desktop or server machines, and are implemented by scripts or applications exposing an HTTP connection and calling into already existing compiling/building software infrastructure.

2.1.3. Player

The **Player** node receives and executes YAAAPI modules, largely abstracting the specifications of the execution environment. Upon receiving a module, it is loaded into memory and used to process audio. A Player processing element set to be permanently in execution: it is designed to run a single module at any given instant or otherwise either act as a bypass, simply copying input to output, or produce silent output, depending on the context.

Players must rapidly and correctly swap modules. The new module must be properly setup before actual usage and the old module must be replaced without interruption and finally removed. This typically involves some form of concurrent programming and requires related synchronization to be appropriately applied.

Players can take many forms: they can be standalone applications, embedded software, or act as plugin wrappers exposing, e.g., a VST3 or Audio Unit interface, and thus running inside a host application. They can also fulfil different purposes, e.g., they could be real-time processors, offline sound generators, or analysis/testing devices.

2.2. Workflow

Standardized interactions between Orchestrator, Player, and Compiler nodes are essential to enable seamless workflow. In PIPES, all communication among nodes is based on the HTTP protocol, ensuring reliability and simplicity in data transmission. This approach allows for straightforward integration with existing network infrastructure and promotes ease of implementation across various platforms. Furthermore, if encryption is needed, it is possible to simply switch to the HTTPS protocol, while authentication can be provided by means of HTTP Authentication[17].

[13] https://refspecs.linuxfoundation.org/elf/TIS1.1.pdf

[14] https://en.wikipedia.org/wiki/Mach-O

[15] https://en.wikipedia.org/wiki/Portable_Executable

[16] https://webassembly.org/

[17] https://datatracker.ietf.org/doc/html/rfc7235

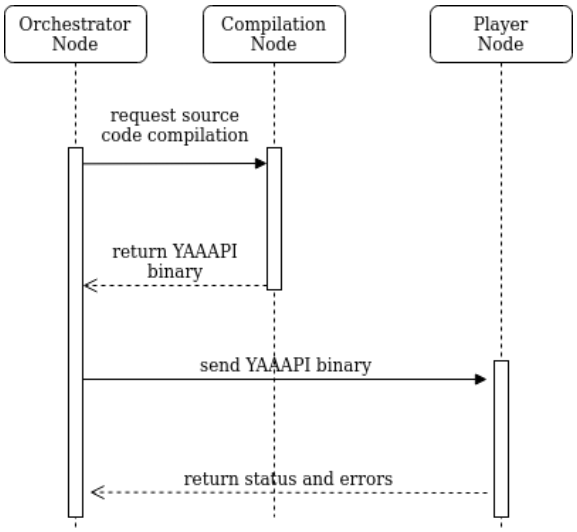

Figure 2: Sequence diagram. Orchestrators delegate code compilation to Compiler nodes and forward the received binaries to Player nodes.

Figure 2 shows the typical sequence of operations in a PIPES network. Developers program the audio application via the Orchestrator node. The source code is sent, via HTTP requests as a set of files wrapped in a JSON object[18], to an arbitrary number of Compiler nodes. These return, by means of HTTP responses, compiled YAAAPI modules, each tailored to a specific target. The Orchestrator, then, sends, through HTTP requests, the received binary modules to Player nodes for execution, matching compilation targets and Player environments. Upon module reception, Players execute the new modules and, eventually, send back some application-specific output response.

All PIPES nodes can reside on the same machine, encompassing the traditional workflow with the developer working on a fully-featured terminal.

2.3. YAAAPI

We defined a C API, named Yet Another Audio API (YAAAPI), to interface with audio modules. It can be seen as a simplified analogy of current plugin APIs covering fundamental functions for audio processing, state reset, and parameter updates. It also includes basic handling of common event messages, such as note on/off, to support the development of synthesizers. YAAAPI modules need to implement these functions and set a number of global variables.

```
// YAAAPI Functions
void yaaapi_init ();
void yaaapi_fini ();
void yaaapi_set_sample_rate (float fs);
void yaaapi_reset ();
void yaaapi_process (const float** x, float**
    y, int n_samples);
void yaaapi_set_parameter (int index, float
    value);
float yaaapi_get_parameter (int index);
void yaaapi_note_on (char note, char vel);
void yaaapi_note_off (char note);
```

[18] https://www.json.org/

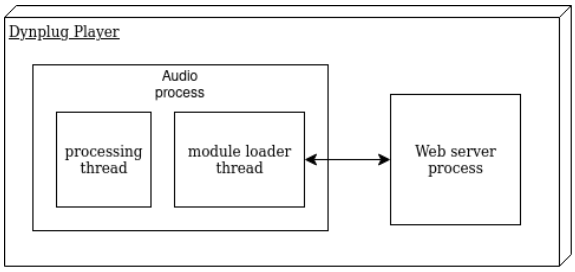

Figure 3: dynplug player structure.

```
void yaaapi_pitch_bend (int value);
void yaaapi_mod_wheel (char value);

void yaaaeapa_get_parameter_info (int index,
    char** name, char** shortName, char**
    units, char* out, char* bypass, int* steps
    , float* defaultValueUnmapped);

// YAAAPI Variables
int yaaapi_parameters_n;
int yaaapi_buses_in_n;
int yaaapi_buses_out_n;
int yaaapi_channels_in_n;
int yaaapi_channels_out_n;
```

Although not yet standardized, the current YAAAPI version suggests a concrete possibility to adopt a single common API for all targets.

3. IMPLEMENTATION

As a proof of concept, we implemented a minimal working PIPES network and successfully tested it on a Local Area Network (LAN) with each node running on a different machine. The following subsections provide relevant implementation details.

3.1. Player node

Any Player necessarily implements two main components: one dedicated to audio processing, i.e., module execution, and another acting as a web server waiting for new YAAAPI audio modules from the network.

We implemented *dynplug*, a Linux x86-64 and arm64 VST3 Player node. The source code of the audio processing part is available at `https://github.com/paolomarrone/dynplug`, while the web server part can be found at `https://github.com/paolomarrone/dynplug_server`. The former is implemented in C and C++ and the latter in Go. Figure 3 represents the general structure of dynplug. The audio and web server parts operate as independent processes. The audio process is composed of two threads, one dedicated to current YAAAPI module execution and another that communicates with the web server process via a named pipe[19] and loads new modules.

When a new module is received, the web server process writes it into a new file and passes its path to the audio process' module loader thread. The audio process relies on the *dlfcn* library[20] to actually load it into memory, retrieve symbol addresses, and eventually unload it. The actual swapping operation is guarded by a *pthreads mutex*[21] to guarantee thread-safety. The swapping in itself simply consists of substituting function and data addresses of the current module with the new one, thus resulting in a very fast operation. In order to also preserve real-time safety, the audio processing thread uses non-blocking *pthread_mutex_trylock()* calls to acquire the mutex.

To the host application, dynplug appears as a regular VST3 plugin with a fixed number of buses and channels (1 input stereo bus and 1 output stereo bus) and set of parameters (30 input parameters with continuous linear $[0, 1]$ mapping). Unluckily, support for changes at runtime in this sense in most hosts is lacking, therefore this was a forced choice.

We successfully tested dynplug inside the Reaper DAW[22] on a x86-64 Void Linux[23] system, as well as within a Sushi[24] instance running on an arm64 Raspberry Pi board[25] with the ELK Audio OS[26].

3.2. Compiler node

We wrote a minimal Compiler node in Go called *yaaapi_compilation_server*, whose source code is available at `https://github.com/paolomarrone/yaaaeapa_compilation_server`, which simply receives one or more C files wrapped in a JSON object, invokes the GCC compiler[27] to build the output binary, and finally responds with the result or conversely with an error.

It is meant to run on an x86-64 Linux machine and can either natively compile for Linux x86-64 or cross-compile to Linux arm64. The Orchestrator node that requests compilation must specify which architecture to compile for using the *Target-Arch* HTTP request header.

3.3. Orchestrator node

We developed a tiny JavaScript library to build Orchestrators that work in web browsers – albeit potentially requiring disabling certain browser security features in some network configurations – or on Node.js[28], whose code can be found at `https://github.com/paolomarrone/ciaramellaToDynplug`. It basically allows to send the C source code of YAAAPI modules to yaaapi_compilation_server nodes, receive binary modules back, and send them to dynplug nodes.

Furthermore, we added YAAAPI as a target to Zampogna, the source-to-source transpiler for the Ciaramella programming language, and very easily integrated it into the Orchestrator library. Thus, it is possible to write Ciaramella code and have it transpiled to C locally before being sent to Compiler nodes. As mentioned before, an alternative solution would involve Zampogna running on a Compiler node instead.

[19] `https://en.wikipedia.org/wiki/Named_pipe`
[20] `https://pubs.opengroup.org/onlinepubs/7908799/xsh/dlfcn.h.html`
[21] `https://pubs.opengroup.org/onlinepubs/9699919799/basedefs/pthread.h.html`
[22] `https://www.reaper.fm/`
[23] `https://voidlinux.org/`
[24] `https://github.com/elk-audio/sushi`
[25] `https://www.raspberrypi.org/`
[26] `https://os.elk.audio/`
[27] `https://gcc.gnu.org/`
[28] `https://nodejs.org/`

A Node.js test script is also provided in which some Ciaramella code goes through the entire process, with Compiler and Player node addresses hardcoded.

4. CONSIDERATIONS

As it stands today, PIPES can be considered to be immature. However, somehow surprisingly it already has more features than several existing current solutions. On the one hand the current protocol has not yet been rigorously defined, furthermore it does not fit many potential use cases since the implementation is limited to one platform, two programming languages, and two target architectures; therefore, it can be considered essentially as a simple proof-of-concept. On the other hand, to the best of our knowledge no other tool is comparable in terms of ease of implementation, modularity, integrability, and heterogeneity of potential applications.

In its current form, with only minor adjustments to the implementation, PIPES could already prove useful for testing and measurement of potentially automated and/or remote DSP algorithms. E.g., a developer could start with an already established DSP routine, then iteratively tweak it and measure its CPU usage, memory footprint, and/or output quality, by compiling and running it on several remote machines through PIPES, hence without needing to restart execution environments.

The outcome might be different when considering the development of new algorithms and deployment to user-facing products. In the former case, the lack of provisions for compiler control, debugging, sandboxed execution, and remote control are severely limiting. In the latter, major challenges to overcome are related to the excessively simplistic nature of YAAAPI and the lack of support for dynamic I/O reconfiguration in common plugin APIs/hosts.

Nevertheless, it seems possible to overcome most of these limitations either with straightforward additions or by leveraging already-existing and widespread solutions. For example:

- compiler flags could just be included in the same JSON object as the one the Orchestrator sends to Compiler nodes;
- GDB's remote debugging[29] and tracing[30] capabilities should be relatively easy to integrate in a Player node;
- remote control could be accomplished via OSC[31];
- YAAAPI could be enhanced as needed, complemented with metadata, or otherwise replaced with an already-existing plugin API such as LV2 which couples a simple core API with advanced extensibility and adaptability features by design.

Further additions might improve the usefulness of the system, such as:

- a service discovery mechanism that reduces manual configuration efforts required from users;
- more granular DSU to dynamically replace, add, or remove only part of the currently running module without affecting the state of the other parts – this would be particularly useful for users of modular environments;
- a mechanism for collaborative editing coupled with automated compilation and deployment;
- the introduction of Translator nodes concerned with intermediate building steps (e.g., source-to-source transpilers, data processors, version control systems), which would allow for language-independent Orchestrator nodes and distributed/automated build chains;
- an analogous and complementary framework for GUIs, perhaps inspired to or even based on web technology.

In any case, as it happens with every protocol, API, or product in general, it would not be obvious to determine an optimal tradeoff between required and optional features, especially in such potentially heterogeneous application contexts. Concentrating on a varied but limited set of concrete use cases would probably be of help in this sense.

5. CONCLUSIONS

In this paper we introduced PIPES, an experimental protocol designed to streamline audio software development and deployment. PIPES clearly defines separate roles w.r.t. coding, building, and deployment, each implemented by a different network node. Communication among nodes is standardized and a common audio processing API is defined for the system to provide a seamless workflow without restricting to specific programming languages or execution environments. A proof-of-concept implementation validates the overall architecture and shows that little effort is required to setup such a system and to integrate it with already-existing technology. Despite its current immaturity and limitations, the PIPES approach shows significant potential to influence future audio DSP development practices.

6. ACKNOWLEDGMENTS

We acknowledge Stefano Zambon at Elk Audio for providing substantial technical advice during this work.

7. REFERENCES

[1] Stefano D'Angelo, "Lightweight virtual analog modeling," in *Proceedings of the 22nd Colloquium on Music Informatics (XXII CIM)*, Udine, Italy, November 2018, pp. 20–23.

[2] Yann Orlarey, Dominique Fober, and Stéphane Letz, "Faust: an efficient functional approach to DSP programming," *New Computational Paradigms for Computer Music*, pp. 65–96, 2009.

[3] Paolo Marrone, Stefano D'Angelo, Federico Fontana, Gennaro Costagliola, and Gabriele Puppis, "Ciaramella: a synchronous data flow programming language for audio DSP," in *Proceedings of the 19th Sound and Music Computing Conference (SMC 2022)*, Saint-Étienne, France, June 2022.

[4] Lynn Andrea Stein, "Interactive programming: Revolutionizing introductory computer science," *ACM Computing Surveys (CSUR)*, vol. 28, no. 4es, pp. 103–es, 1996.

[5] Erik Sandewall, "Programming in an interactive environment: the "LISP" experience," *ACM Computing Surveys (CSUR)*, vol. 10, no. 1, pp. 35–71, 1978.

[29] https://sourceware.org/gdb/current/onlinedocs/gdb.html/Remote-Debugging.html

[30] https://sourceware.org/gdb/current/onlinedocs/gdb.html/Tracepoints.html

[31] https://opensoundcontrol.stanford.edu/

[6] Nick Collins, Alex McLean, Julian Rohrhuber, and Adrian Ward, "Live coding in laptop performance," *Organised Sound*, vol. 8, no. 3, pp. 321–330, 2003.

[7] Ge Wang and Perry R Cook, "On-the-fly programming: Using code as an expressive musical instrument," in *Proceedings of the 2004 International Conference on New Interfaces for Musical Expression (NIME04)*, Hamamatsu, Japan, June 2004, vol. 4, pp. 138–143.

[8] Sarah Denoux, Stéphane Letz, Yann Orlarey, and Dominique Fober, "FAUSTLIVE, Just-In-Time Faust compiler... and much more," in *Proceedings of the Linux Audio Conference 2014 (LAC 2014)*, Karlsruhe, Germany, May 2014, pp. 143–150.

[9] Michael Hicks, Jonathan T. Moore, and Scott Nettles, "Dynamic software updating," in *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation (PLDI '01)*, Snowbird, Utah, USA, June 2001, pp. 13–23.

[10] Emili Miedes and Francesc D. Muñoz-Escoí, "A survey about dynamic software updating," *Instituto Universitario Mixto Tecnológico de Informática, Universitat Politècnica de València*, 2012.