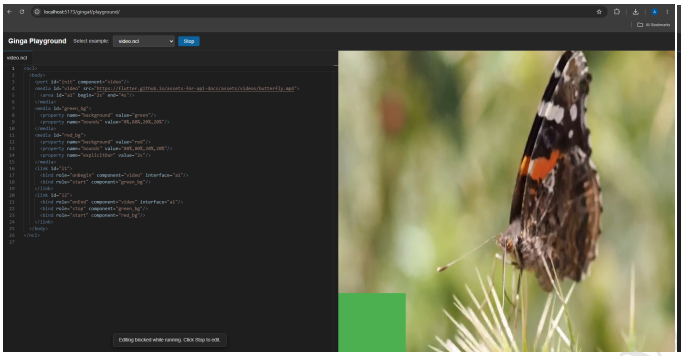
gingaf: A Cross-Platform Architecture to Execute Ginga-NCL and Ginga-HTML5 Applications

Alan L. V. Guedes
a.guedes@reading.ac.uk
University of Reading
UK

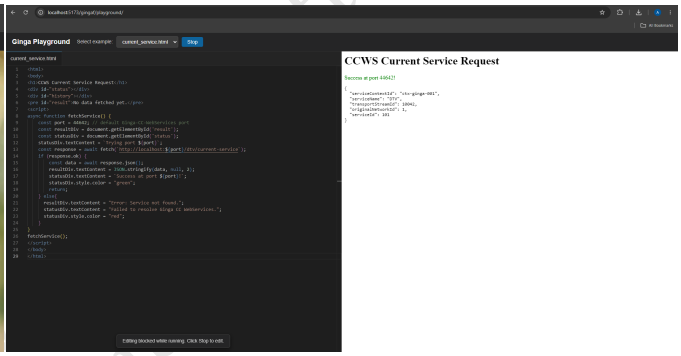
Paulo Victor Borges
paulo.victor.lima15@gmail.com
PUC-Rio
Brazil

Daniel Moraes
danielmoraes@telemidia.puc-rio.br
PUC-Rio
Brazil

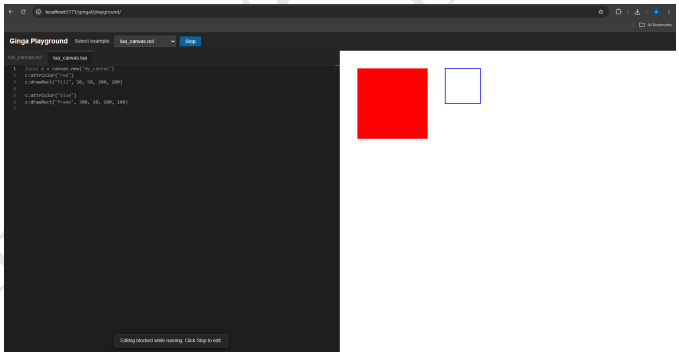
Sergio Colcher
colcher@puc-rio.br
PUC-Rio
Brazil



(a) Execution of Ginga-NCL application `video.ncl` at $t = 2$s.



(b) Execution of Ginga-HTML5 application `current_service.html`.



(c) Execution of Ginga-NCL application `lua_canvas.ncl`

Figure 1: gingaf execution at Web using the GingaPlayground interface

ABSTRACT

Ginga, alongside its Ginga-NCL and Ginga-HTML5 subsystems, is an international standard for interactive digital television defined by the ITU-T/SBTVD. It is used in Brazil and several Latin American countries, and reaches over 60 million people. However, traditional Ginga development tools rely on heavy, platform-specific C++ implementations, creating a steep barrier to entry for developers. This paper presents `gingaf`, a Ginga architecture which focuses on decoupling logic components from UI components to support running both Ginga-NCL and Ginga-HTML5 applications directly from the web browser, mobile, or desktop. On the web, we present a portable Lua interpreter and a Ginga-HTML5 player that intercepts HTTP requests to support Ginga-CC-WebServices. This architecture does not require any native browser plugins, paving the way for web-based development playgrounds and debugging environments.

Unpublished working draft. Not for distribution.

© 2026 SBC – Brazilian Computing Society.
This is an unpublished working draft. Not for distribution. Not for redistribution. The definitive Version of Record was published in *Proceedings of the 32nd Brazilian Symposium on Multimedia and the Web (WebMedia 2026)*, https://doi.org/10.5753/webmedia.2026.30538.

KEYWORDS

TV, Ginga, Ginga-NCL, Ginga-HTML5

1 INTRODUCTION

The Ginga middleware is the international standard for interactive digital television by both the ITU-T [8] and the Brazilian Digital Terrestrial Television System (SBTVD) [2]. Ginga utilizes the Nested Context Language (NCL), an XML-based declarative multimedia language. Ginga standard is legally mandated and installed on 100% of certified Digital TV sets and set-top boxes sold in Brazil and several other Latin American countries adopting the ISDB-T framework. Because it is mandated by local telecommunications entities, every major television manufacturer must embed a Ginga implementation directly into their hardware firmware, making it one of the most widely deployed interactive middleware systems globally. This wide adoption results in over 60 million people impacted daily. It is also widely used for public interest broadcasting; for example, the Brazil4D [5] application targets low-income families to access public services, health information, and educational content. Furthermore, the application coding specification for the next-generation Brazilian digital television standard (TV 3.0) continues to advance the Ginga engine [7].

Despite the critical role and massive scale of Ginga, developers and researchers continue to face the **lack of a Web-based development environment**. This happens because traditional Ginga players, such as the reference PUC-Rio C++ implementations [15, 16], are tightly coupled to the native platform (targeting Linux-based digital TV receivers) and require specialized OS setups, compilers, and heavy multimedia frameworks to run. Moreover, the recent standardisation of new Ginga subsystems (Ginga-HTML5 and Ginga-CC-WebServices) added more complexity and limited the portability of these traditional C++ players. Therefore, developers and researchers cannot easily share, test, or prototype their code online. This creates a steep barrier, which contrasts sharply with the trend in modern programming ecosystems, where major frameworks provide instant web-based playgrounds.

`gingaf` addresses these challenges by introducing a portable Ginga architecture that focuses on decoupling logic and UI components. By separating the logic (headless) execution engine from the visual rendering layer, we support running both subsystems directly from the web browser, mobile, or desktop. As a proof of concept and validation of this decoupled design, we developed GingaPlayground (Figure 1), a web-based editor running directly in the browser without any native plugins or local installation.

To support reproducibility, source code (both `gingaf` and GingaPlayground), unit tests, and demonstration videos of `gingaf` are available[1]. An instance of GingaPlayground is hosted online[2].

We summarize our main contributions below:

- **Cross-Platform Ginga implementation by Decoupling Logic from UI Components**: We introduce an architecture that strictly separates the headless logic state from the presentation components. This decoupling enables deterministic execution of Ginga synchronization logic in pure code environments and cross-platform support across web, mobile, and desktop from a single codebase.
- **NCLua and CCWS Support on the Web**: We present adaptations for embedding NCLua and CCWS to be able to run client-side by using a portable pure Dart Lua interpreter and overloading the browser's `fetch` API inside a sandboxed iframe.

The remainder of this paper is structured as follows. Section 2 presents the background on the Ginga standard, and Section 3 reviews related work. Section 4 introduces the decoupled architecture of `gingaf`. Section 5 and Section 6 detail the execution of the Ginga-NCL and Ginga-HTML5 subsystems. Section 7 presents the GingaPlayground use case. Finally, Section 8 concludes the paper.

2 GINGA BACKGROUND

Figure 2 depicts the architecture of the D-profile Ginga DTV Receiver introduced by the TV 2.5 specifications [11]. Dashed elements are the new components proposed in D-profile receivers, namely, Ginga-HTML5 and Ginga-CC-WebServices. In the following, we first detail the main components[3].

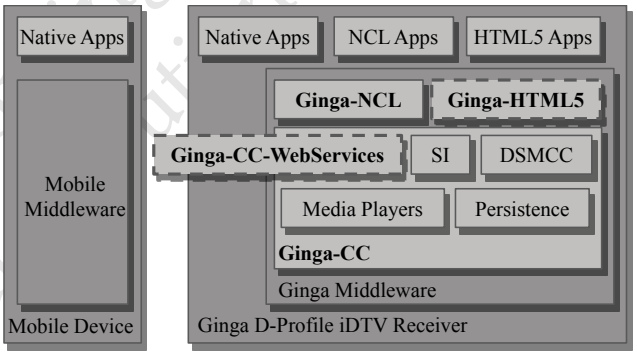


Figure 2: D-Profile Ginga Receiver (Source: [11]).

Ginga-NCL is the subsystem responsible for controlling NCL multimedia applications. NCL [13] is a declarative language that defines temporal synchronization, layout reuse, multi-device presentation, and embedded Lua scripting[4]. It also provides an API for on-the-fly modifications (NCL editing commands) and handles media from both broadcast and broadband channels. To accommodate varying hardware capabilities, the SBTVD Forum defines specific receiver profiles that applications can target to ensure compatibility [2].

To demonstrate the structure of NCL applications, we use a sample application (`video.ncl`) presented in Listing 1. The presentation defines a primary video (`video`) in lines 4–6, a green background media (`green_bg`) in lines 7–10, and a red background media (`red_bg`) in lines 11–15 with an explicit duration of 2 seconds. Temporal behavior is coordinated by link elements: link `l1` starts the green background when area `a1` of the video begins (at 2s), and link `l2` stops the green background and starts the red background when area `a1` ends (at 4s). The application ends when the `red_bg`

[1] https://github.com/ginga-org-br/gingaf
[2] https://ginga.org.br/gingaf/playground/
[3] Other components such as SI and DSMCC were omitted for simplicity.
[4] https://www.lua.org

Unpublished working draft. Not for distribution.

ends at 6s (explicitDur 2s) and the main video ends decoding at 7s (video content duration).

```
<ncl>
<body>
  <port id="init" component="video"/>
  <media id="video" src="https://flutter.github.io/assets-for-api-
       docs/assets/videos/butterfly.mp4">
    <area id="a1" begin="2s" end="4s"/>
  </media>
  <media id="green_bg">
    <property name="background" value="green"/>
    <property name="bounds" value="0%,80%,20%,20%"/>
  </media>
  <media id="red_bg">
    <property name="background" value="red"/>
    <property name="bounds" value="80%,80%,20%,20%"/>
    <property name="explicitDur" value="2s"/>
  </media>
  <link id="l1">
    <bind role="onBegin" component="video" interface="a1"/>
    <bind role="start" component="green_bg"/>
  </link>
  <link id="l2">
    <bind role="onEnd" component="video" interface="a1"/>
    <bind role="stop" component="green_bg"/>
    <bind role="start" component="red_bg"/>
  </link>
</body>
</ncl>
```

Listing 1: video.ncl source code

Ginga-HTML5 [3] supports the execution of Web-based applications (*e.g.*, Listing 2) natively within the Ginga environment. It provides a standard web engine capable of rendering HTML5, CSS3, and executing JavaScript, allowing developers to author interactive digital television applications using modern web technologies.

Ginga-CC-WebServices focuses on integrating the experience of the broadcasting and broadband environments. To do this, it supports a set of services implemented in the form of REST APIs. Thus, as indicated in Figure 2, its functionalities can be used by: 1) native Smart TV applications, web-based applications, and Ginga applications in the DTV receiver; 2) devices present in the home environment, allowing, *e.g.*, second-screen scenarios.

According to its standard [1], Ginga-CC-WebServices listens by default on port 44642 and can be discovered by multicasting an HTTP M-SEARCH request. Its functionalities include not only functions for accessing content from broadcasting, but also functions that standardize access to advanced features of the DTV environment (*e.g.*, platform characteristics, supported DRM, media playback capabilities). We briefly introduce each API and show its main endpoints in what follows. In particular, the identifiers in square brackets should be filled in the API request. For instance, the `GET http(s)://<host>/dtv/current-service` returns information for the selected DTV service.

To demonstrate the structure of Ginga-HTML5 applications using Ginga-CC-WebServices, we provide a sample application in Listing 2. This JavaScript-based HTML application uses the standard `fetch` API to request the `/dtv/current-service` endpoint on the local Ginga-CC-WebServices server at the default port 44642. Once the asynchronous request is resolved, the application parses the JSON response and updates the `div` elements with the retrieved service data or an error message if the service is unreachable.

Unpublished working draft. Not for distribution.

```html
<html>
<body>
<h1>Ginga-CC-WebServices Current Service Request</h1>
<div id="status"></div>
<div id="history"></div>
<pre id="result">No data fetched yet.</pre>
<script>
async function fetchService() {
    const port = 44642; // default Ginga-CC-WebServices port
    const resultDiv = document.getElementById('result');
    const statusDiv = document.getElementById('status');
    statusDiv.textContent = `Trying port ${port}`;
    const response = await fetch(`http://localhost:${port}/dtv/
          current-service`);
    if (response.ok) {
        const data = await response.json();
        resultDiv.textContent = JSON.stringify(data, null, 2);
        statusDiv.textContent = `Success at port ${port}!`;
        statusDiv.style.color = "green";
        return;
    } else{
      resultDiv.textContent = "Error:_Service_not_found.";
      statusDiv.textContent = "Failed_to_resolve_Ginga-CC-
            WebServices.";
      statusDiv.style.color = "red";
    }
}
fetchService();
</script>
</body>
</html>
```

Listing 2: `current_service.html` fetching from Ginga-CC-WebServices

3 RELATED WORK

The development of Ginga players has historically been led by the TeleMídia Lab at PUC-Rio, producing several versions. However, all prior implementations share a fundamental limitation: they are deeply tethered to native desktop libraries, making web compilation impossible. Earlier work such as WebNCL [9] provided a web-based NCL presentation engine but did not support Lua scripting.

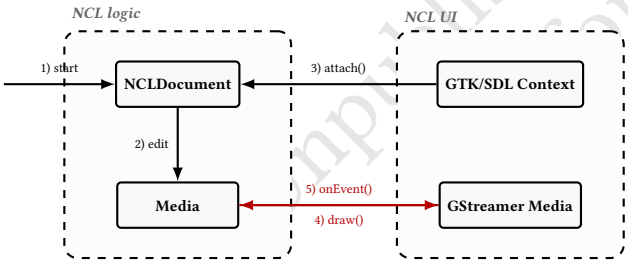


Figure 3: Traditional native C++ NCL player architecture describing PUC-Rio Ginga v0.13 and v1.0, demonstrating tight coupling between logic elements and UI graphics bindings.

PUC-Rio Ginga v0.13.6 [10, 15] was designed for early-generation digital TV receivers with resource constraints. It used SDL[5] as the UI context for rendering graphics and managing window events, and FFmpeg[6] for audio/video decoding. Its execution model relied on event listeners to handle asynchronous media playback: dedicated threads monitored AV decoder callbacks and posted events into the NCL engine, creating concurrency that was difficult to test deterministically. Recently, PUC-Rio Ginga v1.0 [6, 16] modernized the renderer by adopting GTK[7] as the UI context and GStreamer[8] as the AV decoding pipeline. A key architectural advance was the move to a mono-threaded model inspired by synchronous execution: rather than spawning per-media threads, the engine advances state through a central tick loop, reducing race conditions and improving predictability. Another notable early implementation is **WebNCL**, a web-based NCL player fully implemented in JavaScript that did not support Lua.

The fundamental architectural issue preventing these implementations from running on the web is the tight coupling between the NCL logic layer and the native operating system resources. As illustrated in Figure 3, in traditional C++ Ginga architectures, a `Media` object within the NCL Document directly instantiates and drives a native multimedia element (such as a GTK widget or a GStreamer pipeline). The state machine logic blocks on native rendering calls (`draw()`), and the media objects trigger NCL transitions directly via system thread callbacks (`onEvent()`).

This bidirectional, tightly bound architecture makes it exceptionally difficult to virtualize the execution. In a web browser environment (*e.g.*, using WebAssembly), there is no access to raw socket connections for Ginga-CC-WebServices, no native threading model compatible with GStreamer, and no direct frame buffer for GTK to paint onto. Furthermore, JavaScript's single-threaded event loop and strict security sandboxes make blocking NCL state machine evaluations completely incompatible with web execution.

Recent efforts to modernize NCL tooling have explored cross-platform frameworks. One crucial related work is NuGinga Playcode [12], a web-compatible NCL authoring tool supporting NCL and NCLua using cross-compilation techniques. This work demonstrated the feasibility of porting NCL and Lua to the web by using C++ to Wasm toolchains. Our work, `gingaf`, extends this capability by providing a fully decoupled Dart implementation that natively supports both HTML (via Ginga-CC-WebServices integration) and Lua without heavily relying on C++ to Wasm toolchains.

Table 1 compares the features of `gingaf` with traditional implementations. By completely decoupling the logic component from the UI component, and by porting the logic entirely to pure Dart, `gingaf` successfully isolates the synchronization engine. This allows the logic to compile flawlessly to JavaScript and Wasm, completely sidestepping the historical native binding limitations.

4 METHODOLOGY

To overcome the inherent coupling between NCL logic and native rendering pipelines, `gingaf` introduces a novel architecture built using Dart and the Flutter[9] framework. A key architectural decision in our proposed architecture is the strict separation of the NCL state machine from the visual presentation layer, leveraging Dart's multiplatform capabilities. By implementing the core `NCLDocument` engine entirely in pure Dart as a non-UI component, we guarantee that the complex NCL logic can be natively compiled and executed across web, mobile, and desktop environments without requiring platform-specific bindings or C++ bridges. Figure 4 illustrates this high-level layered architecture.

[5] https://www.libsdl.org
[6] https://ffmpeg.org
[7] https://www.gtk.org
[8] https://gstreamer.freedesktop.org
[9] https://docs.flutter.dev/

Unpublished working draft. Not for distribution.

Table 1: Comparison of NCL Player Implementations

Work	Language	Multimedia Engine	Web Deploy	Mobile Deploy	Scheduling	Ginga-NCL	Ginga-HTML5	CCWS
PUC-Rio Ginga v0.13 [10, 15]	C++	SDL/FFmpeg	No	No	Event-Listener	Yes	No	No
PUC-Rio Ginga v1.0 [6, 16]	C++	GTK/Gstreamer	No	No	Synchronous `tick()`	Yes	No	No
WebNCL [9]	JavaScript	HTML5	Yes	No	Event-Listener	Yes	No	No
NuGinga [4, 12]	C++ (WASM)	HTML5	Yes	Yes	Event-Listener	Yes	No	No
`gingaf` (ours)	Dart	Flutter	Yes	Yes	Synchronous `tick()`	Yes	Yes	Yes

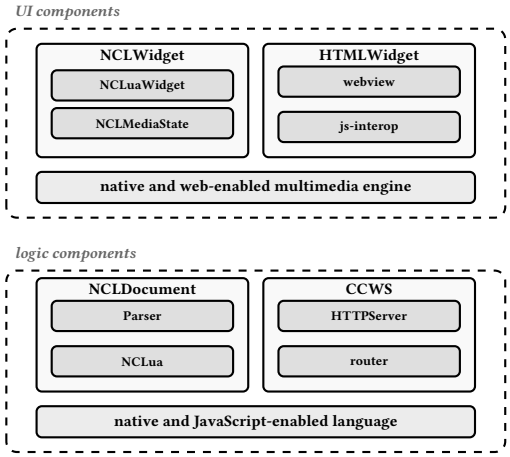


Figure 4: `gingaf` layered architecture organized into logic and UI components

At the bottom, the non-UI `NCLDocument` logic component operates within the logic layer. This layer is implemented in a native and JavaScript-enabled language, which is concretely realized by Dart. Dart enables compiling to native machine code for desktop and mobile, and uses the Dart2JS compiler to target highly optimized JavaScript for web execution. The presentation layer sits above this logic and is driven by a native and web-enabled multimedia engine. In our implementation, this engine is Flutter, which utilizes the CanvasKit renderer[10] (a WebAssembly build of Skia) to achieve high-performance, pixel-perfect layout and rendering in the browser without relying on browser-specific DOM elements. Within this structure, the presentation layer consists of the `NCLWidget` (which embeds the `NCLuaWidget` and `NCLMediaState`), the `HTMLWidget`, and the `webview`. Crucially, `NCLua` runs securely within the `NCLDocument` logic component, while the Ginga-CC-WebServices logic component exposes services via an embedded `HTTPServer` and routes them internally using the `router`.

The subsequent sections detail these architectural layers. Section 5 discusses the cross-platform Ginga-NCL execution, while Section 6 details the cross-platform Ginga-HTML5 and Ginga-CC-WebServices.

5 CROSS-PLATFORM GINGA-NCL

`gingaf` is built around two main components (Figure 5): the NCL Logic, implemented entirely in Dart, acts as a self-contained execution engine that manages the NCL event state machine with no knowledge of any display context; and the NCL UI, implemented as a Flutter widget, drives the engine and maps its output to native platform widgets. We discuss the two components in the next sections.

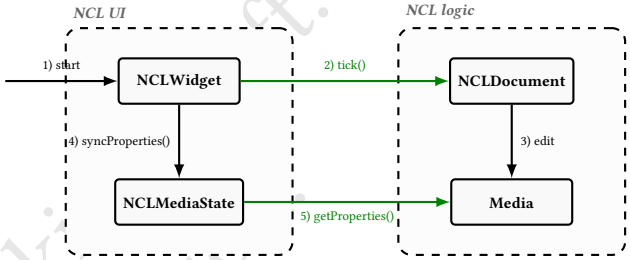


Figure 5: `gingaf` communication diagram

5.1 `gingaf` NCL Logic

The `NCLDocument` class is the core of the NCL logic component. The execution state of `NCLDocument` is managed through three key API methods: `start()` to initialize execution; `tick()` to advance the virtual clock and process NCL logic; and `stop()` to halt execution.

The `start()` function begins by parsing the NCL XML input and validating the resulting tree against XSD rules, checking ID uniqueness and IDREF integrity. When the `start()` ends, `NCLDocument` will have the following data structures:

- `DOM` (Document Object Model) contains `Element` instances in the document structure (Figure 6). Each `Element` contains its respective XML string and its raw XML attributes. `Node` extends `Element` to hold runtime state (sleeping, occurring, or paused) and a property list. `NCLua` is a special Media that executes Lua scripts in pure Dart[11] for multiplatform (other elements were omitted for simplicity).
- `VirtualClock`: the engine maintains an internal logical clock which Media use for their elapsed time. This clock is advanced via `tick()` calls (ideal for debugging).
- `ActionQueue`: inside one `tick()`, actions are queued and drained deterministically to ensure consistent execution behavior.
- `UIQueue`: A shared queue that serves UI and logic without coupling them. At the end of tick, the Document writes entries indicating which `Media` nodes changed visual properties, so the UI component can update rendering accordingly. Moreover, before each tick, the NCL UI component can

[10] https://skia.org/docs/user/modules/canvaskit/

[11] https://pub.dev/packages/lua_dardo_plus

Unpublished working draft. Not for distribution.

write entries to notify the document that a media presentation has ended (*e.g.*, a video file reached its end), allowing the engine to process the corresponding event and propagate any triggered links.

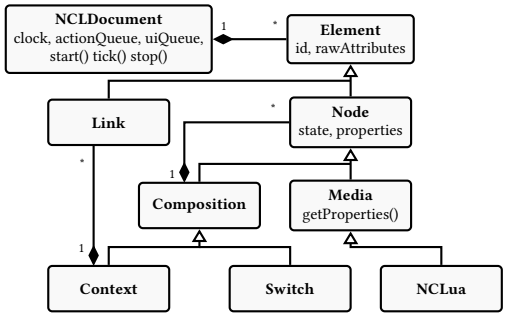


Figure 6: `gingaf` NCL `DOM`

`tick()` function advances an internal virtual clock by the given millisecond increment rather than relying on system time. This clock can be driven manually for testing or automatically by a Flutter periodic timer in a presentation environment. Decoupling time from the renderer enables high-speed headless testing: synchronization scenarios can be validated in-memory using inline XML strings, with no overhead from rendering windows, graphics libraries, or AV decoders.

On each call, `tick()` drains two queues. The `ActionQueue` collects transitions and link executions generated during state changes; these are drained deterministically in XML document definition order to ensure consistent execution behavior. The `UIQueue` serves communication across the component boundary: before the tick, the NCL UI component writes entries reporting that a media presentation has ended, which the engine processes to propagate any triggered links; after the tick, the engine writes entries indicating which `Media` nodes changed visual properties, so the NCL UI component can update rendering accordingly.

An example of the execution log produced during headless execution is shown in Listing 3.

```
> dart run ncl_doc/lib/main.dart examples/video.ncl
[ncl_doc_cli] Executing examples/video.ncl in headless mode
[Clock: 0.0s] NCLDocument will start
[Clock: 0.0s] Node "body": sleeping->occurring
[Clock: 0.0s] Node "video": sleeping->occurring
[ncl_doc_cli] perform 1 ticks per second indefinitely
[Clock: 0.0s] tickIndefinitely (1 ticks per/s)
[Clock: 1.0s] Tick
[Clock: 2.0s] Node "video" (area a1): sleeping->occurring
[Clock: 2.0s] Node "green_bg": sleeping->occurring
[Clock: 2.0s] Tick
[Clock: 3.0s] Tick
[Clock: 4.0s] Node "video" (area a1): occurring -> sleeping
[Clock: 4.0s] Node "green_bg": occurring -> sleeping
[Clock: 4.0s] Node "red_bg": sleeping->occurring
```

Listing 3: Headless execution log of video.ncl using `dart run ncl_doc/lib/main.dart video.ncl`

The reliability of the cross-platform codebase is maintained through an automated test suite. The `ncl_doc` logic package contains 64 unit tests validating the parser and virtual clock progression. Because Dart compiles efficiently to JavaScript, all tests run in milliseconds directly in continuous integration pipelines on the web.

5.2 `gingaf` NCL UI

The `gingaf` NCL UI layer is responsible for visual presentation. `NCLWidget` drives the engine by calling `tick()` on `NCLDocument` (arrow 2 in Figure 5) and delegates rendering to `NCLMediaState`, which reads the current visual properties of each `Media` node directly from the NCL logic component (arrow 5). The interface consists of a root `Ginga` widget that uses a Flutter `Stack` to lay out elements. A background audio/video widget (`main_av_widget`) is placed at the bottom, while the active media elements are rendered on top within the NCL presentation surface. The `WidgetFactory` maps NCL elements to native Flutter widgets.

Currently implemented widgets include `ImageWidget` (which displays images and handles image format caching), `AVWidget` (which renders native video/audio playback and supports state tracking), and `TextWidget` (which renders formatted text blocks using bounds properties). Each widget extends `BaseWidget` to map NCL layout properties (such as `left`, `top`, `width`, `height`, and `background`) to Flutter's `Positioned` and `BoxDecoration` parameters.

A key design aspect of `gingaf` is the bidirectional event feedback loop. `NCLWidget` holds an instance of `NCLDocument` and registers a Flutter periodic timer that periodically calls `doc.tick()` to advance the virtual clock.

When a media widget experiences a state change (for example, when an `AVWidget` reaches the end of its video file), the widget notifies the `NCLWidget` via the UI Queue. The application then retrieves the corresponding `Media` node from the document and triggers its transition event, allowing the headless execution engine to run link conditions and propagate state changes back to the widgets. This ensures a clean separation of concerns: widgets only worry about rendering and local lifecycle events, while the engine handles the declarative NCL coordination logic.

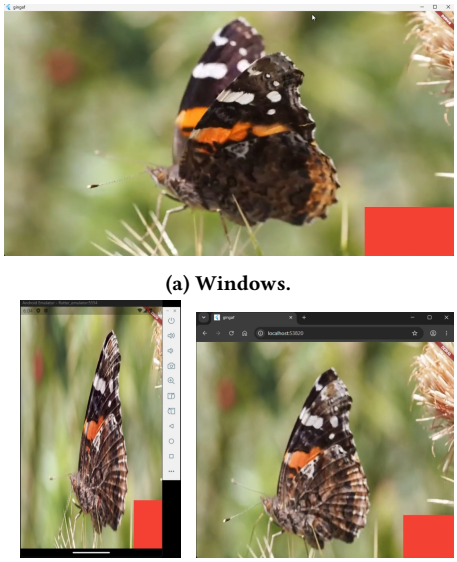
To show the correctness and multiplatform execution capabilities of `gingaf`, we execute the sample NCL application (`video.ncl`) described in Section 2. As a result, Figure 7 presents the execution of this application across three different environments: Windows, mobile (Android), and Web. Additionally, the temporal execution screenshots are shown in Figure 7 and the corresponding headless simulation logs are provided in Listing 3 (Section 5.1).

The reliability of the codebase is maintained through an automated test suite. The `ncl_doc` logic package contains 64 unit tests validating the parser, XML parsing rules, virtual clock progression, and deterministic link execution. The `gingaf` presentation package contains 14 tests verifying widget rendering, layout constraint resolution, focus state transitions, and integration with the NCL execution engine. All tests run in milliseconds due to the headless execution capabilities of the engine.

5.3 `gingaf` NCLua

A key component of the Ginga architecture is NCLua, which enables the execution of imperative Lua scripts within the NCL document. Historically, implementing NCLua on the web has been an insurmountable challenge because it required linking the standard C

Unpublished working draft. Not for distribution.



(a) Windows.

(b) Android. (c) Web.

Figure 7: Multiplatform execution of `video.ncl` across Windows, Android, and Web.

Lua interpreter to the web environment, typically using complex Emscripten toolchains that are difficult to integrate transparently with the DOM and modern frameworks like Flutter.

To overcome this, `gingaf` adopts a novel approach: it implements the NCLua player entirely in Dart using the `lua_dardo` library. `lua_dardo` is a pure Dart implementation of the Lua 5.3 interpreter. Because it is written natively in Dart, it compiles directly to JavaScript and WebAssembly along with the rest of the `gingaf` logic, seamlessly running in any web browser without external C dependencies or WebAssembly bridges.

When an NCL document defines a Lua media element, `gingaf` instantiates a `lua_dardo` state machine and binds the required NCLua APIs. These APIs allow the Lua script to register event listeners, query the NCL document properties, and emit events that the main `NCLDocument` logic can capture during its `tick()` cycle.

A major feature of NCLua is its ability to draw graphics onto a canvas using imperative commands. In a native C++ Ginga player, these commands call down into SDL or GTK canvas primitives. In a web environment, translating these calls synchronously across language barriers (*e.g.*, JavaScript to Dart to C) is cumbersome. `gingaf` overcomes this by using a pure Dart Lua interpreter (`lua_dardo`) to allow portability and a "Command List" abstraction. When the Lua script executes a drawing function (*e.g.*, `canvas:drawLine()`) the Lua interpreter intercepts the call and appends a lightweight, serializable command object to an internal list. This list acts as an intermediate representation of the visual frame. At the end of the Lua script's execution slice, the NCL UI queries this list. The Flutter rendering engine then iterates over the list, translating each stored command directly into native Flutter `Canvas` operations (*e.g.*, `canvas.drawLine()` or `canvas.drawParagraph()`). Figure 8 illustrates this pipeline.

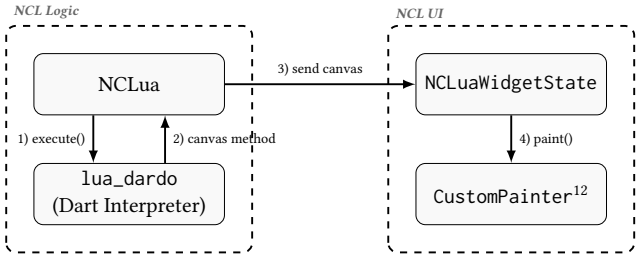


Figure 8: The NCLua Canvas Translation Pipeline. NCLua executes Lua via `lua_dardo`, which calls back canvas methods. NCLua forwards those calls to `NCLuaWidgetState`, which delegates rendering to a `CustomPainter`.

This architecture offers two immense benefits for the web. First, the command list completely decouples the imperative execution speed of Lua from the browser's rendering framerate. Second, it maintains absolute cross-platform consistency, as Flutter guarantees identical canvas rendering semantics whether the application is running as a WebGL application, a Windows desktop app, or an iOS native application.

Handling asynchronous Lua operations, such as network requests or timers, within the Dart event loop requires careful synchronization. `gingaf` maps Lua's coroutine mechanisms to Dart's `Future` and `async/await` paradigms. When a Lua script invokes an asynchronous NCLua API, the `lua_dardo` interpreter suspends execution and yields control back to the `NCLDocument`. Once the asynchronous operation completes, the engine resumes the Lua coroutine, passing the result back. This ensures that long-running Lua tasks do not block the main NCL execution thread, maintaining precise temporal synchronization across the entire document.

6 CROSS-PLATFORM GINGA-HTML5 AND GINGA-CC-WEBSERVICES

The Ginga standard specifies support for embedded HTML5 applications via Ginga-HTML5. In a traditional set-top box environment, this involves integrating an entire WebKit or Chromium instance into the C++ player, creating a massive platform dependency that requires complex cross-platform configuration. `gingaf` reduces this complexity by implementing Ginga-HTML5 using the Dart `webview_all` package[13]. This package abstracts the respective web engines on mobile (*e.g.*, Android WebView[14]) and desktop (OS-specific webviews, *e.g.*, WebView2[15]), while using JavaScript interoperability on the web. Moreover, a critical aspect of Ginga-HTML5 is its integration with CCWS, which defines REST APIs that allow the HTML application to access Ginga-CC capabilities, control the main video, and interact with the broadcast stream.

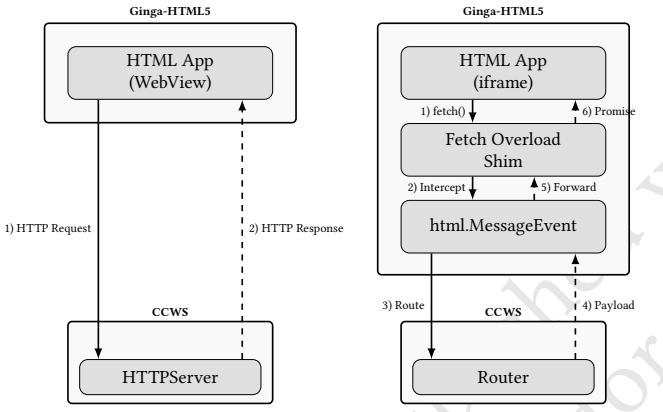
When running natively on desktop or mobile environments (Figure 9a), `gingaf` executes Ginga-HTML5 applications within a native WebView container. To support CCWS, the architecture embeds a local `HTTPServer` listening by default on port 44642. The

[12]https://api.flutter.dev/flutter/rendering/CustomPainter-class.html
[13]https://pub.dev/packages/webview_all
[14]https://developer.android.com/reference/android/webkit/WebView
[15]https://developer.microsoft.com/microsoft-edge/webview2

Unpublished working draft. Not for distribution.

HTML application makes standard HTTP requests targeting this local server, which handles routing, processes requests against the NCL document context, and returns standard HTTP responses.

When running in a web browser environment (Figure 9b), executing a local HTTP server is not possible due to browser security restrictions and sandbox limitations. To bypass this, `gingaf` executes the HTML application in a sandboxed `iframe` and virtualizes the CCWS middleware without an HTTP server. This is achieved by injecting a lightweight JavaScript shim into the `iframe` that overloads the standard browser `fetch` API and `XMLHttpRequest` objects. When the HTML application makes a request targeting a CCWS endpoint, the overloaded `fetch` intercepts it and dispatches a cross-origin message (`postMessage`) to the parent Dart/Flutter application. The parent application receives this message, routes it internally to the CCWS logic manager, and returns the response payload back to the iframe. The overloaded `fetch` then resolves its Promise with this mock response, completely transparent to the HTML application. This architecture successfully virtualizes the CCWS middleware entirely within the browser, eliminating the need for any backend or local HTTP server.



(a) Native Clients **(b) Web Client**

Figure 9: CCWS Request Architecture across different platforms. Native clients bridge WebView requests directly, while the Web client overloads `fetch` in a sandboxed iframe using `postMessage`.

7 USE CASE: GINGAPLAYGROUND

To validate our proposed decoupled architecture and verify that the core logic engine can run deterministically in a pure web sandbox, we developed GingaPlayground. This web-based editor use case serves to confirm the feasibility of our decoupled approach by integrating the Flutter UI for rendering the application, a code editor[16] for modifying NCL and Lua source files in real-time, and the cross-platform NCL logic running entirely in the browser.

This implementation directly validates that separating the NCL state from the UI renderer allows the logic engine to compile to JavaScript and Wasm, enabling instant, configuration-free execution without native desktop bindings or local virtualization. Through this environment, developers can visually inspect document states, manipulate the execution timeline, and dynamically reload modifications without leaving the browser, confirming that our solution provides similar execution capabilities compared to traditional players.

Figure 1 presents the visual execution of the sample Ginga-NCL application (`video.ncl`) from Listing 1, the Ginga-HTML5 application from Listing 2, and a NCLua application with canvas calls(`lua_canvas.ncl`) directly inside Google Chrome without any native plugins. The figure demonstrates the GingaPlayground interface layout, where the left side serves as the web-based code editor and the right side displays the real-time execution of the running Ginga application. This proves that both the temporal synchronization defined in the NCL links and the imperative NCLua drawing commands are rendered correctly in the browser.

8 FINAL REMARKS

`gingaf` demonstrates that decoupling the Ginga logic and UI components provides a highly portable solution that successfully brings the Ginga middleware to cross-platform environments, including the web. By isolating the NCL state machine and execution engine in pure Dart, the logic runs fully in-memory and can compile directly to JavaScript or native binaries without platform-dependent bindings or multimedia pipeline dependencies. This separation of concerns allows developers to execute and test synchronization logic deterministically on any platform, while rendering is handled independently by the presentation layer. Our architecture resolves the historical portability challenges of traditional Ginga middleware by removing the tight coupling between execution logic and native rendering APIs.

Furthermore, this web-native architecture opens new possibilities for collaborative development and continuous integration of digital TV applications. Since the `gingaf` engine runs deterministically in the browser, NCL projects can be easily shared via URL links, embedded in documentation, or integrated into online tutorials. This portability not only benefits educational initiatives but also accelerates industrial prototyping. Broadcasters and application designers can quickly iterate on UI layouts, Lua scripting logic, and CCWS interactions without deploying the code to physical set-top boxes during the early stages of development, ultimately reducing the time-to-market for new interactive services.

The immediate next step in this research is to evolve the GingaPlayground by implementing a Language Server Protocol (LSP)[17] server. Integrating an LSP server is highly beneficial because it decouples the editor client (*e.g.*, Monaco Editor or VS Code) from the core NCL validation and analysis engine. By encapsulating our Dart-based parser into a language server, we can deliver a consistent and feature-rich authoring experience, like real-time error reporting (diagnostics), context-aware autocompletion of NCL tags and attributes, and symbol outline navigation. Additionally, future work includes implementing support to allow Remote Players, for instance NCL360 formats [14].

ETHICAL CONSIDERATIONS

Not applicable.

[16] https://microsoft.github.io/monaco-editor/

[17] https://microsoft.github.io/language-server-protocol/

Unpublished working draft. Not for distribution.

ACKNOWLEDGEMENTS

We are grateful to all the academic members at SBTVD Forum working on the Ginga standard. We also acknowledge that Google Gemini was used to assist this work in: identifying bugs and generating unit tests; and grammar suggestions during writing.

REFERENCES

[1] ABNT. 2023. *ABNT NBR 15606-11: Digital terrestrial television – Ginga CC Web-Services – Ginga Common Core WebServices specification.* Technical Report. Associação Brasileira de Normas Técnicas.

[2] ABNT. 2023. *ABNT NBR 15606-2: Digital terrestrial television – Ginga-NCL for fixed and mobile receivers – XML application language for application coding.* Technical Report. Associação Brasileira de Normas Técnicas.

[3] ABNT. 2026. *ABNT NBR 15606-10: Digital terrestrial television – Ginga-HTML5 – Ginga HTML5 profile specification.* Technical Report. Associação Brasileira de Normas Técnicas.

[4] Rodrigo Braga, Volnei Klehm, Leonardo Santana, Kevin Takano, Lauro Gama, Addison Lima, Vinícius Melo, Taynara Paiva, Diego Gonzalez, Michael Mello, Chak Lee, and Lois Nascimento. 2018. NuGinga: a JavaScript Driven Implementation of Brazilian Digital TV Middleware for Android. In *Proceedings of the 24th Brazilian Symposium on Multimedia and the Web* (Salvador, BA, Brazil) *(WebMedia '18)*. Association for Computing Machinery, New York, NY, USA, 423–426. doi:10.1145/3243082.3267454

[5] Cosette Castro and Cristiana Freitas. 2016. Brazil 4D: An Experience of Interactive Content Production for Free-to-Air Digital Television. In *Handbook of Research on Comparative Approaches to the Digital Age Revolution in Europe and the Americas*. IGI Global Scientific Publishing, 202–215.

[6] J. P. Dodsworth, L. d. M. Terças, A. L. V. Guedes, G. F. Lima, C. d. S. S. Neto, and S. Colcher. 2018. Dynamic Integration of Foreign-Language Parsers into an NCL Player. In *Proceedings of the 24th Brazilian Symposium on Multimedia and the Web*. doi:10.1145/3243082.3243095

[7] Joel dos Santos, Li-Chang Shuen Cristina Silva Sousa, Carlos Soares Neto, Guido Lemos, Sérgio Colcher, Marcelo F. Moreno, and Débora Muchaluat-Saade. 2026. An Overview of the TV 3.0 Application Coding Layer Specification. *IEEE Transactions on Broadcasting* PP, 99 (2026), 1–15. doi:10.1109/TBC.2026.3689351

[8] ITU. 2009. *H.761: Nested Context Language (NCL) and Ginga-NCL for IPTV Services.* Technical Report. ITU, Geneva, Switzerland.

[9] Erick Lazaro Melo, Caio César Viel, Cesar A. C. Teixeira, Alexandre Coelho Rondon, and Daniel de Paula Silva. 2013. WebNCL: a web-based presentation machine for multimedia documents. In *Proceedings of the Brazilian Symposium on Multimedia and the Web (WebMedia)*. doi:10.1145/2382636.2382719

[10] Marcio Ferreira Moreno. 2010. *Conciliando Flexibilidade e Eficiência no Desenvolvimento do Ambiente Declarativo Ginga-NCL*. Ph. D. Dissertation. Pontifícia Universidade Católica do Rio de Janeiro. doi:10.17771/PUCRio.acad.35200

[11] Marcelo F. Moreno, Débora C. Muchaluat-Saade, Guido Lemos de Souza Filho, Raoni Kulesza, and Álan L. V. Guedes. 2020. A Brief Overview about D-Profile of Ginga DTV Receivers. In *IMX-LATAM*. doi:10.48550/arXiv.2608.02565

[12] Dina Nogueira, Lois Nascimento, Michael Mello, and Rodrigo Braga. 2020. NuGinga Playcode: A web NCL/NCLua authoring tool for Ginga-NCL digital TV applications. In *Anais Estendidos do XXVI Simpósio Brasileiro de Sistemas Multimídia e Web* (São Luís). SBC, Porto Alegre, RS, Brasil, 75–78. doi:10.5753/nuginga_playcode20.2020.13066

[13] Luiz Fernando Gomes Soares, Rogério Ferreira Rodrigues, and Romualdo Monteiro de Resende Costa. 2006. Nested Context Model 3.0 Part 6 – NCL (Nested Context Language) Main Profile. (2006).

[14] Gabriel Souza, D. Silva, M. Delgado, and R. Rodrigues. 2020. Interactive 360-degree Videos in Ginga-NCL Using Head-Mounted-Displays as Second Screen Devices. In *Proceedings of the 26th Brazillian Symposium on Multimedia and the Web (WebMedia '20)*. doi:10.1145/3428658.3430972

[15] TeleMídia Lab. 2013. PUC-Rio Ginga v0.13.6. https://github.com/TeleMidia/ginga/releases/tag/v0.13.6.

[16] TeleMídia Lab. 2018. PUC-Rio Ginga v1.0. https://github.com/TeleMidia/ginga/releases/tag/v1.0.

Unpublished working draft. Not for distribution.