Referências Bibliográficas

[1] ABNT. *ABNT NBR 15606-2:2007 Televisão digital terrestre – Codificação de dados especificações de transmissão para radiodifusão digital. Parte 2: Ginga-NCL para receptores fixos e móveis – Linguagem de aplicação XML para codificação de aplicações.* Rio de Janeiro: ABNT - Associação Brasileira de Normas Técnicas, 2007.

[2] ITU-T. *ITU–T Recommendation H.761, 2009: Nested Context Language (NCL) and Ginga-NCL for IPTV services.* Geneva: International Telecommunication Union, 2009.

[3] MORENO, M. F. *Conciliando flexibilidade e eficiência no desenvolvimento do ambiente declarativo Ginga-NCL.* Tese de doutorado - PUC–Rio, Rio de Janeiro, 2010.

[4] MORENO, M. F. *Um middleware declarativo para sistemas de TV digital interativa.* Dissertação de mestrado - PUC–Rio, Rio de Janeiro, 2006.

[5] RODRIGUES, R. F. *Formatação temporal e espacial no sistema HyperProp.* Dissertação de mestrado - PUC–Rio, Rio de Janeiro, 1997.

[6] SOARES, L. F. G.; RODRIGUEZ, N. L. R.; CASANOVA, M. A. NCM: A conceptual model for hyperdocuments. *I Workshop em Sistemas Hipermídia Distribuídos – SBMídia95*, p. 40–46, 1995.

[7] MEIRE, J. A. NCL: Uma linguagem declarativa para especificação de documentos hipermídia na web. *VI Simpósio Brasileiro de Sistemas Multimídia e Hipermídia – SBMídia2000*, p. 79–95, 2000.

[8] VIANU, V. XML: From practice to theory. In: SBBD. 2003. p. 11–25.

[9] LIMA, G. A. F. *The NCL Converter Collection (for NCC version 1.0).* A ser publicado, 2011.

[10] RAYMOND, E. S. *The Art of UNIX Programming.* Pearson Education, 2003.

A
Gramática do Perfil EDTV

A Tabela A.1 apresenta a gramática pelo perfil NCL EDTV. Na tabela, os elementos aparecem listados em ordem alfabética. Parênteses são usados para delimitar listas de elementos. O símbolo '|' denota um conjunto de alternativas e os símbolo '?', '*' e '+' indicam, respectivamente, zero ou uma, zero ou mais, e uma ou mais repetições. Atributos sublinhados são obrigatórios.

Tabela A.1 Gramática do perfil EDTV

Elemento	Atributos	Conteúdo
<area>	*id, coords, begin, end, text, position, first, last, label*	-
<assessmentStatement>	*comparator*	(attributeAssesment, (attributeAssessment | valueAssessment))
<attributeAssessment>	*role, eventType, key, attributeType, offset*	-
<bind>	*id, component, interface, descriptor*	(bindParam)*
<bindParam>	*name, value*	-
<body>	*id*	(port | property | media | context | switch | link | meta | metadata)*
<bindRule>	*constituent, rule*	-

Tabela A.1 (continuação)

Elemento	Atributos	Conteúdo
<causalConnector>	*id*	(connectorParam*, (simpleCondition | compoundCondition), (simpleAction | compoundAction))
<connectorBase>	*id*	(importBase | causalConnector)*
<connectorParam>	*name*, *type*	-
<compositeRule>	*id*, *operator*	(rule |compositeRule)+
<compoundAction>	*operator*, *delay*	(simpleAction | compoundAction)+
<compoundCondition>	*operator*, *delay*	((simpleCondition | compoundCondition)+, (assessmentStatement | compoundStatement)*)
<compoundStatement>	*operator*, *isNegated*	(assessmentStatement | compoundStatement)+
<context>	*id*, *refer*	(port | property | media | context | switch | link | meta | metadata)*

Tabela A.1 (continuação)

Elemento	Atributos	Conteúdo
<defaultComponent>	*component*	-
<defaultDescriptor>	*descriptor*	-
<descriptor>	*id*, *player*, *explicitDur*, *region*, *freeze*, *moveUp*, *moveDown*, *moveLeft*, *moveRight*, *focusIndex*, *focusSrc*, *focusSelSrc*, *focusBorderColor*, *focusBorderWidth*, *focusBorder-Transparency*, *selBorderColor*, *transIn*, *transOut*	descriptorParam*
<descriptorBase>	*id*	(importBase | descriptor | descriptorSwitch)+
<descriptorParam>	*name*, *value*	-
<descriptorSwitch>	*id*	(defaultDescriptor?, (bindRule | descriptor)*)

Tabela A.1 (continuação)

Elemento	Atributos	Conteúdo
<head>	-	(importedDocumentBase?, descriptorBase?, regionBase*, transitionBase?, connectorBase?, ruleBase?, meta*, metadata*)
<importBase>	*alias*,*documentURI*,*region*	-
<imported-DocumentBase>	*id*	(importNCL)+
<importNCL>	*alias*, *documentURI*	-
<link>	*id*, *xconnector*	(linkParam*, bind)+
<linkParam>	*name*, *value*	-
<mapping>	*component*, *interface*	-
<media>	*id*, *src*, *refer*, *instance*, *type*, *descriptor*	(area | property)*
<meta>	*name*, *content*	-
<metadata>	-	RDF tree
<ncl>	*id*, *title*, *xmlns*	(head?, body?)

Tabela A.1 (continuação)

Elemento	Atributos	Conteúdo
<port>	*id*, *component*, *interface*	-
<property>	*name*, *value*, *externable*	-
<region>	*id*, *title*, *left*, *right*, *top*, *bottom*, *height*, *width*, *zIndex*	region*
<regionBase>	*id*, *device*, *region*	(importBase | region)+
<rule>	*id*, *var*, *comparator*, *value*	-
<ruleBase>	*id*	(importBase | rule | compositeRule)+
<simpleAction>	*role*, *delay*, *eventType*, *actionType*, *value*, *min*, *max*, *qualifier*, *repeat*, *repeatDelay*, *duration*, *by*	-
<simpleCondition>	*role*, *delay*, *eventType*, *actionType*, *value*, *min*, *max*, *qualifier*, *repeat*, *repeatDelay*, *duration*, *by*	-

Tabela A.1 (continuação)

Elemento	Atributos	Conteúdo
<switch>	*id*, *refer*	defaultComponent?, (switchPort | bindRule | media | context | switch)*
<switchPort>	*id*	mapping+
<transition>	*id*, *type*, *subtype*, *dur*, *startProgress*, *endProgress*, *direction*, *fadeColor*, *horRepeat*, *vertRepeat*, *borderWidth*, *borderColor*	-
<transitionBase>	*id*	(importBase | transition)+
<valueAssessment>	*value*	-

B
LibPlayer

A LibPlayer é uma biblioteca C para construção de apresentações multimídia interativas. A biblioteca foi projetada para facilitar o mapeamento de operações sobre objetos de mídia NCL em primitivas audiovisuais do sistema. Este capítulo descreve a arquitetura e a implementação dessa biblioteca.[1]

B.1
API de player

A principal abstração da API da LibPlayer é o *player*. Um *player* é um exibidor de mídia associado a algum conteúdo. Todo *player* possui propriedades, recebe ações e notifica eventos. As propriedades controlam como o conteúdo do *player* é apresentado. Por exemplo, o *player* de imagem possui a propriedade "transparency" que controla a transparência da imagem exibida. As ações, por sua vez, são comandos que podem ser enviados aos *players*. Dois exemplos típicos são as ações "start" e "stop" que disparam, respectivamente, o início ou fim da apresentação do *player*. Finalmente, toda ação gera um evento que é notificado através de *callbacks*.

O conceito de *player* é análogo ao objeto de mídia de NCL. A única diferença é que o *player* não define âncoras. De fato, cada âncora do objeto de mídia pode ser vista como uma determinada configuração do conjunto de propriedades do *player*. Por exemplo, uma âncora temporal do objeto de mídia define um intervalo associado à propriedade "time" do *player*. Desta forma, usando as propriedades e eventos de é possível construir o conceito de âncora.

A LibPlayer define dois tipos básicos: `lp_value_t` e `lp_map_t`. O primeiro define um valor genérico (`bool`, `int`, `double`, `char*` ou `void*`), usado para representar o valor das propriedades dos *players*. O último define um mapa do tipo chave-valor em que a chave é uma *string* e o valor é um `lp_value_t`. Esse mapa é usado para representar parâmetros das ações e eventos dos *players*.

A seguir apresentamos as principais funções da API de *player*. Para facilitar a apresentação, o tipo dos parâmetros e do retorno foi omito. Nas funções o parâmetro `p` indica o *player* sobre o qual a função é aplicada. A

[1] Código-fonte disponível em `http://www.telemidia.puc-rio.br/~gflima/software/libplayer`

maioria das chamadas retorna um valor `bool` que indica se a chamada foi bem-sucedida ou não.

`lp_player_new (source, mime)`
: Cria um novo *player* com conteúdo *source* e tipo *mime*, ambos *strings*. Retorna o endereço do novo *player* em caso de sucesso. Senão retorna `NULL`.

`lp_player_free (p)`
: Destrói o *player* `p`.

`lp_player_get (p, property, *value)`
: Armazena o valor da propriedade `property`, tipo *string*, em `*value`, tipo `lp_value_t*`. Retorna `true` se a propriedade está definida, caso contrário retorna `false`.

`lp_player_set (p, property, value)`
: Atribui `value`, tipo `lp_value_t`, à propriedade `property`, tipo *string*.

`lp_player_unset (p, property)`
: Remove a propriedade `property`, tipo *string*, do *player* `p`.

`lp_player_post (p, action, params)`
: Envia a ação `action`, tipo *string*, com parâmetros `params`, tipo `lp_map_t*`, para o *player* `p`. Se a ação for bem-sucedida, a função notifica as *callbacks* registradas em `p` e retorna `true`. Caso contrário, retorna `false`.

`lp_player_register (p, func)`
: Registra a *callback* `func`, cujo protótipo é o mesmo da função anterior, no *player* `p`.

`lp_player_unregister (p, func)`
: Remove a *callback* `func` da lista de funções registradas em `p`.

`lp_player_notify (p, action, params)`
: Notifica as *callbacks* do *player* `p` de que a ação `action`, tipo *string*, com parâmetros `params`, tipo `lp_map_t`, ocorreu.

Players podem conter outros *players*. Desta forma, é possível construir *players* que modificam ou controlam outros *players*. Um caso típico é o do *player screen*, que representa a janela da apresentação. Para que sejam apresentados, outros *players* devem ser adicionados ao *screen*. As funções seguintes manipulam *players* compostos.

`lp_player_add (p, child)`

Adiciona o *player* `child` à lista de filhos de `p`.

`lp_player_remove (p, child)`

Remove o *player* `child` da lista de filhos de `p`.

A interface de programação de *player*, apresentada acima, possui funções para alterar os valores de propriedades e executar ações. Porém, ela não define quais são essas propriedades e ações. Tais informações dependem da implementação de cada *player*. De fato, todo *player* define uma lista de propriedades e ações reconhecidas — i.e. as quais ele associa alguma semântica.

Para evitar inconsistências entre os *players* foram criadas classes de *players* que compartilham as mesmas propriedades e ações. Um *player* pode pertencer a mais de uma classe. Por exemplo, o *player* de vídeo pertence à classe dos *players* visuais e, ao mesmo tempo, à classe dos *players* de mídia contínua. A Tabela B.1 apresenta a lista de propriedades reconhecidas pelas classes atualmente definidas.

Classe	Propriedade	Descrição
Todos	`__name`	nome do plugin
	`__dlpath`	caminho da biblioteca
	`__mime_list`	lista de *mime-types* suportados
Visuais	`source`	conteúdo
	`x`	posição horizontal na tela
	`y`	posição vertical na tela
	`z`	índice de sobreposição
	`width`	largura em *pixels*
	`height`	altura em *pixels*
	`rotation`	rotação
	`transparency`	transparência
	`state`	"playing", "stopped", ou "paused"
	`original_width`	largura original em *pixels*
	`original_height`	altura original em *pixels*
Mídia contínua	`time`	tempo desde o último `start`

Tabela B.1 Propriedades reconhecidas por cada classe de *player*.

Atualmente, os *players* visuais reconhecem apenas as ações "start", que apresenta o conteúdo, e "stop", que pára a apresentação. Os *players* de mídia contínua reconhecem, além das anteriores, as ações "pause", que pausa a apresentação, e "seek" que avança ou retrocede o tempo da apresentação. Todos os *players* reconhecem a ação "step" que atualiza o estado global do *player*.

No caso de uma propriedade (ou ação) desconhecida, i.e. sem semântica associada, é recomendado que *player* trate-a como outra qualquer. Ou seja, armazene a propriedade ou, no caso da ação, notifique os ouvintes sobre a sua ocorrência. Este comportamento mantém a analogia com objeto de mídia NCL, em que propriedades sem semântica funcionam com variáveis do usuário.

O comportamento padrão e as listas de propriedades e ações reconhecidas são apenas guias definidos pela biblioteca. Os *players* distribuídos pela LibPlayer seguem esses padrões. Porém, cada *player* é livre para definir propriedades e ações da maneira que achar conveniente.

```
1  lp_player_t *screen;
2
3  void quit (lp_player_t * p, const char *action, lp_map_t * params) {
4    if (strcmp (action, "stop") == 0)
5      lp_player_post (screen, "stop", NULL);
6  }
7
8  int main (void) {
9    screen = lp_player_new (NULL, "application/x-libplayer-screen");
10   lp_player_t *image = lp_player_new ("sample.png", NULL);
11   lp_player_t *timer = lp_player_new (NULL,
12                         "application/x-libplayer-timer");
13   lp_player_set_d (timer, "duration", 2.0 /* seconds */ );
14   lp_player_register (timer, quit);
15
16   lp_player_add (screen, image);
17   lp_player_add (screen, timer);
18   lp_player_post (image, "start", NULL);
19   lp_player_post (timer, "start", NULL);
20
21   lp_player_post (screen, "start", NULL); /* event-loop */
22
23   exit (0);
24 }
```

Figura B.1 Exemplo de programa LibPlayer.

A Figura B.1 apresenta o código de um programa que usa a LibPlayer para exibir uma imagem durante dois segundos. Como sempre, a execução inicia pela `main`, linha 8. Na linha 9 é criado o *player* `screen`, que representa a janela da apresentação e, na linha 10, é criado o *player* `image`, que representa a imagem. Nas linhas 11–12, o *player* que conta o tempo (`timer`) é criado e inicializado com duração de dois segundos. A linha 11 registra a função `quit`, definida na linha 3, no *player* `timer`. Essa função aguarda o evento "stop" do `timer`, gerado pelo fim da duração, para terminar a apresentação (linha 5). Para que todos os *players* sejam apresentados eles precisam ser adicionados ao

`screen`, linhas 15–16, e iniciados, linhas 17–18. Finalmente, a linha 20 dispara o inicio da apresentação. Desse ponto em diante o controle é transferido para LibPlayer. Ao final de dois segundos, a chamada retorna e o programa termina (linha 22).

B.2 Plugins

Os *players* são implementados através de *plugins* em que cada *player* corresponde a um *plugin*. Um *plugin* é uma biblioteca dinâmica que implementa a API definida pela estrutura `lp_plugin_t`. Esta estrutura possui as seguintes funções.

`free (plugin)`
: Destrói o *plugin* `plugin`.

`get (plugin, property, *value)`
: Análoga à `lp_player_get` do *player*.

`set (plugin, property, value)`
: Análoga à `lp_player_set` do *player*.

`unset (plugin, property)`
: Análoga à `lp_player_unset` do *player*.

`exec (plugin, action, params)`
: Executa a ação `action`, tipo *string*, com parâmetros `params`, tipo `lp_map_t`, no *plugin* `plugin`. Retorna `true` se a ação for bem sucedida, caso contrário retorna `false`.

Além dessa API, todo *plugin* deve vir companhado de um arquivo de descrição que define o nome do *plugin*, o caminho para a biblioteca dinâmica que implementa o *plugin* e a lista de tipos *mime* suportados pelo *plugin*. O arquivo também pode conter uma lista de pares chave-valor que podem ser usados pelo *plugin* no momento da sua inicialização.

Os *plugins* são carregados pela LibPlayer no momento em que a biblioteca é inicializada. O carregamento funciona da seguinte forma. Primeiro, a biblioteca coleta os caminhos de carregamento definidos pela variável `plugin_path` do arquivo de configuração "libplayerrc". Se a variável estiver vazia é utilizado um valor padrão, definido no momento da compilação. Em seguida, para cada diretório especificado, a biblioteca procura subdiretórios contendo

o arquivo de descrição "plugin.desc". Finalmente, o *plugin* associado a cada "plugin.desc" válido é carregado.

Para facilitar a criação de *plugins* a LibPlayer fornece uma biblioteca auxiliar (Auxlib) que implementa diversas funções comuns, por exemplo, tratamento de propriedades e ações, verificação de erros, etc. Apesar de opcional, todos os *plugins* distribuídos com a LibPlayer utilizam essa biblioteca. Outra vantagem da Auxlib é que ela pode ser usada para verificar alguns dos padrões de codificação definidos anteriormente.

B.3 Loop de eventos

O *loop* de eventos controla como as ações do usuário e do ambiente afetam o resultado da apresentação. Na LibPlayer, o *loop* de eventos, contido no *player screen*, é disparado pela ação "start". Esse loop consiste, basicamente de três passos:

1. aquisição da entrada do usuário,
2. atualização do estado dos *players*, e
3. apresentação dos resultados.

Atualmente, o *loop* roda na *thread* principal da aplicação, o que significa que a chamada que dispara a apresentação só retorna quando o *loop* termina. Isso também significa que o controle da apresentação fica a cargo das *callbacks* registradas nos *players*.

```
1  state = "playing";
2  while (state != "stopped")
3    {
4      handle_input ();
5      clear (screen_surface);
6
7      for (i = 0; i < n; i++)
8        lp_player_post (players[i], "step", screen_surface);
9
10     update (screen_surface);
11     update_counters ();
12     synch (target_fps);
13   }
```

Figura B.2 *Loop* de eventos da LibPlayer.

A Figura B.2 apresenta o pseudo-código do *loop* de eventos da LibPlayer. Assim que o *screen* recebe a ação "start" ele muda o seu estado para "playing" e entra no *loop*, que é repetido enquanto o *screen* estiver tocando (linhas 1–2). O primeiro passo do *loop* é tratar os eventos de entrada, função `handle_input`, linha 4. Esta função verifica se há algum evento pendente na fila de eventos de entrada (eventos de teclado, mouse, etc). Se houver, a função retira esse evento da lista e posta sobre o próprio *screen* uma ação "input" passando como parâmetro o evento. Desta forma, as *callbacks* registradas no *screen* são notificadas e executam o código associado ao evento. Após todos os eventos terem sido tratados, a superfície da tela é preparada para um novo passo de renderização (linha 5).

O próximo passo consiste de atualizar o estado dos *players* – i.e avançá-los para a próxima amostra – contidos no *screen* (linhas 7–8). Ou seja, postar a ação "step" passando como parâmetro a superfície da tela. Desta forma, os *players* visuais podem se desenhar na janela de apresentação. Em seguida, nas linhas 10–11, a superfície da janela é atualizada para refletir as alterações e os contadores locais (tempo de apresentação, taxa de *frames*, etc.) são atualizados. Finalmente, a linha 12 chama a função `synch` que sincroniza o *loop* de acordo com a taxa de *frames* desejada. Ou seja, a função "dorme" o tempo necessário para manter a taxa de frames constante (no máximo `target_fps` *frames* por segundo).

Há diversas formas de otimizar a implementação atual do *loop*. A primeira, é controlar as operações de desenho para evitar operações de redesenho (*blits*) desnecessárias. Isso implica em retirar dos *players* a capacidade de desenhar diretamente na tela. Além disso, é possível criar uma *thread* separada para o *loop* e dessa forma permitir que a ação "start" do *screen* retorne imediatamente – o que torna a programação mais "interativa". Entretanto, para evitar problemas de concorrência, é preciso que a comunicação entre as duas *threads* seja controlada, por exemplo, através de uma fila de eventos.

B.4 Dependências

A LibPlayer (sem os *plugins*) depende apenas da biblioteca de portabilidade do TeleMídia (Tmlib)[2]. Atualmente, estão implementados os *plugins* de imagem, cronômetro (*timer*), texto e *screen*. O *screen* utiliza a LibSDL[3] para acessar os dispositivos de áudio e vídeo, e capturar a entrada do usuário. Os *plugins*

[2]Código-fonte disponível em `http://www.telemidia.puc-rio.br/~gflima/tmlib`
[3]`http://www.libsdl.org`

de imagem e texto utilizam a biblioteca Cairo[4] para manipulação de imagens e renderização de textos.

[4] http://www.cairographics.org