A GALS Approach for Programming Distributed Interactive Multimedia Applications

Rodrigo C. M. Santos
PUC-Rio
Rio de Janeiro–RJ 22451-900
rsantos@inf.puc-rio.br

Francisco Sant'Anna
UERJ
Rio de Janeiro–RJ 20550-900
francisco@ime.uerj.br

Noemi Rodriguez
PUC-Rio
Rio de Janeiro–RJ 22451-900
noemi@inf.puc-rio.br

ABSTRACT

Multi-device (or distributed) multimedia applications are programs designed for exploring multiple devices during their execution. Most of these applications allow users to interact with them, defining their flow of execution. We argue that current programming approches still lack proper support for developing these applications. In a previous work we have discussed the use of the synchronous language Céu for programming multimedia, which has led to the development of the library Céu-Media as a partial result of this work. Now we are extending our work for approaching distributed applications. More precisely, we are devising a GALS (*Globally Asynchronous Locally Synchronous*) middleware that supports the development and execution of multi-device multimedia applications and guarantees the consistency between devices.

KEYWORDS

Multi-device Applications, Synchronous Languages, GALS, Céu, Céu-Media, Synchronization, Consistency

1 INTRODUCTION

The proliferation of personal multimedia-enabled devices—such as smartphones, tablets, smartwatches, etc.—has encouraged the development of multimedia applications using multiple devices, the so-called *distributed* or *multi-device* multimedia applications. Here we call *interactive* those applications that allow users to interact with them. There are at least two issues when developing interactive multi-device applications. The first one is their programming, that is, the support in terms of languages and frameworks programmers have for aiding during the development phase. The second regards the runtime support, that is, guarantees provided by the underlying middleware or framework during the execution phase.

To better frame our discussion, let's consider the five-layered synchronization reference model proposed by Costa Segundo and Santos [9] depicted in Figure 1. The media, stream, object and semantic layers cover runtime support techniques, while the specification layer embodies approaches for supporting the specification of programs. In the media layer lies techniques for achieving intra-stream synchronization, while the stream layer refers to approaches regarding inter-stream synchronization.

The object layer involves what the authors call synthetic synchronization: approaches for satisfying relationships that are not encoded directly within media objects. Players of high-level multimedia languages fit in this layer. The semantic layer deals with synchronization in a contextual way, comprising content, spatial and temporal relationships between objects (e.g., inter-destination synchronization). Finally, in the specification layer lies proposals for aiding the programming of multimedia applications.

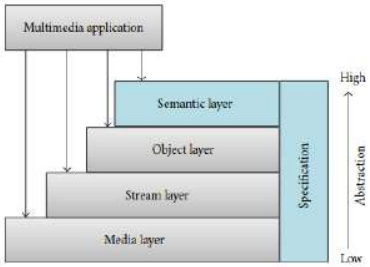

Figure 1: Five-layered synchronization model [9].

Some of the issues regarding the development of multi-device applications have been extensively studied by related work, but most of them proposes or discusses approaches that fit into one of the four horizontal layers. For instance, [22, 31] are comprehensive surveys about distributed multimedia synchronization techniques (stream and semantic layers). Stokking et al. [28] propose a network-based approach for achieving inter-destination synchronization in IPTV settings (stream and semantic layers). Challenges in developing multimedia systems supporting social viewing of shared content are discussed in [6, 13] (semantic layer). Mauve et al. [21] propose the *local-lag* and *timewarp* approaches for maintaining the consistency when executing distributed continuous applications (semantic layer).

Here we are concerned with the specification of the semantic layer, which is a view often disregarded by previous work. More precisely, we are interested in investigating how one can program multimedia interactive applications in a precise (non-ambiguous) and coherent (following consistent concepts) way. Therefore, our main focus is in the high-level support for programming these applications rather than proposing novel runtime support approaches for these systems. Current approaches are either too low-level (multimedia frameworks for general purpose languages) forcing users to program part of the communication layer, or are ambiguous and lack expressiveness (declarative multimedia languages).

Clearly, any effective approach that covers the specification layer will involve most if not all the previous layers—there is no point in proposing a programming framework without providing means for supporting its execution. Hence, although focusing on the specification, we should implement approaches from other layers for

In:XVII Workshop de Teses de Dissertações (WTD 2017), Gramado, Brasil. Anais do XXIII Simpósio Brasileiro de Sistemas Multimídia e Web: Workshops e Pôsteres. Porto Alegre: Sociedade Brasileira de Computação, 2017.

ISBN 978-85-7669-380-2.

guaranteeing that the execution of applications indeed corresponds to what have been specified in their source code.

The *interactive* qualified stated in the last paragraphs actually brings additional complexity to the problem. Most of non-interactive programs can have part of their behavior statically checked, which can be used for compensating network delays in distributed applications. For instance, the streaming of a given content could start before its scheduled time. On the other hand, users' input cannot be known *a priori*, preventing the implementation of techniques as the described above ([21] has an interesting discussion about this).

In this research we are investigating the use of reactive synchronous languages for tackling the interactive multi-device programming problem. Synchronous languages rely on the *synchronous hypothesis* that states that programs take no time for producing outputs when reacting to inputs. Although this abstraction is well suited for programming local applications (as indicated by preliminary results of this thesis), it cannot be directly applied to the distributed domain—the synchronous hypothesis does not hold due to communication latency. For approaching multi-device applications, we are exploring the GALS (*Globally Asynchronous, Locally Synchronous*) architectural style, that considers a distributed system as composed of several synchronous nodes that communicate with each other asynchronously. More precisely, we are devising a GALS middleware that supports the development and execution of multi-device multimedia applications.

This paper is organized as follows: Section 2 discusses approaches for programming multi-device applications. Section 3 presents the theoretical background of this thesis. Section 4 discusses our proposal and the preliminary results. And, Section 5 highlights the expected contributions and points future work.

2 RELATED WORK

There are two common approaches for programming multimedia applications: using general purpose or domain specific languages. The first approach usually implies in using specialized frameworks for supporting the programming of complex low-level operations. GStreamer, FFmpeg, libav, libVLC, DirectShow and AV Foundation are examples of multimedia frameworks for general purpose languages. Several other frameworks are built upon them.

By relying on general purpose languages, expressiveness is a strength of this approach. However, it tends to require expert programmers familiar with details about multimedia processing, coding, decoding, filtering, transcoding, packaging for streaming, etc., hence the use of these frameworks usually demand a non-negligible learning curve of low level concepts. By design, their APIs favor operations at intra-stream level over the composition of multiple objects. As consequence, high-level operations, such as synchronization of different streams, users' interaction, detection of the end of media (considering that a single object may have multiple streams) should be programmed on top of the low-level API, which requires solid background of the underlying framework.

Programmers commonly resort to threads and/or callbacks for developing programs composed of multiple objects when using these frameworks. This introduces another level of complexity, once it brings to the multimedia programming field the well-known problems of understandability, predictability and determinism [18].

In general, these frameworks lack proper support for distributed applications. FFmpeg and libav, for instance, only provides functions for transcoding media content to formats suitable for streaming—the streaming per se should be implemented from scratch or using a third-party library. Others have means for streaming content (libVLC, DirectShow). GStreamer goes a step further by implementing support for clock synchronization in different devices. High-level operations like communication, state synchronization, total or partial event ordering are not natively supported by any of them.

On the other hand, multimedia DSLs (also known as *multimedia languages*) implement a set of constructs to support the programming of applications without exposing too much low level details. They tend to favor the specification of the composition of multiple objects into a unified and coherent presentation. NCL [1], SMIL [32], IPML [14], HTML5 [34], X3D [16], BIFS [17] and SVG [33] are examples of multimedia languages. The well-known tradeoff between high-level abstractions and expressiveness also applies to these languages, that is, they do not have the same expressiveness of general purpose languages.

Most multimedia DSLs are interpreted, demanding the existence of a player (or *execution engine*). Players take as input a source code written in a given language and map high-level constructs to low-level digital signal processing operations producing an audiovisual presentation as output. Conceptually there are two actors involved when using multimedia DSLs: the player implementer and the application programmer. The first one is an expert programmer familiar with low-level multimedia concepts and is able to use a specialized framework to implement the execution engine. The second actor should be familiar only with the constructs offered by the language, which, in general, implement high-level concepts focusing on synchronization and composition of objects.

An well-known problem of widespread multimedia languages is their ambiguity caused by the lack of a deterministic semantics. A lot of works in literature address this problem by proposing alternative semantics [7, 10] or creating tools that statically check presentation properties (audio overlapping, video/images shadowing, contradictory constraints) [8, 24]. However, these works consider just a subset of the languages due to their complexity.

Regarding the programming of distributed applications, SMIL, HTML, SVG, X3D and BIFS have no support. Some works propose extensions to SMIL for allowing the specification of QoS streaming parameters [15, 30]. The W3C Multi-Device Timing Community Group is proposing the *TimingObject* concept as an alternative for precisely timed web applications, which consists of a JavaScript API that provides a synchronized timeline among different devices (at the time of this writing, it has the *draft* status). On the other hand, NCL and IPML implement declarative constructs for supporting interactive distributed applications. However, these constructs have either limited expressiveness or semantic inconsistencies, hindering the use of those languages in real-world applications.

3 BACKGROUND

We are designing a middleware that offers high-level abstractions, some similar to those implemented by multimedia languages, but suitable for the distributed domain. Under the hood, we are exploring approaches to produce an audiovisual presentation that

accurately corresponds to its source code, since there is no point in providing high-level programming abstractions without supporting their execution. Our intended users are both the actors mentioned in Section 2, namely player implementers and application programmers.

In our approach, we are delving into the use of reactive synchronous languages [3] (*synchronous languages*, as shorthand) for programming multimedia applications. These languages rely on the *synchronous hypothesis* [2] that considers that programs produce outputs synchronously with their inputs. Reactive languages divide computations into a sequence of discrete steps called *reactions*. Each reaction executes until its completion before the system can process any other input. The synchronous hypothesis adds the constraint that inside each reaction the time does not advance. In practice, this model assumes that the computation of reactions is faster than the minimum time interval between external events.

The synchronous approach was originally proposed as an alternative for developing safe concurrent real-time embedded systems, where the synchronous assumption is rather common [11]. Even though synchronous languages constitute a suited paradigm for developing interactive reactive programs, an well-known limitation of them regards the programming of distributed applications, because communication latency breaks the synchronous hypothesis. To overcome this issue, we explore the GALS (*Globally Asynchronous, Locally Synchronous*) design in this research for approaching the multi-device setting — more on that later in this section.

Some characteristics and guarantees provided by synchronous languages may be used for solving part of the problems of the multimedia programming field. In these languages, time advances in a sequence of discrete input events, defining what is known as *logical time*. As argued by Lima [19], in synchronous systems the logical notion of time supplants the physical notion, since it favors the specification of operations that should be executed accurately in a given time instant.

Determinism is another feature embraced by most synchronous languages. A program is said to be deterministic if given an initial state and a sequence of inputs, it always executes the same sequence of operations and reaches the same final state. As pointed by Berry and Benveniste, the advantages of deterministic systems should be obvious: there is no reason a programmer should want his/hers programs to behave in some non-deterministic manner [2].

Synchronous languages have native support for concurrency, while preserving determinism. Safe concurrency and determinism are important features for real-time embedded systems that these languages have been originally designed for.

Support for event handling, in general, is a major concern of reactive languages. The programming of event-driven applications using traditional programming models is typically performed around the notion of asynchronous callbacks. One of the main issues when using callbacks is that the program control jumps around multiple callbacks, leading to codes that are hard to follow and/or understand (the so-called *Callback Hell* problem). In fact, the control flow is driven by events and not by an order specified by the programmer. Synchronous languages overcome these and other problems by proving abstractions to express how programs should react to events. Compilers usually guarantee safe access to shared variables, which yields the advantage that programmers do not need to worry about the order of events and computation dependencies.

Furthermore, as the synchronous approach has been proposed based on mathematically sound tools, it provides means for statically checking properties of programs. In the scope of this research, this supports the effective development of tools that check whether the final presentation holds a given set of properties, as for instance audio or video overlapping, contradictory constraints, time conflicts, etc.

The approach of applying synchronous languages in the multimedia field is not novel. At the 90's, some authors have explored the use of these languages for addressing the problem of real-time synchronization of streamed media contents [4, 5, 12]. There are proposals of using these languages for programming applications: ChucK [35], Pure Data [25], and Faust [23] are some examples of synchronous DSLs developed for audio processing (also know as music programming languages). As the human hearing can detect even small latencies and delays in audio signals, the use of the synchronous approach represents an interesting alternative for providing timing guarantees over sample-level operations in the audio signal.

Smix [19] is a more recent proposal for high-level multimedia programming that also relies on the synchronous hypothesis. A Smix program is composed of a set of media objects and a list of links. Links causally relate events with media object operations (start, stop, pause, set the value of a property, etc.). The language has been proposed as an alternative for traditional informal and ambiguous high-level multimedia languages, therefore since its conception Smix was designed to have a formal and deterministic semantics.

These works help to illustrate how the multimedia research community for long has been investigating the use of synchronous languages for approaching problems of the field. However, none of them has explored the use of these languages in the context of programming interactive multi-device applications, at least at the abstraction level we are interested in this research. They all have in common the assumption that the characteristics of synchronous languages constitute a suitable framework for programming the control part of multimedia systems. Here we borrow this assumption under the programming perspective and apply it in the distributed domain.

Approaching distributed applications using synchronous languages is not straightforward. One of the main issues is that in a distributed setting one cannot assume the synchronous hypothesis due to the non-negligible communication latency. In fact, some authors consider that defining the precise semantics and consistency guarantees for reactive programming in distributed systems is an open research problem [20]. We call a distributed system consistent if all devices perceive the events of interest in the same order.

A proposed modeling for distributed synchronous systems is the so-called GALS design. A GALS [29] system is composed of several synchronous parts that communicate with each other using an asynchronous medium. In practice, this design is an attempt to model systems in which individual modules take advantage of the synchronous approach and the communication latency is usually the only source of non-determinism. The GALS design has been

originally proposed for programming multi-clock digital circuits, in which each synchronous block has its own clock running in its own frequency, and interconnected through an asynchronous bus.

In essence, we are approaching the problem of programming of multi-device applications in two levels. For guaranteeing the synchronization in each device, we are using a multimedia synchronous framework that supports the development of applications using high-level abstractions and accurately supports their execution (that its, respecting the timing expressed by users in the source code)—this framework discussed in Section 4 is a partial result of this research. For the distributed part, we are using a middleware for coordinating the communication between devices. This middleware follow the GALS style and one of its main goals is to provide consistency to the system, accordingly to the definition above.

4 PRELIMINARY RESULTS

We have been exploring the synchronous language Céu [26] for programming multimedia, which has led to the development of the framework Céu-Media [27] as a preliminary result of this ongoing research. Céu is a structured synchronous reactive programming language that provides native support for event handling and concurrency. As most of other synchronous languages, Céu's semantics is also deterministic. Céu-Media explores Céu characteristics for providing a multimedia framework capable of accurately programming inter-media synchronization in a local application.

There are two main advantages in using Céu-Media. The first is its high-level abstractions combined with the expressiveness of Céu. The framework implements abstractions similar to those of traditional multimedia languages NCL and SMIL, but avoids their ambiguity and synchronization problems due to the synchronous and deterministic semantics of Céu. Additionally, Céu's support for event handling and concurrency has been seen as a useful alternative for programming multimedia in a general-purpose imperative reactive language.

The second advantage is its accuracy. One of our main concerns when designing Céu-Media has been to reproduce in the final multimedia presentation the synchronous semantics expressed in the program's source code. Hence, Céu-Media guarantees that the presentation clock does not advance while the program is reacting to a given event — a discussion of practical implications of this feature can be found in [19]. Under the hood, we managed to implement this by developing and attaching to the presentation a deterministic monotonic clock which is enslaved to the program's logical time. As backend, Céu-Media uses the industry-grade multimedia framework GStreamer.

The Céu-Media framework consists of three main concepts: `Scene`, `Media`, and `Player`. A `Scene` represents a top-level OS window with audio and (possibly) video output. A `Media` holds the description of a media object. And a `Player` renders a `Media` on a `Scene`.

Consider the Céu-Media program depicted in Listing 1. It declares five `Media` descriptions. `vid1`, `vid2`, `vid3`, and `vid4` (lines 1–4) represent videos (`muted_video.gov`, a muted video) and `audio` represents an audio (`audio.ogg`, the corresponding audio track). The program waits for five seconds due to the **`await`** statement in line 6 and then creates a `Scene` (lines 7–20) for rendering the presentation.

The **`par/or`** composition creates concurrent execution *trails* when evaluated. The execution of such trails is necessarily deterministic. Once executed, the **`par/or`** statement starts its trails in parallel and terminates when one them terminates. Thus, the composition in lines 9–19 creates five `Player`s, one in each of its trails. Each of the first four trails creates a `Player` with a distinct video description. The last trail creates a `Player` to execute the `audio`. The program ends when one of these `Player` ends.

```
1  var Media vid1 = Media.VIDEO ("muted_video.ogv", ...);
2  var Media vid2 = Media.VIDEO ("muted_video.ogv", ...);
3  var Media vid3 = Media.VIDEO ("muted_video.ogv", ...);
4  var Media vid4 = Media.VIDEO ("muted_video.ogv", ...);
5  var Media audio = Media.AUDIO ("audio.ogg", 1.0);
6  await 5s;
7  var IScene scene;
8  watching Scene (Size (1080, 720)) -> (&scene) do
9    par/or do
10     await Play(vid1, &scene);
11   with
12     await Play(vid2, &scene);
13   with
14     await Play(vid3, &scene);
15   with
16     await Play(vid4, &scene);
17   with
18     await Play(audio, &scene);
19   end
20 end
```

Listing 1: Céu-Media guarantees that all players starts with the same clock reference and keeps them in-sync.

Conceptually, the logical time does not pass when the program awakes for the **`await`** in line 6 until all trails reach their corresponding **`await`**. It means that all Céu-Media `Player`s should start at the same logical time and, therefore, be executed in-sync. Céu-Media guarantees that this property holds during the presentation, because its clock advances at the same pace as the program's logical time. While this may lead to glitches — especially in the audio — due to the drift between real and logical time, this approach is capable of realize the synchronous semantics in the final output. Thus, we say that Céu-Media promotes the accurate programming of inter-media synchronization relationships.

4.1 Mars: Céu-Media for Distributed Applications

To approach the distributed scenario, we have designed and implemented a GALS middleware (called Mars) whose main goal is to provide the programming support for developing interactive multi-device applications upon Céu-Media. Internally, Mars takes care of implementing all low-level communication and synchronization functionalities among devices.

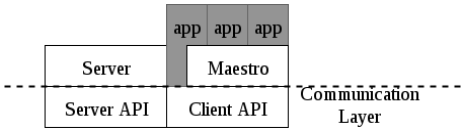

Figure 2: Mars architecture.

Mars is actually composed of a set of APIs and a tiny server application. Figure 2 depicts the internal minimalist architecture of the middleware. The server has mainly two functionalities: to manage sections and to broadcast events and clock ticks to all devices within a section. We call *section* a group of devices that are communicating with each other to execute jointly and collaboratively a given distributed program.

Within a section, all Céu-Media events generated in any device are forwarded to all others, that is, these events become global to all devices of that section. The *Maestro* component is responsible for intercepting these events and sending them to the server. Likewise, this same component receives events coming from the server and forwards them to applications. The low-level client and server APIs implement the communication layer of the middleware. Even though the low-level communication API has been designed to meet the middleware requirements, programmers interested in more control over these low-level operations may optionally build their programs using directly the client communication API.

To better illustrate the use of Mars, consider the source code of a simple Céu-Media program depicted in Listing 2. This program creates a `Player` for executing a video and finishes when the video ends or when one presses any button of the mouse. For didactic purposes, let's call this program *P*.

```
1  var Media.Video video = val Media.Video (<...>);
2  var& IScene scene;
3  watching Scene (Size (640, 480)) -> (&scene);
4  do
5    par/or
6      await Play (&scene, &video);
7    with
8      await CM_SCENE_MOUSE_CLICK;
9    end
10 end
```

Listing 2: A simple Céu-Media program that plays a video and finishes when one presses a mouse button.

When compiling this source code using Mars libraries, the output is a modified version of the program *P*, that we will call P'. *P* and P' have similar behavior: both execute a video until one presses a mouse button, but P' has been compiled for the distributed setting. Consider a section in which there are several devices running their own instance of P'. When any of these instances generates the mouse click event, the Maestro component sends it to the server, which in turn forwards it to all devices. Therefore, all instances of P' will receive the `CM_SCENE_MOUSE_CLICK` event and thus awake from the **`await`** in line 8. It worth highlighting that this source code has no explicit constructs for communicating or synchronizing with remote instances, but Mars hides these operations offering an API that resembles the programming of local applications.

The server has an important role in the system, being vital for guaranteeing the consistency between devices. In Mars's architecture, the server forwards events in the same order it receives, ensuring a total order of events within a section. Furthermore, when a Céu-Media event is generated in a given device, the Maestro intercepts it before the application has the chance to react to it. The application is notified about that event only when the Maestro receives it back from the server. While this approach indeed guarantees the global consistency, a drawback is the decrease in the level of responsiveness of programs, once events are processed after the server has received and forwarded back them to the device.

Considering the wall-clock time, each device reacts at a different moment to broadcast events. To overcome this issue, the server sends periodic tick events based on its internal clock (we are assuming a local network with low-latency) to emulate a notion of global clock. Thus, as these ticks are delivery respecting the ordering of events, logically all devices react at the same time. The deployment of applications that use this global clock in networks with high latency can lead to executions having several glitches, but the consistency is guaranteed. While we agree that these glitches can make some programs infeasible to use, we have chosen to prioritize the consistency over the QoE in this research.

4.2 Ongoing work: Asymmetric Programs

Until now we have discussed how Mars takes as input programs designed for being executed in a single device and adapts them to its distributed client-server architecture. As these programs react in the exactly same way, we call them *symmetric*. However, some programs explicitly developed for the distributed setting should react differently depending on the event source—which is why we call them *asymmetric*.

To illustrate, a simple 2-players game consisting of a hero and a monster. If the player controlling the hero issues a key event, the hero avatar should be updated in both devices. Likewise, if the player controlling the monster issues the event, the monster avatar should be updated. Considering that each device has a unique ID and that is possible to identify the device that has generated the key event, using a sequence of **`if`**-`than`-`else` one can implement this game. However, this tends to become a complex and tedious programming task as the number of roles and/or devices increases.

Consider now that we have the events `MONSTER_UPDATE` and `HERO_UPDATE` to indicate which avatar should be updated. Thus, one could write the program as in Listing 3. The **`every`** statement (line 3—5) in the first trail of the **`par`** composition (lines 2—10) awakes in each occurrence of the `MONSTER_UPDATE` event, calling the function that updates the monster avatar (line 4). Likewise, the second trail has another **`every`** block that awakes in occurrences of the `HERO_UPDATE` event for updating the hero (lines 7—9).

```
1  <...>
2  par do
3    every (key, press) in MONSTER_UPDATE do
4      call Update_Monster (key, press);
5    end
6  with
7    every (key, press) in HERO_UPDATE do
8      call Update_Hero (key, press);
9    end
10 end
```

Listing 3: A modified version of the monster-hero game.

Now the problem becomes how to properly generate the `MONSTER_UPDATE` and `HERO_UPDATE` events. In practice, the program in Listing 3 would work if each occurrence of the key event (`CM_SCENE_KEY`) coming from the device with id 0 (assuming this device controls the monster) was forwarded as being the `MONSTER_UPDATE` event and if occurrences of that same event, but coming from the device with id 1 (this one controlling the hero) was forwarded as the `HERO_UPDATE` event.

We are working on ways for allowing the programming of event mappings as discussed above. Our initial approach is to use a Lua table that express this mapping, as in Listing 4. Each entry follows the skeleton `DEVICE_ID, EVENT, MAPPING`, indicating that the `EVENT` from device with id `DEVICE_ID` is to be interpreted as the event `MAPPING`.

```
1  return {
2    {0, CM_SCENE_KEY, MONSTER_UPDATE},
3    {1, CM_SCENE_KEY, HERO_UPDATE}
4  }
```

Listing 4: A Lua file specifying event mappings.

Note that one could develop different programs, one for controlling the monster and other for controlling the hero, and execute them in the same section. The first one would be as simple as the first trail of Listing 3 and the second would be as simple as the second trail. In this case, we could also have different Lua files, one for each device and each having only the necessary mapping.

5 FUTURE WORK AND EXPECTED CONTRIBUTIONS

Mars currently provides a GALS programming model for multimedia applications. However it still lacks support for synchronizing media objects in different devices. In literature, most of the work addressing this problem approaches it by means of clock synchronization. Céu-Media has a deterministic clock that controls the pace of the multimedia presentation according to the program's logical clock. We intend to support this feature by investigating approaches for synchronizing clock sources in each device within a section using classical algorithms as NTP or PTP.

The evaluation of the results of this research should follow a qualitative analysis. By using realistic multi-device scenarios proposed in literature, we intent to discuss how they could be implemented using our solution and highlight the main strengths and weaknesses of this approach when compared with the use of current programming frameworks for multimedia (both, imperative and declarative).

Summarizing, at the end of this research we expect the following main contributions: an alternative programming model exploring the synchronous hypothesis, high-level abstractions and a general-purpose imperative language for the multimedia domain; an investigation of the suitability of the GALS style for programming interactive distributed multimedia applications (the outcome of this investigation should be implemented in the Mars middleware); an alternative for players implementers to develop execution engines either by developing players having Mars as backend or by compiling programs in high-level DSLs to Céu-Media; an in-depth study about how the features of the Céu language can properly support programmers to express the control part of multimedia applications.

REFERENCES

[1] ABNT. NBR 15606-2:2011 "Digital Terrestrial Television - Data Coding and Transmission Specification for Digital Broadcasting - Part 2: Ginga-NCL for Fixed and Mobile Receivers - XML Application Language for Application Coding. Technical report, São Paulo, Brazil, 2011.

[2] A. Benveniste and G. Berry. The Synchronous Approach to Reactive and Real-Time Systems. *Proceedings of the IEEE*, 79(9):1270–1282, 1991.

[3] A. Benveniste, P. Caspi, S. A. Edwards, N. Halbwachs, P. Le Guernic, and R. De Simone. The synchronous languages 12 years later. *Proceedings of the IEEE*, 91(1):64–83, 2003.

[4] G. S. Blair, G. Coulson, M. Papathomas, P. Robin, J. B. Stefani, F. Horn, and L. Hazard. A programming model and system infrastructure for real-time synchronization in distributed multimedia systems. *IEEE Journal on Selected Areas in Communications*, 14(1):249–263, 1996.

[5] G. S. Blair, M. Papathomas, G. Coulson, P. Robin, J. B. Stefani, F. Horn, and L. Hazard. Supporting real-time multimedia behaviour in open distributed systems: an approach based on synchronous languages. *ACM Multimedia'94*, 1994.

[6] F. Boronat, R. Mekuria, M. Montagud, and P. Cesar. Distributed Media Synchronisation for Shared Video Watching: Issues, Challenges and Examples. *Computer Communications and Networks*, pages 393–431, 2013.

[7] A. Bossi and O. Gaggi. Enriching smil with assertions for temporal validation. In *Proceedings of the 15th ACM International Conference on Multimedia*, MM '07, pages 107–116, New York, NY, USA, 2007. ACM.

[8] A. Y. Chang. An Intelligent Analysis and Verification Model for Consistent SMIL Presentations. *Journal of Convergence Information Technology*, 7(7):332–341, apr 2012.

[9] R. M. Costa Segundo and C. A. S. Santos. Systematic Review of Multiple Contents Synchronization in Interactive Television Scenario. *ISRN Communications and Networking*, 2014:1–17, 2014.

[10] J. dos Santos, C. Braga, and D. C. Muchaluat-Saade. A rewriting logic semantics for ncl. *Sci. Comput. Program.*, 107(C):64–92, Sept. 2015.

[11] J.-p. T. Dumitru Potop-butucaru, Robert De Simone. The synchronous hypothesis and synchronous languages. In *Embedded Systems Handbook*. CRC Press, 2005.

[12] J. M. Eyzell and J. Farines. Using ESTEREL for building synchronization mechanisms in multimedia systems. *IEEE Conference on Protocols for Multimedia Systems - Multimedia Networking, 1997*, pages 269–272, 1997.

[13] D. Geerts, I. Vaishnavi, R. Mekuria, O. van Deventer, and P. Cesar. Are We in Sync?: Synchronization Requirements for Watching Online Video Together. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '11, pages 311–314, New York, NY, USA, 2011. ACM.

[14] J. Hu and L. Feijs. IPML: Extending SMIL for Distributed Multimedia Presentations. In *VSMM 2006. Lecture Notes in Computer Science*, pages 60–70. Springer Berlin Heidelberg, 2006.

[15] Ing-Chau Chang and Sheng-Wen Hsieh. An adaptive QoS guarantee framework for SMIL multimedia presentations with ATM ABR service. In *Global Telecommunications Conference, 2002. GLOBECOM '02. IEEE*, volume 2, pages 1784–1788. IEEE, 2002.

[16] ISO. X3D Architecture and base components V3, 2013. ISO/IEC IS 19775-1:2013.

[17] ISO. Information technology – Coding of audio-visual objects – Part 11: Scene description and application engine, 2015. ISO/IEC 14496-11:2015(E).

[18] E. Lee. The Problem with Threads. *Computer*, 39(5):33–42, may 2006.

[19] G. F. Lima. *A synchronous virtual machine for multimedia presentations*. PhD thesis, Department of Informatics, PUC-Rio, Rio de Janeiro, RJ, Brazil, 2015.

[20] A. Margara and G. Salvaneschi. We Have a DREAM: Distributed Reactive Programming with Consistency Guarantees. *Proceedings of the 8th ACM International Conference on Distributed Event-Based Systems*, pages 142–153, 2014.

[21] M. Mauve, J. Vogel, V. Hilt, and W. Effelsberg. Local-Lag and Timewarp: Providing Consistency for Replicated Continuous Applications. *IEEE Transactions on Multimedia*, 6(1):47–57, feb 2004.

[22] M. Montagud, F. Boronat, H. Stokking, R. van Brandenburg, and R. Brandenburg. Inter-destination multimedia synchronization: schemes, use cases and standardization. *Multimedia Systems*, 18(6):459–482, jul 2012.

[23] Y. Orlarey, D. Fober, and S. Letz. FAUST: An efficient functional approach to DSP programming. In *New Computational Paradigms for Computer Music*, 2009.

[24] D. Picinin, Jr., J.-M. Farines, and C. Koliver. An approach to verify live ncl applications. In *Proceedings of the 18th Brazilian Symposium on Multimedia and the Web*, WebMedia '12, pages 223–232, New York, NY, USA, 2012. ACM.

[25] M. Puckette. Pure data: another integrated computer music environment. In *in Proceedings, International Computer Music Conference*, pages 37–41, 1996.

[26] F. Sant'Anna, N. Rodriguez, R. Ierusalimschy, O. Landsiedel, and P. Tsigas. Safe System-Level Concurrency on Resource-Constrained Nodes. In *SenSys '13*, New York, New York, USA, nov 2013. ACM Press.

[27] R. C. Santos, G. F. Lima, F. Sant'Anna, and N. Rodriguez. CÉu-media: Local inter-media synchronization using cÉu. In *Proceedings of the 22Nd Brazilian Symposium on Multimedia and the Web*, Webmedia '16, pages 143–150, New York, NY, USA, 2016. ACM.

[28] H. Stokking, M. van Deventer, O. Niamut, F. Walraven, and R. Mekuria. IPTV inter-destination synchronization: A network-based approach. oct 2010.

[29] P. Teehan, M. Greenstreet, and G. Lemieux. A Survey and Taxonomy of GALS Design Styles. *IEEE Design & Test of Computers*, 24(5):418–428, sep 2007.

[30] Y. Terashima, K. Yasumoto, T. Higashino, K. Abe, T. Matsuura, and K. Taniguchi. Integration of QoS guarantees into SMIL and its flexible implementation. In *2000 Eighth International Workshop on Quality of Service. IWQoS 2000 (Cat. No.00EX400)*, pages 164–166. IEEE, 2000.

[31] M. O. van Deventer, H. Stokking, M. Hammond, J. Le Feuvre, and P. Cesar. Standards for multi-stream and multi-device media synchronization. volume 54, pages 16–21, mar 2016.

[32] W3C. Synchronized Multimedia Integration Language (SMIL 3.0), 2008. W3C Recommendation https://www.w3.org/TR/smil/.

[33] W3C. Scalable Vector Graphics (SVG) 1.1 (Second Edition), 2011. W3C Recommendation. http://www.w3.org/TR/SVG/.

[34] W3C. HTML5 - A vocabulary and associated APIs for HTML and XHTML, 2014. W3C Recommendation. http://www.w3.org/TR/html5/.

[35] G. Wang and P. R. Cook. On-the-fly programming: Using code as an expressive musical instrument. In *Proceedings of the 2004 Conference on New Interfaces for Musical Expression*, NIME '04, pages 138–143, Singapore, Singapore, 2004. National University of Singapore.