



UNIVERSIDADE FEDERAL DA PARAÍBA
CENTRO DE CIÊNCIAS EXATAS E DA NATUREZA
DEPARTAMENTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM INFORMÁTICA

LUATV: UMA API PARA EXTENSÃO DE APLICAÇÕES NO AMBIENTE GINGA-NCL

RAFAEL ROSSI DE MELLO BRANDÃO

JOÃO PESSOA, PB
AGOSTO DE 2010

UNIVERSIDADE FEDERAL DA PARAÍBA
CENTRO DE CIÊNCIAS EXATAS E DA NATUREZA
DEPARTAMENTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM INFORMÁTICA

**LUATV: UMA API PARA EXTENSÃO DE APLICAÇÕES
NO AMBIENTE GINGA-NCL**

RAFAEL ROSSI DE MELLO BRANDÃO

JOÃO PESSOA, PB
AGOSTO DE 2010

RAFAEL ROSSI DE MELLO BRANDÃO

LUATV: UMA API PARA EXTENSÃO DE APLICAÇÕES NO AMBIENTE GINGA-NCL

Dissertação apresentada ao Centro de Ciências Exatas e da Natureza da Universidade Federal da Paraíba, como requisito parcial para obtenção do título de mestre em informática (Sistemas de Computação).

Orientação: Prof. Dr. Guido Lemos de S. Filho

Co-orientação: Carlos Eduardo C. F. Batista

JOÃO PESSOA, PB

AGOSTO DE 2010

Ata da Sessão Pública de Defesa de Dissertação de
Mestrado do RAFAEL ROSSI DE MELLO
BRANDÃO, candidato ao Título de Mestre em
Informática na Área de Sistemas de Computação,
realizada em 03 de agosto de 2010.

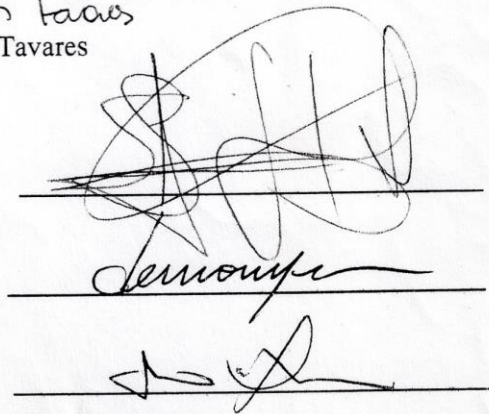
Aos três dias do mês de agosto do ano dois mil e dez, às dez horas, na Escola de Rede do Centro de Ciências Exatas e da Natureza da Universidade Federal da Paraíba, reuniram-se os membros da Banca Examinadora constituída para examinar o candidato ao grau de Mestre em Informática, na área de "*Sistemas de Computação*", na linha de pesquisa "*Computação Distribuída*", o Sr. Rafael Rossi de Mello Brandão. A comissão examinadora composta pelos professores doutores: Guido Lemos Guido Lemos de Souza Filho (DI - UFPB), Orientador e Presidente da Banca Examinadora, Dênio Mariz Timóteo de Souza (DI-UFPB), como examinador interno e Angelo Perkusich (UFCG), como examinador externo. Dando início aos trabalhos, o Prof. Guido Lemos de Souza Filho, cumprimentou os presentes, comunicou aos mesmos a finalidade da reunião e passou a palavra ao candidato para que o mesmo fizesse, oralmente, a exposição do trabalho de dissertação intitulado "*LuaTV: Uma API para extensão de aplicações no ambiente GINGA-NCL*". Concluída a exposição, o candidato foi argüido pela Banca Examinadora que emitiu o seguinte parecer: "*aprovado*". Assim sendo, deve a Universidade Federal da Paraíba expedir o respectivo diploma de Mestre em Informática na forma da lei e, para constar, o professora Tatiana Aires Tavares, Sra. Coordenadora do PPGI, lavrou a presente ata, que vai assinada por ela, e pelos membros da Banca Examinadora. João Pessoa, 03 de agosto de 2010.

Tatiana Aires Tavares
Tatiana Aires Tavares

Prof. Dr. Guido Lemos de Souza Filho
Orientador (DI-UFPB)

Prof. Dr. Dênio Mariz Timóteo de Souza
Examinador Local (DI-UFPB)

Prof. Dr. Angelo Perkusich
Examinador Externo (UFCG)



Dedico este trabalho a toda minha família, em especial meus pais Murillo e Marília e meus irmãos Marcelle e Murillo pelo carinho, atenção e amor incondicional;

À minha noiva Chrystinne, minha maior companheira, pelo apoio nos momentos mais difíceis e estressantes desta jornada.

Agradecimentos

Ao Criador, pela possibilidade de vislumbrar, ainda que de forma singela, a grandeza do universo;

Ao meu orientador e amigo, Guido, pelo apoio, lições e diversas oportunidades que me deu ao longo de quatro anos de parcerias;

Ao grande amigo Carlos Eduardo (Bidu), pela co-orientação e grande ajuda na idealização deste trabalho e diversas outras atividades;

A todos os amigos do LAViD, profissionais dedicados e incansáveis, pelos anos de convivência na batalha do dia a dia;

Aos colegas Fernando Brito e Lucenildo Júnior, pela vontade de aprender e superar as dificuldades encontradas no desenvolvimento do projeto LuaTV;

A todos que, direta ou indiretamente, contribuíram com o sucesso deste trabalho.

Resumo

O ambiente Ginga-NCL é responsável pela apresentação de aplicações declarativas (escritas na linguagem NCL) no Sistema Brasileiro de Televisão Digital (SBTVD), e é também a Recomendação ITU-T H.761 para *middlewares* IPTV. A linguagem Lua é utilizada como linguagem de *scripting* da NCL. A série de recomendações ITU-T H.760 define também dois conjuntos de API NCLua para o desenvolvimento de aplicações IPTV: as APIs *Core* e *Extended*. A API *Core* compreende a API NCLua básica, parte integrante da especificação original do ambiente Ginga-NCL no SBTVD. A API *Extended* tem como objetivo prover outras funcionalidades relevantes incorporando algumas das facilidades encontradas em ambientes imperativos. Este trabalho visa especificar um conjunto de APIs Lua denominado LuaTV, que será parte da especificação inicial para a API NCLua *Extended*.

As facilidades da API são divididas em quatro pacotes funcionais: *metadata*, com funcionalidades relacionadas ao acesso de metadados presentes em um fluxo de TV Digital; *security*, que provê mecanismos para encriptação, autenticação e integridade de dados; *multidevices*, oferecendo acesso em alto-nível a recursos comumente presentes em dispositivos remotos; e *widget*, que tem como objetivo oferecer suporte ao desenvolvimento de componentes gráficos.

Palavras-chave: TV Digital; *Middleware* Ginga; Ginga-NCL; Lua; Aplicações Interativas.

Abstract

The Ginga-NCL environment is responsible for the presentation of declarative (NCL) applications on the Brazilian Digital TV System (SBTVD), and it is also the ITU-T H.761 Recommendation for IPTV middleware. Lua is the scripting language of NCL. ITU-T H. 760 series also defines two sets of NCLua API for the development of IPTV applications: the Core and the Extended API. The Core comprises the basic NCLua API, which are part of the Ginga-NCL original specification for the Brazilian terrestrial DTV system. The Extended aims to provide other relevant functionalities incorporating some of the features commonly present on other imperative environments. This work aims to present a set of Lua APIs named LuaTV, which will be part of the draft specification for the NCLua Extended API.

LuaTV features are divided into four functional categories: metadata, with functionalities related to the accessing Digital TV metadata information; security, providing mechanisms for encrypting and authenticating data; multidevices, offering high-level access to resources commonly available on remote devices; and widget, aimed at graphical support to applications.

Keywords: Digital TV; Ginga Middleware; Ginga-NCL; Lua; Interactive Applications.

Lista de ilustrações

FIGURA 1. CENÁRIO DE TELEVISÃO DIGITAL COM TRANSMISSÃO POR DIFUSÃO, RECEPTORES E DISPOSITIVOS DE INTERAÇÃO (SOUZA FILHO, ET AL., 2007)	16
FIGURA 2. ARQUITETURA DO MIDDLEWARE GINGA (ABNT 15606-1, 2009).....	22
FIGURA 3. EXEMPLO DE TABELA DE UM EVENTO NCLUA	27
FIGURA 4. ELO NCL QUE DEPENDE DE UM NCLUA	27
FIGURA 5. ELO NCL QUE ACIONA UM NCLUA	27
FIGURA 6. UMA TABELA DE EVENTO DA CLASSE NCL DO TIPO PRESENTATION	28
FIGURA 7. EXEMPLO DE TABELA DE EVENTO DA CLASSE NCL DO TIPO ATTRIBUTION	28
FIGURA 8. EXEMPLO DE TABELA COM EVENTO DA CLASSE KEY	28
FIGURA 9. EXEMPLO DE TABELA DE EVENTO DA CLASSE USER	29
FIGURA 10. POSTAGEM DE UM EVENTO DA CLASSE TCP PARA INICIAR UMA CONEXÃO	29
FIGURA 11. EVENTO QUE ENCAPSULA O RESULTADO DA TENTATIVA DE CONEXÃO	29
FIGURA 12. ENVIO DE DADOS UTILIZANDO A CLASSE DE EVENTOS TCP	29
FIGURA 13. EVENTO DA CLASSE TCP QUE ENCAPSULA DADOS RECEBIDOS.....	30
FIGURA 14. DIAGRAMA DE CLASSES DO FRAMEWORK LUAONTV (SOUZA JÚNIOR, 2009)	33
FIGURA 15. DIAGRAMA DE CLASSES DOS COMPONENTES DA API HAVI (SCHWALB, 2003)	34
FIGURA 16. HIERARQUIA DE WIDGETS NA API LWUIT	35
FIGURA 17. DIAGRAMA DE CLASSES DA API DE INTEGRAÇÃO DE DISPOSITIVOS DO PACOTE BR.ORG.SBTVD.INTEGRATIONDEVICES.....	36
FIGURA 18. ARQUITETURA DO COMPONENTE DE MÚLTIPLOS DISPOSITIVOS NO GINGANCL	38
FIGURA 19. FUNCIONAMENTO DA CLASSE MESSAGEDIGEST	40
FIGURA 20. UTILIZAÇÃO DA CLASSE SIGNATURE NA API JAVA SECURITY	41
FIGURA 21. UTILIZAÇÃO DA CLASSE CIPHER NA API JAVA SECURITY	42
FIGURA 22. CLASSE SIDATABASE RESPONSÁVEL PELA FUNCIONALIDADE DE SI NO JAVADTV.....	46
FIGURA 23. CONTEXTO DA API LUATV	50
FIGURA 24. API LUATV NA ARQUITETURA DO MIDDLEWARE GINGA	51
FIGURA 25. API WIDGET COM OS MÓDULOS WIDGET, CURSOR E TEXTINPUT	53
FIGURA 26. EXEMPLO DE EVENTO GERADO PELA ENTRADA DO CURSOR NA REGIÃO DO NCLUA.....	55
FIGURA 27. EVENTO DE TECLA QUE ENCAPSULA MODIFICADORES	56
FIGURA 28. A API MULTIDEVICE COM OS MÓDULOS DEVICEMANAGER E DEVICESERVICE.....	57
FIGURA 29. EXEMPLO DE EVENTO GERADO PELA ENTRADA DE UM DISPOSITIVO EM UMA CLASSE REGISTRADA...	59
FIGURA 30. EXEMPLO DE EVENTO GERADO CHEGADA DE DADOS DE UM DADO SERVIÇO	59
FIGURA 31. ARQUITETURA ELABORADA PARA O COMPONENTE DEVICEINTEGRATION.....	61
FIGURA 32. API SECURITY COM OS MÓDULOS SIGNATURE, DIGEST E CYPHER	64
FIGURA 33. API METADATA COM SEU MÓDULO METADATA UTILIZANDO O MÓDULO EVENT E O COMPONENTE SERVICE INFORMATION DO NÚCLEO COMUM	68
FIGURA 34. EXEMPLO DE EVENTO GERADO PELO MÓDULO METADATA	69
FIGURA 35. WIDGET PARA EXIBIÇÃO DE MAPAS	72
FIGURA 36. WIDGET PARA ANOTAÇÕES RÁPIDAS.....	73
FIGURA 37. EXECUÇÃO DA APLICAÇÃO PÊNALTÍ INTERATIVO EM DOIS DISPOSITIVOS SECUNDÁRIOS E NO DISPOSITIVO BASE	74
FIGURA 38. CENÁRIO DE EXECUÇÃO DA APLICAÇÃO ToV	75

Lista de tabelas

TABELA 1. FUNÇÕES DO MÓDULO <i>WIDGET</i>	54
TABELA 2. FUNÇÕES DO MÓDULO <i>CURSOR</i>	55
TABELA 3. FUNÇÕES DO MÓDULO <i>TEXTINPUT</i>	56
TABELA 4. FUNÇÕES DO MÓDULO <i>DEVICEMANAGER</i>	58
TABELA 5. FUNÇÕES DO MÓDULO <i>DEVICESERVICE</i>	58
TABELA 6. FUNÇÕES DO MÓDULO <i>SIGNATURE</i>	65
TABELA 7. FUNÇÕES DO MÓDULO <i>DIGEST</i>	67
TABELA 8. FUNÇÕES DO MÓDULO <i>CYPHER</i>	67
TABELA 9. FUNÇÕES DO MÓDULO <i>METADATA</i>	69
TABELA 10. EQUIVALÊNCIA FUNCIONAL (GINGA-J x GINGA-NCL x LUATV)	70

Lista de abreviaturas e siglas

3DES	<i>Triple Data Encryption Standard</i>
3G	Terceira Geração de padrões e tecnologias de telefonia móvel
ACAP	<i>Advanced Common Application Platform</i>
AES	<i>Advanced Encryption Standard</i>
AIT	<i>Application Information Table</i>
API	<i>Application Programming Interface</i>
ARIB	<i>Association of Radio Industries and Businesses</i>
ATSC	<i>Advanced Television Systems Committee</i>
AWT	<i>Advanced Windowing Toolkit</i>
BAT	<i>Bouquet Association Table</i>
BIT	<i>Broadcaster Information Table</i>
C&A	Captura & Acesso
CBC	Comissão Brasileira de Comunicação
CC/PP	<i>Composite Capabilities/Preference Profiles</i>
CFB	<i>Cipher FeedBack</i>
CRC	<i>Cyclic Redundancy Check</i>
DAVIC	<i>Digital Audio Video Council</i>
DES	<i>Data Encryption Standard</i>
DTV	<i>Digital Television</i>
DVB	<i>Digital Video Broadcast</i>
EIT	<i>Event Information Table</i>
EPG	<i>Electronic Program Guide</i>
FUCAPI	Fundação Centro de Análise, Pesquisa e Inovação Tecnológica
GEM	<i>Globally Executable MHP</i>
GIF	<i>Graphics Interchange Format</i>
GINGA-CC	<i>Ginga Common Core</i>
GINGA-CDN	<i>Ginga Code Development Network</i>
GUI	<i>Graphical User Interface</i>
HAVi	<i>Home Audio Video Interoperability</i>
HDTV	<i>High Definition Television</i>
IETF	<i>Internet Engineering Task Force</i>
IPTV	<i>Internet Protocol Television</i>
ISDB	<i>Integrated Services Digital Broadcasting</i>
ITU	<i>International Telecommunication Union</i>
JMF	<i>Java Media Framework</i>
JNI	<i>Java Native Interface</i>
JPEG	<i>Joint Photographic Experts Group</i>
JVM	<i>Java Virtual Machine</i>
LAViD	Laboratório de Aplicações de Vídeo Digital
LDT	<i>Linked Description Table</i>
LWUIT	<i>LightWeight User Interface Toolkit</i>
MD5	<i>Message-Digest algorithm 5</i>
MHP	<i>Multimedia Home Platform</i>

MP3	<i>MPEG Audio Layer 3</i>
MPEG	<i>Moving Picture Experts Group</i>
NBIT	<i>Network Board Information Table</i>
NCL	<i>Nested Context Language</i>
NIT	<i>Network Information Table</i>
OC	<i>Object Carousel</i>
OCAP	<i>OpenCable Application Platform</i>
P2P	<i>Peer-to-Peer</i>
PCAT	<i>Partial Content Announcement Table</i>
PDA	<i>Personal Data Assistant</i>
PDF	<i>Portable Document Format</i>
PSI	<i>Program Specific Information</i>
RDF	<i>Resource Description Framework</i>
RSA	<i>Rivest, Shamir and Adleman algorithm</i>
RST	<i>Running Status Table</i>
SBTVD	<i>Sistema Brasileiro de Televisão Digital</i>
SDT	<i>Service Description Table</i>
SFTP	<i>Secure File Transfer Protocol</i>
SHA	<i>Secure Hash Algorithm</i>
SI	<i>Service Information</i>
SSH	<i>Secure Shell</i>
SSL	<i>Secure Sockets Layer</i>
ST	<i>Stuffing Table</i>
STB	<i>Set-top Box</i>
SVN	<i>Subversion</i>
TDT	<i>Time and Date Table</i>
TLS	<i>Transport Layer Security</i>
TOT	<i>Time Offset Table</i>
ToV	<i>Torcida Virtual</i>
TS	<i>Transport Stream</i>
TV	<i>Televisão</i>
TV-A	<i>TV-Anytime</i>
TVD	<i>TV Digital</i>
TVDI	<i>TV Digital Interativa</i>
TVi	<i>TV Interativa</i>
TXT	<i>Text</i>
UAProf	<i>User Agent Profile</i>
URI	<i>Uniform Resource Identifier</i>
WMA	<i>Windows Media Audio</i>
XHTML	<i>eXtensible Hypertext Markup Language</i>
XML	<i>eXtensible Markup Language</i>

Sumário

1. Introdução	15
1.1 Motivação	17
1.2 Objetivo geral	18
1.3 Objetivos específicos	18
1.4 Estrutura da dissertação	19
2. Fundamentação teórica	20
2.1 O <i>middleware</i> Ginga	20
2.2 Ginga-NCL	23
2.3 API NCLua	24
2.3.1 Módulo <i>event</i>	26
2.3.2 Módulo <i>canvas</i>	30
3. Trabalhos relacionados	32
3.1 Interface gráfica	32
3.1.1 LuaOnTV	32
3.1.2 HAVi GUI	33
3.1.3 Ginga-J (API JavaDTV)	34
3.2 Integração de dispositivos	35
3.2.1 Ginga-J (API SBTVD)	36
3.2.2 Ginga-NCL	37
3.2.3 ARIB Peripheral	39
3.3 Segurança	40
3.3.1 Java Security	40
3.3.2 Lua MD5	42
3.4 Informações sobre serviço televisivo	43
3.4.1 MHP (DVB SI)	43
3.4.2 ARIB B.23 (API ARIB SI)	45
3.4.3 Ginga-J (API JavaDTV Service)	45
3.4.4 TV-Anytime (TVA)	46
3.5 Análise comparativa das soluções	47
4. API LuaTV	50
4.1 Visão geral	50
4.2 Arquitetura	50
4.3 Pacotes funcionais	52
4.3.1 API Widgets	52

4.3.2	API Multidevice	57
4.3.2.1	Componente <i>Device Integration</i> no núcleo comum	60
4.3.3	API Security	62
4.3.4	API Metadata.....	68
4.4	Equivalência funcional	69
5.	Cenários de uso da API	71
5.1	<i>Desktop widgets</i>	71
5.2	Pênalti Interativo.....	73
5.3	Torcida Virtual (ToV).....	74
5.4	Guias Eletrônicos de Programação (EPGs)	76
5.5	Transferência segura de dados	76
6.	Conclusão	77
6.1	Resultados e contribuições.....	77
6.2	Trabalhos futuros	78
6.3	Considerações finais	79
7.	Referências bibliográficas	80
	APÊNDICE A: Especificação da API LuaTV	86
A1.	Widgets API	87
A1.1	widget object.....	87
A1.2	cursor object	89
A1.2.1	Constructors.....	89
A2.	Multidevice API	93
A2.1	devicemanager object	93
A2.2	deviceservice object.....	95
A3.1	metadata object.....	96
A4.	Security API	98
A4.1	signature object.....	98
A4.2	digest object.....	99
A4.3	cypher object.....	99
	APÊNDICE B: Especificação da API do componente Device Integration.....	101
B1.	IDeviceMonitor.....	102
B2.	DeviceMonitorListener	105
B3.	DeviceMonitorEvent.....	107

1. Introdução

Tradicionalmente, o conteúdo televisivo vinha sendo concebido com o propósito de comunicação em um só sentido (da estação emissora para telespectador) com intuito de ser exibido em um único dispositivo. Este paradigma mudou com a introdução da TV Digital Interativa (TVDI) que possibilita a transmissão de aplicações interativas junto ao fluxo audiovisual. Pode-se dizer que o conceito de TV interativa não é novo, apesar de o termo só vir a ser mais utilizado recentemente com a TV Digital, o conceito vem sendo explorado desde o início dos anos 70, quando o grupo americano Warner-Qube realizou experimentos interativos em um sistema de vídeo sob demanda (MORRIS, et al., 2005).

De forma simplificada, um sistema de TVDI é composto por: uma estação transmissora, um meio sobre o qual o sinal é propagado (ar ou meios físicos como cabo coaxial, fibra óptica, etc.) e um receptor ou *set-top-box* (STB). Dentre outras coisas, o receptor é responsável por recuperar informações e aplicações interativas enviadas junto ao fluxo, além de demultiplexar os dados de áudio e vídeo transmitidos e decodificá-los para reprodução. O receptor pode executar aplicações recebidas e ocasionalmente enviar dados de volta para a emissora através de um canal de retorno. Para a execução de uma mesma aplicação em diferentes plataformas de hardware de receptores é necessária a presença de uma camada de software intermediária entre o sistema operacional e as aplicações, chamada de *middleware*

Os principais *middlewares* com especificação aberta são o MHP (*Multimedia Home Platform*) do sistema europeu DVB (*Digital Video Broadcast*), os americanos OCAP (*OpenCable Application Platform*) e ACAP (*Advanced Common Core Application Platform*) do sistema americano ATSC (*Advanced Television Systems Committee*), e as especificações ARIB (*Association of Radio Industries and Businesses*), do sistema japonês ISDB (*Integrated Services Digital Broadcasting*).

O *middleware* especificado para o Sistema Brasileiro de Televisão Digital (SBTVD) foi denominado Ginga. Durante sua fase de especificação percebeu-se a possibilidade de estender a exibição do conteúdo televisivo para dispositivos com

capacidade de processamento e de interação, freqüentemente existentes nas residências, tais como celulares, PDAs (*Personal Data Assistants*), notebooks, etc. Desta forma, interações de um ou mais usuários poderiam acontecer paralelamente sem que a apresentação do conteúdo no televisor sofresse qualquer alteração. A Figura 1 ilustra possíveis cenários de transmissão do sinal de televisão digital do SBTVD.

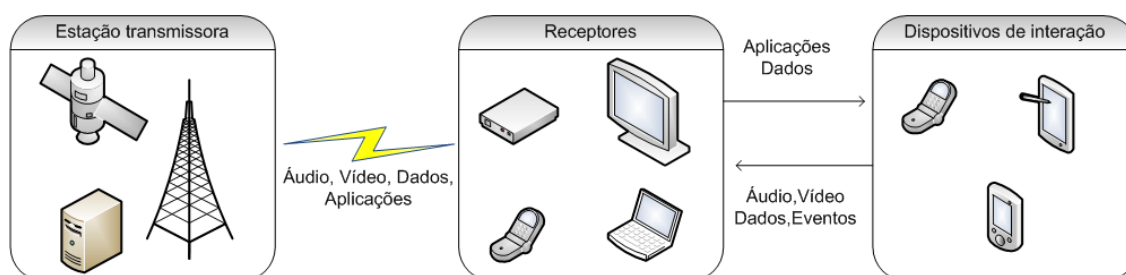


Figura 1. Cenário de televisão digital com transmissão por difusão, receptores e dispositivos de interação (SOUZA FILHO, et al., 2007)

São várias as possibilidades de aplicações no contexto televisivo em um cenário como o apresentado acima, desde simples enquetes respondidas por múltiplos usuários, até aplicações mais complexas que envolvam a captura e *streaming* de dados como, por exemplo, um chat com voz ou uma videoconferência. Este cenário se torna ainda mais interessante quando percebemos o crescente poder computacional presente nos dispositivos atualmente e a melhoria da infraestrutura de conexões com a Internet (AUN, 2007).

O SBTVD define dois ambientes em seu *middleware* para receptores fixos, um ambiente imperativo (Ginga-J) e um ambiente declarativo (Ginga-NCL). As aplicações imperativas são baseadas na plataforma Java e tem à disposição um conjunto de APIs (*Application Programming Interfaces*) direcionadas ao desenvolvimento de aplicativos para TV Digital (*Xlets*). As aplicações declarativas utilizam a linguagem NCL (*Nested Context Language*) (SOARES, et al., 2006), uma linguagem de marcação baseada em XML (*eXtensible Markup Language*). A NCL define dois perfis: um avançado, o *Enhanced Language Profile*, e outro mais básico, o *Basic Language Profile* utilizado por receptores móveis. Assim como outras linguagens de marcação, a NCL permite a utilização de uma linguagem de *script* para realizar computações de natureza imperativa (ABNT 15606-5, 2009). A linguagem de *script* especificada para NCL é a linguagem Lua (IERUSALIMSKY,

2006), sobre a qual foi idealizada a API proposta neste trabalho.

1.1 Motivação

Uma das principais motivações para a realização do presente trabalho é o fato de que no SBTVD o único ambiente obrigatório para o perfil de receptores móveis (também chamado de perfil *one-seg*) é o Ginga-NCL (ABNT 15606-5, 2009). Desta forma, aplicações interativas a serem executadas em dispositivos móveis devem ser desenvolvidas apenas em linguagem NCL com *scripts* Lua. Contudo, certas funcionalidades disponíveis para o ambiente imperativo Ginga-J não estão presentes para o desenvolvedor de aplicações NCLua.

Com a padronização do ambiente Ginga-NCL pela ITU (*International Telecommunication Union*) para ambientes IPTV sob recomendação internacional H.761 (ITU, 2009), surgiu a necessidade de complementar a API NCLua com funcionalidades originalmente oferecidas apenas pelo ambiente Ginga-J. A ITU estabelece através de suas recomendações um ambiente padronizado para IPTV com objetivo de harmonizar os diversos sistemas disponíveis. A recomendação H.761 (*Nested context language (NCL) and Ginga-NCL for IPTV services*) (ITU, 2009) define como um dos requisitos de padronização a utilização da linguagem Lua em ambientes IPTV, utilizando as APIs definidas para o Ginga-NCL (ABNT 15606-2, 2009). A API da linguagem Lua, porém, foi revisada e dois conjuntos de APIs foram definidos: *Core* e *Extended*, onde o primeiro contempla praticamente todas as APIs originais do Ginga-NCL e o segundo novas classes de eventos NCL e novos módulos. O grupo de trabalho GRN-6 (Grupo Relator de Normatização), intermediado pela Anatel, é responsável pela definição da API *Extended* Lua (que fará parte da especificação atualmente designada por H.IPTV.MAFR.14). Este grupo faz parte da CBC-3 (Comissão Brasileira de Comunicação), e conta com colaboradores do Laboratório Telemídia da PUC-Rio, do Laboratório de Aplicações de Vídeo Digital (LAViD) da Universidade Federal da Paraíba (UFPB), da Fundação Centro de Análise, Pesquisa e Inovação Tecnológica (FUCAPI) além da própria Anatel. O presente trabalho é parte das contribuições deste grupo a serem analisadas para a padronização da linguagem.

A padronização internacional do Ginga-NCL foi um passo importante para que

o ambiente possa ser adotado isoladamente por outros sistemas, o que de fato já é a opção escolhida por alguns países (destacando-se a Argentina¹).

1.2 Objetivo geral

O objetivo geral deste trabalho é expandir o escopo de desenvolvimento de aplicações NCLua no *middleware* Ginga através da especificação de uma nova API em Lua. Funcionalidades imperativas para o âmbito da TV Digital que não são oferecidas para o desenvolvedor na atual especificação Ginga-NCL (em conjunto com sua linguagem de *script* Lua) são o foco desta API.

1.3 Objetivos específicos

O objetivo principal do trabalho proposto é a especificação e validação da API LuaTV. Uma especificação inicial da API é proposta neste trabalho e conta com quatro pacotes funcionais, descritos brevemente abaixo.

- Widget: Um pacote de API para criação de elementos gráficos para estender as possibilidades em aplicações NCLua que lidam com a interface gráfica e entrada de dados do usuário;
- Multidevice: API que integra múltiplos dispositivos em uma rede doméstica para utilização de seus recursos possibilitando, por exemplo, reprodução ou captura de mídias e interação do usuário através de teclado ou cursor;
- Metadata: Módulo para recuperação de informações acerca dos serviços televisivos presentes no fluxo transmitido, tais como informações sobre os fluxos elementares, horário da programação, sinopses, informações sobre a classificação indicativa etc.;
- Security: Provê facilidades relacionadas à encriptação e segurança na transmissão de dados (requisitos de certas aplicações como T-Bank (SILVA, et al., 2007), votações, etc.).

Além da especificação da API LuaTV que será enviada como contribuição para a padronização da linguagem Lua em âmbito IPTV pela ITU-T (setor normativo

¹ <http://www.ginga.org.ar>

da ITU), os seguintes outros objetivos específicos foram definidos para a obtenção do objetivo geral:

- Estudo aprofundado sobre a comunicação entre Lua e NCL e implementações existentes;
- Refinamento da API LuaTV especificada inicialmente;
- Definição da versão final da arquitetura LuaTV (integração com a arquitetura do *middleware* Ginga);
- Refinamento da arquitetura e especificação da API do componente para integração de múltiplos dispositivos a ser utilizado pela API Multidevice e outras APIs do *middleware* Ginga;
- Elaboração de casos de uso com cenários e aplicações para TV Digital que se utilizem da API LuaTV;
- Implementação de referência da API para validação e testes.

1.4 Estrutura da dissertação

Esta dissertação está estruturada em cinco capítulos. Após a presente introdução, o capítulo 2 traz a fundamentação teórica que contextualiza este trabalho, onde será detalhado o *middleware* Ginga e suas APIs para desenvolvimento de aplicações interativas. O capítulo 3 apresenta alguns trabalhos relacionados a este trabalho comparando suas funcionalidades, características e limitações. O capítulo 4 apresenta a API LuaTV e sua arquitetura integrada ao *middleware* Ginga. O capítulo 5 traz cenários inovadores de aplicações possíveis com as funcionalidades da API. Por fim, no capítulo 6 tem-se as considerações finais, os resultados obtidos e a conclusão do trabalho proposto.

2. Fundamentação teórica

Este capítulo apresenta o contexto no qual a API proposta se encontra: o *middleware* Ginga, seus ambientes Ginga-J e Ginga-NCL e a API NCLua que é utilizada por *scripts* Lua. Tais conceitos são fundamentais para a compreensão do trabalho ora descrito, visto que esse se propõe a estender tal plataforma.

2.1 O *middleware* Ginga

O Ginga é a especificação de *middleware* do SBTVD que resultou da fusão dos *middlewares* FlexTV e MAESTRO, desenvolvidos por projetos liderados pela UFPB e PUC-Rio, respectivamente. O FlexTV (LEITE, et al., 2005), *middleware* imperativo de referência do SBTVD, incluía um conjunto de APIs Java estabelecidas internacionalmente juntamente com funcionalidades inovadoras, como a possibilidade de gerenciamento de conexões com múltiplos dispositivos, permitindo que os mesmos fossem utilizados para interagir simultaneamente como uma única aplicação. Já o MAESTRO (SOARES, 2006), o *middleware* declarativo, era responsável por processar documentos escritos em linguagem NCL juntamente com *scripts* Lua, tendo como principal facilidade o sincronismo espaço-temporal entre objetos multimídia.

O Ginga integrou os *middlewares* FlexTV e MAESTRO redefinindo-os em seus ambientes Ginga-J (SOUZA FILHO, et al., 2007) (sua máquina de execução) e Ginga-NCL (SOARES, et al., 2007) (sua máquina de apresentação), tomando por base as recomendações internacionais da ITU, J.200 (ITU, 2001) (ITU, 2004) e J.202 (ITU, 2003). A recomendação J.200 contém definições com propósito de harmonização para um núcleo comum (*Common Core*) a ser utilizado pelos dois ambientes, enquanto as recomendações J.201 e J.202 contém recomendações para harmonização do formato de conteúdo declarativo e procedural para aplicações TVDI.

A especificação inicial do ambiente Ginga-J contava com um pacote de APIs baseado no GEM (*Globally Executed MHP*) que incluía as APIs JavaTV², DAVIC³

² <http://java.sun.com/javame/reference/apis/jsr927/>

³ <http://www.davic.org>

(*Digital Audio Visual Council*), HAVi⁴ (*Home Audio Video Interoperability*) e DVB⁵. A utilização de um modelo baseado no GEM objetivava a interoperabilidade de aplicações desenvolvidas para o Ginga com outros padrões de *middleware*. Contudo, questões relacionadas ao modelo de pagamento de *royalties* envolvendo as APIs GEM levantaram a discussão no Fórum SBTVD⁶ se tal solução era realmente a mais adequada ao sistema brasileiro. Em uma parceria com a Sun Microsystems, o Fórum SBTVD trabalhou na especificação de um conjunto de APIs livre de *royalties* com funcionalidades equivalentes as do GEM, a API JavaDTV. Além da JavaDTV, também estão presentes na nova especificação do Ginga-J a API JMF 2.1 (*Java Media Framework*), parte das APIs ARIB B.23 e um conjunto de APIs específico do SBTVD. Com a adoção da API JavaDTV, a interoperabilidade foi postergada e dependerá de futuras adoções da API por outros sistemas televisivos.

A Figura 2 apresenta a arquitetura do *middleware* Ginga com seus ambientes declarativo e imperativo. Seguindo as recomendações da ITU, os ambientes utilizam um núcleo comum que se comunica com o sistema operacional. A máquina de apresentação utiliza a camada de serviços específicos Ginga para ter acesso ao núcleo comum enquanto a máquina de execução utiliza a JVM (*Java Virtual Machine*). Existe ainda uma ponte de comunicação (também especificada nas recomendações da ITU) que possibilita integração entre os ambientes. Essa API de ponte permite que aplicações declarativas utilizem serviços disponíveis para aplicações imperativas e vice-versa.

⁴ <http://www.havi.org>

⁵ <http://www.dvb.org/technology/standards/>

⁶ <http://www.forumsbtvd.org.br>

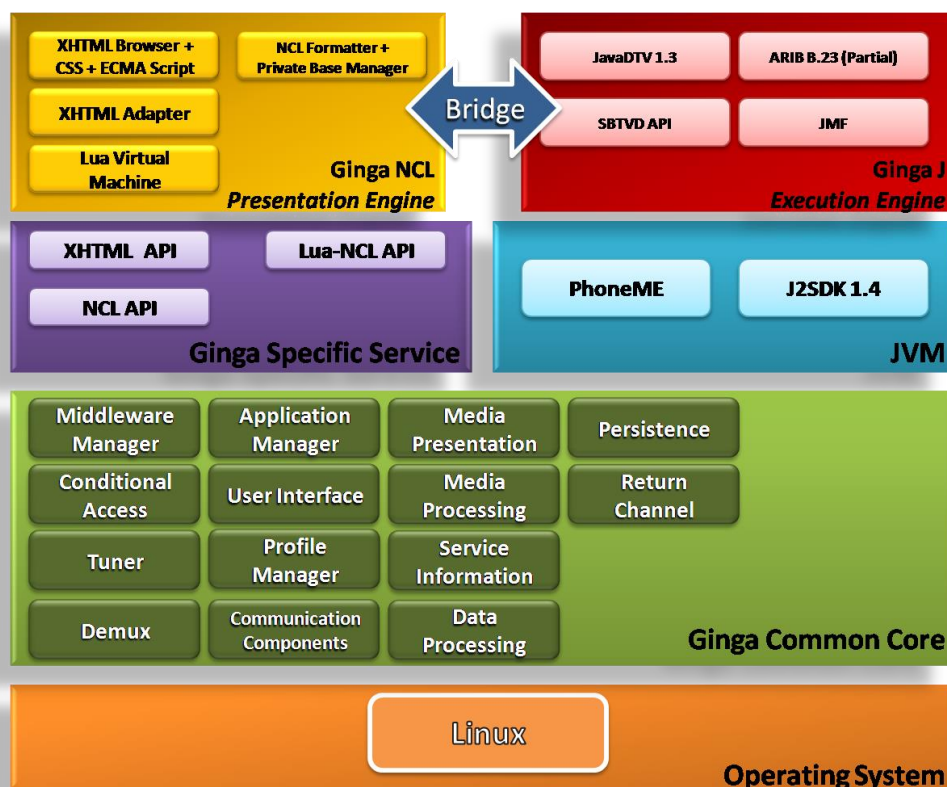


Figura 2. Arquitetura do *middleware* Ginga (ABNT 15606-1, 2009)

Da arquitetura do *middleware* Ginga destacam-se os seguintes componentes relevantes para a API LuaTV: o formatador NCL (*NCL Formatter*) na máquina de apresentação, componente-chave para a reprodução de documentos NCL; a API NCLua na camada de serviços específicos utilizada por *scripts* Lua presentes em documentos NCL; e os componentes do núcleo comum que podem ser utilizados pelos dois ambientes. Esta arquitetura é apresentada como sugestão de implementação em (ABNT 15606-1, 2009) e não possui caráter normativo, arquiteturas alternativas podem contemplar os requisitos e capacidades do *middleware* sem necessariamente seguir o modelo sugerido.

A implementação de referência de código-aberto OpenGinga⁷ que vem sendo desenvolvida pelo laboratório LAVID⁸ utiliza a arquitetura de implementação sugerida. Em 2009, o projeto GingaCDN⁹ (*Ginga Code Development Network*) iniciou-se com o propósito de desenvolvimento distribuído e colaborativo de *software*, com isso o OpenGinga passou a adotar o modelo de componentes FlexCM (FREIRE FILHO, 2008) o que possibilitou o desenvolvimento modular através de um

⁷ <http://www.openginga.org>

⁸ <http://www.lavid.ufpb.br>

⁹ <http://gingacdnlavid.ufpb.br>

esquema de definição de interfaces e conexões de componentes. Componentes do núcleo comum do OpenGinga podem ser implementados isoladamente e posteriormente conectados a outros componentes através do ambiente de execução do FlexCM.

2.2 Ginga-NCL

Como mencionado anteriormente, a linguagem NCL (SOARES, et al., 2007) é baseada na linguagem XML, porém o foco da linguagem declarativa NCL é mais amplo do que o oferecido por linguagens de propósito similar, como XHTML (XHTML, 2010) e SMIL (SMIL, 2010). A sincronização espaço-temporal, definida genericamente pelos *links* NCL; adaptabilidade, definida pelos elementos *switch* e *descriptor switch* da NCL; e suporte a múltiplos dispositivos de exibição, definidos por regiões NCL, são funcionalidades chave dessa linguagem declarativa (ABNT 15606-2, 2009).

Um documento NCL define como os objetos de mídia são estruturados e relacionados no tempo e espaço. Como uma linguagem de cola, ela não restringe ou prescreve os tipos de conteúdo dos objetos de mídia. Nesse sentido, podem-se ter objetos de imagem (como GIF e JPEG), de vídeo (MPEG, MOV, por exemplo), de áudio (MP3, WMA, etc.), de texto (TXT, PDF), de aplicações imperativas, entre outros, inclusive objetos de mídia NCL. O suporte a formatos de mídia pelo ambiente de execução NCL depende dos exibidores de mídia que estão acoplados ao formatador NCL (exibidor NCL). Um objeto de mídia NCL que deve obrigatoriamente ser suportado é o objeto de mídia baseado em XHTML. A NCL não substitui, mas embute documentos (ou objetos) baseados em XHTML (ABNT 15606-2, 2009).

Objetos imperativos podem ser inseridos em documentos NCL para trazer capacidades computacionais adicionais aos documentos declarativos. O Ginga-NCL permite que dois tipos de mídia com aplicações imperativas sejam associados a um documento NCL: *scripts* imperativos Lua e aplicações imperativas Java (Xlets), estes nós de mídia imperativos são chamados de NCLua e NCLet respectivamente. A Listagem 1 mostra o código de uma aplicação NCL simplificada que utiliza uma mídia imperativa NCLua.

```

<?xml version="1.0" encoding="ISO-8859-1"?>
<ncl id="nclMaps" xmlns="http://www.ncl.org.br/NCL3.0/EDTUPProfile">

<head>
  <regionBase>
    <region width="50%" height="50%" left="0%" top="0%" id="rgLua"/>
  </regionBase>

  <descriptorBase>
    <descriptor id="dsLua" region="rgLua" focusIndex="luaIdx"/>
  </descriptorBase>
</head>

<body>
  <port id="entryPoint" component="lua"/>

  <media id="lua" src="maps.lua" descriptor="dsLua">
    <area id="fim"/>
  </media>
</body>

</ncl>

```

Listagem 1. Exemplo de documento NCL que insere um objeto Lua com mídia imperativa

O código começa com a definição das URIs (*Uniform Resource Identifier*) dos esquemas da NCL através de atributos do elemento *ncl*. Em seguida são declarados o cabeçalho e o corpo do programa. No cabeçalho, são definidas as regiões na tela onde as mídias serão apresentadas (elementos *region*) e também a forma que estas mídias serão apresentadas (elementos *descriptor*), uma ou mais regiões e descritores podem ser definidos dentro dos elementos *regionBase* e *descriptorBase* respectivamente. No corpo do documento NCL são definidas as mídias e a estrutura do programa em si. O elemento *port* indica em qual componente o documento inicia sua apresentação. Mídias podem ser declaradas através do elemento *media* que define o local dos arquivos da mídia e a associa com seus descritores. Adicionalmente, conectores ou elos podem também ser declarados no corpo para sincronização da apresentação das mídias através do elemento *link*.

2.3 API NCLua

O ciclo de vida de um objeto NCLua no *middleware* Ginga é controlado pelo formatador NCL. O formatador é responsável por iniciar a execução de um objeto NCLua e por mediar a comunicação entre esse objeto e outros objetos em um

documento NCL (ABNT 15606-5, 2009).

Depois da iniciação, a execução do objeto NCLua torna-se orientada a eventos, em ambas as direções. Isto é, qualquer ação comandada pelo formatador NCL é dirigida aos tratadores de evento registrados e qualquer notificação de mudança de estado de eventos NCL é enviada como um evento ao formatador NCL (como, por exemplo, o fim da execução do código imperativo) (ABNT 15606-5, 2009).

Além da biblioteca padrão de Lua, os seguintes módulos estão presentes na API NCLua do *middleware* Ginga:

- Módulo *canvas*: oferece uma API para desenhar primitivas gráficas e manipular imagens;
- Módulo *event*: permite que aplicações NCLua comuniquem-se com o *middleware* através de eventos (classe de eventos *ncl* e *key*) e com aplicações remotas (classe de eventos *tcp*);
- Módulo *settings*: exporta uma tabela com variáveis definidas pelo autor do documento NCL e variáveis de ambiente reservadas em um nó de mídia "application/x-ginga-settings";
- Módulo *persistent*: exporta uma tabela com variáveis persistentes, que estão disponíveis para manipulação apenas por objetos imperativos.

Uma característica apresentada nestes módulos é o de serem modelados de forma minimalista. Eles não tentam definir um modelo de programação ou esgotar em APIs os possíveis tipos de uso (SANT'ANNA, et al., 2008). O que eles oferecem de maneira geral são acessos a primitivas disponíveis em qualquer sistema computacional. Algumas vantagens dessa abordagem levantadas por (SANT'ANNA, et al., 2008) são:

- Define uma forma flexível de desenvolvimento de aplicações.
- A definição é pequena, sucinta e de fácil entendimento.
- Atualmente são aproximadamente 30 funções ou tipos.
- A implementação é bastante óbvia, com um mapeamento próximo de 1:1 com o sistema de implementação do *middleware*, portanto pequena e com pouca

margem a comportamentos não padronizados.

- Fomenta o desenvolvimento de *frameworks*, *game engines*, etc, portáveis e com utilidades diferentes sobre a mesma API.
- Por ter uma interface mais baixo-nível permite o desenvolvimento de novos *players* e até mesmo aplicações nativas portáveis entre receptores.

Os módulos *event* e *canvas* desempenham um papel crucial para o desenvolvimento da API tema deste trabalho e, portanto serão detalhados aqui.

2.3.1 Módulo *event*

O modelo de execução de scripts NCLua é orientado a eventos, de fato, *scripts* Lua no *middleware* Ginga são tratadores de eventos tais como: eventos de sincronia em documentos NCL, eventos de teclas do usuário, eventos de notificação de dados no canal de retorno, notificação de chegada de informações sobre serviços, entre outros. O módulo *event* fornece as facilidades para que um *script* NCLua possa tratar estes eventos. As seguintes funções estão definidas no módulo:

- **event.register([pos,] f):** Registra a função passada como um tratador de eventos, isto é, sempre que ocorrer um evento, *f* será chamada. A função passada deve receber como parâmetro uma tabela que irá encapsular os dados do evento.
- **event.unregister(f):** Remove a função passada da lista de tratadores de eventos, isto é, novos eventos não serão mais passados a *f*.
- **event.post([dst,] evt):** Posta o evento passado.
- **event.timer(time, f):** Cria um temporizador que expira após *time* milissegundos e então chama a função *f*.
- **event.uptime():** Retorna o número de milissegundos decorridos desde o início da aplicação.

Um evento é representado através de uma tabela Lua, sendo o campo *class* o

único obrigatório. Existem quatro classes de eventos: *ncl*, *key*, *user* e *tcp*. Um exemplo de evento válido é exibido na Figura 3:

```
{class = 'tcp' , type = 'data', value = 'dados',  
connection = 13 }
```

Figura 3. Exemplo de tabela de um evento NCLua

Na linguagem NCL, nós de mídia podem relacionar-se através de um esquema de condições e ações, estes relacionamentos são descritos através de elos. Elos podem acionar ou depender de um *script* NCLua, elos que acionam o *scripts* NCLua podem estar vinculados, por exemplo, à exibição de outras mídias e ao fim da execução destas, serem acionados através do recebimento de um evento no tratador registrado. Já os elos que dependem de um NCLua ficam à espera de um evento que chegará no momento em que o *script* postar o evento relacionado (função *event.post*). Exemplos de elos na linguagem NCL que dependem e acionam um NCLua são mostrados nas Figuras 4 e 5. No primeiro, o elo reproduzirá um áudio ao fim da execução de um script NCLua. O segundo elo inicia um NCLua no início da reprodução de um vídeo.

```
<link xconnector = "onBeginStart">  
  <bind role="onEnd" component="luaId" / >  
  <bind role="start" component="audioId"/ >  
</link>
```

Figura 4. Elo NCL que depende de um NCLua

```
<link xconnector = "onBeginStart">  
  <bind role = "onBegin"  
component="videoId"/ >  
  <bind role = "start"  
component="luaId"/ >  
</link>
```

Figura 5. Elo NCL que aciona um NCLua

Através da classe de eventos *ncl* é possível a comunicação de um *script* NCLua com o documento NCL em que ele está inserido. Existem dois tipos de eventos na classe *ncl*, eventos de apresentação e eventos de atribuição. O tipo do evento é

determinado pelo campo *type* que pode ser '*presentation*' ou '*attribution*'.

Eventos do tipo *presentation* são utilizados para controlar a exibição do nó NCLua, eles podem estar associados a áreas específicas (chamadas de âncoras de apresentação) ou ao nó como um todo. O campo *area* identifica a âncora em questão, quando está ausente (valor *nil*) representa todo o nó. O evento carrega ainda a ação a ser tomada com relação à apresentação do NCLua através do campo *action*, podendo assumir os valores: '*start*', '*stop*', '*abort*', '*pause*' e '*resume*'. A Figura 6 mostra um exemplo evento do tipo *presentation* da classe *ncl*.

```
{ class = 'ncl' , type =  
'presentation', action = 'start' }
```

Figura 6. Uma tabela de evento da classe *ncl* do tipo *presentation*

Os eventos do tipo *attribution* são utilizados para controlar as propriedades de um nó NCLua. Através do campo *property* é definida qual propriedade em questão está sendo controlada, enquanto o campo *value* define o valor da propriedade. O campo *action* também está presente em eventos de atribuição e pode assumir os mesmos valores citados. A Figura 7 ilustra um evento do tipo *attribution* da classe *ncl*.

```
{ class = 'ncl' , type =  
'attribution', property = 'myProp'  
, action = 'start' , value = '10' }
```

Figura 7. Exemplo de tabela de evento da classe *ncl* do tipo *attribution*

A classe de eventos *key* é utilizada para receber notificações do uso do controle remoto pelo usuário. Eventos desta classe não devem ser gerados, devem ser somente recebidos já que o controle remoto é um dispositivo somente de entrada. A Figura 8 exibe um exemplo de evento da classe *key*.

```
{ class = 'key' , type = 'press',  
key = 'R' }
```

Figura 8. Exemplo de tabela com evento da classe *key*

Um evento da classe *user* possui um único campo obrigatório, o campo *class*, de forma que uma aplicação pode se utilizar desta classe para uso interno, definindo os campos de seu interesse, como mostrado na Figura 9.

```
{class = 'user' , myfield = 'mydata' }
```

Figura 9. Exemplo de tabela de evento da classe user

A classe de eventos *tcp* oferece facilidades para envio e recebimento de dados pelo canal de retorno. Para iniciar uma conexão um evento como o mostrado na Figura 10 deve ser postado. O resultado desta conexão será retornado como um evento para um tratador registrado. A estrutura deste evento é mostrada na Figura 11.

```
event.post {  
    class = 'tcp',  
    type = 'connect',  
    host = '192.168.0.1',  
    port = '80',  
    [timeout = <number>]  
}
```

Figura 10. Postagem de um evento da classe *tcp* para iniciar uma conexão

```
{class = 'tcp', type = 'connect',  
host = '192.168.0.1' , port = '80',  
connection = '23', error = 'unable to  
connect' }
```

Figura 11. Evento que encapsula o resultado da tentativa de conexão

Os campos *connection* e *error* identificam o resultado da tentativa de conexão, caso ela tenha sido bem sucedida o campo *connection* trará um identificador único para esta conexão, caso resulte em erro o campo *error* conterá a mensagem associada. Os campos *connection* e *error* são mutuamente exclusivos. Como mostrado nas Figuras 12 e 13, um script NCLua envia e recebe dados de maneira similar ao início de uma conexão.

```
event.post {  
    class = 'tcp',  
    type = 'data',  
    value = 'GET /image.png  
HTTP/1.1',  
    connection = 212,  
    timeout = 120  
}
```

Figura 12. Envio de dados utilizando a classe de eventos *tcp*

```
{class = 'tcp', type = 'data', value =  
'someData', connection = '23', error =  
'403 Forbidden'}
```

Figura 13. Evento da classe *tcp* que encapsula dados recebidos

2.3.2 Módulo *canvas*

Um NCLua realiza todas as operações gráficas através do módulo *canvas*, que disponibiliza um conjunto de funcionalidades para desenho de primitivas gráficas. Uma variável global *canvas* é instanciada sempre que um NCLua é iniciado, este objeto representa a região associada ao nó de mídia NCLua. As seguintes funções estão definidas no módulo *canvas*:

- **canvas:new(...):** A partir do objeto *canvas* é possível criar novos objetos gráficos e combiná-los através de operações de composição.
- **canvas:attrSize():** Retorna as dimensões do *canvas*.
- **canvas:attrColor(...):** Acessa o atributo de cor do *canvas*.
- **canvas:attrClip(...):** Acessa o atributo que limita a área do *canvas* para desenho.
- **canvas:attrCrop(...):** Acessa o atributo de recorte do *canvas*.
- **canvas:attrFont(...):** Acessa o atributo de fonte do *canvas*.
- **canvas:drawLine(x1, y1, x2, y2):** Desenha uma linha com extremidades em (x1,y1) e (x2,y2).
- **canvas:drawRect(mode, x, y, width, height):** Desenha um retângulo no *canvas*.
- **canvas:drawText(x, y, text):** Desenha o texto passado na posição (x,y) do

canvas.

- **canvas:measureText(text):** Retorna as dimensões do texto passado.
- **canvas:compose(x, y, canvas, src_x, src_y, src_width, src_height):** Faz a composição pixel a pixel entre dois *canvas*.
- **canvas:flush():** Atualiza o *canvas* após operações de desenho e de composição.

A notação **module:attrFunction(...)** empregada denota que existe um par de funções para leitura e escrita com o mesmo nome, porém com assinaturas diferentes. Por exemplo, o método *canvas:attrColor(...)* possui as seguintes assinaturas:

- *canvas:attrColor(R<number>, G <number>, B <number>, A <number>)*
- *canvas:attrColor() --> R, G, B, A*

3. Trabalhos relacionados

Este capítulo apresenta alguns trabalhos correlatos às diferentes funcionalidades da API proposta. Cada uma das seções abaixo discute trabalhos que implementam soluções relacionadas aos pacotes funcionais da API LuaTV: interface gráfica, integração de dispositivos, segurança e recuperação de metadados de serviços televisivos. Uma comparação entre as soluções abordadas é apresentada na seção final deste capítulo.

3.1 Interface gráfica

Os trabalhos relacionados à interface gráfica aqui discutidos limitam-se àqueles que implementam suas funcionalidades no âmbito de TV Digital uma vez que a API proposta está inserida neste contexto. Foram consideradas as propostas do LuaOnTV, as especificações HAVi, (utilizadas por *middlewares* imperativos baseados no GEM), além da API gráfica utilizada pelo *middleware* Ginga-J.

3.1.1 LuaOnTV

O LuaOnTV (LUAONTV, 2008) é um *framework* para utilização de componentes gráficos para entrada e saída de dados em aplicações interativas, implementado sob paradigma da programação orientada a objetos. Este *framework* foi desenvolvido no laboratório de TV Digital do ENE/UnB sob licença GPL com o propósito de facilitar a criação de aplicativos de tal sorte que o autor de documentos NCL fique responsável apenas pelo sincronismo de mídias (SOUZA JÚNIOR, 2009). A Figura 14 exibe o diagrama de classes presentes no *framework*.

Os componentes do *framework* podem ser divididos em componentes visuais e não-visuais. Os componentes não-visuais são: *EventManager* (para controle de eventos de foco), *Graphics* (para comunicação com a API NCLua), o repositório base *Pane* (container dos componentes gráficos) e uma super-classe para todos os componentes gráficos, o *DefaultComponent*. Os componentes visuais são: *Label*, *TextArea*, *NumberField*, *VirtualKeyboard*, *Button*, *List*, *ComboBox*, *ScrollBar*, *CheckBox*, *CheckBoxGroup*, *RadioButton* e *RadioButtonGroup*.

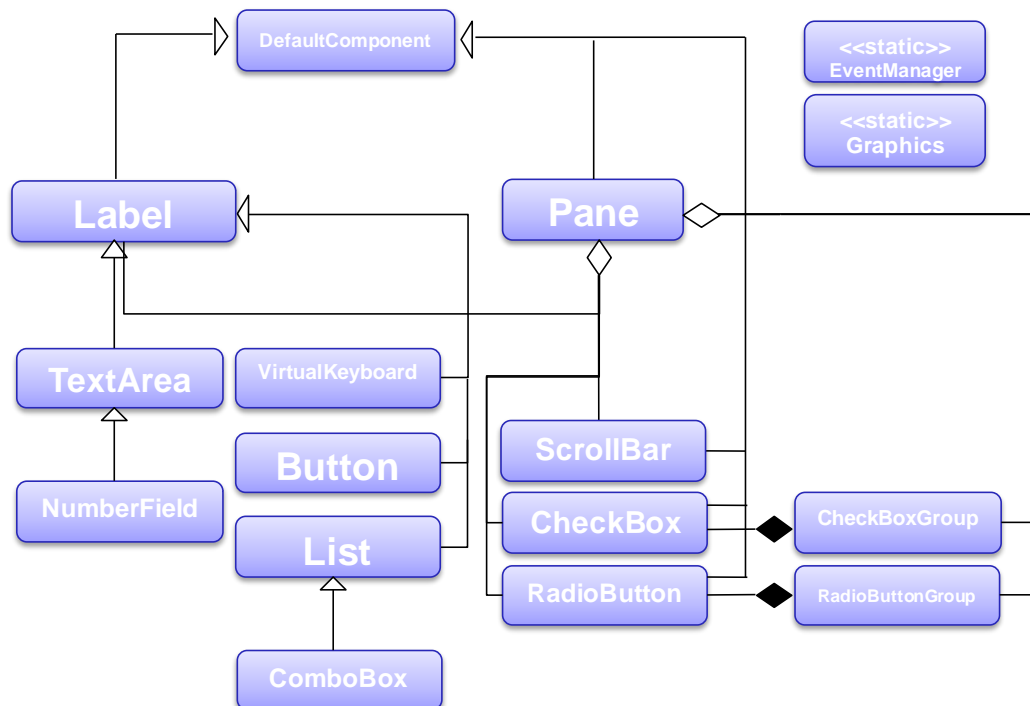


Figura 14. Diagrama de classes do *framework* LuaOnTV (SOUZA JÚNIOR, 2009)

3.1.2 HAVi GUI

O padrão HAVi define um conjunto de APIs Java voltado para interfaces gráficas conhecido como HAVi Level 2 GUI que compreende um conjunto de *widgets* que não necessitam de um sistema de janelas e um conjunto de classes para gerência de recursos escassos para possibilitar que aplicações compartilhem a tela sem um gerenciador de janelas.

O pacote `org.havi.ui` provê funcionalidades equivalentes ao *toolkit* gráfico Java AWT (*Advanced Windowing Toolkit*)¹⁰, API de elementos gráficos da linguagem Java que é voltada para desenvolvimento em PC. A API HAVi GUI faz uso de aspectos do pacote `java.awt` que não são dependentes de plataforma estendendo a especificação adicionando suporte através de controle remoto e gerenciamento e recuperação de informações do sistemas gráfico.

A API inclui *widgets* como botões, *check-boxes*, *radio buttons*, campos de entrada de texto, textos estaticos, ícones, caixas de diálogo e animações, são disponibilizadas também uma classe container (`HContainer`) e uma super-classe de componentes (`HComponent`) equivalentes ao *toolkit* AWT. A classe `HScreen`

¹⁰ <http://java.sun.com/j2se/1.4.2/docs/api/java/awt/package-summary.html>

juntamente com a classe HScreenDevice abstraem os planos gráficos existentes (ex., plano de *background*, plano de vídeo, plano de gráficos), com isso é possível configurar separadamente cada camada com o formato do pixel, resolução, tamanho da tela e proporções.

A API conta também com a classe HScene (semelhante a classe `java.awt.Window`) que estende a classe HContainer e oferece facilidades para a exibição dos componentes adicionados. Essencialmente, a classe HScene funciona como um mecanismo de requisição de certa área da tela. A Figura 15 exibe o diagrama de classes com a hierarquia dos componentes da API HAVi.

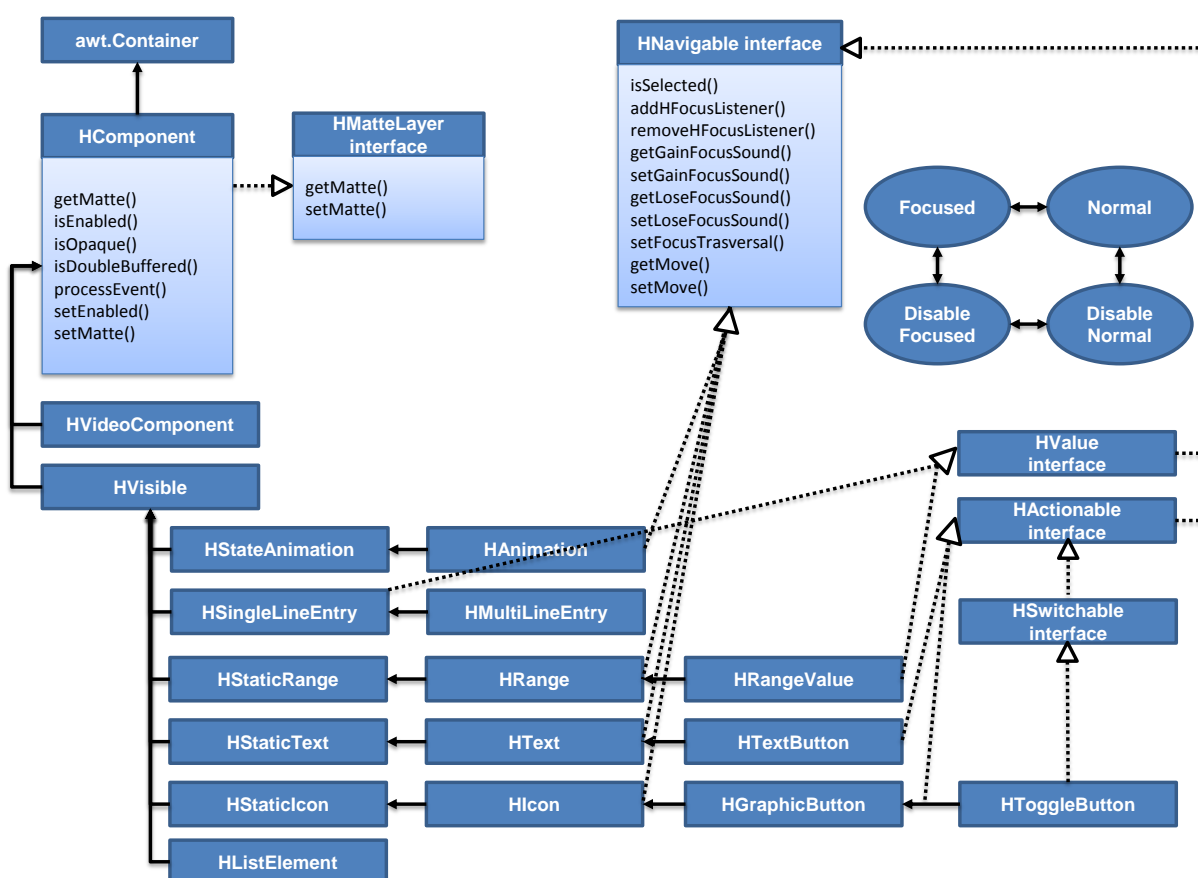


Figura 15. Diagrama de classes dos componentes da API HAVi (SCHWALB, 2003)

3.1.3 Ginga-J (API JavaDTV)

A API JavaDTV conta com dois pacotes relacionados a interface gráfica, a API de *widgets* LWUIT (*Light-Weight User Interface Toolkit*) e a API DTV UI. O pacote `com.sun.dtv.lwuit` se baseia no esquema de composição de componentes e containers com terminologia e design similar ao toolkit AWT/Swing. Já o pacote

com.sun.dtv.ui dá suporte a funcionalidades específicas de TV.

Na API LWUIT, componentes podem ser inseridos em containers com gerentes de layout associados para determinar seu posicionamento. Os containers podem ser aninhados em vários níveis assim como no Swing/AWT. Todos os componentes são desenhados por um gerente (classe UIManager) o que possibilita a utilização de temas e estilos sendo possível uma customização mais elaborada através da implementação de uma classe LookAndFeel que sobrescreva métodos para desenho e redimensionamento dos componentes. A Figura 16 exibe a hierarquia de classes dos *widgets* na API LWUIT.

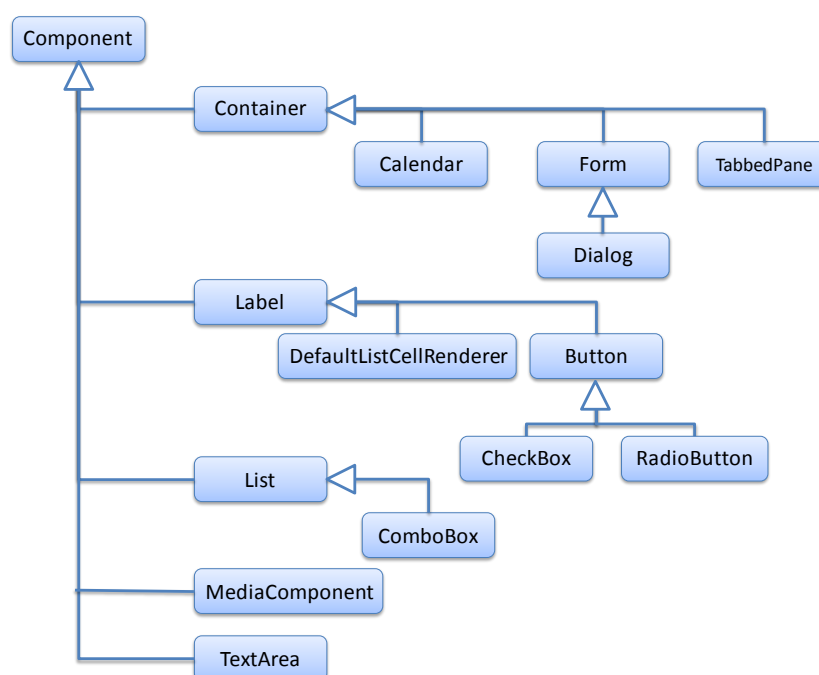


Figura 16. Hierarquia de *widgets* na API LWUIT

A API DTV UI oferece, entre outras coisas, facilidades para a utilização de dispositivos de entrada como teclado, mouse e controle remoto, além de prover abstrações para a tela e seus planos gráficos (de vídeo, *background*, *closed caption*, etc.). O pacote DTV UI oferece ainda um container para cada plano gráfico, onde componentes podem ser inseridos e terem seus layouts gerenciados.

3.2 Integração de dispositivos

APIs de integração de dispositivos fornecem acesso para interação a vários usuários simultaneamente através dispositivos de interação. Esta seção apresenta trabalhos relacionados ao uso de múltiplos dispositivos nos ambientes declarativo e

imperativo do *middleware* Ginga e das especificações ARIB.

3.2.1 Ginga-J (API SBTVD)

A API *Interaction Devices* presente no pacote de API específicas do SBTVD (br.org.sbtvd.interactiondevices) foi proposta por (SILVA, 2008) durante a fase de especificação do ambiente Ginga-J. A API utiliza o poder computacional do receptor de forma que toda a interação realizada pelo usuário do dispositivo seja processada no próprio STB, sendo apenas o resultado deste processamento enviado ao dispositivo remoto. A Figura 17 exibida abaixo ilustra o modelo conceitual da API.

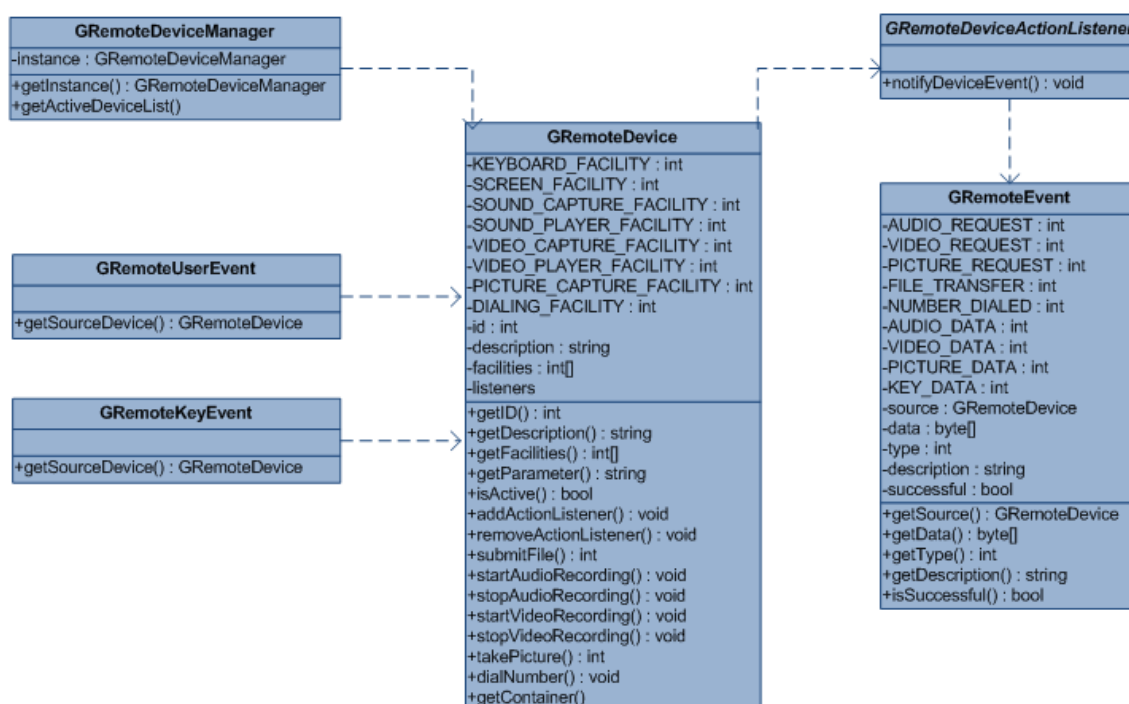


Figura 17. Diagrama de classes da API de integração de dispositivos do pacote br.org.sbtvd.integrationdevices

A API conta com um gerente de dispositivos (GRemoteDeviceManager) para recuperação da lista de dispositivos (GRemoteDevice) conectados e registrados no receptor. A classe GRemoteDevice define um objeto que é a representação abstrata de um dispositivo de interação. Esta classe oferece métodos que possibilitam a recuperação de informações acerca dos dispositivos registrados (tipo do dispositivo, funcionalidades disponíveis etc.), bem como explorar as funcionalidades disponíveis do mesmo (utilizar recursos de gravação de áudio, vídeo, captura de imagens). Cada dispositivo possui uma classe container, da API gráfica, associada (classe DTVContainer na API JavaDTV) que pode ser utilizada para compor interfaces a

serem exibidas de forma transparente nos dispositivos. Ouvintes (GRemoteDeviceActionListener) podem ser registrados nos dispositivos para notificação de eventos de dados e de interação do usuário. Eventos podem encapsular dados de áudio e vídeo, requisições de certos tipos de dados ou códigos de teclas pressionadas.

3.2.2 Ginga-NCL

O *middleware* declarativo Ginga-NCL define dois modelos de comunicação com dispositivos secundários de exibição para a apresentação de aplicações distribuídas (SOARES, et al., 2009). No primeiro modelo um mesmo conteúdo é apresentado nos dispositivos sobre controle de navegação único; o segundo modelo permite que os dispositivos controlem a apresentação do conteúdo de forma independente, permitindo um controle de navegação completamente individualizado a partir de cada dispositivo.

Esses dois modelos são refletidos em classes de dispositivos que podem ser utilizadas nas aplicações NCL (SOARES, et al., 2009). O primeiro modelo define uma classe chamada passiva, enquanto o segundo é aplicado à classe de dispositivos ativos. Outras classes podem ser definidas, com base nos recursos disponíveis nos dispositivos, porém deverão ser extensões das classes ativa e passiva. Cada dispositivo pode pertencer a mais de uma classe, porém um recurso só poderá utilizar um modelo de comunicação durante a execução de uma aplicação distribuída.

O conjunto de dispositivos registrados junto ao dispositivo base (único) forma um domínio, e a base orquestra a exibição da aplicação utilizando os recursos registrados. O dispositivo base deverá transmitir amostras de áudio e/ou vídeo já prontas para serem exibidas para os dispositivos passivos, de forma que todos os dispositivos apresentem sempre o mesmo conteúdo. Cabe ao receptor transmitir para os dispositivos de classe ativa partes da aplicação - que são objetos de mídia, podendo ser inclusive código NCL - para que sejam apresentados e localmente controlados. A NCL não restringe os tipos de mídia que compõem uma aplicação, apenas define seus relacionamentos no tempo e espaço. O modelo permite inclusive a criação de novos domínios a partir de aplicações em dispositivos ativos, de forma que a mesma possa ser distribuída novamente entre outros dispositivos (ativos e

passivos) recorrentemente. A Figura 18 exibe a arquitetura do componente de múltiplos dispositivos do Ginga-NCL.

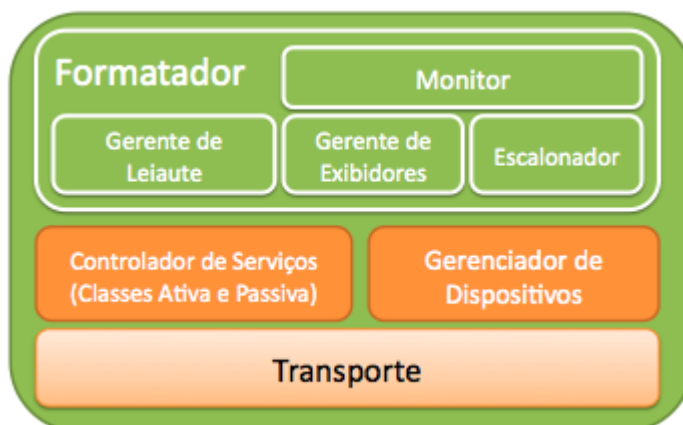


Figura 18. Arquitetura do componente de múltiplos dispositivos no GingaNCL

As funcionalidades da NCL relacionadas à utilização de recursos de dispositivos externos são suportadas por módulos específicos na arquitetura do Ginga-NCL (SOARES, et al., 2009). Os módulos que compõem o componente de integração com múltiplos dispositivos referem-se ao registro de dispositivos e à comunicação entre os dispositivos para apresentação de um documento multimídia distribuído. O *Controlador de Serviços* controla os serviços dos dispositivos em um domínio, que é definido pelo conjunto dos dispositivos em suas classes. O *Gerenciador de Dispositivos* (SOARES, et al., 2009) controla o registro de dispositivos junto a um conjunto de classes de dispositivo pré-definidas: a classe "0", que se refere ao dispositivo base (que executa o código NCL inicial); a classe "1" refere-se aos dispositivos passivos, que receberão fluxos de mídia prontos para serem exibidos; e finalmente a classe "2", que é a designada para os dispositivos ativos, que recebem código NCL (parte da aplicação primária) a ser executado de forma independente. Mais classes podem ser criadas a partir das classes padrão, e o registro de classes pode também ser feito dinamicamente, a partir do uso de scripts Lua (SOARES, et al., 2009). O modelo de definição de classes, porém, não é definido pelas especificações do Ginga-NCL, podendo ser especificado pelo fabricante do dispositivo base.

O *Gerenciador de Dispositivos* guarda todas as informações relativas à configuração dos dispositivos para todas as classes registradas, bem como a associação dos dispositivos e classes. Informações sobre os recursos dos

dispositivos que serão utilizados pelas aplicações NCL (ex. tamanho de tela, número de canais de áudio, entre outras coisas) estão associados a um conjunto de variáveis de ambiente do Ginga-NCL, que podem ser acessadas através do seu "nó de *settings*" (SOARES, et al., 2009). Essas informações podem ser utilizadas para adaptação de aplicações às características de uma determinada classe de dispositivos.

3.2.3 ARIB Peripheral

A especificação ARIB STD-B23 (*Application Execution Engine Platform for Digital Broadcasting*) define um ambiente para execução de aplicações para TV Digital baseadas em Java (análogo ao Ginga-J), sendo aderente às especificações ITU J.202, porém definindo um conjunto de API Java específicas para o sistema ISDB. Uma dessas APIs foi definida para a integração de dispositivos conectados com o receptor de TV Digital, e o pacote núcleo dessa API é denominado `jp.or.arib.tv.peripheral` (ARIB, 2009).

Esse pacote oferece classes e interfaces com funcionalidades para descoberta e registro de dispositivos, obtenção de propriedades e estado dos dispositivos, além de funcionalidades de leitura e escrita para comunicação. As principais classes e interfaces são apresentadas abaixo:

- *Device* - interface que provê métodos para acessar dispositivos periféricos - obtenção de propriedades, leitura/escrita e registro de *listeners* (*DeviceStateListener*) para detecção de mudança de estado (eventos *DeviceStateEvent*).
- *DeviceManager* - interface que gerencia os dispositivos conectados e capazes de se comunicar com o receptor executando a aplicação (Xlet). Através dessa interface é possível obter informações sobre quais são os dispositivos disponíveis e registrar *listeners* (*DeviceDiscoveryListener*) para notificação quando da descoberta de dispositivos (eventos *DeviceDiscoveryEvent*).
- *DevicePermission* - classe que define o conjunto de permissões disponíveis para acessar um dispositivo em uma rede doméstica (estende `java.security.Permission`). Uma instância dessa classe representa o conjunto

de permissões para um dispositivo específico.

- *PeripheralException* - essa classe serve para notificação de exceções ocorridas em operações realizadas em conjunto com os dispositivos conectados.

3.3 Segurança

APIs de segurança são de suma importância para aplicações que necessitam de confidencialidade e garantia de integridade nos dados manipulados. Aqui serão abordadas as soluções de segurança da API Java Security e da biblioteca Lua MD5 apresentando as funcionalidades mais relevantes para a especificação da API proposta.

3.3.1 Java Security

A API de segurança Java Security (JAVA SECURITY, 2010) (pacote *java.security* ¹¹) define um conjunto de APIs juntamente com implementações de algoritmos de segurança, mecanismos e protocolos mais utilizados. A API cobre vários pontos relacionados à segurança incluindo criptografia, infra-estrutura de chaves públicas, comunicação segura, autenticação e controle de acesso. Desta API destacam-se três classes relevantes para o trabalho proposto: a classe *MessageDigest*, a classe *Signature* e a classe *Cipher*.

A classe *MessageDigest* como o próprio nome diz, provê funcionalidade de *message digest* (ou *hashing*). A classe recebe uma entrada de tamanho arbitrário (um *array* de bytes) e gera uma saída de tamanho fixo que identifica de forma única os dados. A saída produzida é chamada às vezes de "*checksum*" ou "*digital fingerprint*" dos dados. A Figura 19 exibe o funcionamento da classe.



Figura 19. Funcionamento da classe *MessageDigest*

A classe funciona através da alimentação dos dados com chamadas ao método *update* e geração do *hash* pelo método *digest*, se utilizando de algoritmos

¹¹ <http://java.sun.com/javase/6/docs/api/java/security/package-summary.html>

como o MD5 (RIVEST, 1992) e SHA-1 (NIST, 1995). *Message digests* podem ser utilizados em várias tarefas e podem determinar quando um dado foi modificado, intencionalmente ou não.

A classe `Signature` provê funcionalidades de algoritmos de assinatura digital criptográfica como o DSA (NIST, 1994) ou RSA (RIVEST, et al., 1983) (em conjunto com MD5). Um algoritmo de assinaturas seguras recebe como entrada uma chave privada e dados de tamanho arbitrário, e produz como saída uma string de bytes relativamente pequena (em geral de tamanho fixo), a assinatura. A Figura 20 apresenta a utilização desta classe na API de segurança do Java.

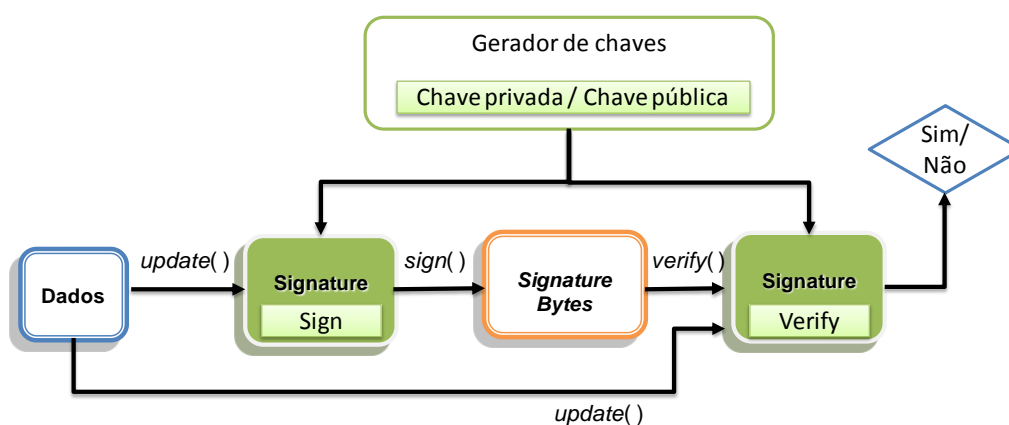


Figura 20. Utilização da classe `Signature` na API Java Security

A classe `Cipher` disponibiliza facilidades para encriptação e decriptação de dados. A encriptação é o processo de tomar como entrada dados (textos puros) e uma chave, e produzir dados (texto cifrado) sem um significado para terceiros que desconheçam a chave. A decriptação é o processo inverso, tomar um texto cifrado e uma chave e reproduzir o texto puro inicial. A Figura 21 apresenta a utilização da classe `Cipher` na API.

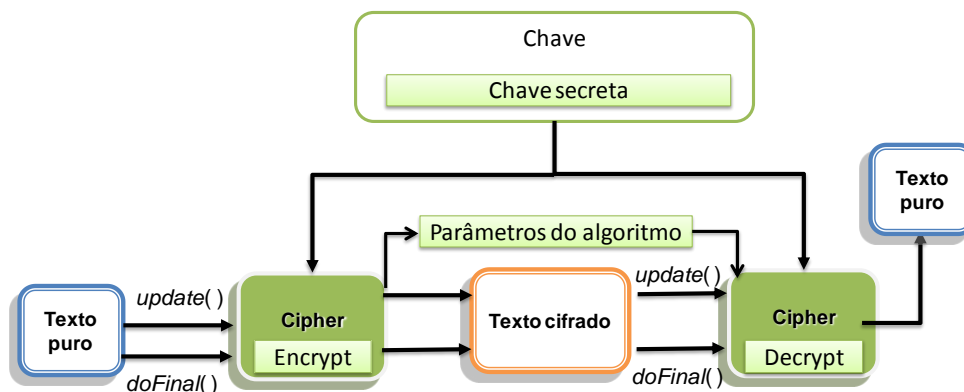


Figura 21. Utilização da classe Cipher na API Java Security

3.3.2 Lua MD5

Lua MD5 (LUA-MD5, 2010) é uma biblioteca aberta que oferece facilidades criptográficas básicas para Lua 5.1. Estão presentes na biblioteca o módulo md5, com funcionalidades para *digest* (*hash*), encriptação/decriptação utilizando o algoritmo MD5 e também o módulo des56 que utiliza o algoritmo DES com chave de 56 bits para encriptação/decriptação de dados.

O design desta biblioteca, assim como as outras bibliotecas padrão da linguagem Lua, é bastante simples e direto. A tabela md5 provê a seguintes funcionalidades:

- **md5.sum (message):** computa o *message digest* de uma string baseado no algoritmo MD5. Recebe uma mensagem de entrada com tamanho e conteúdo arbitrário e retorna uma saída de 128-bits com o *fingerprint* dos dados.
- **md5.sumhexa (message):** semelhante a md5.sum mas o retorno é uma string hexadecimal de 32 bytes.
- **md5.crypt (message, key [,seed]):** Encripta uma string utilizando o algoritmo MD5 no modo CFB (*Cipher-Feedback Mode*) para encriptação em blocos de tamanhos fixos. A mensagem (*message*) é uma string de tamanho arbitrário assim como a chave (*key*). Opcionalmente, um *seed* pode ser utilizado. O retorno é uma string com o texto cifrado.

- **md5.decrypt (message, key):** Decrypta uma string tomando como entrada uma string resultante de uma chamada prévia a *crypt()*.
- **md5.exor (s1, s2):** Realiza um OU-exclusivo bit a bit das strings s1 e s2 (ambas devem ter o mesmo tamanho).

As seguintes funções estão registradas na tabela des56:

- **des56.crypt (message, key):** semelhante a md5.encrypt() porém encripta uma string utilizando o algoritmo DES com uma chave de 56 bit e uma chave de 8-bytes deve ser utilizada.
- **des56.decrypt (message, key):** semelhante a md5.decrypt(). A string de entrada deve ser resultante de uma chamada prévia a des56.crypt().

3.4 Informações sobre serviço televisivo

As APIs de informações de serviço, ou APIs de SI (*Service Information*), são responsáveis por fornecer às aplicações o acesso a dados presentes nas tabelas de informações de serviço, geralmente em um fluxo do padrão MPEG-2/System (ISO/IEC 13818-1, 2001). Tais informações incluem identificadores para os fluxos elementares (áudio, vídeo e dados) presentes em cada serviço multiplexado, descrição textual dos serviços e eventos que o compõem, entre outros.

O MPEG-2/System, conhecido formalmente por ISO/IEC 13818-1 especifica um esquema de informação de serviços chamado PSI (*Program Specific Information*). O PSI provê informações para possibilitar a configuração automática do receptor para demultiplexar e decodificar os vários fluxos de programas presentes em um *transport stream* (container utilizado para transmissão de fluxo de TV digital) ou em um *program stream* (um container para mídias baseadas em arquivos como o DVD).

3.4.1 MHP (DVB SI)

A DVB SI (DVB, 2003) é a API utilizada pelo middleware MHP para tratar a

recuperação de informações específicas do sistema. Ela está inserida no pacote org.dvb.si. A API possui uma classe central org.dvb.si.SIDataBase que dá acesso a todas as informações das diversas tabelas a uma aplicação. As informações podem ser acessadas de forma síncrona ou assíncrona dependendo da finalidade da informação para a aplicação. Se a aplicação precisa acessar uma informação com o menor tempo possível, a API checa primeiramente por dados em *cache* e somente caso não estejam disponíveis, irá extrair as informações que vêm no fluxo. A recuperação das informações se dá através de ouvintes que são notificados quando determinada requisição é concluída. Além disso, A API provê um monitor para observar mudanças em um objeto SI específico através da classe SIMonitoringListener.

A API DVB SI adiciona algumas tabelas às originalmente definidas pelo padrão ISO MPEG2-System, algumas delas são obrigatórias e outras opcionais. As tabelas obrigatórias definidas são:

- **NIT** (*Network Information Table*): Descreve organização física do multiplexador, características da rede e os *Transport Streams* associados a cada rede;
- **SDT** (*Service Description Table*): Informa o nome do provedor de serviços, seu tipo Rádio, TV, Teletexto, etc;
- **EIT** (*Event Information Table*): Descreve nome, início e duração de um programa;
- **TDT** (*Time and Date Table*): Tempo UTC;

As tabelas especificadas como opcionais são as seguintes:

- **BAT** (*Bouquet Association Table*): Descreve uma associação lógica de serviços, utilizada para agrupar serviços (pacotes de programação *pay-per-view*, por exemplo)

- **RST** (*Running Status Table*): Informações de evento atual e próximo podem ser um pouco desatualizadas devido ao delay de transmissão do SI, esta tabela pode ser utilizada para recuperar tais informações de forma mais precisa;
- **TOT** (*Time Offset Table*): Informações sobre hora e data;
- **ST** (*Stuffing Table*): Utilizada para manter a integridade de tabelas que tiveram seções sobrescritas;

3.4.2 ARIB B.23 (API ARIB SI)

A solução adotada pela ARIB é baseada na DVB SI, porém, novas tabelas são utilizadas para adequação do sistema a informações específicas. Quem provê o fornecimento destas informações específicas do padrão japonês ISDB é a ARIB SI (ARIB, 2004). A API adiciona as seguintes novas tabelas ao MPEG2-System, sendo todas opcionais:

- **PCAT** (*Partial Content Announcement Table*): Indica o agendamento de transmissão de conteúdo parcial na transmissão de dados;
- **BIT** (*Broadcaster Information Table*): Utilizada para enviar informações sobre a rede difusora;
- **NBIT** (*Network Board Information Table*): Descreve a informação da bancada de rede e informação de referência de como adquirir as informações de grupo de rede;
- **LDT** (*Linked Description Table*): Carrega informações que referenciam outras tabelas;

3.4.3 Ginga-J (API JavaDTV Service)

O acesso a informações sobre serviços na API JavaDTV é disponibilizado

pelo pacote com.sun.dtv.service (SUN, 2009). Uma única classe é definida neste pacote, a SIDatabase, através dela é possível obter uma referência para o acesso ao componente de SI de mais baixo-nível (dependente de plataforma). Esta classe é uma forma de acesso genérica aos dados de SI e prevê sua utilização para os padrões SBTVD, DVB e ARIB. Uma implementação desta API deve estender a classe SI Database e prover métodos de acesso adicionais para a recuperação de tabelas individuais. A Figura 22 a seguir exhibe a classe SIDatabase.

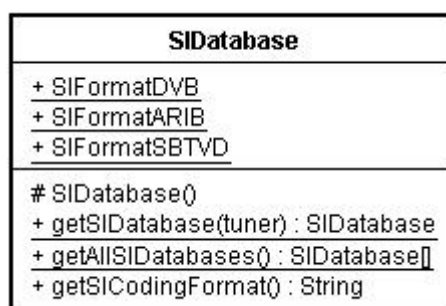


Figura 22. Classe SIDatabase responsável pela funcionalidade de SI no JavaDTV

Cada instância da classe SIDatabase é associada a uma interface de rede. A classe oferece o método getSIDatabase() para requisição de uma SIDatabase específica ao Tuner. O método getAllSIDatabases() retorna todas as SIDatabases disponíveis no sistema. Adicionalmente, o método getSICodingFormat() pode ser utilizado para verificação do tipo de SIDatabase.

3.4.4 TV-Anytime (TVA)

Na maioria dos sistemas de TVD a transmissão de informações é baseada nas definições de metadados das tabelas de SI. No entanto, tabelas de SI são consideradas metadados rígidos, destinados a um propósito específico e que não passam por uma padronização de metadados passível de ser customizável. Serviços mais elaborados podem necessitar de informações detalhadas e que dificilmente seriam definidos de forma satisfatória com tabelas SI. Assim, surgiu a necessidade de se padronizar metadados flexíveis, ou seja, metadados que apresentem estruturas customizáveis para atender os novos serviços em cenários mais complexos (ALVES, 2006).

O TV-Anytime (TV-Anytime, 1999) foi desenvolvido de 1999 a 2005 em uma

iniciativa de várias organizações com o intuito de desenvolver um conjunto de especificações abertas para sistemas interoperáveis. O objetivo era possibilitar que provedores de conteúdo, provedores de serviço e usuários fossem capazes de manipular conteúdo digital em dispositivos com capacidade de armazenamento. Diferentemente dos padrões baseados em PSI, o TVA adota XML como formato para representação dos metadados e a definição formal da estrutura e da sintaxe dos metadados é realizada por XML Schema.

Existem quatro categorias de metadados no TVA: metadados de descrição de conteúdo, de descrição de instância, de usuário e de segmentação (TV-Anytime, 1999). A primeira categoria descreve a mídia em si, estão nesta categoria informações como: gênero, idioma, informações sobre áudio e vídeo, etc. A segunda inclui informações para dar suporte a mecanismos de localização e anúncio de serviços, por exemplo, um Guia Eletrônico de Programação ou *Electronic Programming Guide* (EPG). A categoria de usuário identifica grupos, perfis e histórico de utilização dos usuários, podendo ser utilizada para uma visualização personalizada do conteúdo de acordo com o perfil do usuário. A última categoria permite o acesso e manipulação das mídias em intervalos temporais aleatórios.

O TVA não define tecnologias e mecanismos de recepção a serem aplicados, podendo ser adotado por qualquer padrão de TVD.

3.5 Análise comparativa das soluções

As soluções HAVi GUI e JavaDTV LWUIT+DTVUI para criação de interface gráfica com a linguagem Java são funcionalmente equivalentes, ambas são *toolkits* completos para composição de *widgets* e gerenciamento de *leiaute*. Como já mencionado, a API JavaDTV foi especificada para substituir as APIs baseadas no GEM no SBTVD, que incluía a HAVi GUI. Ambas as APIs contam com uma gama de diferentes componentes visuais e containers possibilitando virtualmente qualquer design de interface desejado. As duas APIs dão suporte a captura de eventos do usuário através de teclado, apenas a API JavaDTV suporta interação através de mouse.

A outra proposta analisada para criação de interfaces gráficas foi o *framework* LuaOnTV, cujo foco está no ambiente Ginga-NCL. Esta solução oferece uma API para a criação de componentes visuais em Lua visando substituir o esquema de

composição de interfaces oferecido naturalmente pela NCL, deixando para a linguagem apenas a sincronização das mídias presentes no documento. O *framework* emprega um esquema de orientação a objetos em sua arquitetura. Os *widgets* suportados são baseados em componentes comumente encontrados em *toolkits* tradicionais.

Dentre as soluções abordadas para integração de dispositivos, nota-se a semelhança entre as APIs Java ARIB Peripherals e Interaction Devices do ambiente Ginga-J. A proposta do sistema japonês baseou-se na API do ambiente imperativo do Ginga e dá suporte basicamente às mesmas funcionalidades. O foco destas APIs é a recuperação de dispositivos conectados para utilização individual de seus recursos, sendo a maior parte do processamento realizada no dispositivo base. Em contraste, a proposta do ambiente Ginga-NCL provê mecanismos para distribuição de partes da aplicação e do conteúdo multimídia em um esquema de hierarquia de classes. O processamento das mídias acontece tanto no dispositivo base quanto nos dispositivos da chamada classe ativa.

Os dois trabalhos discutidos relativos a segurança de informações são soluções distintas que podem ser utilizadas em ambientes declarativos e imperativos. O Java Security contempla um grande número de dispositivos de segurança que cobre diversos requisitos de uma aplicação interativa segura. Já a proposta Lua-MD5, apesar de limitada em suas funcionalidades, demonstra que é possível disponibilizar uma API de segurança a ser utilizada por *scripts* Lua, que eventualmente poderiam estar inseridos em documentos NCL. Sua sintaxe simples e direta facilita o entendimento e utilização da API por desenvolvedores de Lua.

As APIs DVB SI, ARIB SI e JavaDTV Service para recuperação de informações sobre serviços televisivos são semelhantes em termos de utilização e design. Todas as três propostas foram baseadas em PSI e podem ser vistas como parte de uma primeira geração para transmissão de metadados de programação televisiva (os chamados metadados rígidos). Estes padrões se desenvolveram naturalmente para suprir as necessidades de aplicações que necessitavam de metadados presentes em fluxos de *broadcasting* digital.

Por outro lado, o TV-Anytime é visto como uma grande promessa para se tornar um padrão chave para metadados em um próximo passo evolutivo nos sistemas televisivos digitais. Um dos motivos está na tendência das aplicações

interativas atuais utilizarem cada vez mais conteúdo baseado na *web* e outras fontes de dados em grande volume. Outro ponto relevante é o fato de que o TVA foi desenvolvido de forma extensível (metadados flexíveis).

4. API LuaTV

Neste capítulo é apresentada a API proposta neste trabalho detalhando as funcionalidades de seus módulos, sua arquitetura e a integração com o *middleware* Ginga. A especificação completa da API LuaTV está presente no Apêndice A deste trabalho para consulta detalhada dos pacotes funcionais e todos os seus módulos e funções.

4.1 Visão geral

Essencialmente, objetos NCLua estendem as capacidades de um documento NCL oferecendo facilidades imperativas ao seu desenvolvedor. A API LuaTV fornece facilidades adicionais à API NCLua através de abstrações para funcionalidades de outras camadas de software do *middleware* Ginga-NCL. A Figura 23 ilustra o contexto onde a API está inserida.

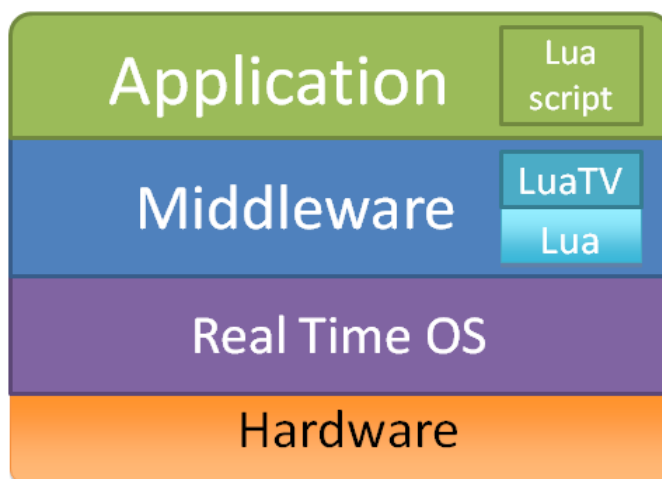


Figura 23. Contexto da API LuaTV

4.2 Arquitetura

A modelagem da arquitetura segue o paradigma minimalista empregado na especificação dos módulos da API NCLua, porém com um nível maior de abstração e genericidade, oferecendo facilidades em alto-nível para a utilização de recursos sem a necessidade de conhecimento de detalhes da plataforma em que está

inserida. A API se utiliza das funcionalidades da API NCLua (módulos *canvas*, *event*, *settings*, *persistent* e biblioteca padrão Lua) e das APIs dos componentes do núcleo comum. Quando necessária, a comunicação com o documento NCL se dá através de eventos do módulo *event*. A Figura 24 mostra a integração da API LuaTV com a arquitetura do *middleware* Ginga. Os módulos de cada componente funcional juntamente com suas arquiteturas e funcionalidades são discutidos em seguida.

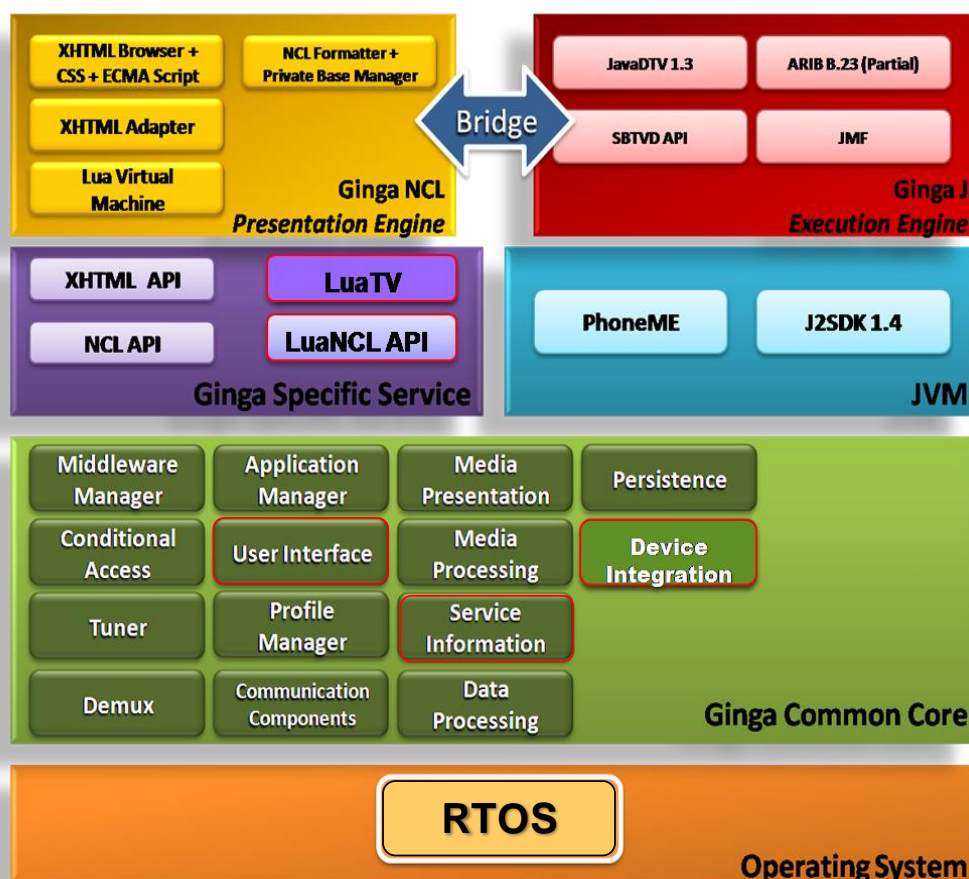


Figura 24. API LuaTV na arquitetura do *middleware* Ginga

A API situa-se na camada de serviços específicos do Ginga e conta com quatro pacotes de APIs que são funcionalmente independentes: *Widget*, *Multidevice*, *Metadata* e *Security*. Os quatro pacotes funcionais definidos para a primeira versão da API contemplam funcionalidades análogas à APIs correspondentes presentes no ambiente Ginga-J, mas que não estão disponíveis para o autor de documento NCL em conjunto com *scripts* Lua na especificação atual.

4.3 Pacotes funcionais

Os pacotes funcionais contemplados pela API foram baseados em funcionalidades que se mostram de difícil implementação em documentos NCL. Para o levantamento de tais requisitos, uma análise foi realizada contrastando as funcionalidades oferecidas pelo ambiente Ginga-NCL (para o autor de documentos NCL) e as funcionalidades oferecidas pelo ambiente imperativo Ginga-J (oferecidas ao desenvolvedor de Xlets Java). Observou-se que o ambiente declarativo poderia dar melhor suporte ao autor de documentos NCL em certos pontos como: melhor captura de dados de entrada do usuário, melhor controle de recursos em dispositivos remotos, controle sobre a segurança de dados manipulados pela aplicação e a recuperação de informações específicas sobre o fluxo transmitido.

Estas funcionalidades foram refletidas diretamente na especificação dos pacotes funcionais da API LuaTV. As sessões apresentadas a seguir discutem detalhadamente cada um dos pacotes funcionais.

4.3.1 API Widgets

O termo *widget* vem de uma corruptela (*Window Gadget*). *Widgets* são componentes de interface gráfica do usuário (GUI, do inglês *Graphical User Interface*), o que inclui objetos como janelas, botões, menus, ícones, barras de rolagem, etc. O objetivo desta API, porém, não é fornecer uma coleção completa de componentes gráficos em um esquema orientado a objetos como é normalmente um *toolkit* de *widgets*. O objetivo é prover mecanismos simples para a construção de componentes visuais arbitrários com foco no tratamento de entrada do usuário, uma vez que, a linguagem NCL já oferece uma gama de mecanismos para a composição de interfaces, porém carece de alguns elementos interativos. Por este motivo, foi elaborada uma API com modelagem diferente das abordadas na seção de trabalhos relacionados, de forma que o esquema de *toolkit* gráfico tradicional foi desconsiderado e uma modelagem que estende as funcionalidades do ambiente Ginga-NCL foi adotada. A Figura 25 traz a arquitetura da API Widgets.

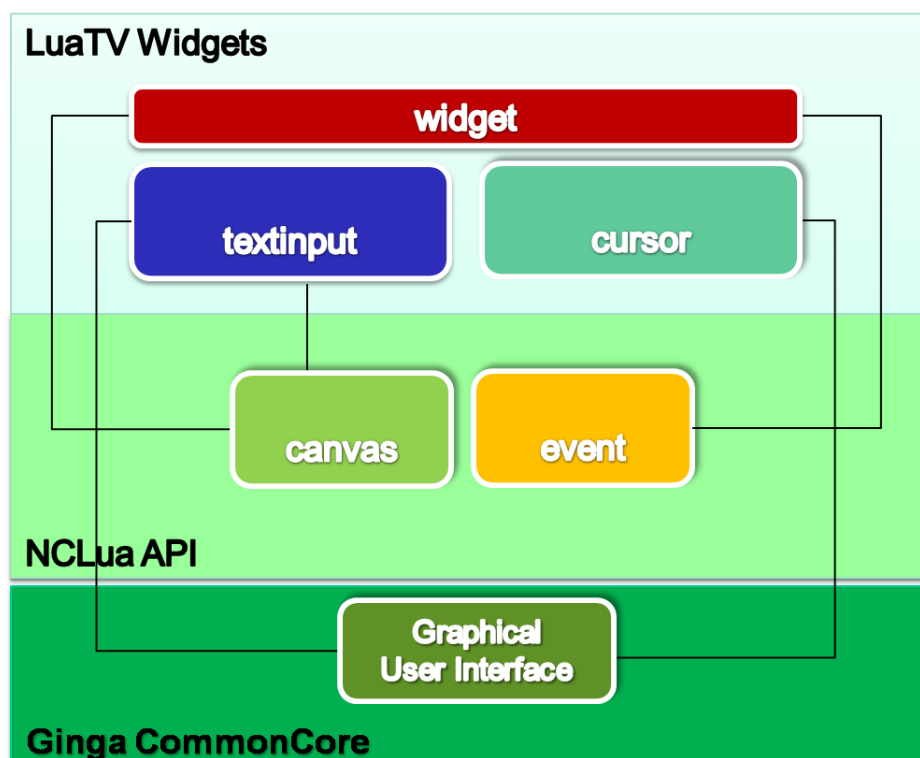


Figura 25. API Widget com os módulos *widget*, *cursor* e *textinput*

Existem três módulos presentes nesta API: *widget*, *cursor* e *textinput*. Os módulos *widget* e *textinput* têm associado uma instância de *canvas* da API NCLua que é utilizada pelo usuário da API para composição da parte visual de seu componente. Os módulos desta API também fazem uso do módulo *event* da API NCLua para envio e recebimento de eventos.

O módulo *widget* é responsável por preencher um dado *canvas* (comumente associado a uma região NCL) que pode ser maior que a região definida para o NCLua no documento NCL. O módulo *widget* permite a rolagem (*scrolling*) deste *canvas* (quando for maior que a região definida) provendo assim uma espécie de *canvas* virtual que pode ter partes fora da área visível da aplicação NCLua. Opcionalmente, o usuário pode registrar tratadores de eventos para receber eventos de entrada como teclas e cursores, recebidos através dos módulos *textinput* e *cursor*. Os eventos Lua repassados são da classe *widget*. Outra funcionalidade presente neste módulo é a possibilidade de criação de *tooltips* com mensagem e área definidas pelo usuário. A Tabela 1 traz todas as funções do módulo *widget*. A notação *module:function()* (com dois pontos) indica que podem existir várias instâncias (ou objetos) do módulo, a linguagem Lua recupera de forma transparente a referência do objeto em questão como primeiro parâmetro para a função, já a

notação *module.function()* indica que existe apenas uma instância de um dado módulo.

Tabela 1. Funções do módulo *widget*

Módulo <i>widget</i>
widget:new ([canvas: object, textinput: object]) → widget: object Cria um novo objeto <i>widget</i> . Um <i>canvas</i> opcional pode ser utilizado (inclusive de tamanho maior que a região do NCLua para <i>scrolling</i>). Um objeto <i>textinput</i> pode ser passado para que seus eventos sejam administrados por este <i>widget</i> . Eventos de cursor também são capturados e repassados.
widget:register (f: function) Registra uma função tratadora para receber eventos de entrada do usuário (mouse e teclado)
widget:unregister (f: function) Desregistra a função tratadora para recebimento de eventos
widget:addTooltip(x, y, w, h: number, msg: string) → tooltip_id: number Adiciona um <i>tooltip</i> a ser exibido quando o cursor estiver na área especificada
widget:removeTooltip(tooltip_id: number) Remove o <i>tooltip</i> através de seu identificador
widget:scroll (x, y: number) Desloca o <i>canvas</i> do <i>widget</i> compondo-o com o <i>canvas</i> da região do NCLua
widget:attrCanvas([canvas: object]) Recupera ou define o <i>canvas</i> deste <i>widget</i>
widget:attTextInput([textinput: object]) Recupera ou define o <i>textinput</i> deste <i>widget</i>

O módulo *cursor* é responsável por oferecer a facilidade de ponteiros ou cursores que podem ser representados por uma imagem arbitrária. O módulo lança seis tipos de eventos da classe *widget*: 'cursor_over', 'cursor_in', 'cursor_out', 'button_press', 'button_release' e 'button_double_click' que encapsulam também os valores das coordenadas x e y de posicionamento do cursor. Os eventos do tipo 'button_press', 'button_release' e 'button_double_click' também possuem um campo 'button' com um valor numérico definindo qual botão do dispositivo controlador do cursor gerou o evento. O valor para o campo 'button' é numérico para permitir que a API seja estendida, de forma que dispositivos com três botões, por exemplo, teriam o valor 0 para o botão esquerda, 1 para o central e 2 para o botão direito. Desta

forma, dispositivos com mais teclas ou botões também podem ser utilizados. Os eventos 'cursor_over', 'cursor_in', 'cursor_out' levam em consideração a área do NCLua definida pelo elemento *region* do documento NCL.

O módulo *cursor* não é criado diretamente pelo usuário da API, uma instância do *cursor* é disponibilizada para cada objeto NCLua no momento de sua execução no documento NCL. A Tabela 2 apresenta as funções deste módulo em seguida a Figura 26 exibe um exemplo de evento gerado pelo módulo *cursor*.

Tabela 2. Funções do módulo *cursor*

Módulo <i>cursor</i>
cursor:register (f: function) Registra uma função tratadora para recebimento de eventos do cursor
cursor:unregister (f: function) Desregistra a função tratadora para recebimento de eventos do cursor
cursor:attrImage ([path: string]) Define ou recupera a imagem associada ao cursor
cursor:attrPosition([x,y: number]) Define ou recupera as coordenadas atuais do cursor
cursor:click () → x,y: number Executa um <i>click</i> na posição atual do cursor, gerando um evento de <i>click</i>

```
{ class='widget', src='cursor',
  type='cursor_in, x=12, y=105 }
```

Figura 26. Exemplo de evento gerado pela entrada do cursor na região do NCLua

O módulo *textinput* oferece ao utilizador da API uma forma de capturar entrada de texto do usuário do sistema e também provê um mecanismo visual para criação de caixas de texto (*text boxes*). Tanto este módulo quanto o módulo *cursor* podem ser utilizados individualmente ou em conjunto com um objeto *widget* repassando eventos capturados de forma transparente. A Tabela 3 apresenta as funções disponibilizadas pelo módulo *textinput*.

Tabela 3. Funções do módulo *textinput*

Módulo <i>textinput</i>	
<code>textinput:new</code> ([<i>canvas</i>: object])	Cria um novo objeto <i>textinput</i> opcionalmente com um <i>canvas</i> para exibição do texto digitado
<code>textinput:register</code> (f: function)	Registra uma função tratadora para recebimento de eventos do teclado
<code>textinput:unregister</code> (f: function)	Desregistra a função tratadora
<code>textinput:attrCanvas</code> ([<i>canvas</i>: object])	Recupera ou define um objeto <i>canvas</i> utilizado para exibir o texto de entrada

A captura de eventos de teclas do módulo *textinput* estende a captura do módulo *event* possibilitando a notificação de modificadores de teclas (*key modifiers*). Modificadores são teclas de controle pressionadas em conjunto com outras teclas, por exemplo, teclas CTRL+C, ALT+F4, etc. Modificadores são notificados através da tabela *modifiers* encapsulada em um evento do módulo *textinput*. Dois tipos de eventos podem ser lançados: 'key_press' e 'key_release'. A Figura 27 exibe um exemplo de evento lançado por um objeto *textinput*.

```
{  class='widget',  src='textinput', type =
  'key_press', key=65 modifiers={'alt', 'ctrl'} }
```

Figura 27. Evento de tecla que encapsula modificadores

Uma implementação da API Widgets não pode ser feita completamente em Lua devido à necessidade de se comunicar com o componente de interface gráfica presente no núcleo comum. Através deste componente se dá a captura dos eventos de mouse e teclado, já que o núcleo comum tem acesso a funcionalidades dependentes de plataforma. A camada de *software* do núcleo comum utiliza a linguagem C++ como padrão. Por se tratar de uma linguagem de extensão, Lua pode acessar naturalmente código C/C++ através da sua pilha de comunicação global (IERUSALIMSCHY, 2006).

Alguns *widgets* que fazem uso das funcionalidades providas por esta API serão apresentados no capítulo 5 como forma de ilustrar cenários possíveis com as inovações proporcionadas pela API LuaTV.

4.3.2 API Multidevice

O objetivo principal desta API é a integração de múltiplos dispositivos através do suporte ao esquema de hierarquia de classes e serviços utilizado pela linguagem NCL conforme abordado na subseção 3.2.2 de trabalhos correlatos. Os dois módulos que fazem parte da API Multidevice oferecem funcionalidades relacionadas à utilização de recursos em dispositivos remotamente conectados a um dispositivo base, comumente um receptor de TV Digital. O módulo *devicemanager* oferece entre outras coisas registro de classe de dispositivos, recuperação de dispositivos e instanciação de serviços. Já o módulo *deviceservice* provê mecanismos para requisição ou envio de dados para um dispositivo ou grupo de dispositivos de uma classe, ele é instanciado através do módulo *devicemanager*. A arquitetura da API é ilustrada na Figura 28.

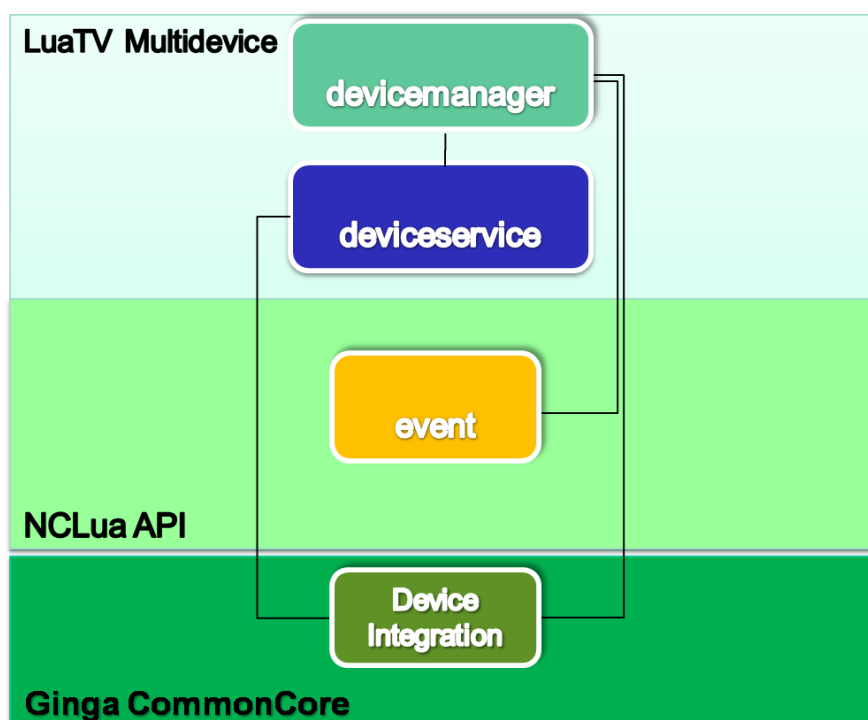


Figura 28. A API Multidevice com os módulos *devicemanager* e *deviceservice*.

O módulo *devicemanager* é responsável por gerenciar dispositivos e suas classes. É possível recuperar uma lista de dispositivos de uma dada classe bem como uma lista com todas as classes registradas. Outra facilidade provida pelo módulo é a recuperação de informações sobre classes e os metadados de dispositivos, que são encapsulados em tabelas Lua. A API oferece mecanismos para que os serviços oferecidos por uma classe possam ser utilizados, através de uma

abordagem de requisições de sintaxe livre que geram respostas associadas. Dessa forma, estabelece-se um canal de comunicação entre o dispositivo executando uma aplicação LuaTV com um grupo de dispositivos que suportam um determinado serviço. Os serviços instanciados são objetos *deviceservice*. As Tabelas 4 e 5 mostram as funções definidas no módulo *devicemanager* e *deviceservice* respectivamente.

Tabela 4. Funções do módulo *devicemanager*

Módulo <i>devicemanager</i>
<i>devicemanager.listDevices</i> (<i>device_class</i>: string) → <i>devices</i>: table Recupera uma lista de dispositivos conectados associados à classe desejada
<i>devicemanager.registerClass</i> (<i>device_class_path</i>) → <i>device_class_id</i>: string Registra uma classe de dispositivos a partir de um arquivo XML no formato RDF
<i>devicemanager.listDeviceClasses</i> () → <i>classes</i>: table Recupera uma tabela com todas as classes de dispositivos registradas
<i>devicemanager.classMetadata</i> (<i>device_class_id</i>: string) → <i>class_metadata</i>: table Recupera os metadados de uma classe registrada
<i>devicemanager.deviceMetadata</i> (<i>device_id</i>: string) → <i>device_metadata</i>: table Recupera informações acerca de um dispositivo conectado
<i>devicemanager.newDeviceService</i> (<i>device_class</i>, <i>service</i>: string) → <i>deviceservice</i>: object Instancia um serviço para utilização de determinado recurso
<i>devicemanager.register</i> (<i>handler</i>: function) Registra um tratador para recebimento de eventos de entrada e saída de dispositivos em classes
<i>devicemanager.unregister</i> (<i>handler</i>: function) Desregistra o tratador para recebimento de eventos

Tabela 5. Funções do módulo *deviceservice*.

Módulo <i>deviceservice</i>
<i>deviceservice.request</i> (<i>request</i>: string, <i>handler</i>: function [, <i>device_id</i>: string]) Requisita/envia dados deste serviço que serão recebidos de forma assíncrona através da função tratadora. Requisições são realizadas para grupos de dispositivos ou um dispositivo específico

O módulo *devicemanager* lança eventos notificando a entrada ou saída de dispositivos em classes (eventos do tipo 'join_class' e 'leave_class') (BATISTA, 2010). Serviços instanciados (objetos *deviceservice*) também lançam eventos (do tipo 'data') para notificar a chegada de dados requisitados a grupos ou dispositivos

individuais. As Figuras 29 e 30 exibem exemplos de eventos lançados pelos módulos da *devicemanager* e *deviceservice*.

```
{    class    = 'multidevice',  src    =  
'11:22:33:44',  type    = 'join_class',  
deviceclass = 'someClass'  }
```

Figura 29. Exemplo de evento gerado pela entrada de um dispositivo em uma classe registrada

```
{    class    = 'multidevice',  src    =  
'11:22:33:44',  type    = 'data',  
deviceclass = 'someClass',  service =  
'someService',  data    = {someData,...}  
}
```

Figura 30. Exemplo de evento gerado chegada de dados de um dado serviço

As classes de dispositivos utilizadas por esta API são representadas utilizando o perfil *User Agent Profile* (UAProf) (UAProf, 2001), uma implementação do arcabouço W3C CC/PP (*Composite Capabilities/Preference Profiles*) (KLYNE, et al., 2004) que utiliza como base o *Resource Description Framework* (RDF) (RDF, 2004). O RDF é um modelo de descrição de dados baseado em XML definido por um conjunto de especificações do W3C (*World Wide Consortium*), o qual se tornou um método genérico para descrição e modelagem de dados semânticos presentes na *web* geralmente com diferentes formatos e sintaxes.

A descrição de classes deve seguir a sintaxe do UAProf juntamente com as extensões definidas por (BATISTA, 2010) que visa incorporar outras semânticas de descrição de capacidades como as descrições de dispositivos oferecidas pelas especificações UPnP (UPNP, 2010). As subclasses *HardwarePlatform*, *SoftwarePlatform* e *NetworkCharacteristics* do UAProf (subclasses do componente genérico definido pelo CC/PP) receberam novos atributos, também uma nova classe *DeviceGroup* foi acrescida com o intuito de permitir a definição das classes de dispositivos NCL. Um novo *namespace* (“gncl”) é usado para a definição das novas classes, bem como os novos parâmetros específicos do Ginga-NCL e atributos genéricos para suportar outros modelos de descrição diferentes do UAProf (BATISTA, 2010).

A Listagem 2, mostrada a seguir, exhibe a descrição da classe ativa do Ginga-

NCL (identificador “ClasseAtivaGingaNCL”, uma instância da classe *DeviceGroup*) que faz uso das extensões propostas.

```
<?xml version="1.0"?>
<rdf:RDF>
  <rdf:Description rdf:ID="ClasseAtivaGingaNCL">
    <gncl:minDevices>2</gncl:minDevices>
    <prf:component>
      <rdf:Description rdf:ID="HardwarePlatform">
        <prf:ScreenSize>320x240</prf:ScreenSize>
      </rdf:Description>
    </prf:component>
    <prf:component>
      <rdf:Description rdf:ID="SoftwarePlatform">
        <prf:CcppAccept>
          <rdf:Bag>
            <rdf:li>application/xhtml+xml</rdf:li>
            <rdf:li>image/png</rdf:li>
            <rdf:li>image/x-png</rdf:li>
            <rdf:li>text/html</rdf:li>
            <rdf:li>application/x-ginga-NCL</rdf:li>
          </rdf:Bag>
        </prf:CcppAccept>
        <gncl:supportedNCLProfiles>
          <rdf:Bag>
            <rdf:li>NCL3.0/EDTV</rdf:li>
            <rdf:li>NCL3.0/BDTV</rdf:li>
          </rdf:Bag>
        </gncl:supportedNCLProfiles>
        <gncl:supportedServices>
          <rdf:Bag>
            <rdf:li>NCL Active Class Service</rdf:li>
          </rdf:Bag>
        </gncl:supportedServices>
      </rdf:Description>
    </prf:component>
    <prf:component>
      <rdf:Description rdf:ID="NetworkCharacteristics">
        <gncl:registrationMethod>GingaNCLSalutation</gncl:registrationMethod>
      </rdf:Description>
    </prf:component>
  </rdf:Description>
</rdf:RDF>
```

Listagem 2. Exemplo de descrição da classe ativa NCL com o UAProf e extensões (BATISTA, 2010)

4.3.2.1 Componente *Device Integration* no núcleo comum

O componente chamado *Device Integration* (mostrado na Figura 28) não está presente na arquitetura sugerida do *middleware* Ginga descrita na Norma ABNT NBR 15606-1 (ABNT 15606-1, 2009). A elaboração deste componente tornou-se fundamental para a especificação inicial da API LuaTV, pois uma implementação puramente em Lua da API Multidevice não é viável na atual implementação de referência do Ginga-NCL. A arquitetura do componente será incorporada no projeto GingaCDN, sua especificação fará parte das interfaces disponíveis para implementação no modelo FlexCM (FREIRE FILHO, 2008). A Figura 31 a seguir

descreve a arquitetura elaborada.

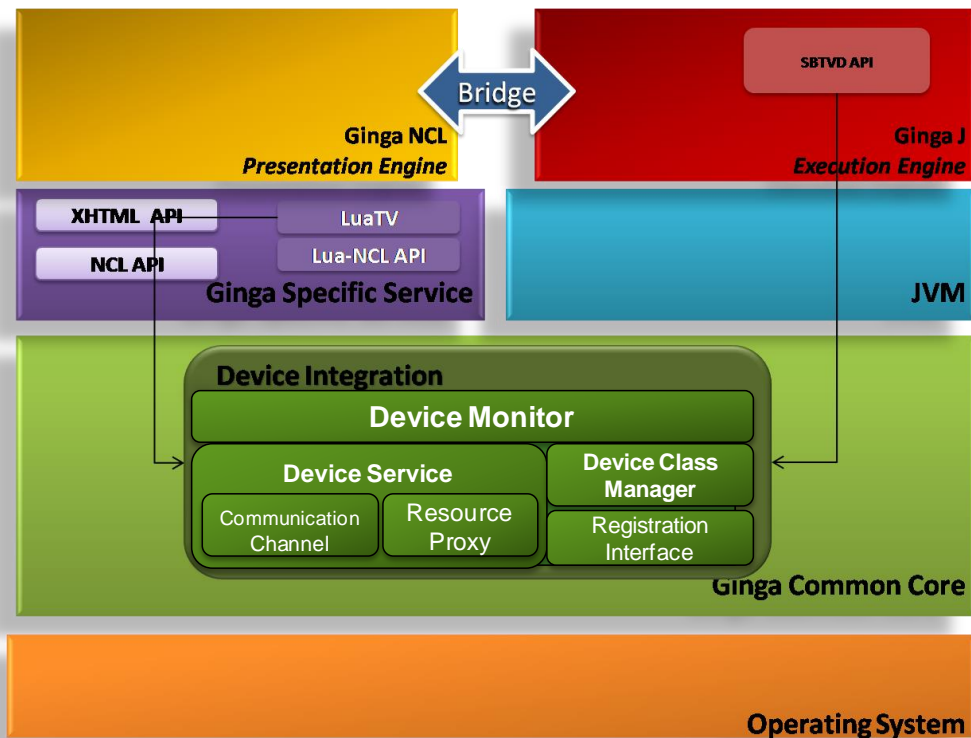


Figura 31. Arquitetura elaborada para o componente *DeviceIntegration*

A especificação da arquitetura do componente *Device Integration* foi baseada nos seguintes requisitos levantados de acordo com as necessidades do ambiente Ginga-NCL (BATISTA, 2010): suporte à definição de classes de dispositivos de forma genérica e extensível; possibilidade de registro dinâmico de dispositivos às classes definidas; e suporte à utilização de serviços e canais de comunicação. As seguintes entidades funcionais foram definidas para a arquitetura do componente *Device Integration*:

- *Device Service*: Componente que oferece uma abstração para utilização dos serviços disponibilizados pelos dispositivos. Dados podem ser requisitados ou enviados através de um canal de conexão (*Communication Channel*), os recursos utilizados durante a comunicação devem ser alocados através do *Resource Proxy*.
- *Communication Channel*: Provê mecanismos para o envio e recebimento de dados abstraindo as tecnologias utilizadas no canal de comunicação.
- *Resource Proxy*: Oferece uma abstração para acesso a recursos. Recursos podem ser arquivos armazenados, acesso a captura de fluxos (*streams*) de áudio e vídeo, etc.

- *Registration Interface*: Interface para registro de dispositivos juntamente com seus perfis e capacidades. A implementação desta interface deve ajustar a semântica e os mecanismos de descrição utilizados ao modelo UAProf estendido. O componente *Device Class Manager* pode agrupar os dispositivos registrados nas classes associadas a uma aplicação NCL em execução.

- *Device Class Manager*: Gerencia as classes de dispositivos disponíveis juntamente com a associação de dispositivos a uma ou mais delas.

- *Device Monitor*: componente responsável pela comunicação com o formatador do ambiente Ginga-NCL. É o componente responsável por traduzir os eventos do formatador para sinais de controle dos serviços oferecidos pelos dispositivos conectados.

Apesar de idealizado para suportar o ambiente Ginga-NCL, o componente *Device Integration* pode ser utilizado pela API de integração de dispositivos presente no pacote de APIs SBTVD do Ginga-J. A especificação da arquitetura do componente é genérica o suficiente para a utilização de quaisquer recursos presentes em dispositivos. A integração com o ambiente imperativo se dá através do *framework* de programação JNI (*Java Native Interface*) que permite que a máquina virtual Java acesse bibliotecas construídas com código nativo.

No Apêndice B desta dissertação podem ser encontrados os cabeçalhos que definem a interface do componente *Device Integration* utilizando o modelo FlexCM. Estão presentes a interface da classe principal *IDeviceMonitor*, um ouvinte (classe *DeviceMonitorListener*) e uma classe para a representação de um evento (*DeviceMonitorListenerEvent*). As demais entidades definidas na arquitetura do componente devem ser implementadas como classes internas, sem uma interface para utilização de suas funcionalidades diretamente por outros componentes. A definição da API deste componente faz parte das contribuições do presente trabalho.

4.3.3 API Security

Para melhor entendimento da API *Security* são necessários conceitos relacionados a funções de *hash*, algoritmos de criptografia simétricos (também chamados de algoritmos de chave simétrica) e assimétricos (também chamados de algoritmos de chave assimétrica ou pública).

Uma função de embaralhamento criptográfico (*cryptographic hash function*, em inglês) é um procedimento determinístico que recebe como entrada um bloco arbitrário de dados e retorna uma cadeia de bytes com tamanho fixo, de tal forma que qualquer mudança nos dados irá alterar o valor do *hash*, quando calculado de novo. Os dados a serem codificados muitas vezes são chamados de "mensagem" e o resultado do embaralhamento é chamado de resumo da mensagem (*message digest*). Nesta categoria estão algoritmos como o MD5 (RIVEST, 1992) (*Message Digest algorithm 5*) e os algoritmos da família SHA (NIST, 1995) (*Secure Hash Algorithm*).

Os algoritmos de encriptação simétrica utilizam chaves criptográficas relacionadas para as operações de cifragem ou decifragem. A chave de cifragem pode ser idêntica à de decifragem ou poderá existir uma transformação simples entre as duas chaves. As chaves, na prática, representam um segredo, compartilhado entre duas ou mais partes, que podem ser usadas para manter um canal confidencial de informação. Usa-se uma única chave, compartilhada por ambos os interlocutores, na premissa de que esta é conhecida apenas por eles. Os algoritmos 3DES (NIST, 1999) (*Triple Data Encryption Standard*) e AES (NIST, 2001) (*Advanced Encryption Standard*) pertencem a esta categoria.

Algoritmos de criptografia assimétricos utilizam um par de chaves: uma chave pública e uma chave privada. A chave pública é distribuída livremente enquanto a chave privada é conhecida apenas por seu dono. Uma mensagem cifrada com a chave pública somente pode ser decifrada pela sua chave privada correspondente. Esta categoria de algoritmos pode ser utilizada para autenticação e confidencialidade. Para confidencialidade, a chave pública é usada para cifrar mensagens, com isso apenas o dono da chave privada pode decifrá-la. Para autenticação, a chave privada é usada para cifrar mensagens, com isso garante-se que apenas o dono da chave privada poderia ter cifrado a mensagem que foi decifrada com a chave pública correspondente. O algoritmo RSA (RIVEST, et al., 1983) (que leva o nome de seus criadores Rivest, Shamir e Adleman) é o representante mais difundido desta categoria.

Em um documento NCL os aspectos relacionados à segurança não são tratados pelo autor do documento. Assume-se que os *players* de mídia devem suportar protocolos de segurança se utilizados. Neste sentido, a API Security provê

facilidades para geração e verificação de assinaturas digitais, geração de *message digest* e cifragem para transmissão segura de dados para aplicações que desejem maior segurança de seus dados. Alguns exemplos de aplicações em que a segurança dos dados é necessária são: aplicações de *T-Bank* ou *T-Commerce* (para transações financeiras), aplicações de votação (que primam pela confidencialidade de seus dados), aplicações de utilidade pública como declaração de imposto de renda, entre outras.

A Figura 32 apresenta a arquitetura elaborada para a API de segurança na especificação LuaTV.

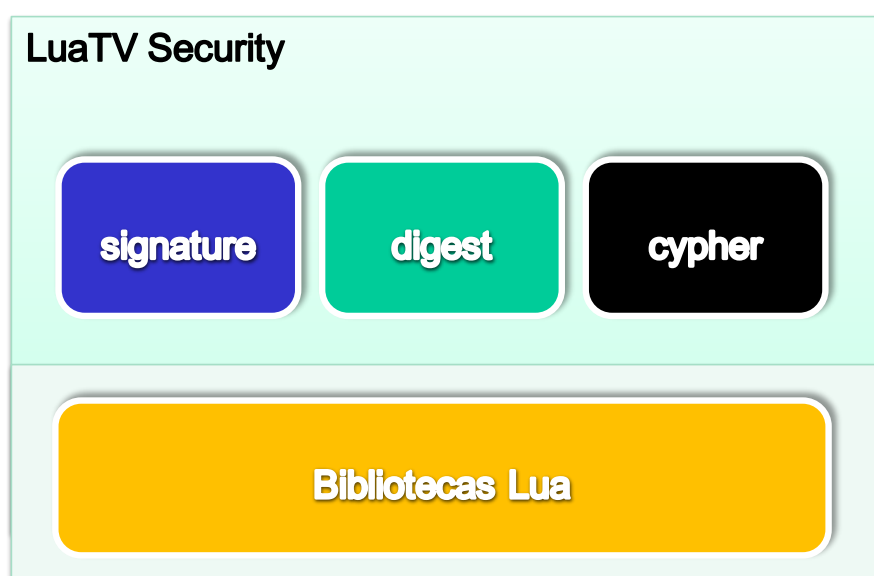


Figura 32. API Security com os módulos *signature*, *digest* e *cypher*

São três os módulos presentes nesta API: *signature*, *digest* e *cypher*. A API Security pode se utilizar de funcionalidades já providas por bibliotecas Lua como manipulação de bits e implementações de certos algoritmos. A biblioteca Lua MD5, discutida na seção de trabalhos relacionados, poderia ser utilizada por uma implementação da API, contudo a implementação atual da biblioteca tem partes em C e a API de segurança proposta pode ser implementada puramente em Lua.

Uma implementação exclusivamente em Lua é possível e essa característica pode ser explorada na transmissão do fluxo de dados pela emissora. As bibliotecas de segurança utilizadas podem ser enviadas juntamente com aplicações NCL no carrossel de dados transmitido, sem a necessidade de qualquer adaptação no código do *middleware* presente no receptor do usuário.

Outras propostas como a biblioteca Lua Bit (LUA BIT, 2007) (para operações bit-a-bit implementada puramente em Lua) e Lua Lash (com implementações dos algoritmos CRC32, SHA-1 e MD5) (LUA LASH, 2010) também podem ser consideradas em implementações da API LuaTV. Estas três bibliotecas (Lua MD5, LuaBit e Lua Lash) estão disponíveis sob licença MIT, a mesma utilizada por Lua.

Na API de segurança proposta, o módulo *signature* oferece métodos para geração e verificação de assinaturas digitais. A Tabela 6 mostra a descrição das funções oferecidas pelo módulo.

Tabela 6. Funções do módulo *signature*.

Módulo <i>signature</i>	
signature:new (digest_algorithm: string) → signature: object	Cria uma nova instância do módulo utilizando o algoritmo especificado.
signature:sign (data, key: string [, key_password: string]) → signed_data: string	Assina os dados com uma chave no formato do algoritmo especificado. Uma senha pode ser utilizada para descriptografar a chave.
signature:verify(data, signature: string [, cert_path: string]) → b: boolean	Verifica se os dados (<i>data</i>) informados foram assinados com determinada chave (<i>signature</i>). Um certificado contendo a chave pública a ser verificada pode ser utilizado.
signature:listAlgorithms() → algorithms: table	Recupera uma lista dos algoritmos suportados por este módulo
signature:listCertificates() → certificates: table	Recupera uma lista dos formatos de certificados suportados por este módulo

O usuário pode definir qual algoritmo de *hash* deverá ser utilizado para geração da assinatura, uma lista com uma identificação dos algoritmos suportados pode ser recuperada. A verificação pode ser realizada através de um certificado, por exemplo, um certificado x509 (ITU, 2008) padrão ITU-T para infra-estruturas de chaves públicas (PKI, do inglês *Public Key Infrastructure*). Uma recomendação X.509 especifica, entre outras coisas, identificação dos algoritmos utilizados, validade do certificado, o formato dos certificados digitais e uma lista de certificados revogados, de tal maneira que se possa amarrar firmemente um nome a uma chave pública. Uma lista com os formatos de certificados suportados pode ser recuperada no módulo *signature*. A Listagem 3 exhibe um exemplo de certificado X.509

(decodificado) para um sítio (www.freesoft.org) emitido pela entidade certificadora Thawte Consulting.

```
Certificate:
Data:
  Version: 1 (0x0)
  Serial Number: 7829 (0x1e95)
  Signature Algorithm: md5WithRSAEncryption
  Issuer: C=ZA, ST=Western Cape, L=Cape Town, O=Thawte
Consulting cc,
  OU=Certification Services Division,
  CN=Thawte Server CA/emailAddress=server-
certs@thawte.com
  Validity
    Not Before: Jul  9 16:04:02 1998 GMT
    Not After : Jul  9 16:04:02 1999 GMT
  Subject: C=US, ST=Maryland, L=Pasadena, O=Brent Baccala,
  OU=FreeSoft,
  CN=www.freesoft.org/emailAddress=baccala@freesoft.org
  Subject Public Key Info:
    Public Key Algorithm: rsaEncryption
    RSA Public Key: (1024 bit)
      Modulus (1024 bit):
        00:b4:31:98:0a:c4:bc:62:c1:88:aa:dc:b0:c8:bb:
        33:35:19:d5:0c:64:b9:3d:41:b2:96:fc:f3:31:e1:
        66:36:d0:8e:56:12:44:ba:75:eb:e8:1c:9c:5b:66:
        70:33:52:14:c9:ec:4f:91:51:70:39:de:53:85:17:
        16:94:6e:ee:f4:d5:6f:d5:ca:b3:47:5e:1b:0c:7b:
        c5:cc:2b:6b:c1:90:c3:16:31:0d:bf:7a:c7:47:77:
        8f:a0:21:c7:4c:d0:16:65:00:c1:0f:d7:b8:80:e3:
        d2:75:6b:c1:ea:9e:5c:5c:ea:7d:c1:a1:10:bc:b8:
        e8:35:1c:9e:27:52:7e:41:8f
      Exponent: 65537 (0x10001)
    Signature Algorithm: md5WithRSAEncryption
    93:5f:8f:5f:c5:af:bf:0a:ab:a5:6d:fb:24:5f:b6:59:5d:9d:
    92:2e:4a:1b:8b:ac:7d:99:17:5d:cd:19:f6:ad:ef:63:2f:92:
    ab:2f:4b:cf:0a:13:90:ee:2c:0e:43:03:be:f6:ea:8e:9c:67:
    d0:a2:40:03:f7:ef:6a:15:09:79:a9:46:ed:b7:16:1b:41:72:
    0d:19:aa:ad:dd:9a:df:ab:97:50:65:f5:5e:85:a6:ef:19:d1:
    5a:de:9d:ea:63:cd:cb:cc:6d:5d:01:85:b5:6d:c8:f3:d9:f7:
    8f:0e:fc:ba:1f:34:e9:96:6e:6c:cf:f2:ef:9b:bf:de:b5:22:
    68:9f
```

Listagem 3. Exemplo de certificado X.509 emitido para um website

O módulo *digest* provê a facilidade de geração de *message digest* utilizando quaisquer implementações de algoritmos para *hashing*. A lista com identificadores

dos algoritmos suportados pode ser recuperada.

Tabela 7. Funções do módulo *digest*.

Módulo <i>digest</i>
digest:new (digest_algorithm: string) → digest: object Cria uma nova instância do módulo utilizando o algoritmo especificado
digest:generate (data: string) → message: string Gera um <i>message digest</i> dos dados para verificação de integridade
digest:listAlgorithms() → algorithms: table Recupera uma lista dos algoritmos suportados por este módulo

O módulo *cypher* disponibiliza funções para a cifragem e decifragem de dados baseado em chaves simétricas e assimétricas. A Tabela 8 traz a descrição das funções oferecidas pelo módulo.

Tabela 8. Funções do módulo *cypher*.

Módulo <i>cypher</i>
cypher:new (algorithm: string) → cypher: object Cria um novo objeto <i>cypher</i> utilizando o algoritmo especificado
cypher:encrypt (data, key: string [, key_password: string]) → encrypted_data: string Criptografa os dados <i>data</i> com uma chave <i>key</i>
cypher:decrypt (data, key: string [, key_password: string]) → decrypted_data: string Descriptografa os dados <i>data</i> com a chave <i>key</i>
cypher:listAlgorithms() → algorithms: table Recupera uma lista dos algoritmos suportados por este módulo
cypher:listCertificates() → certificates: table Recupera uma lista dos formatos de certificados suportados por este módulo

Caso o algoritmo especificado seja de criptografia simétrica, *key* será considerada uma chave simétrica. Caso contrário, *key* será considerada uma chave privada (para decifragem) ou um certificado (para cifragem). Se a chave especificada for uma chave simétrica ou chave privada, o parâmetro *key_password* pode ser utilizado para descriptografá-la caso necessário.

A API Security não restringe ou “amarra” algoritmos para as funcionalidades de encriptação, geração de chaves e verificação de integridade de dados. A definição dos algoritmos suportados em cada módulo é dependente da

implementação da API. Com essa estrutura é possível inclusive suportar esquemas de criptografia mais elaborados como o TLS (*Transport Layer Security*) (TLS, 2008), padrão IETF (*Internet Engineering Task Force*), e seu predecessor SSL (*Secure Socket Layer*) (SSL, 1996) desenvolvido pela Netscape Communications, que comumente se utilizam de certificados X.509 para transmitir dados seguros entre aplicações.

4.3.4 API Metadata

A API Metadata provê um mecanismo para a recuperação de informações sobre os serviços de TVD presentes no fluxo transmitido. As informações são requisitadas através de tratadores (ou ouvintes) registrados no objeto *metadata* e entregue de forma assíncrona.

A API foi modelada com requisito de genericidade com relação às informações oferecidas, fazendo com que o módulo possa aderir às especificações de SI dos diversos padrões independentemente da estrutura das tabelas. A API pode ser implementada para recuperar tabelas presentes em sistemas como DVB, ATSC e ARIB ou ainda ser utilizada para um esquema de metadados baseado em XML como é o caso do TV-Anytime. A arquitetura deste pacote funcional é mostrada na Figura 33.

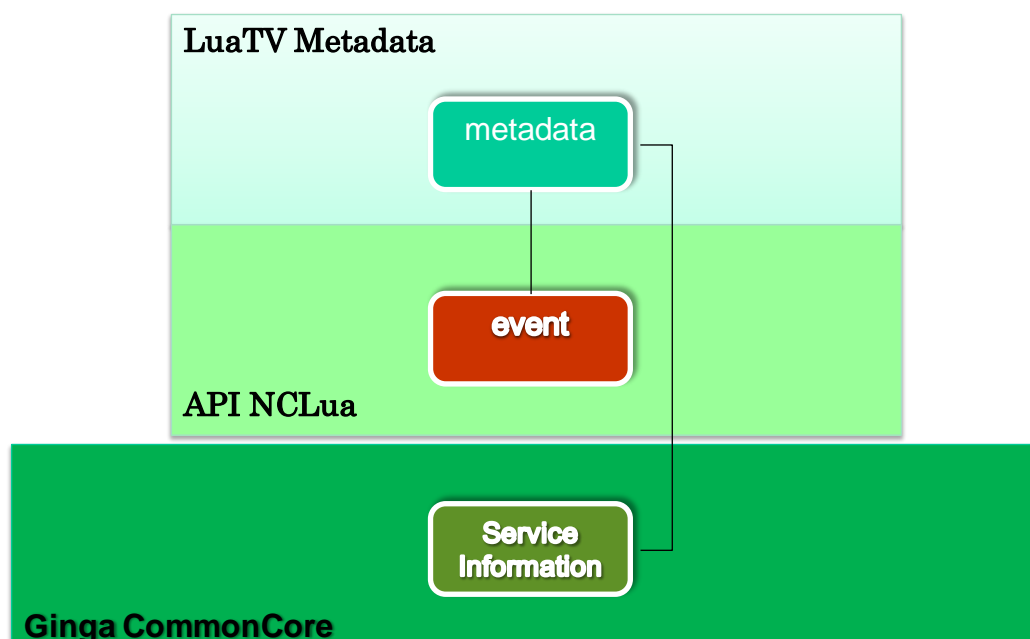


Figura 33. API Metadata com seu módulo *metadata* utilizando o módulo *event* e o componente *Service Information* do núcleo comum

Na arquitetura sugerida e nas implementações encontradas do Ginga, esta API necessita interagir com o componente *Service Information* presente no núcleo comum para a recuperação das informações junto ao *middleware*. Este componente faz parte da arquitetura sugerida na Norma ABNT NBR 15606-1 (ABNT 15606-1, 2009). Porém, a API é abstrata o suficiente para se adaptar a qualquer forma de transmissão de metadados.

Um único módulo *metadata* está presente onde podem ser registrados tratadores para recuperação dos dados desejados. É possível a recuperação de toda uma tabela (identificada através de uma string como “PAT”, “PMT”, “AIT”, ou um identificador numérico como “0x160”) ou de um campo específico de uma tabela. Outra facilidade presente neste módulo é a possibilidade de recuperação de dados em *cache* com a finalidade de evitar a espera por dados vindos no fluxo. A Tabela 9 exhibe as funcionalidades do módulo *metadata* em seguida a Figura 34 exhibe um exemplo de evento da classe ‘*metadata*’ lançado pelo módulo.

Tabela 9. Funções do módulo *metadata*.

Módulo cipher
metadata.register (f: function, table_id: string [, field: string, cached: boolean]) Registra uma função tratadora para recebimento de eventos com dados de <i>SI</i>
metadata.unregister (f: function) Desregistra a função tratadora

```
{ class = 'metadata', tableID = 'AIT',  
  field = 'application_type', data = '1' }
```

Figura 34. Exemplo de evento gerado pelo módulo *metadata*

A API Metadata foi especificada de tal maneira que torna possível uma implementação que suporte tanto o esquema de metadados rígidos baseados em PSI quanto padrões de metadados flexíveis baseados em XML (como o TV-Anytime), uma vez que, tabelas Lua podem encapsular ambos os formatos.

4.4 Equivalência funcional

Os pacotes funcionais da API LuaTV têm como objetivo complementar as funcionalidades da API NCLua oferecendo facilidades imperativas que não estão presentes no ambiente Ginga-NCL sem a utilização de recursos do ambiente

imperativo Ginga-J. A Tabela 10 apresenta um comparativo de funcionalidades oferecidas pelo ambiente Ginga-J, pelo Ginga-NCL (isoladamente) e pelo Ginga-NCL utilizando as extensões da API LuaTV.

Tabela 10. Equivalência funcional (Ginga-J x Ginga-NCL x LuaTV)

Funcionalidade	Ginga-J	Ginga-NCL	Ginga-NCL + LuaTV
Suporte a dispositivos apontadores (<i>mouse</i>)	Sim (API DTV UI)	Não	Sim (API Widget)
Suporte a criação de caixas-de-texto	Sim (API LWUIT)	Não	Sim (API Widget)
Suporte a múltiplos dispositivos de interação	Sim (API SBTVD)	Sim	Sim (API Multidevice)
Captura e utilização de fluxos de dados de dispositivos de interação	Sim (API SBTVD)	Não	Sim (API Multidevice)
Suporte e controle de segurança de dados	Opcional (APIs JSSE ¹² e JCE ¹³)	Não*	Sim (API Security)
Recuperação de informação sobre serviços televisivos	Sim (API JavaDTV Service)	Parcialmente, através do nó de <i>settings</i>	Sim (API Metadata)

* Dependente de suporte do *player* associado, não há controle direto por parte da aplicação.

¹² <http://java.sun.com/products/archive/jsse/>

¹³ <http://java.sun.com/products/archive/jce/>

5. Cenários de uso da API

Este capítulo descreve cenários de uso representativos e aplicações que podem tomar proveito dos pacotes funcionais presentes na API LuaTV. O objetivo é ilustrar aplicações que não seriam facilmente implementadas com a atual especificação do ambiente Ginga-NCL isoladamente.

5.1 *Desktop widgets*

A API Widgets pode ser útil no desenvolvimento de *desktop widgets*, mini-aplicativos geralmente utilizados na área de trabalho de ambientes gráficos de sistemas operacionais que poderiam também ser utilizados na tela de um televisor. Desta forma, o conceito seria modificado para TV *widgets*.

Alguns exemplos de *engines* para a criação e utilização de *desktop widgets* são: Dashboard (DASHBOARD, 2010) desenvolvido pela Apple; Konfabulator (KONFABULATOR, 2010) desenvolvido pela Yahoo!; e o Google Desktop Gadgets (GOOGLE, 2010). Em geral, esses tipos de *widgets* são visualmente compactos e podem ser “arrastados” em um esquema *drag and drop* para uma disposição arbitrária. São inúmeras as finalidades destes *widgets*, tais como, calendários, anotações rápidas, relógios, previsão do tempo, agenda, entre outras.

O desenvolvimento desses tipos de componentes visuais certamente é mais fácil com o uso da API Widgets, já que ela dá suporte à utilização de dispositivos de entrada como teclado e mouse para interação do usuário. Além dos *widgets* tradicionalmente encontrados, dois outros exemplos representativos de *widgets* poderiam ser desenvolvidos em NCL com extensões Lua e incorporados para utilização em TV:

- Componente gráfico para leitura de RSS (*Really Simple Syndication*): O formato de RSS *feed* é amplamente utilizado na *web* para compartilhar últimas notícias ou textos completos e até mesmo arquivos multimídia. Informações podem ser recuperadas através da classe de eventos *tcp* do módulo *event* e exibidas através do módulo *widget*.

- Componente gráfico para exibição de mapas: Um *widget* para exibição de mapas que o usuário possa interagir através do cursor de seu dispositivo. As imagens poderiam ser exibidas em nós de mídia NCLua. Imagens maiores do que a área designada para o NCLua podem ser utilizadas fazendo uso da funcionalidade de *scrolling*.

Todos estes *widgets* poderiam ser arrastados através de um dispositivo apontador. *Widgets* com entrada de texto, como anotações rápidas e agendas, também podem se beneficiar das funcionalidades da API LuaTV.

As Figuras 35 e 36 abaixo ilustram dois exemplos *widgets* implementados com a API proposta: o *widget* para exibição de mapas e o *widget* para anotações rápidas. O primeiro mostra a área geográfica afetada pelo recente vazamento de óleo da empresa British Petrol, o usuário pode interagir com seu dispositivo apontador rolando a imagem e realizando *zoom* através de duplo-clique. O segundo *widget* mostra o uso das facilidades de caixa-de-texto providas pelo módulo *textinput* e a utilização de *tooltips* oferecida pelo módulo *widget*.



Figura 35. *Widget* para exibição de mapas

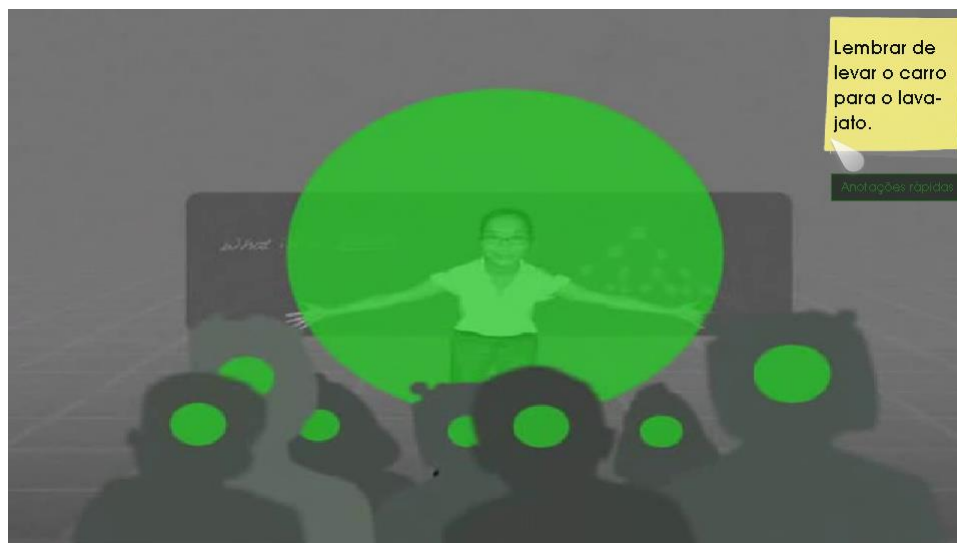


Figura 36. Widget para anotações rápidas

5.2 Pênalti Interativo

A aplicação Pênalti Interativo apresentada em (BATISTA, 2010) pôde ser adaptada sem maiores dificuldades para utilizar a API LuaTV Multidevice. A classe "ClasseAtivaGingaNCL" mostrada na subseção 4.3.2 é utilizada, disponibilizando um serviço para execução de documentos NCL nos dispositivos conectados. O dispositivo base inicia a execução de objetos de mídia NCL nos dispositivos secundários e oferece durante um período fixo de tempo a simulação de um pênalti de futebol.

A utilização da aplicação é simples, dois usuários participantes escolhem qual a direção do chute e para qual lado o goleiro irá pular. Existe a necessidade da identificação individual dos dispositivos, pois os jogadores exercem papéis distintos (batedor e goleiro). Após as escolhas dos usuários, uma animação é exibida no dispositivo base. A Figura 37 mostra a execução da aplicação em dois dispositivos e a animação resultante no dispositivo base.

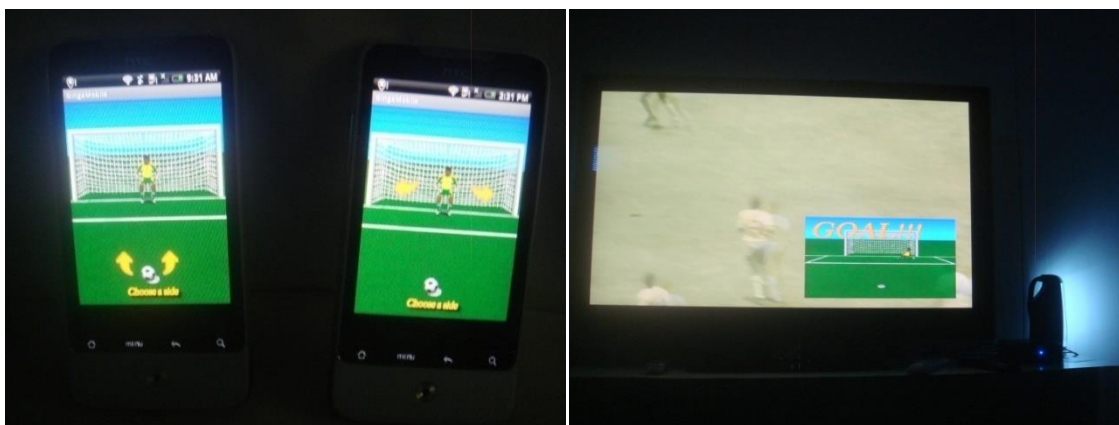


Figura 37. Execução da aplicação Pênalti Interativo em dois dispositivos secundários e no dispositivo base

5.3 Torcida Virtual (ToV)

A Torcida Virtual (TAVARES, et al., 2004) (ToV) é uma aplicação interativa que cria um conceito de espaços acústicos compartilhados. O objetivo é possibilitar que telespectadores interajam acusticamente durante a transmissão de eventos esportivos ou artísticos. A aplicação disponibiliza "setores" em que se podem escolher assentos para assistir ao evento. A localização do usuário é importante para determinar com quais outros usuários a interação por meio do áudio se dará. Na especificação da ToV foi definido que o áudio será ouvido pelos usuários sentados próximos virtualmente, implementando uma espécie de ambiente acústico virtual.

A usabilidade da aplicação é bem direta, o telespectador poderá escolher o setor que deseja ocupar, escolher um assento identificado por um número e posteriormente poderá "torcer" interagindo com os demais usuários. A Figura 38 apresenta o cenário de execução da aplicação mostrando o contexto em que ela está inserida, posteriormente serão listados os componentes de *software* que compõem a aplicação.

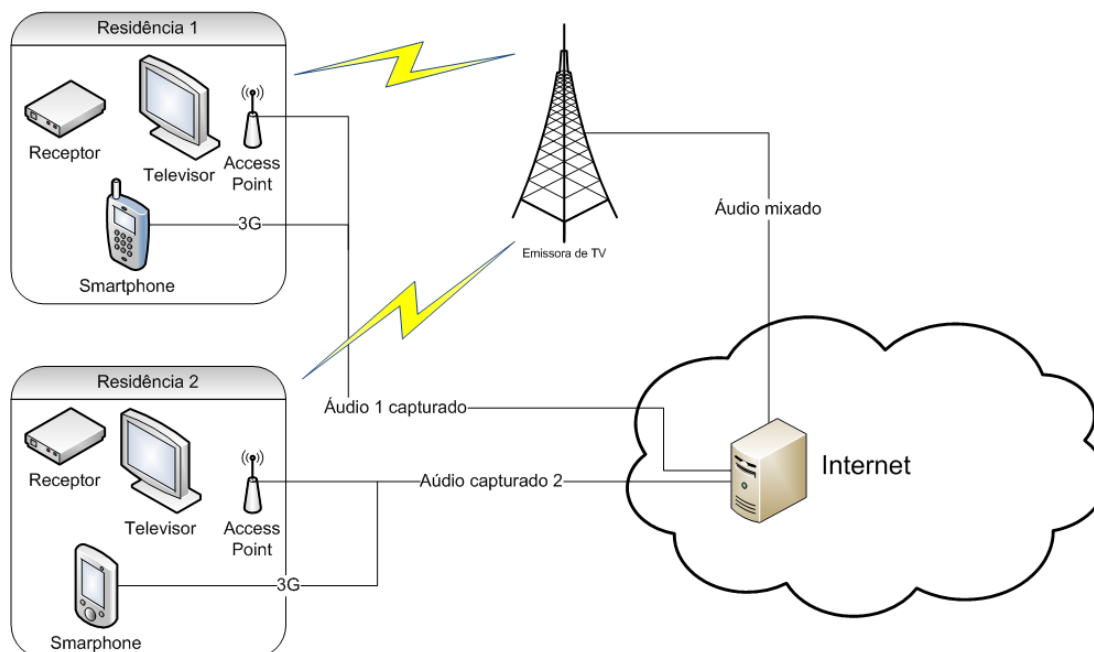


Figura 38. Cenário de execução da aplicação ToV

O desenvolvimento desta aplicação não é viável na atual especificação do ambiente declarativo do *middleware* Ginga sem o ambiente imperativo, uma vez que, a linguagem NCL não tem mecanismos para identificação individualmente dos dispositivos conectados e para recebimento de fluxos de dados transmitidos. Por outro lado, com as extensões da API LuaTV a implementação desta aplicação torna-se possível. Uma implementação desta aplicação no ambiente Ginga-NCL necessitaria dos seguintes componentes de *software*:

- Interface gráfica construída em linguagem NCL;
- *Scripts* NCLua para a comunicação com os dispositivos remotos através da API Multidevice;
- Aplicação cliente que será executada no dispositivo remoto (podendo ser desenvolvida para sistemas como Symbian ou Android) responsável por capturar o áudio e enviá-lo (através de rede 3G ou *wi-fi*) para um servidor na internet;
- *Software* que será executado no servidor e terá o papel de mixar os vários fluxos de áudio recebidos e posteriormente enviar o resultado

para ser retransmitido pela emissora.

5.4 Guias Eletrônicos de Programação (EPGs)

A API Metadata pode ser utilizada para o desenvolvimento de aplicações de EPG para exibição de informações sobre a programação transmitida. O autor de documentos NCL se beneficiaria amplamente das funcionalidades de recuperação de metadados na API LuaTV. *Scripts* NCLua podem ser utilizados para ter acesso a tabelas inteiras de forma assíncrona e posteriormente exibir os dados coletados no próprio *script* ou repassá-los para o documento NCL através do módulo *event*.

5.5 Transferência segura de dados

A segurança na manipulação de dados é um requisito presente em diversos cenários comumente encontrados. Uma série de aplicações incluindo aplicações financeiras (*T-Bank*, *T-Commerce*, etc.), aplicações de votação, aplicações de utilidade pública (como declaração de imposto de renda), aplicações exclusivamente para transferência de dados como, clientes SFTP (*Secure File Transfer Protocol*) e P2P (*Peer-to-Peer*) com utilização protocolos seguros, são apenas alguns exemplos de como a API Security pode ser explorada por aplicações NCL. Na especificação atual do ambiente declarativo, uma implementação destas aplicações puramente em NCL é inviável.

Assim como mencionado para aplicações EPG, o autor do documento NCL pode implementar todo o esquema de segurança necessário para sua aplicação em *scripts* NCLua e utilizar os dados da forma como desejar, enviando-os de volta para o NCL através do módulo *event* ou tratando-os diretamente em Lua.

6. Conclusão

Este trabalho apresentou a API LuaTV, uma proposta para extensão das funcionalidades disponíveis para o desenvolvedor de *scripts* Lua a serem embutidos em documentos NCL no ambiente Ginga-NCL. Este capítulo apresenta os resultados e contribuições obtidos durante a especificação da API, além dos possíveis trabalhos futuros e as considerações finais.

6.1 Resultados e contribuições

A especificação da API evoluiu ao longo do desenvolvimento do presente trabalho com constantes discussões e reuniões colocando em pauta sua finalidade e modelagem. Paralelamente à especificação da API, uma equipe de pesquisadores do laboratório LAViD iniciou um projeto para a implementação de referência da mesma. Todas as atividades deste projeto podem ser visualizadas através do ambiente de acompanhamento e participação do desenvolvimento de projetos relacionados ao GingaCDN. Além disso, todos os códigos gerados pela equipe de desenvolvimento podem ser acessados através de um serviço SVN (*Subversion*). A seguir encontra-se a listagem de todas as contribuições geradas ao longo do desenvolvimento do presente trabalho.

- Especificação completa da API LuaTV com seus quatro pacotes funcionais (presente no Apêndice A desta dissertação);
- Especificação de um componente de integração de dispositivos para o núcleo comum do *middleware* Ginga;
- Criação de um projeto público¹⁴ para acompanhamento da implementação de referência;
- Elaboração de cenários de uso e aplicações representativas para uso da API LuaTV;
- Disponibilização dos códigos desenvolvidos através do serviço de

¹⁴ <http://dev.openginga.org/projects/luatv-project>

versionamento SVN¹⁵;

- Disponibilização do código da interface do componente *Device Integration* no modelo de componentes FlexCM (presente no Apêndice B desta dissertação);

Adicionalmente, um artigo (BRANDÃO, et al., 2010) foi publicado na “*19th International Conference on Computer Communications and Networks (ICCCN 2010)*” promovido pela IEEE Communications Society e sediado na cidade de Zurique, Suíça, no ano de 2010. Este artigo aborda a especificação da API e também faz parte dos resultados obtidos.

6.2 Trabalhos futuros

O trabalho futuro mais direto será o encaminhamento da API para padronização no órgão ITU-T. Certamente novas discussões serão levantadas pelo Grupo de Trabalho GRN-6 que conseqüentemente contribuirão para o refinamento das funcionalidades contempladas na API. Algumas partes da API inclusive já fazem parte das contribuições do grupo como é o caso da API Widget que foi muito bem vista pelos pesquisadores e pelo coordenador do grupo.

Outro trabalho futuro surgiu com a realização de uma proposta de plano de doutorado para integração de múltiplos dispositivos de interação com aplicações de captura e acesso (C&A). As chamadas aplicações de captura e acesso exploram o paradigma de Computação Ubíqua para dar apoio à captura automática de informação em experiências ao vivo e à correspondente geração de documentos passíveis de armazenamento, recuperação, visualização e extensão. Tipicamente, o foco destas aplicações abrange desde a captura das informações de eventos como reuniões, palestras e aulas, até procedimentos médicos ou potencialmente qualquer outro evento que seja interessante recuperar e disponibilizar as informações neles abordadas. A idéia é realizar a captura das informações através de computadores e dispositivos de entrada e saída, tais como câmeras, microfones, tinta digital (*ink*), e sensores diversos. Posteriormente, as informações capturadas são disponibilizadas para acesso através de um documento hipermídia (como documentos NCL). Desta forma, permite-se que os usuários se concentrem exclusivamente nas informações

¹⁵ <https://svn.lavid.ufpb.br/svnroot/luatv/>

passadas e não na sua captura, como é realizado atualmente, por exemplo, nas salas de aula tradicionais.

A integração sugerida mostra-se interessante, pois levantará requisitos em uma nova ótica já que a API será empregada em um contexto diferente do ambiente de TV Digital. A utilização da API servirá para refinar suas funcionalidades, bem como guiar parte do desenvolvimento da sua implementação de referência. Este plano de trabalho com a integração proposta foi aceito como parte dos requisitos para admissão do autor desta dissertação no curso de doutorado da PUC-Rio.

6.3 Considerações finais

Este trabalho propôs a especificação de uma API para expandir o escopo de desenvolvimento de aplicações NCLua no ambiente declarativo do *middleware* Ginga. As funcionalidades contempladas foram levantadas de acordo com requisitos do ponto de vista de um autor de documentos NCL sem acesso a recursos disponibilizados pelo ambiente imperativo do *middleware* Ginga. Aplicações e cenários de uso foram elaborados para ilustrar as possibilidades de utilização da API proposta. As funcionalidades oferecidas pela API tornam possível o desenvolvimento de certas aplicações outrora impossíveis de serem especificadas isoladamente no ambiente Ginga-NCL.

7. Referências bibliográficas

ABNT NBR 15606-1. 2008. *Televisão digital terrestre — Codificação de dados e especificações de transmissão para radiodifusão digital – Parte 1: Codificação de dados.* 2008.

ABNT NBR 15606-2. 2009. *Televisão digital terrestre – Codificação de dados e especificações de transmissão para radiodifusão digital – Parte 2: Ginga-NCL para receptores fixos e móveis – Linguagem de aplicação XML para codificação de aplicações.* 2009.

ABNT NBR 15606-5. 2009. *Televisão digital terrestre — Codificação de dados e especificações de transmissão para radiodifusão digital – Parte 5: Ginga-NCL para receptores portáteis – Linguagem de aplicação XML para codificação de aplicações.* 2009.

ALVES, L. G. P.; KULESZA, R.; SILVA, F. S. da; JUCÁ, P.; BRESSAN, G. 2006. Análise Comparativa de Metadados em TV Digital. *Brazilian Symposium on Computer Networks (II Digital TV Workshop)* – SBRC. 2006.

ARIB. 2004. Association of Radio Industries and Businesses. *ARIB STD-B23 Version 1.1: Application Execution Engine Platform for Digital Broadcasting (English Translation).* 2004.

ARIB. 2009. Association of Radio Industries and Businesses. *ARIB STD-B23 Version 1.2: Application Execution Engine Platform for Digital Broadcasting (English Translation).* 2004.

AUN, F. 2007. Brazil, Russia, India and China to Lead Internet Growth Through 2011. *ClickZ: News and expert advice for the digital marketer.* [Online] 2007. <http://www.clickz.com/3626274>

BATISTA, C. E. C. F.; SOARES, L. F. G.; SOUZA FILHO, G. L. 2010. Estendendo o uso das classes de dispositivos Ginga-NCL. *Brazilian Symposium on Multimedia and the Web – WebMedia.* 2010.

- BRANDÃO R. R. de M.; BATISTA, C. E. C. F.; SOARES, L. F. G.; SOUZA FILHO, G. L. 2010.** *Extended features for the Ginga-NCL environment - Introducing the LuaTV API*. Proceedings of the 19th International Conference on Computer Communication Networks (2nd Workshop on Multimedia Computing and Communications). 2010 (To Appear)
- DASHBOARD, 2010.** Dashboard Widgets. *Amazing widgets for your Mac OS X Dashboard*. [Online] 2010. <http://www.apple.com/downloads/dashboard/>
- DVB. 2003.** Digital Vídeo Broadcasting (DVB). *Digital video broadcasting (DVB) multimedia home platform (MHP)*. Padrão ETSI TS 102 812, ETSI, 2003
- FREIRE FILHO, S. L. de M. 2008.** *FlexCM: Um Modelo de Componentes para Sistemas Adaptativos*. Departamento de Informática, Universidade Federal da Paraíba. 2008. p. 74, Dissertação (Mestrado).
- GOOGLE, 2010.** Google Desktop Gadgets. *Google Desktop Gadget API*. [Online] 2010. <http://code.google.com/intl/en/apis/desktop/docs/gadgetapi.html>
- IERUSALIMSKY, Roberto. 2006.** *Programming in Lua*. 2. ed. Rio de Janeiro. s.l. : Lua.org, 2006. p. 308.
- ISO/IEC 13818-1. 2001.** *Information technology — Generic coding of moving pictures and associated audio information: Systems*. 2009.
- ITU. 2001.** *ITU-T Recommendation J.200: Worldwide common core - Application environment for digital interactive television services*. 2001.
- ITU. 2004.** *ITU-T Recommendation J.201: Harmonization of declarative content format for interactive television applications*. 2004.
- ITU. 2003.** *ITU-T Recommendation J.202: Harmonization of procedural content formats for interactive TV applications*. 2003.
- ITU. 2008.** *ITU-T Recommendation X.509: Information technology – Open systems interconnection – The Directory: Public-key and attribute certificate frameworks*. 2008
- ITU. 2009.** *ITU-T Recommendation H.761: Nested context language (NCL) and Ginga-NCL for IPTV services*. 2009

JAVATV. 2010. Java ME Technology - Java TV API. [Online] 2010.
<http://java.sun.com/javame/technology/javatv/>

JAVATV API. 2008. Overview (JavaTV API1.1). [Online] 2008.
<http://java.sun.com/javame/reference/apis/jsr927/>

JAVA SECURITY. 2010. Java SE Security. [Online] 2010.
<http://java.sun.com/javase/technologies/security/>

KLYNE, G.; REYNOLDS, F. ; WOODROW, C.; OHTO, H.; HJELM, J.; BUTLER, M. H., TRAN, L. 2004. *Composite Capability/Preference Profiles (CC/PP): Structure and vocabularies*. W3C working draft, 2004.

KONFABULATOR, 2010. Yahoo! Widgets. *Konfabulator Tools and Documentation*. [Online] 2010. <http://widgets.yahoo.com/tools/>

LEITE, L. E. C.; SOUZA FILHO, G. L. de; BATISTA, C. E. C. F.; KULESZA, R.; BRESSAN, G.; ALVES, L. G. P.; RODRIGUES, R. P; SOARES, L. F. G. 2005.
FlexTV - Uma Proposta de Arquitetura de Middleware para o Sistema Brasileiro de TV Digital. *Revista Engenharia de Computação e Sistemas Digitais*. 2005, Vol. 2, pp. 29-50.

LUA BIT. 2007. Bitwise operation library in Lua. [Online] 2007.
<http://luaforge.net/projects/bit>

LUA LASH. 2010. Lua Hashing Library. [Online] 2010.
<http://code.google.com/p/lash-lua/>

LUA MD5. 2010. MD5 Cryptographic Library for Lua. [Online] 2010.
<http://www.keplerproject.org/md5/>

LUAONTV. 2008. LuaOnTV Project Info. [Online] 2008.
<http://luaforge.net/projects/luaontv/>

MORRIS, S.; SMITH-CHAIGNEAU, A. 2005. *Interactive TV Standards – A Guide to MHP, OCAP and JavaTV*. s.l. : Elsevier, Focal Press, 2005.

- NIST. 1994.** National Institute of Standards and Technology. *Federal Information Processing Standards Publication (FIP186): Digital Signature Standard*. [Online] 1995. <http://www.itl.nist.gov/fipspubs/fip186.htm>
- NIST. 1995.** National Institute of Standards and Technology. *Federal Information Processing Standards Publication (FIP180-1): Secure Hash Standard*. [Online] 1995. <http://www.itl.nist.gov/fipspubs/fip180-1.htm>
- NIST. 1999.** National Institute of Standards and Technology. *Federal Information Processing Standards Publication (FIP46-3): Data Encryption Standard*. [Online] 1999. <http://csrc.nist.gov/publications/fips/fips46-3/fips46-3.pdf>
- NIST. 2001.** National Institute of Standards and Technology. *Federal Information Processing Standards Publication (FIP197): Advanced Encryption Standard*. [Online] 2001. <http://www.csrc.nist.gov/publications/fips/fips197/fips-197.pdf>
- RDF, 2004.** Resource Description Framework (RDF). *Semantic Web Standards*. [Online] 2010. <http://www.w3.org/RDF>
- RIVEST, R. 1992.** The MD5 Message-Digest Algorithm. *Request for Comments: 1321*. [Online] 1992. <http://tools.ietf.org/html/rfc1321>
- RIVEST, R.; SHAMIR, A.; ADLEMAN, L. 1983.** *Cryptographic Communications System and Method*. Cambridge, MA: U.S. Patent 4,405,829, 1983.
- SANT'ANNA, F., CERQUEIRA, R.; SOARES, L. F. G. 2008.** NCLua - Objetos Imperativos Lua na Linguagem Declarativa NCL. *Brazilian Symposium on Multimedia and the Web – WebMedia*. 2008.
- SCHWALB, E. M. 2003.** *iTV Handbook: Technologies and Standards*. s.l. : Prentice Hall PRT, 2003.
- SILVA, L. D. N.; BATISTA, C. E. C. F.; LEITE, L. E. C.; SOUZA FILHO, G. L. de. 2007.** Suporte para desenvolvimento de aplicações multiusuário e multidispositivo para TV Digital com Ginga. *T&C Amazônia Magazine*. N. 12, 2007, pp. 75-84.
- SILVA, L. D. N. 2008.** *Uma Proposta de API para Desenvolvimento de Aplicações Multiusuário e Multidispositivo para TV Digital Utilizando o Middleware Ginga*.

Departamento de Informática, Universidade Federal da Paraíba. 2008. p. 77, Dissertação (Mestrado).

SMIL, 2010. Synchronized Multimedia Integration Language (SMIL). *W3C Synchronized Multimedia Home page*. [Online] 2010. <http://www.w3.org/AudioVideo>

SOARES, L. F. G., RODRIGUES, R. F. 2006. Nested Context Language 3.0 Part 8 - NCL Digital TV Profiles. *Technical Report*. Departamento de Informática da PUC-Rio, MCC 35/06. <http://www.ncl.org.br/documentos/NCL3.0-DTV.pdf>

SOARES, L. F. G. 2006. MAESTRO: The Declarative Middleware Proposal for the SBTVD. *Proceedings of the 4th European Interactive TV Conference*. 2006.

SOARES, L. F. G.; RODRIGUES, R. F.; MORENO, M. F. 2007. Ginga-NCL: the Declarative Environment of the Brazilian Digital TV System. *Journal of the Brazilian Computer Society*, v. 12, p. 37-46, 2007.

SOARES, L. F. G.; COSTA, R. M. de R.; MORENO, M. F. 2009. Ginga-NCL: Suporte a Múltiplos Dispositivos. *Brazilian Symposium on Multimedia and the Web – WebMedia*. 2009.

SOUZA FILHO, G. L. de; LEITE, L. E. C.; BATISTA, C. E. C. F. 2007. Ginga-J: The Procedural Middleware for the Brazilian Digital. *Journal of the Brazilian Computer Society*, v. 12, p. 47-56, 2007.

SOUZA JÚNIOR, P. J. 2009. *LuaComp: Uma ferramenta de autoria de aplicações para TV digital*. , Departamento de Engenharia Elétrica, Universidade de Brasília. 2008. p. 143, Dissertação (Mestrado).

SSL. 1996. The SSL Protocol Version 3.0. *DRAFT302 - The SSL Protocol Version 3.0*. [Online] 2010. <http://www.mozilla.org/projects/security/pki/nss/ssl/draft302.txt>

SUN. 2009. Java DTV API 1.0 Specification. s.l.: Sun Microsystems, 2009.

TAVARES, T. A.; BURLAMAQUI, A.; ALBINO, D. A. da S.; SIMONETTI, C.; LEITE, L. E. C.; FERNANDES, J. H. C.; SOUZA FILHO, G. L. de. 2004. Sharing Virtual Acoustic Spaces over Interactive TV Programs – Presenting “Virtual Cheering” Application. *ICME 2004*. Taipei, Taiwan : s.n., 2004.

TLS, 2008. The Transport Layer Security (TLS) Protocol Version 1.2. *RFC 5246 - The Transport Layer Security (TLS) Protocol Version 1.2*. [Online] 2010. <http://tools.ietf.org/html/rfc5246>

TV-Anytime, 1999. TV Anytime (TV-A). *The TV Anytime Forum*. [Online] 2010. <http://www.tv-anytime.org>

UAPROF, 2001. WAG UAProf. *Technical Report*. WAP-248-UAPROF-20011020-a. 2001.

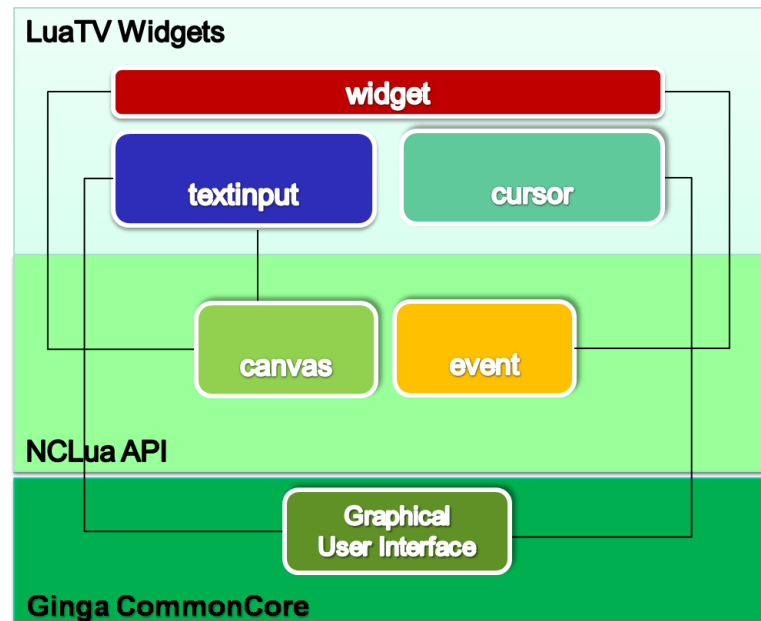
UPNP, 2010. Universal Plug and Play. *Universal Plug and Play Forum*. [Online] 2010. <http://www.upnp.org>

XHTML, 2010. eXtensible Hypertext Markup Language (XHTML). *XHTML2 Working Group Home Page*. [Online] 2010. <http://www.w3.org/MarkUp>

APÊNDICE A: Especificação da API LuaTV

A1. Widgets API

The Widgets API is used to provide widget related functionalities to the NCLua environment, since the NCL language supports almost all kind of graphical interface composition, this API focuses mainly on user interaction (I/O).



A1.1 widget object

The `widget` object is responsible for filling an optional given canvas (commonly associated to a NCL region) with a graphical representation and also for posting input events to the registered handlers.

A1.1.1 Constructors

`widget:new () → widget: object`

Creates a new `widget` object.

Return values:

`widget` - the object representing a new `widget`.

`widget:new (canvas: object, textinput: object) → widget: object`

Creates a new `widget` object.

Arguments:

`canvas` - a `canvas` object to be associated with this `widget`. This canvas can be larger than the defined NCLua region, therefore it is possible to scroll over it.

`textinput` - a `textinput` object that will be handled by this `widget`.

Return values:

`widget` - the object representing a new `widget`.

A1.1.2 Attributes

The attribute methods are used to access and modify the attributes' values (with some exceptions specified). Upon calling the method without parameters, the current attribute value is retrieved and returned. When the method is called with input parameters, they must be set as the new attribute values.

`widget:attrCanvas (canvas: object)`

Defines the canvas to be associated with this graphical component.

Arguments:

canvas - a canvas object that will display the graphical representation of this component.

widget:attrCanvas () → canvas: object

Retrieves the canvas associated with this widget.

Return values:

canvas - a canvas object used to display the graphical representation of this widget.

widget:attrTextInput (textinput: object)

Defines the `textinput` object to be associated with this widget.

Arguments:

cursor - a `textinput` object that will be used to input text by the user in this component.

widget:attrTextInput () → textinput: object

Retrieves the `textinput` object associated with this widget.

Return values:

cursor - the `textinput` object used to display the entered text by the user in this widget.

A1.1.3 Event handler

widget:register (f: function)

Registers a new user event listener for this `widget` component. The registered function will receive an event of the class 'widget' when the component's associated `textinput` or `cursor` objects fire any event (see `cursor:register()` and `textinput:register()` for all possible event types).

Arguments:

f - the function to be registered as event listener for this widget component.

f should be a function with the following signature:

```
function f (evt: table)
  -- returns boolean end
end
```

The parameter `evt` is a Lua table containing information about the event, for example:

```
{class='widget',src='textinput',type='key_press', key=65,
modifiers={alt=true, ctrl=true} }
```

widget:unregister (f: function)

Removes the handler from the event listener list

Arguments:

f - the function previously registered as event listener for this widget

A1.1.4 Miscellaneous

widget:scroll(x,y: number)

Scrolls the widget's canvas to a given x,y positions by composing it along with the NCLua default canvas.

Arguments:

x, y - number values for scrolling the associated canvas.

widget:addTooltip(x,y,w,h: number, msg: string) → tooltipID: number

Adds a tooltip to be displayed when the cursor stops over the specified area.

Arguments:

- x, y, w, h - number values defining the area of this tooltip.
- msg - a string containing the message to be displayed by this tooltip.

Return values:

- tooltipID - a unique number value for this tooltip identification

widget:removeTooltip(tooltipID: number)

Remove the tooltip identified by `tooltipID`.

Arguments:

- tooltipID - a unique number value identifier for a tooltip

A1.2 cursor object

The `cursor` object is responsible for offering a pointer cursor (associated to an image) facility, to be exhibited over an NCLua canvas. This object will fire six types of 'widget' class events: 'cursor_over', 'cursor_in', 'cursor_out', 'button_press', 'button_release' and 'button_double_click', which carry the values of positioning for the event. The 'button_press', 'button_release' and 'button_double_click' events will also encapsulate a 'button' field with a number value specifying which of the pointer's button is involved (identifying, in case of a mouse pointer, the left, right and middle buttons, for instance).

A1.2.1 Constructors

This module is automatically created for each NCLua script executed, so it's always available for the Lua developer using LuaTV, so it may have many instances (i.e. one cursor instance for each NCLua media node). Although normally only one cursor pointer is available and displayed by the system, NCLua scripts can independently customize it while inside of its exhibiting canvas.

A1.2.2 Attributes

The attribute methods are used to access and modify the attributes values (with some exceptions specified). Upon calling the method without parameters, the current attribute value is retrieved and returned. When the method is called with input parameters, they must be set as the new attribute values.

cursor:attrImage(path: string)

Sets the cursor image source path to a given string value, and thus replaces the image being used to represent the cursor.

Arguments:

- url - a string containing the image source path.

cursor:attrImage()→url:string

Returns the cursor image path.

Return Values:

- url - a string containing the content source path for the cursor image.

cursor:attrPosition(x,y:number)

Sets the position of the cursor on the given x,y positions over the screen. Sends up an event of the class 'widget', with the type 'cursor_over' and the number values for the vertical and horizontal positions.

Arguments:

- x, y - number values for the position on the canvas.

cursor:attrPosition()→x,y:number

Returns the position of the cursor on the given number position over the cursor canvas.

Return Values:

x, y - current number values for the position on screen.

A1.2.3 Event handler

cursor:register (f: function)

Registers a new user event listener for a cursor. The registered function will receive events of the class 'widget' when the cursor changes its position and when it is clicked (as described on the *attrPosition* and *click* methods).

Arguments:

f - the function to be registered as event listener for this cursor widget.

f should be a function with the following signature:

```
function f (evt: table)
  -- returns boolean end
end
```

The parameter `evt` is a Lua table containing information about the event, for example:

```
{ class='widget', src='cursor', type='cursor_over', x=12,
  y=105 }
```

The valid values for the `type` field of the events are 'cursor_over', 'cursor_in', 'cursor_out', 'button_press', 'button_double_click' and 'button_release'. The 'button_press', 'button_double_click' and 'button_release'. The events encapsulate a 'button' field defining which of the pointer's button was related in this event, the values are numbers (0 for left button, 1 for middle button, 2 for right button, on a mouse pointer, for instance).

cursor:unregister (f: function)

Removes the handler from the event listener list

Arguments:

f - the function previously registered as event listener for this widget

A1.2.4 Miscellaneous

cursor:click(button: number)

Clicks on the current position the cursor is over and sends up two events of the class 'widget', with *types* 'button_press' and 'button_release' encapsulating the number values for the vertical and horizontal positions.

Arguments:

button - the button to be clicked

cursor:press(button: number)

Presses the specified button on the current position the cursor is over and sends up a 'button_press' event of the class 'widget'

Arguments:

button - the button to be pressed

cursor:release(button: number)

Released the specified button and sends up a 'button_release' event of the class 'widget'

Arguments:

button - the button to be pressed

A1.3 textinput object

The `textinput` object is responsible for offering a mechanism to capture text from user input so that it can be

eventually dispatched to a `widget` object.

A1.3.1 Constructors

`textInput:new ()→ textInput: object`

Returns a new `textInput` object which allow the user to input text. This object will fire the events generated by user input interaction to the registered handlers.

Return values:

`textInput` - a new `textInput` object

`textInput:new (canvas: object)→ textInput: object`

Returns a new `textInput` object allowing the user to handle text input. This object will fire the events generated by user input interaction to registered handlers. The `canvas` object parameter can also be used to display the captured text.

Arguments:

`canvas` - the `canvas` object to be associated with this widget, all entered text will be displayed over this `canvas`' defined configuration (i.e. color, font, etc)

Return values:

`textInput` - a new `textInput` object

A1.3.2 Attributes

`textInput:attrCanvas(canvas: object)`

Sets the `canvas` object related to this `textInput`, in order to display the entered text on the `canvas`.

Arguments:

`canvas` - a `canvas` object to be associated with this `textInput`.

`textInput:attrCanvas()→ canvas: object`

Retrieves the associated `canvas` object of this object

Return values:

`canvas` - a `canvas` object to be associated with this `textInput`.

`textInput:attrText(text: string)`

Sets this object's text content.

Arguments:

`text` - a string parameter containing arbitrary text for this `textInput`.

`textInput:attrText() → canvas: object`

Retrieves this object's text content.

Arguments:

`text` - a string parameter containing arbitrary text for this `textInput`.

A1.3.3 Event handler

`textInput:register (f: function)`

Registers a new user event listener for this `textInput`.

Arguments:

`f` - the function to be registered as event listener for this widget

`f` should be a function with the following signature:

```
function f (evt: table)
  -- returns boolean end
end
```

The parameter `evt` is a Lua table containing information about the event, for example:

```
{class='widget',    src='textinput',    type='key_press',    key=60,
modifiers={alt=true, ctrl=true} }
```

The possible values for the 'type' field are 'key_press' and 'key_release'. The 'key' field will encapsulate the code associated to the key been pressed or released, the 'modifiers' table will carry any of the modifiers keys been pressed at the same time.

textinput:unregister (f: function)

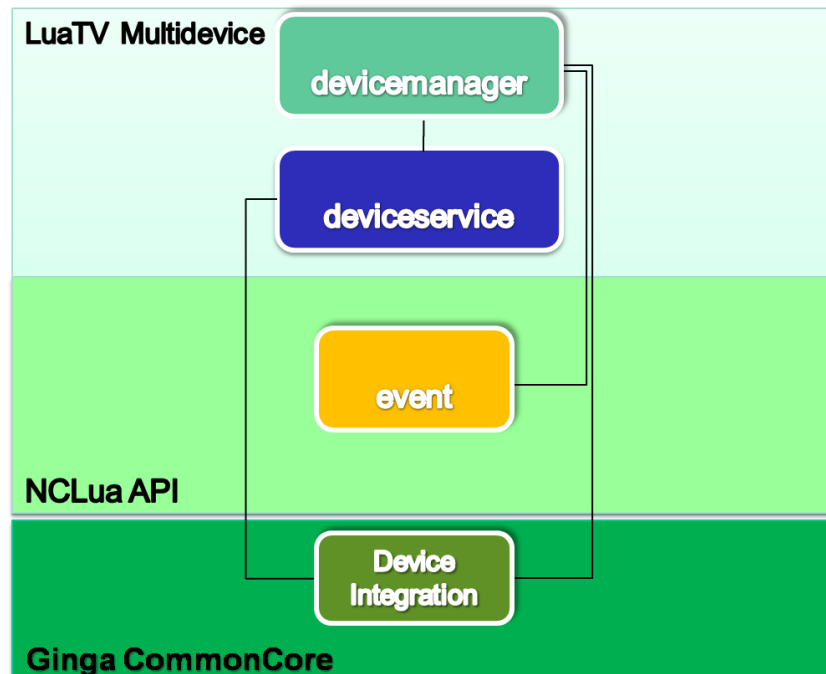
Removes the specified handler from the event handler list.

Arguments:

- f - the function previously registered as event listener for this widget

A2. Multidevice API

The two modules that are part of the Multidevice API are related to the usage of resources located on remote devices connected to the application host device. The `devicemanager` module is used to handle device class registration, device retrieving and service instantiation, while the `deviceservice` is used to request/send data to a group of devices of a given class/service or a single device.



A2.1 devicemanager object

The `devicemanager` is an object responsible for managing devices and device classes*. It's possible to retrieve a list of devices of a given class and also the list of all registered classes. Another facility provided by this module is the retrieving of classes and devices metadata conveniently represented as a Lua table. A service of a given class can be instantiated so the user can request/send data for a group of devices that support such service; this is accomplished through the `deviceservice` module.

* The classes used in this API are represented using the User Agent Profile (UAProf) which relies on the Resource Description Framework (RDF). RDF is defined by a family of World Wide Web Consortium (W3C) specifications. It has come to be used as a general method for conceptual description or modeling of information that is implemented in web resources, using a variety of syntax formats.

A2.1.1 Constructor

This module is automatically created when an NCLua script is executed, so it's always available to the Lua developer. Only one instance of the `devicemanager` is created and made available, even in different NCLua scripts contexts.

A2.1.2 Attributes

`devicemanager:registerClass(device_class_path : string)`

Registers a new device class from a UAProf RDF format XML file path

Arguments:

`device_class_path` - path to a XML file using the UAProf format specification

`devicemanager:listDeviceClasses()` → `classes: table`

Returns a table containing a list of all registered classes in the host device

Return values:

classes - a table listing all the classes' identifiers

devicemanager:listDevices(class_id: string) → devices: table

Returns a table containing a list of all connected devices of a given class_id

Arguments:

class_id - string identifying the class to retrieve the connected devices

Return values:

devices - a table listing all connected devices of the specified class

A2.1.3 Event handler

devicemanager.register(f: function)

Registers a new user event listener for this devicemanager. The registered function will receive an event of the class 'multidevice' when a device join or leave a class.

Arguments:

f - the function to be registered as event listener for this remote manager.

f should be a function with the following signature:

```
function f (evt: table)
  -- returns boolean end
end
```

The parameter evt is a Lua table containing information about the event, for example:

```
{class='multidevice', type='join_class', class='class_id',
device='192.168.1.101' }
```

The possible types for this event are: 'join_class' and 'leave_class'

devicemanager.unregister(f: function)

Removes the handler from the event listener list

Arguments:

f - the function previously registered as event listener for this remote manager.

A2.1.4 Miscellaneous

devicemanager.classMetadata(device_class_id: string) → class_metadata: table

Returns a table with the metadata of a previously registered class identified by class_id

Arguments:

device_class_id - a string with a device class identifier

Return values:

class_metadata - a table with the device class metadata accordingly to the UAProf profile used in registration

devicemanager.deviceMetadata(device_id: string) → device_metadata: table

Retrieves the metadata associated to the device identified by the string device_id.

Arguments:

device_id - a string identifying a device.

Return values:

device_metadata - a table containing the device metadata

devicemanager.newDeviceService(device_class, service: string) → deviceservice: object

Instantiates a new `deviceservice` so the user can perform requests for the group of devices associated to a class supporting such service.

Arguments:

device_class - a string identifying a device class.

service - a string identifying a service of the given device_class.

Return values:

deviceservice - a new `deviceservice` instance for the specified service.

A2.2 deviceservice object

The `deviceservice` is the object responsible for requesting data from a service. Data is received in an asynchronous approach through registered handlers.

A2.2.1 Miscellaneous

deviceservice:request(request: string, f: function [, device_id: string])

Requests data from a service, being the data delivered to the given function `f` handler.

Arguments:

request - a string with a service request (the semantics of this request is service dependent).

f - a handler function that will receive the request's response data

device_id - an optional string identifying a single device

`f` should be a function with the following signature:

```
function f (evt: table)
  -- returns boolean end
end
```

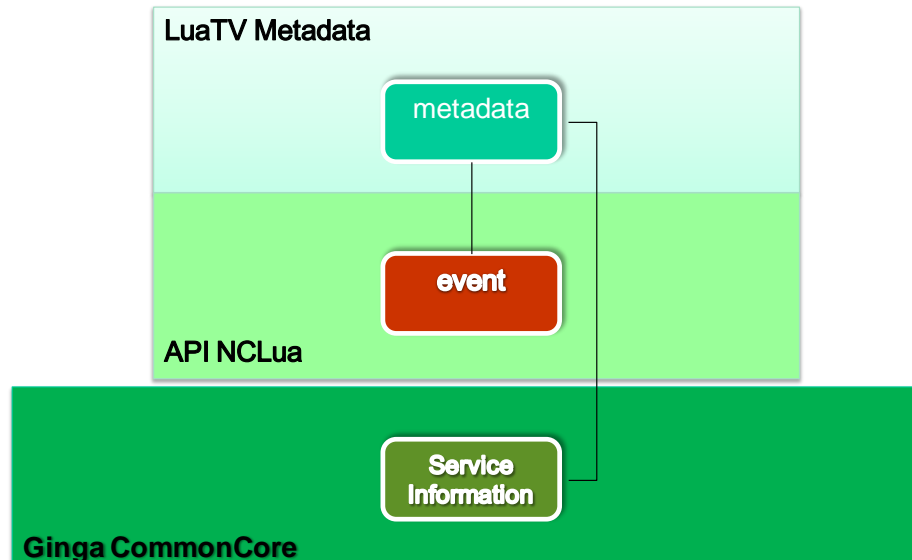
The parameter `evt` is a Lua table containing information about the event, for example:

```
{class='multidevice', type='data', deviceclass='class_id',
service='service_id', device='192.168.1.101', data={someData}
}
```

Only the 'data' type is possible for this kind of event.

A3. Metadata API

The Metadata module is used to retrieve information about services present in the stream. Data is requested through listeners registered on `metadata` object and is delivered in an asynchronous fashion. Alternatively, the optional *boolean* parameter *cached* can be flagged so that results (previously stored in cache) are returned immediately.



A3.1 metadata object

The `metadata` object is responsible for retrieving service information (e.g. now/next info, parental rating, or any other service metadata present on stream).

A3.1.1 Constructors

This module is automatically created when an NCLua script is executed, so it is always available for the Lua developer using LuaTV. Only one instance of the `metadata` is created, which may be accessed by different NCLua scripts.

A3.1.2 Event handler

`metadata:register (f: function, tableID: string [, field: string, cached: boolean])`

Registers a new metadata event listener. The registered function will receive an event of the class 'metadata' when the requested metadata is ready. Optional parameter *cached* can be used to retrieve data from cache instead of waiting new data on stream.

Arguments:

f - the function to be registered as event listener.

f should be a function with the following signature:

```
function f (evt: table)
  -- returns boolean end
end
```

The parameter `evt` is a Lua table containing information, for example: { `class='metadata'`, `tableID='AIT'`, `field='application_type'`, `data='1'` }

`tableID` - a string representing the table identification, for example: '17' for a table with such integer id or equivalently '0x11' in hexadecimal, also mnemonics like 'AIT', 'PAT', 'PMT' can be used.

`field` - a optional string specifying that only this field should be retrieved from the table. The data field from the `evt` table will carry the requested value as a string; if no field is specified on registration then data will

contain the entire requested table (as a Lua table).

cached - a optional boolean defining that (whenever possible) user wants a cached version of the information,.

metadata:unregister (f: function)

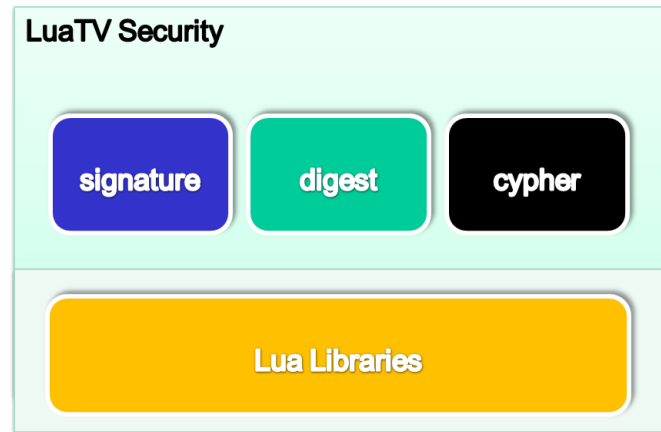
Removes the handler from the event listener list

Arguments:

f - the function previously registered as event listener for this widget

A4. Security API

The objects that are part of the Security module are responsible for offering functionalities related to data security, such as digital signature generation and verification, message digest generation and data encryption. The implementation of this API could extend other security libraries available in Lua (as pictured below).



A4.1 signature object

The signature is an object responsible for the generation and verification of digital signatures.

A4.1.1 Constructor

signature:new (digital_signature_algorithm: string)→ signature: object

Returns a new signature object, using the digest algorithm identified by the given string. If digital_signature_algorithm is not support a null object shall be returned.

Arguments:

digital_signature_algorithm – string identifier for the algorithm to be used in order to to generate and verify signatures.

Return values:

signature – a new object signature using a specific digest algorithm.

A4.1.2 Miscellaneous

signature:sign (data, key: string [, key_password: string]) → signed_data: string

Generates a signature for a set of bytes contained in a string.

Arguments:

data – the bytes from the container (from a file, a string) of bytes to be signed.

key – string containing the a valid key, using the encrypted format defined by the algorithm defined..

key_password – an optional string containing a password to decrypt the private key.

Return values:

signed_data – a string containing the generated signature. An empty string indicates an error.

signature:verify(data, signature: string [, certificate: string])→ b: boolean

Used to verify a signature based on the input data and signature, optionally a public key contained in a given certificate may be used.

Arguments:

data – the bytes from the container (a file, a string) of bytes, used as input data for the verification.

signature – a string containing the signature to be verified.

certificate – an optional path for a certificate containing the public key to be used to verify the signature.

Return values:

b – true if verified, false if not.

signature:listAlgorithms() → algorithms: table

Returns a table listing all algorithms supported by the module.

Return values:

algorithms – a table with all supported algorithms.

signature:listCertificates() → certificates: table

Returns a table listing all certificates formats supported by the module.

Return values:

certificates – a table with all supported certificate formats.

A4.2 digest object

The digest is an object responsible for the generation of message digests.

A4.2.1 Constructor

digest:new (digest_algorithm: string) → digest: object

Returns a new digest object, using the digest algorithm identified by a string.

Arguments:

digest_algorithm – string identifier of the hash algorithm used to generate and verify signatures. If digest_algorithm is not support a null object will be returned.

Return values:

digest – a new object digest using a specific digest algorithm.

A4.2.2 Miscellaneous

digest:generate (data: string) → message: string

Generates a message digest for a set of bytes contained in a string.

Arguments:

data – the bytes from the container (a file, a string) to be used to generate the message digest.

Return values:

message – a string containing the message digest. An empty string indicates an error.

digest:listAlgorithms() → algorithms: table

Returns a table listing all algorithms supported by the module.

Return values:

algorithms – a table with all supported algorithms.

A4.3 cypher object

The cypher is an object responsible for data encryption and decryption.

A4.3.1 Constructor

cypher:new (algorithm: string) → cypher: object

Returns a new cypher object, using the encryption algorithm identified by a string.

Arguments:

algorithm – string identifier of the hash algorithm used to generate and verify signatures. If the string algorithm contains a non-supported value, a null object shall be returned.

Return values:

cypher – a new object cypher using a specific encryption algorithm.

A4.3.2 Miscellaneous

cypher:encrypt (data, key: string [, key_password: string]) → encrypted_data: string

Encrypts a set of bytes contained in a string.

Arguments:

data – the data from the container (a file, a string) of bytes to be encrypted.

key – string for a certificate path or containing a symmetric key, depending upon the algorithm specified for this object.

key_password – if the key is encrypted, this parameter shall be contain the password to access it.

Return values:

encrypted_data – a string containing the encrypted data. An empty string indicates an error.

cypher:decrypt (data, key: string [, key_password: string]) → decrypted_data: string

Decrypts a set of bytes contained in a string.

Arguments:

data – the data from the container (a file, a string) of bytes to be decrypted.

key – string containing a private key or a symmetric key, depending of the algorithm used for the creation of the object.

key_password – if the key is encrypted, this parameter shall contain the password to access it.

Return values:

decrypted_data – a string containing the encrypted data. An empty string indicates an error.

cypher:listAlgorithms()→ algorithms: table

Returns a table listing all algorithms supported by the module.

Return values:

algorithms – a table with all supported algorithms.

cypher:listCertificates()→ certificates: table

Returns a table listing all certificates formats supported by the module.

Return values:

certificates – a table with all supported certificate formats.

APÊNDICE B: Especificação da API do componente Device Integration

B1. IDeviceMonitor

O cabeçalho apresentado abaixo representa a interface principal do componente *Device Integration* no modelo de componentes FlexCM. Depois de implementado, este componente poderá ser instanciado, recuperado e conectado à arquitetura de execução do *middleware* OpenGinga através do ambiente de execução do FlexCM.

```
#ifndef GINGA_DEVICEINTEGRATION_IDVICEMONITOR_H
#define GINGA_DEVICEINTEGRATION_IDVICEMONITOR_H

#include <string>
#include <vector>
#include "iunknown.h"

#define IID_IDeviceMonitor "82223263-0f23-40e1-b6ba-474dcd60cdc7"

namespace deviceintegration {

/**
 * \brief Interface IDeviceMonitor é a definição do monitor para
 * recuperação de dispositivos remotos conectados e utilização
 * de seus serviços
 *
 * \author Rafael Rossi (rafael@lavid.ufpb.br)
 * \author Carlos Eduardo Batista (bidu@telemidia.puc-rio.br)
 */
class IDeviceMonitor : public virtual flexcm::interface::IUnknown{

public:
    /**
     * \brief Destrutor virtual.
     */
    virtual ~IDeviceMonitor()
    {
    }

    /**
     * \brief Recupera os metadados RDF de uma classe de
     * dispositivos NCL através de um identificador único
     *
     * \param id Identificador único da classe
     *
     * \return Metadados RDF em uma string
     */
    virtual std::string getClass(std::string id) = 0;

    /**
     * \brief Recupera uma lista com os identificadores de
     * todas as classes de dispositivos NCL registradas junto
     * ao monitor de dispositivos
     */
}
```

```

    * \return Vetor com identificadores das classes de
    * dispositivos registradas
    */
    virtual std::vector<std::string> listClasses() = 0;

    /**
    * \brief Recupera uma lista identificadores dos
    * dispositivos conectados
    *
    * \return Vetor com identificadores dos dispositivos
    * conectados
    */
    virtual std::vector<std::string> getDevices() = 0;

    /**
    * \brief Requisita dados para um serviço de uma classe
    * de dispositivos e retorna um identificador para ele
    *
    * \param service Identificador único do serviço
    * \param request String com a requisição a ser feita
    * ao serviço especificado
    *
    * \param listener Ouvinte que será notificado à medida
    * que os dados forem recebidos
    *
    * \return Identificador para a requisição realizada
    */
    virtual std::string serviceRequest(std::string service,
        std::string request,
        DeviceMonitorListener *listener) = 0;

    /**
    * \brief Registra uma classe de dispositivos (no modelo de
    * descrição RDF) através do caminho de arquivo e retorna um
    * identificador para ela
    *
    * \param deviceClassPath Caminho para um arquivo XML com
    * os metadados RDF de uma classe de dispositivos a ser
    * registrada
    *
    * \return Identificador da classe
    */
    virtual std::string registerClass(std::string deviceClassPath) = 0;

    /**
    * \brief Registra uma classe de dispositivos (no modelo de
    * descrição RDF) baseado no conteúdo passado e retorna um
    * identificador para ela
    *
    * \param deviceClassContent Referência para conteúdo com
    * metadados RDF de uma classe de dispositivos a ser registrada

```

```
    *  
    * \return Identificador da classe  
    */  
    virtual std::string registerClass(std::string deviceClassContent) = 0;  
  
};  
  
}  
  
#endif
```


B2. DeviceMonitorListener

Este cabeçalho representa um ouvinte a ser utilizado em requisições e envio de dados e informações para dispositivos. O método *requestService()* presente na interface do componente *IDeviceMonitor* recebe como parâmetro um ouvinte deste tipo e o notifica à medida que dados de requisições cheguem.

```
#ifndef GINGA_DEVICEINTEGRATION_DEVICEMONITORLISTENER_H
#define GINGA_DEVICEINTEGRATION_DEVICEMONITORLISTENER_H

#include "deviceintegration/devicemonitorevent.h"

namespace deviceintegration {

/**
 * \brief A classe <i>DeviceMonitorListener</i> é o ouvinte de
 * eventos relacionados ao envio e recebimento de informações
 * para/de dispositivos (eventos do tipo <i>DeviceMonitorEvent</i>).
 *
 * \author Rafael Rossi (rafael@lavid.ufpb.br)
 * \author Carlos Eduardo Batista (bidu@telemedia.puc-rio.br)
 */
class DeviceMonitorListener{

public:
    /**
     * \brief Destrutor virtual
     */
    virtual ~DeviceMonitorListener()
    {
    }

    /**
     * \brief Este método recebe eventos DeviceMonitorEvent do
     * tipo CLASS_JOIN. Este tipo de evento indica o registro
     * de dispositivos em uma classe.
     *
     * O objeto <i>event</i> recebido no parâmetro não deve ser
     * removido da memória dentro deste método. É papel da fonte
     * disparadora de eventos remover os eventos criados. Além disso,
     * o ouvinte não deve guardar referência para o evento fora do
     * contexto de execução de <i>joinedClass</i> pois o disparador
     * de evento é livre para remover o evento da memória a qualquer
     * momento após o retorno do método <i>joinedClass</i>.
     *
     * \param event O objeto <i>DeviceMonitorEvent</i> carregando
     * informações a respeito do evento
     */
    virtual void joinedClass(DeviceMonitorEvent *event) = 0;

    /**
     * \brief Este método recebe eventos DeviceMonitorEvent do
     * tipo CLASS_LEAVE. Este tipo de evento indica o desregistro
     * de dispositivos de uma classe.
     *
     * O objeto <i>event</i> recebido no parâmetro não deve ser
     * removido da memória dentro deste método. É papel da fonte

```

```

* disparadora de eventos remover os eventos criados. Além disso,
* o ouvinte não deve guardar referência para o evento fora do
* contexto de execução de <i>leftClass</i> pois o disparador
* de evento é livre para remover o evento da memória a qualquer
* momento após o retorno do método <i>leftClass</i>.

*
* \param event O objeto <i>DeviceMonitorEvent</i> carregando
* informações a respeito do evento
*/
virtual void leftClass(DeviceMonitorEvent *event) = 0;

/**
* \brief Este método recebe eventos DeviceMonitorEvent do tipo
* REQUEST_ANSWER. Este tipo de evento indica o recebimento de
* dados a uma requisição realizada.
*
* O objeto <i>event</i> recebido no parâmetro não deve ser
* removido da memória dentro deste método. É papel da fonte
* disparadora de eventos remover os eventos criados. Além disso,
* o ouvinte não deve guardar referência para o evento fora do
* contexto de execução de <i>requestAnswered</i> pois o disparador
* de evento é livre para remover o evento da memória a qualquer
* momento após o retorno do método <i>requestAnswered</i>.

*
* \param event O objeto <i>DeviceMonitorEvent</i> carregando
* informações a respeito do evento
*/
virtual void requestAnswered(DeviceMonitorEvent *event) = 0;

/**
* \brief Este método recebe eventos DeviceMonitorEvent do tipo
* REQUEST_ERROR. Este tipo de evento indica o recebimento de um erro
* a uma requisição realizada.
*
* O objeto <i>event</i> recebido no parâmetro não deve ser removido
* da memória dentro deste método. É papel da fonte disparadora de
* eventos remover os eventos criados. Além disso, o ouvinte não deve
* guardar referência para o evento fora do contexto de execução de
* <i>requestError</i>, o disparador de evento é livre para remover
* o evento da memória a qualquer momento após o retorno do método
* <i>requestError</i>.

*
* \param event O objeto <i>DeviceMonitorEvent</i> carregando
* informações a respeito do evento
*/
virtual void requestError(DeviceMonitorEvent *event) = 0;

};

}

#endif

```

B3. DeviceMonitorEvent

Este cabeçalho representa um evento que encapsula os dados de uma requisição, o tipo de serviço, a classe de dispositivos e a origem dos dados.

```
#ifndef GINGA_DEVICEINTEGRATION_DEVICEMONITOREVENT_H
#define GINGA_DEVICEINTEGRATION_DEVICEMONITOREVENT_H

namespace deviceintegration {

/**
 * \brief <i>DeviceMonitorEvent</i> carrega informações
 * sobre requisição e envio de dados para dispositivos
 * ou grupos de dispositivos conectados
 *
 * \author Rafael Rossi (rafael@lavid.ufpb.br)
 * \author Carlos Eduardo Batista (bidu@telemidia.puc-rio.br)
 */
class DeviceMonitorEvent{

public:
    /** \brief Tipos de eventos suportados:
     * CLASS_JOIN: Indica o registro de um dispositivo
     * em uma classe
     *
     * CLASS_LEAVE: Indica o desregistro de um dispositivo
     * em uma classe
     *
     * REQUEST_ANSWER: Indica o recebimento de dados como
     * resposta a uma requisição
     *
     * REQUEST_ERROR: Indica erro como resposta a uma requisição
     * */
    enum event_type_t {
        CLASS_JOIN,
        CLASS_LEAVE,
        REQUEST_ANSWER,
        REQUEST_ERROR
    };

private:
    /** \brief Identificação da fonte relacionada a este evento */
    std::string _source;

    /** \brief Identificador da classe de dispositivo relacionado a este
    evento */
    std::string _deviceclass;

    /** \brief Identificador do serviço relacionado a este evento */
    std::string _service;

    /** \brief Referência para os dados encapsulados por este evento */

```

```

char *_payload;

/** \brief Tipo do evento*/
event_type_t _type;

public:
/**
 * \brief Construtor da classe <i>DeviceMonitorEvent</i>
 *
 * \param source Referência para a origem deste evento
 * \param service Identificador do serviço relacionado a este evento
 *
 * \param deviceclass Identificador da classe de dispositivos
 *
 * \param type Tipo do evento
 *
 * \param payload Referência para os dados deste evento
 */
DeviceMonitorEvent(std::string source,
                   std::string service,
                   std::string deviceclass,
                   event_type_t type,
                   char *payload)
{
    _source = source;
    _deviceclass = deviceclass;
    _service = service;
    _type = type;
    _payload = payload;
}

/**
 * \brief Destrutor virtual
 */
virtual ~DeviceMonitorEvent()
{
}

/**
 * \brief Recupera o identificador da fonte deste evento
 *
 * \return Fonte que gerou o evento
 */
std::string getSource()
{
    return _source;
}

/**
 * \brief Recupera a classe de dispositivos relacionada
 * a este evento
 *
 * \return Classe de dispositivo que gerou o evento
 */

```

```

std::string getDeviceClass()
{
    return _deviceclass;
}

/**
 * \brief Recupera o serviço relacionado a este evento
 *
 * \return Serviço que gerou o evento
 */
std::string getService()
{
    return _service;
}

/**
 * \brief Recupera a referência para o payload deste evento
 *
 * \return Dados deste evento
 */
char * getPayload()
{
    return _payload;
}

/**
 * \brief Recupera o tipo deste evento
 *
 * \return Tipo do evento
 */
event_type_t getType()
{
    return _type;
}

};

}

#endif

```