3
Trabalhos Relacionados

Este capítulo apresenta os principais trabalhos relacionados a esta dissertação. A Seção 3.1 aborda duas soluções de reprodutores hipermídia para Web e as linguagens que lhes dão suporte, comparando-as com a proposta deste trabalho. Em especial, são discutidos os *plugins* reprodutores: Ambulant, baseado na linguagem SMIL; e o Adobe Flash Plugin, baseado na linguagem ActionScript. Já a Seção 3.2 discute a arquitetura e os componentes pertinentes ao *middleware* Ginga, com ênfase no subsistema declarativo Ginga-NCL. O objetivo dessa seção é destacar os principais módulos e componentes da arquitetura, que serão utilizados na adaptação do *middleware* como *plugin* no novo ambiente, a Web.

Por fim, na Seção 3.3 é discutido o framework empregado na adaptação do Ginga como um *plugin* para navegadores Web, mais especificamente o Gecko SDK (MOZILLA, 2010), utilizado pelos navegadores compatíveis com o Mozilla Firefox. Essa Seção tem como objetivo destacar os aspectos referentes a construção de *plugins* e ao mecanismo que viabiliza a ponte de comunicação entre as duas máquinas de apresentação (Ginga-NCL e Gecko).

3.1.
Soluções de Reprodutores Hipermídia para Web

Atualmente existem várias soluções para o emprego do conceito de hipermídia com o objetivo de enriquecer ainda mais as páginas da Web. As subseções a seguir apresentam duas soluções de maior relevância. A primeira, o Ambulant Webkit Plugin, baseada na linguagem SMIL e a segunda, o Flash Plugin, amplamente utilizada, baseada na linguagem ActionScript. Além disso, ao final desta seção, na Subseção 3.1.3, será apresentado uma comparação entre as linguagens que dão suporte aos reprodutores mencionados e a linguagem escolhida na proposta deste trabalho.

3.1.1.
Ambulant / SMIL State

O Ambulant (BULTERMAN, JANSEN, *et al.*, 2004) é um reprodutor hipermídia para aplicações especificadas na linguagem SMIL. Além de ser um reprodutor Desktop, ele pode ser facilmente integrado a outros reprodutores devido a sua abordagem baseada em componentes.

O Ambulant Webkit *Plugin*, ou simplesmente Ambulant Plugin, é um exemplo de integração entre o reprodutor Ambulant e o navegador Webkit (The WebKit Open Source Project, 2010). Originalmente, o reprodutor Ambulant oferece as aplicações clientes uma API para controle de apresentação da aplicação SMIL embutida. Porém, para que o Ambulant Plugin eleve o suporte a um nível de integração mais elevado ele deve fazê-lo a parte. Nesse sentido, o SMIL State (JANSEN e BULTERMAN, 2009) propõe adicionar ao Ambulant Plugin o suporte a especificação de estados definidos pelo usuário em linguagens declarativas baseadas no tempo, como SMIL ou SVG (W3C, 2003). Mais ainda, o SMIL State pode ser usado como um mecanismo de comunicação entre essas linguagens, permitindo a fácil comunicação com componentes externos (objetos de mídia, componentes de interface, etc.) e a página Web na qual está embutida.

Portanto, o objetivo principal do SMIL State é o de oferecer às aplicações SMIL a capacidade de se adaptar a mudanças ocorridas no decorrer da apresentação quando embutidas em uma página Web. Tais adaptações podem se apresentar na forma de interação com o usuário ou como mudanças no ambiente (tais como as informações de localização ou largura de banda disponível). Para atingir tal adaptação, ele prevê o uso de um modelo de dados comum, definido pelo usuário, como ilustra a Figura 7.

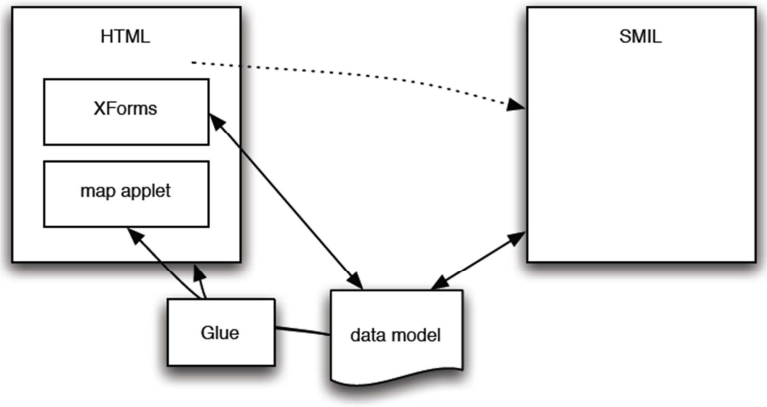

Figura 7 – Modelo de dados comum. Retirado de (JANSEN e BULTERMAN, 2009).

O SMIL State utiliza a definição de ganchos na linguagem SMIL para permitir a inclusão de um modelo de dados. O modelo é especificado na linguagem XML (W3C, 2008) e utiliza a linguagem de expressão XPATH (W3C, 2000) para fazer referencia a elementos do modelo dentro da aplicação SMIL ou fora dela. A manipulação do modelo por parte da pagina Web ocorre através de formulários dinâmicos XForms (W3C, 2009). Por outro lado, a manipulação do modelo por parte da aplicação SMIL utiliza um conjunto de elementos que compõe os ganchos da linguagem, sendo eles:

- o elemento <state>, especificado na seção do elemento <head>, cuja função é definir o modelo;
- o atributo *expr*, que pode ser usado em qualquer elemento SMIL (temporizado) para pular a apresentação do elemento mediante uma condição;
- os elementos <setvalue> , <newvalue> e <delvalue> são utilizados para manipular o modelo;
- os elementos <submission> e <send> são usados para envio e recebimento de partes do modelo;
- a expressão {*expression*} nos valores dos atributos para referenciar os valores definidos no modelo.

O emprego dos elementos que compõe os ganchos da linguagem e a ajuda dos formulários dinâmicos estabelece um canal de comunicação entre a página Web e a aplicação SMIL embutida.

A Figura 8 abaixo mostra um exemplo do uso do SMIL State em uma aplicação SMIL. No exemplo, quando o vídeo (match.mp4) começar a ser reproduzido, depois de 10 segundos, a imagem (banner.png) é exibida durante os 5 segundos seguintes. Se, durante esse intervalo, o usuário clicar sobre a figura (banner.png) o valor do elemento <wantAd>, declarado no modelo de dados comum, definido mais adiante, será alterado para (adv.mp4).

```
<smil>
<head>
</head>
<body>
  <seq>
    <par expr="wantAd">
      <video src="match.mp4"/>
      <img id="ban" begin="10s" end="15s" src="banner.png"/>
      <setvalue begin="ban.activateEvent"
                ref="wantAd" value="adv.mp4"/>
    </par>
    <video expr="wantAd" src="{wantAd}"/>
  </seq>
</body>
```

```
</smil>
```

Figura 8 – Aplicação SMIL que utiliza o SMIL State. Retirado e adaptado de (JANSEN e BULTERMAN, 2009).

Ainda na Figura 8**,** o exemplo demonstra a manipulação do modelo de dados a partir da aplicação SMIL, porém quando embutido em uma página Web o modelo também alterado por ela. A Figura 9 apresenta como exemplo uma página Web que embute a aplicação SMIL especificada na Figura 8. O exemplo define ainda o modelo de dados XML comum a aplicação embutida e a página Web. O modelo comum consiste de um elemento XML raiz <data> e tem como filho um elemento <wantAd>, ambos contidos em uma instancia de um formulário XForms (<form:model>). Ainda na Figura 9, é especificado uma caixa de seleção <form:select>, que, quando selecionada, altera o valor do elemento <wantAd> para (adv2.mp4), assim como ocorreu no evento de clique dentro da aplicação SMIL no exemplo da Figura 8.

```
<html>
<head>
  <form:model ...>
    <form:instance id="advInstance">
      <data xmlns="">
       <wantAd></wantAd>
      </data>
    </form:instance>
  </form:model>
</head>
<body>
  <embed id="sPlugId" src="ex.smil" width="400" height="100"/>
  <form:select ref="wantAd" ...>
    <form:label>Want Advertisement</form:label>
    <form:item>
      <form:label></form:label>
      <form:value>adv2.mp4</form:value>
    </form:item>
  </form:select>
</body>
</html>
```

Figura 9 – Aplicação SMIL embutida em uma página Web.

3.1.2. Flash Plugin

O Adobe Flash Player (ADOBE, 2010) é um reprodutor hipermídia proprietário usado para apresentação de aplicações especificados nas linguagens ActionScript (ADOBE, 2000). Os documentos ActionScript são compilados em um formato binário proprietário, chamado de SWF (*Shockwave*

Flash). O Adobe Flash Player está focado na especificação de animações vetoriais e na reprodução de mídia audiovisual em alta qualidade.

Além do reprodutor *standalone*, o Flash Player possui uma adaptação, na forma de *plugin,* para vários navegadores Web, chamado de Flash Plugin. O Flash Plugin permite que aplicações SWF sejam embutidas em páginas Web, além de fornecer uma API para controle, comunicação e edição.

Com a finalidade mencionada, a API *ExternalInterface* é fornecida pelo reprodutor aos desenvolvedores a fim de viabilizar uma ponte de comunicação entre a página Web e a aplicação SWF embutida. Tal API permite a página Web, através de código JavaScript, invocar procedimentos definidos na aplicação SWF, e, por sua vez, permite a aplicação SWF invocar procedimentos JavaScript definidos na página Web. Com esta finalidade, o método *call* da *ExternalInterface* determina em seu primeiro parâmetro qual método JavaScript deve ser invocado e , em seu segundo parâmetro, quais argumentos devem ser passados a no momento da chamada. Por outro lado, de posse da referencia para o reprodutor, o código JavaScript invoca os métodos especificados na aplicação SWF mediante o registro prévio através do método *addCallback*.

A Figura 10 exemplifica o uso da ExternalInterface, registrando uma função *ASCall* como externa e outra função como *handle* do evento de mouse (*onMouseDown*). A função *ASCall* exibe uma janela de alerta contendo a *string* passada como parâmetro, já a função *handle* do evento (*onMouseDown*) invoca a função JavaScript externa *asToJs*.

```
import flash.external.ExternalInterface

function ASCall(str:String){
  Alert.show(str);
}
onMouseDown = function(){
     ExternalInterface.call("asToJs", "Ola JS!");
}

ExternalInterface.addCallback( "ASCall", this, ASCall);
```

Figura 10 – Documento ActionScript que utiliza a ExternalInterface.

Depois de compilado, o documento ActionScript da Figura 10 passa a ser representada por um arquivo SWF (*ex.swf*) e como tal, pode ser embutido em uma página Web como uma aplicação SWF. Conforme exemplifica a Figura 11, uma vez embutida a aplicação SWF, o código JavaScript da página Web que a referencie especifica uma função JavaScript (*asToJs*), que exibe uma janela de alerta com uma mensagem contendo o parâmetro passado pela aplicação SWF. Além dessa, a página Web especifica também outra função JavaScript (*jsToAs*)

para realizar as chamadas a função ActionScript (*ASCall*), que por sua vez é invocada em resposta ao evento de click (*onclick*) do botão, cujo atributo *id* tem valor igual a *butId*, também contido na página Web.

```
<html>
<head>
  <script language = "JavaScript">
      var fPlugin = getElementById(“fPlugId”);

      function asToJs(str){
        alert(str);
      }
      function jsToAs(str){
        gPlugin.ASCall(str);
      }
  </script>
</head>
<body>
  <embed id="fPlugId" src="ex.swf" width="400" height="100"/>
  <button id="butId" type="button" onclick="jsToAs("Olá AS!");"/>
</body>
</html>
```

Figura 11 – Aplicação SWF embutida em uma página Web.

3.1.3. Comparação

Uma vez apresentadas as soluções existentes e considerando a solução proposta é possível analisar alguns detalhes relativos à integração entre os domínios, segundo a linguagem e a máquina de apresentação que lhes dão suporte.

A Tabela 1 apresenta uma comparação quanto ao suporte algumas funcionalidades, segundo a notação: “--” (muito pouco), “-” (pouco), “+” (limitado) e “++” (muito), entre as soluções (ActionScript/JavaScript), adotada pelo Flash Plugin, (SMIL/SMIL State), adotada pelo Plugin Ambulant e a (NCL/JavaScript), adotada pela solução proposta por este trabalho.

	ActionScript JavaScript	**SMIL SMIL State**	**NCL JavaScript**
Estruturada	--	++	++
Baseada no Tempo	-	++	++
Sincronismo	--	+	++
Granularidade	--	++	++
Ponte de comunicação	++	+	++

Adaptabilidade	-	++	++
Acessibilidade	-	++	++
Depende de linguagem script	++	--	++
Reuso	-	+	++
Encapsulamento	+	++	++
Múltiplos Dispositivos	--	--	++
Edição ao vivo	+	-	++

Tabela 1 – Tabela Comparativa entre as soluções apresentadas e a solução proposta.

De acordo com a Tabela 1, alguns aspectos envolvendo as linguagens e os formatos devem ser levados em consideração. Neste âmbito, por adotar um formato binário e não estruturado para representação de seus documentos ActionScript a solução baseada em Flash apresenta significativa desvantagem em relação às demais soluções. Isso ocorre porque as soluções baseadas em NCL e SMIL adotam o paradigma declarativo **estruturado** para a especificação e representação de suas aplicações, o que lhes concede a possibilidade de **reuso** de aplicações (ou fragmentos), maior **granularidade**, acessibilidade aos dados e independência de dispositivos (**adaptabilidade**). Por essa razão, o autor de um documento ActionScript deve especificar explicitamente o reuso, a adaptabilidade e a acessibilidade a serem consideradas na aplicação. Além disso, apesar de ser baseado no paradigma temporal (**baseadas no tempo**) a linguagem para especificação de aplicações SWF (ActionScript) oferece pouco suporte ao **sincronismo** entre mídia e intermídia. Dessa forma, ao contrário do que acontece com as linguagens NCL e SMIL, o autor do documento ActionScript também é obrigado a definir esse tipo de sincronismo no momento da autoria da aplicação.

Ambas as soluções, baseadas em NCL e em Flash, fazem uso da linguagem imperativa, o JavaScript, como **ponte de comunicação** entre os ambientes, o que as diferencia da solução baseada em SMIL, que utiliza um modelo da dados declarativo comum para tal finalidade. Tal abordagem, no caso da NCL, pode comprometer as vantagens alcançadas com a adoção do modelo declarativo estruturado (como o reuso, adaptabilidade e acessibilidade). Por outro lado, para contornar essa situação a presente proposta utiliza a interface direta com o *player* NCL, através de uma camada em código JavaScript. Tal camada abstrai os detalhes da aplicação NCL do ponto de vista da página Web e, deste modo, cria a ilusão de que aplicação NCL está contida em outra

aplicação NCL (na verdade uma página Web), como será visto no Capitulo 4. Isso representa uma grande vantagem, pois permite que a comunicação entre os ambientes ocorra de acordo como especificado pelo autor da aplicação NCL e não introduz nenhuma mudança na linguagem, ao contrário da solução baseada em SMIL, que especifica ganchos na linguagem para poder criar e manipular o modelo de dados comum.

No que diz respeito ao suporte a edição em tempo de exibição (**edição ao vivo**), a solução baseada em SMIL não apresenta nenhum suporte a não ser o decorrente da manipulação do modelo comum, definido pelo usuário. A solução baseada em Flash oferece suporte à edição de forma direta e ampla, o que limita a edição aos procedimentos pré-definidos pelo autor da aplicação SWF. Por outro lado, a solução baseada em NCL oferece suporte à edição de forma indireta e controlada, através dos comandos de edição, que são previamente especificados pelo autor da aplicação NCL. Por esse motivo a solução proposta e a baseada em SMIL possuem um nível de encapsulamento maior do que a solução baseada em ActionScript. Outro aspecto pertinente envolve o suporte a **múltiplos dispositivos**, o que favorece a solução proposta, uma vez que a NCL possui extenso suporte a esse recurso.

3.2. Middleware Ginga

O *middleware* Ginga é a camada de *software* responsável por prover um ambiente de execução às aplicações da TV Digital. Para isso, o *middleware* deve fornecer às aplicações uma interface de programação para a apresentação de mídias, o controle de suas reproduções, chamadas ao sistema e o acesso a dispositivos (como os de captura, sintonização e interação). Segundo (SOARES e BARBOSA, 2009, p. 26), uma vez atendidos os requisitos exigidos pela aplicação NCL, o *middleware* deve prover um bom suporte para:

- o sincronismo de uma forma geral e, como caso particular, a interação do usuário;
- a definição de relacionamentos de sincronismo espacial e temporal separada da definição do conteúdo dos objetos de mídia relacionados;
- a adaptação do conteúdo e da forma como o conteúdo é exibido;
- o uso múltiplos dispositivos de exibição;
- a edição ao vivo (em tempo de exibição).

Esses recursos fazem do *middleware* Ginga um ambiente ideal para a apresentação de aplicações hipermídia que especifiquem algum tipo de sincronismo. Tal condição permite a criação de aplicações como as que especificam o aparecimento de uma imagem contendo um anúncio durante um programa televisivo. Ou ainda, aplicações mais sofisticadas, que especifiquem momentos de interação com usuário, adaptação do conteúdo exibido segundo o contexto do telespectador, ou mesmo a exibição em múltiplos dispositivos. Isso se deve em grande parte pela adoção de uma arquitetura de *software* bem definida, como será apresentado na seção a seguir.

3.2.1. Arquitetura

O *middleware* Ginga possui uma arquitetura modular baseada em componentes de *software*, onde cada componente representa um conjunto de funcionalidades comum que, quando agrupadas, delimitam o escopo de um módulo. A Figura 12 ilustra os principais módulos que compõe a arquitetura do *middleware*.

Figura 12 – Arquitetura do *middleware* Ginga. Retirada de (MORENO, 2010).

Conforme destaca a Figura 12, o *middleware* Ginga está dividido em dois subsistemas lógicos: o Ambiente de Apresentação ou Máquina de Apresentação, chamado de *Ginga-NCL*, e o Núcleo Comum Ginga, chamado de *Ginga-CC*. Um terceiro subsistema, opcional na arquitetura Ginga, genericamente chamado de Ginga-Imp, poderia dar um suporte direto ao controle de aplicações geradas com

base em uma linguagem imperativa. O *Ginga-NCL*, que será detalhado na Subseção 3.2.1.2, é responsável pelo controle de apresentação das aplicações NCL. Já o *Ginga-CC*, que será detalhado na Subseção 3.2.1.1, é responsável por prover um conjunto de serviços, dependente de plataforma, ao Ambiente de Apresentação, às aplicações residentes e ao possível ambiente imperativo adicional.

3.2.1.1. Subsistema Ginga-CC

O módulo *Sintonizador* está incumbido de receber as aplicações interativas que venham multiplexadas em fluxo(s) de transporte[1] (TS – Transport Stream) provenientes do produtor de conteúdo. Isso pode se dar por meio de interfaces de sintonização ISDB-T (ABNT, 2007) ou por outras interfaces (como uma interface de rede Ethernet). É competência de o módulo de Transporte gerenciar as conexões relativas aos objetos de mídia, quando o recebimento for feito através de uma interface de rede.

As aplicações interativas e objetos de mídia transmitidos multiplexados no fluxo de transporte pelas emissoras são organizados na forma de uma árvore de diretórios e transportados utilizando um protocolo de oferta continua, denominado carrossel de objetos DSM-CC (*Digital Storage Media – Command and Control*). É responsabilidade do módulo *Processador de Dados* monitorar as interfaces de rede a procura de eventos que sinalizem a existência e a origem de aplicações. Uma vez extraídas as aplicações, o módulo *Gerente de E/S* torna-se responsável por gerenciar a persistência temporária das aplicações e de seus objetos de mídia associados.

Durante a apresentação da aplicação, os objetos de mídia referenciados pelo documento NCL devem ser exibidos tal qual foram especificados pela aplicação. Cabe ao módulo *Exibidores* oferecer um exibidor (ou também chamado de *player*) apropriado para cada tipo de mídia, que deve ser capaz de decodificar, controlar e apresentar o objeto de mídia correspondente referenciado. O acesso à API de decodificação de conteúdo dos exibidores é abstraído pelo módulo *Adaptadores*, oq ue garante o acesso uniforme por parte do Ambiente de Apresentação. Como destacado na Seção 1.1, o *Ginga-CC* possui suporte a inserção de objetos imperativos *NCLua* através de um módulo

[1] Um fluxo de transporte é composto pelo conteúdo audiovisual principal além de dados como aplicações NCL e objetos de mídia referenciados por essas aplicações.

interpretador Lua — *Máquina Lua*, que, associado a um exibidor Lua, é capaz de desenhar imagens e gráficos primitivos, além de gerar e tratar eventos de E/S ou internos ao Ambiente de Apresentação. Esse modulo, em especial, será utilizado no Capitulo 4, onde é discutido a definição de uma interface comum para o *plugin*, por esses seguirem a mesma interface.

O conteúdo decodificado pelo módulo *Exibidor* deve ser renderizado de acordo com a posição espacial especificada. Para isso, o módulo *Gerente Gráfico* deve realizar o controle do processo de renderização do conteúdo referenciado, inclusive do conteúdo audiovisual principal. A implementação de referência utiliza como *backend* gráfico do Gerente Gráfico, o DirectFB (DIRECTFB, 2010). Porém, como propõe o Capitulo 5 e apresenta o Seção 5.1, a fim de adequar o *plugin* ao ambiente do navegador alvo (Desktop). Como consequência um novo *backend* gráfico deve que ser escolhido, o DirectX, apresentado no Apêndice , o qual poderá servir de base para trabalho futuros. Além do *backend* gráfico, a Subseção 5.1.2 apresenta seus novos respectivos exibidores.

O suporte a múltiplos dispositivos de exibição é oferecido pelo módulo *Gerente de Dispositivos*, que coordena o registro de dispositivos, a comunicação entre os dispositivos, a construção de domínios de apresentação e garante a preservação do sincronismo das aplicações distribuídas.

Ao módulo *Gerente de Contexto* é atribuída a tarefa de obter e gerenciar as informações referentes ao sistema como um todo e ao perfil dos usuários. Por fim, o módulo *Gerente de Evolução* deve gerenciar as atualizações dos componentes pertencentes aos módulos de forma independente, provenientes do canal de difusão, portador do fluxo de transporte, ou por meio do módulo *Transporte,* quando for utilizada outra interface.

3.2.1.2. Subsistema Ginga-NCL

O módulo *Formatador* é responsável por receber e controlar as aplicações NCL entregues pelo *Ginga-CC*. Ao receber uma aplicação, o *Formatador* solicita ao módulo *Parser XML* que interprete o documento NCL. Durante o processo, o módulo *Parser XML* requisita ao módulo *Conversores* a conversão das entidades NCL interpretadas em estruturas de dados que representem as entidades do modelo NCM.

As estruturas de dados NCM resultantes da interpretação das aplicações NCL são agrupadas em uma estrutura de dados denominada *Base Privada*. O módulo *Gerente de Base Privadas* tem como uma de suas principais funções o controle da *Base Privada* associada aos documentos NCL, podendo iniciar, parar, resumir e interromper a apresentação das aplicações, além de coordenar as referências entre documentos. Outra função importante exercida pelo módulo *Gerente de Base Privadas* envolve o suporte a edição em tempo de exibição, sendo de sua responsabilidade receber, validar, interpretar e aplicar os Comandos de Edição ao documento associado à *Base Privada* de destino.

Como propõe este Trabalho, os Comandos de Edição podem ser executados, por meio de uma linguagem script, a partir de uma página Web. Por esse motivo, além do suporte a transporte através de seções DSM-CC o módulo em questão também deve suportará a submissão de comandos no lado do cliente, como discute a Subseção 4.2.3.

O módulo *Escalonador de Aplicações* realiza a orquestração da apresentação da aplicação NCL durante sua execução, solicitando ao módulo *Conversores* a tradução das estruturas NCM em objetos de execução para facilitar sua apresentação. Para que os objetos de mídia sejam apresentados corretamente, o módulo *Escalonador de Aplicações* requisita ao módulo *Gerente de Exibidores* que instancie o exibidor, de acordo como o tipo de objeto de mídia referenciado. Quando na ocorrência de eventos dos objetos de mídia, através de suas interfaces NCM, o modulo *Escalonador de Aplicações* é notificado sobre tais eventos. Portanto, para que haja o sincronismo, na sua forma mais ampla, entre os objetos de mídia contidos na página Web e os contidos na aplicação NCL é necessário a notificação externa desses eventos. A Subseção 4.2.1 discute uma maneira de fazê-lo para as *âncoras de conteúdo* e as *âncoras de propriedades*.

De posse do exibidor apropriado o *Escalonador de Aplicações* inicia a apresentação do objeto de mídia de acordo com a região especificada na aplicação. O módulo *Gerente de Leiaute* deve realizar a associação entre o conteúdo fornecido pelo exibidor e a região NCL. Caso a região tenha sido especificada em outro dispositivo, o que implica em uma exibição distribuída, o módulo *Gerente de Leiaute* deve utilizar o serviço provido pelo módulo *Gerente de Dispositivos* para comunicação com o ponto remoto.

O módulo *Gerente de contexto* é encarregado de adaptar o conteúdo no momento de sua apresentação, segundo o módulo *Gerenciador de Contexto* do *Ginga-CC* e das informações contidas na própria aplicação. Por fim, o módulo

Gerente de Recuperação é encarregado de realizar o planejamento para a busca dos objetos de mídia.

3.3. Mozilla Firefox e Gecko SDK

Em 1998 a Netscape, maior concorrente da Microsoft no mercado de navegadores para Web na época, decidiu liberar o código fonte de seu navegador Netscape Communicator 5.0 para que desenvolvedores do mundo todo pudessem colaborar em um novo projeto, chamado Mozilla (NETSCAPE, 1998). Inicialmente o projeto lançou um novo navegador Web chamado Mozilla Browser, que depois passou a ser chamado de Phoenix Browser, em seguida de Firebird Browser e, finalmente, trocou seu nome para Firefox, em 2004. O navegador Mozilla Firefox sempre foi reconhecido pelo seu caráter inovador, tendo como uma de suas principais características a flexibilidade. Tal flexibilidade é atingida através da incorporação ou aprimoramento de funcionalidades através de mecanismos voltados para a extensão e personalização (como *temas*, *plugins* e *extensores* — chamados de Mozilla *add-ons*).

Os *extensores* permitem ao navegador agregar novas funcionalidades no que concerne a aplicação navegadora e suas interfaces (como a inclusão de barras de procura e botões de acesso). Por outro lado, os *plugins* permitem ao navegador incorporar funcionalidades externas a ele (como a integração com reprodutores de mídia, visualizadores, antivírus e editores). Esta dissertação se atém, entretanto, somente no mecanismo de *plugin*, visto que, dentre os mecanismos de extensão, ele é o único capaz de oferecer suporte para a integração entre o navegador Web e um reprodutor hipermídia nativo.

Mozilla Gecko é um motor de leiaute (máquina de apresentação), de código aberto, adotado pelo Mozilla Firefox e por outros navegadores (como lock, Netscape, SeaMonkey, Camino e K-Meleon). Sua principal função é a de interpretar conteúdos para Web (como documentos HTML, CSS, XUL e JavaScript) e apresentá-los de forma correta para o usuário. Por consequência, outra função importante do motor de leiaute é a de gerenciar os mecanismos de extensão, garantida que uma vez requisitada, a extensão seja corretamente apresentada.

3.3.1. NPAPI

Em particular, o Gecko SDK oferece uma API, chamada de NPAPI (*Netscape Plugin Application Programming Interface)*, voltada para o desenvolvimento de *plugins*. Um *plugin* Gecko é distribuído na forma de uma biblioteca de ligação dinâmica, em conformidade com o padrão C de sintaxe e com a interface especificada pela NPAPI.

Um *plugin*, além das funcionalidades inerentes ao seu propósito, deve especificar um conjunto de procedimentos para sua inicialização, execução e término. Em contra partida, o Gecko SDK deve oferecer ao *plugin* procedimentos para o acesso aos recursos do navegador. Decorrente disso, a NPAPI define dois grupos de funções e um grupo de estruturas de dados compartilhadas para o desenvolvimento de *plugins*:

- O primeiro grupo é composto por funções implementadas pelo Gecko e invocadas pelo *plugin*. Essas funções têm seus nomes iniciados pelo prefixo "*NPN*" (como as funções *NPN_Write* e *NPN_Read*).
- O segundo grupo é formado por funções implementadas pelo *plugin* e invocadas pelo Gecko. Já essas têm seu nome iniciado pelo prefixo "*NPP*", caso estejam no escopo do plugin, ou têm seus nomes iniciados pelo prefixo "*NP*", caso estejam no escopo da biblioteca (como as funções *NPP_New* e *NP_Initialize*).
- O terceiro grupo é formado por algumas estruturas de dados compartilhadas definidas pelo Gecko. Elas têm sem nomes iniciadas pelo prefixo "*NP*" e são compartilhadas com o *plugin* (como a estrutura *NPWindow*, que encapsula a estrutura de uma janela nativa).

No que concerne a ponte de comunicação entre as máquinas de apresentação, o grupo de funções de prefixo "NPN" permitem a chamada de rotinas scripts (JavaScript), definidas no corpo de uma página Web. A Subseção 4.2.1 discute uma forma de aproveitar esse mecanismo para realizar a notificação de eventos entre a máquina de apresentação Ginga-NCL e a maquina de apresentação Gecko.

Com o uso da NPAPI, *plugins* podem exercer a função de reprodutores direcionados a um tipo particular de mídia, como veremos na seção seguinte, beneficiando-se de recursos da plataforma alvo. Somente o navegador, através

de seu motor de leiaute, não seria capaz de apresentar o conteúdo de mídia referenciado, levando em consideração somente os recursos que detém.

3.3.2. Modelo de Execução

O padrão HTML define dois elementos específicos para a inserção de conteúdos de mídia não previstos em uma página Web. O elemento <object> e o elemento <embed>. Apesar da divergência existente sobre qual dos dois elementos deve ser padronizado, a última versão do padrão (W3C, 2010) prevê a ocorrência de ambos. Porém, neste trabalho, será dado foco somente ao elemento <embed>.

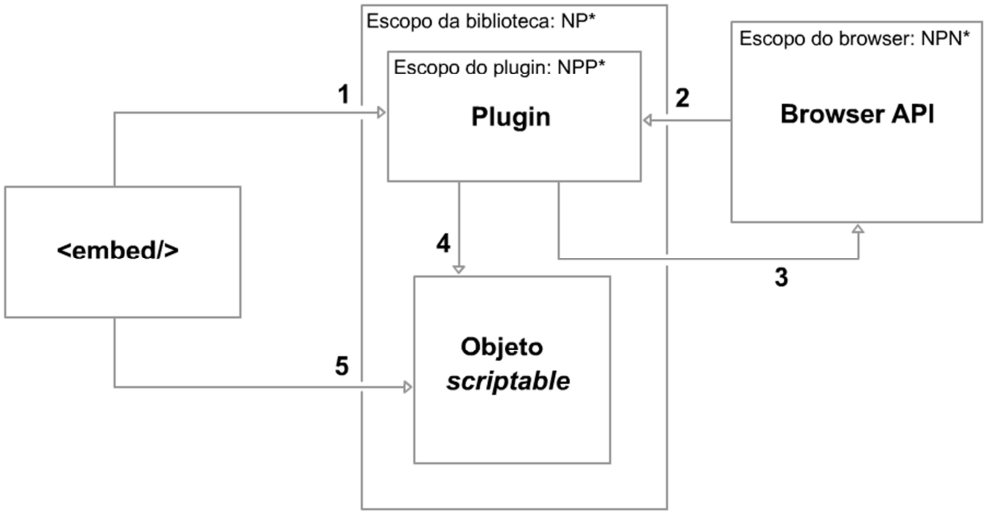

Figura 13 – Ciclo de execução de um plugin Gecko, segundo a NPAPI.

Como ilustra a Figura 13, o ciclo de execução de um *plugin* Gecko é determinado pela página Web no qual está inserido. Portanto, no momento em que o Gecko é solicitado a exibir uma página Web, ele realiza a interpretação do documento HTML. Durante esse processo, caso encontre algum elemento <object> ou <embed>, o Gecko obtém o valor do atributo *type* ou checa a extensão do arquivo no atributo *src*, pelos quais descobre qual o *Mime type* da mídia a ser reproduzida. Em seguida, o Gecko consulta a lista de *plugins* registrados em busca do qual deve ser carregado, se ainda não estiver. Caso seja encontrado, o *plugin* é carregado em memória, veja Figura 13(1), pelo navegador, que então chama a rotina *NP_Initialize* para realizar a inicialização global do *plugin* e a alocação de recursos necessários a execução.

A partir desse ponto, o *plugin* está pronto para ser instanciado, o que ocorre com a chamada a função do *plugin NPP_New*, veja Figura 13(2). Nesse

momento todos os recursos relacionados a inicialização do *middleware* como reprodutor devem ser alocados, assim como os procedimentos para a inicialização do *backend* gráfico. A Seção 4.1 discute os aspectos referentes a inicialização e configuração do *middleware* enquanto *plugin*.

Uma vez o *plugin* carregado e instanciado, o Gecko chama a função *NP_GetValue* em busca do valor da variável *NPPVpluginScriptableNPObject*, veja Figura 13(4). Tal variável determina a existência de um objeto *scriptable*, que é capaz de oferecer uma interface JavaScript para o código nativo contido no *plugin*, veja Figura 13 (5). A extensão da interface de uma linguagem para além do seu escopo é discutida na Seção 4.2.1 como forma de viabilizar a comunicação no sentido da máquina de apresentação Gecko para a máquina de apresentação Ginga-NCL.

Após a destruição da última instância do *plugin*, não é mais necessário mantê-lo em memória. Assim, o navegador chama a função *NP_ShutDown*, que se encarrega de liberar os recursos alocados na instanciação e encerrar o *plugin*.

3.3.3. Modos de Apresentação

Para um *plugin* Gecko dois modos de apresentação estão disponíveis, o modo *windowed* e *windowless*. A escolha do modo apresentação está a cargo do desenvolvedor, influenciando no processo de renderização e no tratamento de eventos (tanto originados do usuário quanto do navegador).

No modo *windowed*, o *plugin* possui uma janela nativa própria. A área de renderização é vista pelo *plugin* como um retângulo opaco que sobrepõe todos os objetos HTML que estejam sendo exibidos atrás dele. Isso ocorre, pois o *plugin* é quem determina quando e como a renderização deve ocorrer e não o navegador. Nesse processo, o navegador cria uma janela nativa, encapsulada pela estrutura *NPWindow*, para cada instância do *plugin*, notificando-o sobre a existência da janela através da função *NPP_SetWindow*, tão logo for instanciado. O tratamento dos eventos cabe ao *plugin*, sendo ele o responsável por receber e tratar todos os eventos seja os eventos relacionados à janela nativa como os eventos vindos do usuário.

No modo *windowsless*, o *plugin* não possui uma janela nativa própria e sim uma área de renderização abstrata (*bitmap buffer*), criada pelo navegador. Ao contrário do que acontece no modo *windowed,* a área de renderização pode

conter partes transparentes e opacas, fazendo com que o plugin assuma aspectos visuais irregulares, o que torna objetos HTML sobrepostos visíveis. Nesse caso, como o *plugin* não tem uma janela própria, o navegador fica encarregado de encaminhar, por meio da chamada a função *NPP_HandleEvent*, os eventos pertinentes à área de renderização e os eventos do usuário.

Para o controle e sincronismo do processo de renderização no modo *windowless*, o navegador envia eventos do tipo desenho (*Paint Message*) ao *plugin* em intervalos de tempo regulares. Após receber esse evento, visando evitar um conflito de renderização, o *plugin* deve especificar em quais partes da área de renderização ele deseja atuar, passando os retângulos correspondentes nas chamadas as funções *NPN_InvalidateRect* ou *NPN_InvalidateRegion*.

No contexto de um *plugin* Gecko, o modo *windowless* oferece uma opção de leiaute mais flexível em relação ao modo *windowed*, decorrente do fato de poder assumir formas irregulares e apresentar vários níveis de opacidade. Em contrapartida, o modo *windowed* apresenta uma forma mais estável e controlada devido ao tratamento nativo de eventos e ao controle total do processo de renderização. Por esse motivo o modo *windowed* foi escolhido neste trabalho, sua utilização é discutida na Seção 4.1.