UNIVERSIDADE FEDERAL DE JUIZ DE FORA
INSTITUTO DE CIÊNCIAS EXATAS
PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

Marcos Paulo Mendes

DoHyPE: Arquitetura de máquina de apresentação voltada à depuração de conteúdo hipermídia

Juiz de Fora

2018

UNIVERSIDADE FEDERAL DE JUIZ DE FORA
INSTITUTO DE CIÊNCIAS EXATAS
PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÃĞÃČO

Marcos Paulo Mendes

DoHyPE: Arquitetura de máquina de apresentação voltada à depuração de conteúdo hipermídia

Dissertação apresentada ao Programa de Pós-Graduação em Ciência da Computação, do Instituto de Ciências Exatas da Universidade Federal de Juiz de Fora como requisito parcial para obtenção do título de Mestre em Ciência da Computação.

Orientador: Marcelo Ferreira Moreno

Juiz de Fora

2018

Ficha catalográfica elaborada através do programa de geração
automática da Biblioteca Universitária da UFJF,
com os dados fornecidos pelo(a) autor(a)

Mendes, Marcos Paulo.
DoHyPE: Arquitetura de máquina de apresentação voltada à depuração de conteúdo hipermídia / Marcos Paulo Mendes. -- 2018.
73 f. : il.

Orientador: Marcelo Ferreira Moreno
Dissertação (mestrado acadêmico) - Universidade Federal de Juiz de Fora, Instituto de Ciências Exatas. Programa de Pós Graduação em Ciência da Computação, 2018.

1. Depuração Hipermídia. 2. Máquina de Apresentação. 3. Ambiente Declarativo. I. Moreno, Marcelo Ferreira, orient. II. Título.

Marcos Paulo Mendes

DoHyPE: Arquitetura de máquina de apresentação voltada à depuração de conteúdo hipermídia

Dissertação apresentada ao Programa de Pós-Graduação em Ciência da Computação, do Instituto de Ciências Exatas da Universidade Federal de Juiz de Fora como requisito parcial para obtenção do título de Mestre em Ciência da Computação.

Aprovada em 20 de Setembro de 2018.

BANCA EXAMINADORA

Prof. D.Sc. Marcelo Ferreira Moreno - Orientador
Universidade Federal de Juiz de Fora

Prof. Dr. Eduardo Barrére
Universidade Federal de Juiz de Fora

Prof. Dr. Débora C. Muchaluat Saade
Universidade Federal Fluminense

A Deus em primeiro lugar. Aos meus pais, namorada e amigos pelo apoio incondicional.

AGRADECIMENTOS

Agradeço em primeiro lugar a Deus pela força, proteção e por ser o principal responsável por mais esta conquista, estando presente durante todos os momentos.

Agradeço aos meus pais por ensinarem os valores da vida, pelo apoio e dedicação incondicional na minha educação e pelo amor que foi essencial em toda minha formação: vocês são meus referenciais de vida e fé em Deus. Ao meu irmão, pela força, apoio e confiança depositada em mim.

Agradeço a minha namorada que me auxiliou e me apoiou, dedicando amor e alegria durante todo esse tempo, pelos conselhos e por todos os momentos que passamos juntos.

Agradeço a todos os professores do Programa de Pós-Graduação em Ciência da Computação pelos ensinamentos transformadores. Em especial agradeço ao professor Marcelo Ferreira Moreno pela paciência, dedicação e incentivo.

Agradeço a todos os colegas do Laboratório de Aplicações e Inovação em Computação (LApIC), que fizeram parte da minha trajetória durante estes anos.

"Sabemos o que somos, mas não sabemos o que poderemos ser".

William Shakespare

RESUMO

O consumo de conteúdo hipermídia vem crescendo devido à recente evolução tecnológica e o surgimento de diversos recursos interativos, que os tornam cada vez mais atraentes. Com isso, a autoria de documentos hipermídia tem aumentado cada vez mais. Essa autoria é composta de diversas etapas, desde a criação do documento até a validação e testes, e apresenta diversos desafios. Um desses desafios está na validação do documento hipermídia que não deve ser feita apenas através da inspeção visual da reprodução do documento, mas também através da obtenção de informações e controle do comportamento da apresentação. Pelo fato das ferramentas de autoria terem recursos limitados para a tarefa de depuração, o processo de autoria se tornou cada vez mais desafiador. Assim, este trabalho visa propor uma arquitetura, denominada DoHyPE, para a máquina de apresentação focada no desenvolvimento e depuração dessas aplicações. A arquitetura é agnóstica em relação ao modelo hipermídia utilizado pelo autor do conteúdo. Para isso, a arquitetura é baseada em eventos, permitindo que a comunicação entre seus componentes seja monitorada ou mesmo sintetizada. Além disso, a comunicação entre os componentes pode ser realizada não apenas local, mas também remotamente. Neste trabalho é apresentado um protocolo para a comunicação entre os módulos componentes da arquitetura. Além disso, é apresentada uma instanciação da DoHyPE em duas etapas. No primeiro passo de instanciação, utiliza-se o *framework* Qt para a implementação de tarefas comuns em Sistemas Multimídia, como reprodução de conteúdo por players. Na segunda etapa, tal instanciação toma a *Nested Context Language* (NCL) como modelo de base. A solução final permite que módulos componentes possam ser desenvolvidos e anexados à máquina de apresentação, tornando possível a criação de agentes que facilitem o processo de depuração. É também apresentada uma análise da carga na utilização dos barramentos de comunicação durante apresentação e sincronia de documentos hipermídia.

Palavras-chave: Depuração Hipermídia. Máquina de Apresentação. Ambiente Declarativo.

ABSTRACT

Hypermedia content consumption has grown due to recent technological evolution and several emerging interactive resources, making it increasingly attractive. In this way the authoring of hypermedia documents has been increasing. This authoring process consists of several steps, from document creation to validation and tests, and presents several challenges. One of these challenges is in the validation of the hypermedia document that should not be done only by visual inspection of document presentation, but through obtaining information and control of the presentation behavior. Developing hypermedia applications has become increasingly challenging because authoring tools have limited resources for the debugging task. Thus, this work aims to propose an architecture, called DoHyPE, for the presentation engine focused on the development and debugging of these applications. The architecture is agnostic regarding the hypermedia model used by the content author. For this, the architecture is based on events, allowing the communication between its components to be monitored or even synthesized. In addition, communication between components can be performed not only locally but also remotely. This work presents a protocol for the communication between the components of the architecture. In addition, this work presents an instantiation of DoHyPE in two steps. In the first step, the Qt framework is used for the implementation of common tasks in Multimedia Systems, such as the presentation of content by media players. In the second step, that instantiation takes the Nested Context Language (NCL) as the underlying hypermedia model. The final solution allows component modules to be developed and attached to the presentation engine, making it possible to create agents that facilitate the debugging process. A data load analysis on the communication buses usage during the presentation and synchronization of hypermedia documents is also discussed.

Keywords: Hypermedia Debugging. Presentation Engine. Declarative Environment.

LISTA DE FIGURAS

2.1 Entidades básicas do NCM 19
2.2 Separação na implementação de máquinas de apresentação 22
2.3 Monitoramento de apresentações hipermídia 23
2.4 *Timeline* de rederização no Web Inspector do Safari 25

3.1 Arquitetura DoHyPE 27

4.1 Cabeçalho da mensagem 39
4.2 Estrutura do corpo das mensagens 39
4.3 Diagrama de classe da biblioteca DoHyPE(Qt) 41
4.4 Diagrama de classe da *DoHyPE(Qt) for NCL* 49

5.1 Documento NCL para o caso de uso Trilha de Eventos 58
5.2 Ilustração de visão temporal dos eventos 59
5.3 Apresentação do documento NCL (Figura 5.1) no *player* DoHyPE(Qt) for NCL 59

6.1 Documento NCL utilizado no experimento 61
6.2 Gráfico do atraso na reprodução do elemento *media* 62
6.3 Gráficos da quantidade de conexões, mensagens e dados nos barramentos por elementos <*media*> 64

LISTA DE TABELAS

3.1 Tipos de mensagens da arquitetura DoHyPE 36

3.2 Exemplo de mensagem de notificação a um *player* para carregamento do conteúdo 37

4.1 Propriedades suportadas pelos componentes *Player* DoHyPE(Qt) 45

4.2 Classes que representam componentes da arquitetura DoHyPE em DoHyPE(Qt) 46

4.3 Tipos de mensagens de DoHyPE(Qt) for NCL 50

4.4 Tipos de mensagens associadas às transições das máquinas de estado de eventos NCL 51

4.5 Tipo de mensagem associada a cada *Role* da linguagem NCL 53

6.1 Tráfego de dados nos barramentos 62

LISTA DE ABREVIATURAS E SIGLAS

API Application Programming Interface

ECMA European Computer Manufacturers Association

GDB GNU Debugger

DDD Data Display Debugger

GIF Graphic Interchange Format

GUI Graphical User Interface

HTG Hypermedia Temporal Graph

IPTV Internet Protocol Television

IBB Integrated Broadcast-Broadband

IPC Interprocess Communication

ITU International Telegraph Union

ITU-T Telecommunication Standardization Sector

JSON JavaScript Object Notation

JPEG Joint Photographic Experts Group

LTS Long-term Support

NCL Nested Context Language

NCM Nested Context Model

PNG Portable Network Graphics

PTP Precision Time Protocol

QoE Quality of Experience

QoS Quality of Service

RMI Remote Method Invocation

RPC Remote Procedure Call

SMIL Synchronized Multimedia Integration Language

SOAP Simple Object Access Protocol

STorM Scene- and Track-oriented Hypermedia Model

UDP User Datagram Protocol

XML Extensible Markup Language

SUMÁRIO

1 INTRODUÇÃO 14

2 FUNDAMENTAÇÃO TEÓRICA E TRABALHOS RELACIONADOS 17

2.1 NESTED CONXTEXT LANGUAGE (NCL) 19

2.2 IMPLEMENTAÇÃO DE REFERÊNCIA GINGA-NCL 20

2.3 SEPARAÇÃO DE INTERESSES NA IMPLEMENTAÇÃO DE MÁQUINAS DE APRESENTAÇÃO 21

2.4 MONITORAMENTO DE APRESENTAÇÕES HIPERMíDIA 22

2.5 ABORDAGEM SíNCRONA PARA APRESENTAÇÕES HIPERMíDIA 24

2.6 MÁQUINAS DE APRESENTAÇÃO COM FACILIDADES VOLTADAS AOS DESENVOLVEDORES 24

3 ARQUITETURA DOHYPE 26

3.1 RESOLVEDOR DE CONTEXTO 26

3.2 MÁQUINA DE APRESENTAÇÃO 28

3.3 RECURSOS DE ENTRADA E SAíDA 29

3.4 COMPONENTES DE MONITORAMENTO E CONTROLE 29

3.5 COMUNICAÇÃO ENTRE COMPONENTES DOHYPE 31

3.5.1 Descrição das mensagens 32

3.6 SUPORTE A DEPURAÇÃO 34

4 PLAYER DE PROVA DE CONCEITO - DOHYPE(QT) FOR NCL . 38

4.1 PROTOCOLO DE COMUNICAÇÃO 38

4.2 DOHYPE(QT) 40

4.3 DOHYPE(QT) FOR NCL 45

5 CASOS DE USO 55

5.1 AGENTE DE DEPURAÇÃO COM SUPORTE A SALTO TEMPORAL ... 55

5.2 AGENTE DE DEPURAÇÃO COM SUPORTE A ENCADEAMENTO DE EVENTOS 56

6 RESULTADOS OBTIDOS 60
6.1 ATRASOS NA REPRODUÇÃO DO CONTEÚDO 61
6.2 AUMENTO DOS DADOS TRAFEGADOS POR ELEMENTOS <*MEDIA*> 62

7 CONCLUSÃO 65

REFERÊNCIAS 67

1 INTRODUÇÃO

O uso de aplicações multimídia distribuídas é atualmente uma crescente realidade para milhões de usuários. Dentre elas, incluem-se não somente aplicações restritas à apresentação de conteúdos de áudio e vídeo sob demanda, mas também aplicações hipermídia que definem relacionamentos espaço-temporais entre objetos de mídia, e, ainda, permitem a intervenção do usuário em meio a uma estrutura de objetos ligados. Com a evolução dos meios e protocolos de comunicação, a transmissão de conteúdos hipermídia com uma satisfatória qualidade de experiência (QoE)(HOSSFELD et al., 2018), ou até mesmo com suporte a qualidade de serviço (QoS) (ITU-T, 2008) tem se tornado cada vez mais possível.

Essa crescente popularidade e a variedade de cenários de uso tem motivado cada dia mais o desenvolvimento de aplicações hipermídia avançadas. Diversas pesquisas e trabalhos têm sido conduzidos com o intuito de aprimorar e/ou facilitar a tarefa de desenvolvimento/autoria. Linguagens declarativas têm surgido com o intuito de aproximar a autoria de conteúdo hipermídia de domínios específicos, inclusive mais adequados a profissionais não-especializados em programação. Por exemplo, certas linguagens estruturadas visam oferecer, a partir de seus construtos, um alto nível de abstração, além de facilitar o reúso em certos casos. Algumas delas podem ser consideradas mais restritas em poder de expressão, conforme descrito em (ANTONACCI et al., 2000), como a *Synchronized Multimedia Integration Language* (BULTERMAN; RUTLEDGE, 2008), enquanto outras, como a *Nested Context Language* (SOARES; RODRIGUES, 2006), permitem a estruturação do conteúdo em composições aninhadas e a especificação de relacionamentos baseados em eventos (paradigma de causalidade) e com semântica flexível.

Apesar das facilidades que uma linguagem declarativa pode oferecer, o processo de autoria não se baseia apenas em escrever código direta ou indiretamente por meio de uma ferramenta visual. Ao longo do processo de autoria, existe a necessidade de geração de hipóteses sobre o comportamento observável e, em seguida, o teste dessas hipóteses com o propósito de encontrar um defeito que tenha causado um comportamento anormal da aplicação (SOMMERVILLE, 2011). Essa verificação não consiste apenas em reproduzir a apresentação hipermídia por um exibidor (ou máquina de apresentação), mas também conceder ao autor informações e controle de todos os elementos que compõem a apresen-

tação.

Trabalhos têm sido desenvolvidos com intuito de permitir a validação sintática, e até mesmo semântica, de documentos hipermídia (SANTOS, 2012; SANTOS et al., 2015a; MEKAHLIA et al., 2016). No entanto, no que diz respeito a processos de depuração, são poucas as proposições no âmbito de linguagens declarativas hipermídia, diferentemente do que observa-se para o desenvolvimento com linguagens imperativas. Isso se deve ao fato de que aquele tipo de linguagem possui normalmente um alto nível de abstração, apresentando desafios conhecidos (MORENO et al., 2016), como a suspensão de uma aplicação realçando o trecho do código relacionado a tal instante ou até mesmo a verificação de corretude da aplicação depender de eventos imprevisíveis. Sendo assim, torna-se necessária a busca por uma arquitetura para máquinas de apresentação voltada não apenas à simples reprodução do conteúdo, mas, em especial, ao auxilio na depuração hipermídia, estando apoiada em componentes que permitam a captura de informações relativas aos elementos da apresentação em cada instante, além de permitir o controle destes elementos.

Como forma de contribuir com o processo de desenvolvimento hipermídia, o presente trabalho tem como objetivo propor uma arquitetura para máquinas de apresentação hipermídia voltadas especificamente para o processo de autoria, que possibilitem a aquisição de informações relevantes a uma depuração mais detalhada, incluindo tanto o reconhecimento quanto a alteração dos estados, propriedades e atributos das instâncias de entidades que formam o modelo hipermídia adotado pelo autor. Dessa forma, a aquisição dessas informações permite que, além da validação sintática do documento hipermídia, o comportamento da apresentação seja validado. A arquitetura proposta, denominada *DoHyPE* (***D****ebug-****o****riented* ***H****ypermedia* ***P****resentation* ***E****ngine*), é caracterizada pelo baixo acoplamento entre seus módulos componentes, alcançada pelo emprego de uma abordagem orientada a eventos e troca de mensagens. Com essa abordagem, todo o comportamento da apresentação, desde o carregamento de sua especificação até seu fim, pode ser capturado ou mesmo modificado em tempo real.

De uma forma geral, *DoHyPE* busca uma definição genérica de componentes arquiteturais, canais de comunicação e formatos de mensagens independentes de modelos e *frameworks* específicos. A arquitetura deve, portanto, ser instanciada sobre plataformas de software multimídia, além de ser especializada para o suporte às entidades de um ou mais modelos hipermídia. Com a definição dessa arquitetura, o presente trabalho apre-

senta uma instanciação da arquitetura DoHyPE em duas etapas. Na primeira etapa o propósito é resolver problemas comuns ao desenvolvimento de máquinas de apresentação hipermídia, como o controle e apresentação do conteúdo. Para isso a instanciação é feita utilizando um *framework* que facilita o desenvolvimento multimídia. Com essa etapa concluída, é realizada a instanciação da DoHyPE para a linguagem NCL, permitindo que agentes possam ser implementados com propósitos específicos, como os agentes de depuração, medição de audiência, dentre outros. Além dessa instanciação, o presente trabalho apresenta casos de uso da arquitetura DoHyPE com foco na autoria de documentos hipermídia.

Para uma descrição abrangente de tal proposta, esta dissertação está estruturada como se segue: O Capítulo 2 apresenta os trabalhos relacionados. O capítulo 3 descreve de forma detalhada a arquitetura *DoHyPE*. O Capítulo 4 é mostrado uma instanciação da arquitetura DoHyPE utilizando o framework Qt, além de apresentar como o *framework* Qt pode facilitar tal instanciação. Nesse capítulo, é descrito como prova de conceito o *player DoHyPE(Qt) for NCL*, que toma a *Nested Context Language* (NCL) como modelo de base para a autoria. O Capítulo 5 apresenta os casos de uso da arquitetura DoHyPE, aplicados ao desenvolvimento hipermídia. O Capítulo 6 contém a avaliação da solução proposta e os resultados obtidos. Finalmente, o Capítulo 6 apresenta as conclusões e os trabalhos futuros.

2 FUNDAMENTAÇÃO TEÓRICA E TRABALHOS RELACIONADOS

Os *frameworks* para o desenvolvimento de aplicações hipermídia podem ser classificados de acordo com o paradigma de programação oferecido, sendo dois os mais comumente usados: o paradigma declarativo e o paradigma imperativo. Aplicações declarativas são definidas através de linguagens que enfatizam a descrição declarativa da solução de um problema, ao invés da decomposição do problema em uma implementação por algoritmos. Aplicações imperativas são baseadas em algoritmos que estabelecem passos a serem executados (WATT et al., 1990). Em geral, aplicações não precisam ser puramente declarativas ou imperativas. Em particular, linguagens declarativas podem permitir o uso de conteúdos do tipo *scripts*, que são imperativos por natureza. Similarmente, aplicações imperativas podem referenciar ou construir e inicializar a apresentação de conteúdos declarativos (SOARES et al., 2007).

Linguagens declarativas são usualmente projetadas para domínios específicos, provendo um alto nível de abstração e mais próximo do que se deseja representar, diferentemente do que a implementação imperativa pode oferecer. No domínio hipermídia, linguagens declarativas, como as várias baseadas em *Extensible Markup Language* (XML), são atrativas para serem utilizadas, pois permitem definir as relações entre as entidades de um modelo por meio de declarações que compõem um documento. É o caso de NCL e *Synchronized Multimedia Integration Language* (SMIL), cujas especificações possuem um alto nível de abstração, herdam a estruturação bem formada de XML e oferecem funcionalidades relevantes para a autoria hipermídia.

O processo de autoria de aplicações hipermídia declarativas envolve a geração de código-fonte que especifica a estrutura que definirá o comportamento da apresentação. Tal especificação pode se dar de forma direta, escrevendo código com editor de texto (e.g. NCL Eclipse (AZEVEDO et al., 2011)), ou de forma indireta, gerando código a partir da manipulação de abstrações visuais ou procedimentos de automação (e.g. NCL Composer (AZEVEDO et al., 2014), *NCL Editor Supporting XTemplate* - NEXT (MATTOS et al., 2013) e o EDITEC (DAMASCENO et al., 2011). Independentemente da forma, o processo de autoria define quais objetos de mídia e respectivos relacionamentos farão

parte da aplicação. Esses relacionamentos podem se concretizar por meio de ligações condicionadas a eventos determinísticos, como aquelas baseadas no tempo de reprodução de objetos de mídia (elos temporais). Podem também ser especificadas condicionadas a eventos não-determinísticos, como por exemplo quando dependem da interação realizada pelo usuário.

A autoria determina também características gerais da apresentação de um conjunto de objetos de mídia. São definidas as regiões que objetos vão ocupar no(s) dispositivos(s) de exibição do usuário e propriedades como o volume de um áudio a ser reproduzido. Dependendo do modelo hipermídia adotado como base da autoria, adaptações da apresentação condicionadas a características do perfil do usuário, de seu dispositivo ou de seu ambiente também podem ser especificadas.

No entanto, ainda que a autoria seja baseada em linguagem de domínio específico, erros de especificação podem ocorrer, levando a comportamentos inesperados, que somente podem ser detectados em tempo de apresentação. A depuração de aplicações hipermídia é fundamental para a rápida e correta identificação de tais erros.

Quando se analisa o desenvolvimento de aplicações por meio de linguagens imperativas, o processo de depuração fornece informações e controles referentes a elementos da linguagem, como a pilha de execução, valores de variáveis, dentre outras informações (SILVA, 2008; PARNIN; ORSO, 2011). Ao contrário desse tipo de desenvolvimento, a autoria de aplicações hipermídia com linguagens declarativas se baseia na descrição de comportamentos, estados, propriedades e atributos que são dependentes de ligações espaciais, temporais e de eventos não determinísticos. Quando se desenvolve uma aplicação hipermídia, o foco não está na sequência de comandos, mas sim no encadeamento de ações que ocorrem devido a eventos de comportamento dos objetos de mídia e do telespectador.

Com o advento da TV Digital no Brasil e a necessidade de se desenvolver aplicações hipermídia interativas para esse ambiente, a NCL se tornou a linguagem declarativa no padrão do Sistema Brasileiro de Televisão Digital (SBTVD) (ABNT, 2008). Além disso, em 2009, se tornou uma recomendação ITU-T (ITU-T, 2014) para sistemas IPTV. A NCL permite definir a estrutura de objetos de mídias e seus relacionamentos.

2.1 NESTED CONXTEXT LANGUAGE (NCL)

A NCL é uma linguagem declarativa baseada em XML para a autoria de documentos hipermídia. Ela se baseia no modelo conceitual *Nested Context Model* (NCM) (SOARES; RODRIGUES, 2005), que tem seu foco na representação e tratamentos de documentos hipermídia. NCM descreve dois tipos de entidades básicas, que são os Nós e os Elos. As entidades básicas são mostradas na Figura 2.1.

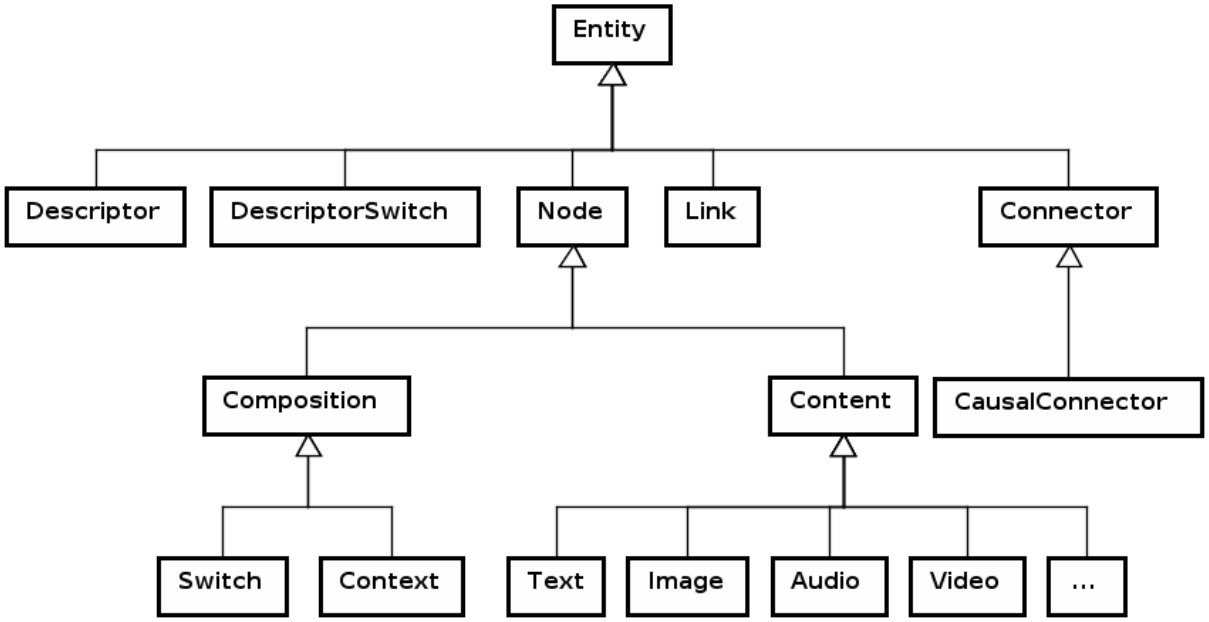

Figura 2.1: Entidades básicas do NCM

Os Nós podem ser Nós de Conteúdo, Nós de Contexto, Nós *Switch* e possuem uma lista ordenada de Âncoras, as quais definem, cada uma, um subconjunto de unidades de informação do conteúdo do Nó. Os Nós de Conteúdo representam objetos de mídia como texto, vídeo, áudio, imagem, aplicações, dentre outros.

O Nó de Composição é aquele que define uma coleção de Nós de Conteúdo e/ou de outros Nós de Composição recursivamente. Além da lista ordenada de âncoras definida em todo Nó, Nós de Composição possuem uma lista ordenada de Portas, que definem mapeamentos entre a composição e seus nós internos. Portas e Âncoras são classificadas como interfaces de um Nó de Composição. Nós de Composição podem ser Nós de Contexto, que representam um conjunto de Nós de Conteúdo, Nós de Contexto e Nós *Switch*.

O Nó *Switch* é uma especialização do Nó de Composição voltada à definição de um conjunto de nós alternativos, na qual a seleção é feita baseada em regras declaradas no

documento. Esse nó permite a adaptação de conteúdos através da análise de variáveis do sistema, por exemplo, o idioma do sistema, idade do usuário, dentre outras.

Os Elos têm o papel de descrever como os nós se conectam, definindo a semântica das relações hipermídia. Através deles a reprodução, por exemplo, de um vídeo pode ser parada após o término de um áudio.

2.2 IMPLEMENTAÇÃO DE REFERÊNCIA GINGA-NCL

Um caso de interesse no suporte a aplicações hipermídia é o *middleware* Ginga, adotado como padrão no Sistema Brasileiro de TV Digital (ABNT, 2008), além de ser recomendado para sistemas *Internet Protocol Television* (IPTV) (ITU-T, 2014) e sistemas *Integrated Broadcast-Broadband* (IBB) (ITU-R, 2017). Ele é baseado na arquitetura Ginga (SOARES et al., 2007) que é divida em três maiores módulos, que são: Ginga Common-Core, Ginga-NCL e Ginga-J. Ginga Common-Core é composto pelos decodificadores de conteúdo comuns e deve suportar o modelo de visualização conceitual especificado para o sistema de TV Digital brasileiro. Nele estão representados os responsáveis pela apresentação de objetos de mídia, como *Media Players*, os responsáveis pela interface do usuário, pela parte gráfica e até mesmo pela gestão de aplicações. O Ginga-NCL é um subsistema lógico do Sistema Ginga que processa documentos NCL. O Ginga-J também é um subsistema lógico do Sistema Ginga, porém processa objetos com conteúdo Xlet. Tanto o Ginga-NCL quanto Ginga-J são desenvolvidos utilizando estes serviços que compõem o Ginga Common-Core.

A implementação de referência existente para Ginga-NCL (PUC-RIO, 2013) foi desenvolvida de forma modular. Cada módulo representa um componente integrante do Ginga Common-Core ou Ginga-NCL. Além disso, essa implementação utiliza técnicas de interface comum a certos subcomponentes, permitindo a especialização entre eles. Sendo assim, os subcomponentes devem implementar certas ações comuns. Isso é feito através da definição de Interfaces que o paradigma Orientado a Objetos possibilita. Com intuito de organizar a definição dos componentes, aqueles que são integrantes do Ginga Common-Core possuem o prefixo *gingacc*. Os componentes que são integrantes do interpretador NCL possuem o prefixo *ncl30*. Pelo fato de o foco dessa implementação estar na transferência de tecnologia para o embarque da máquina de apresentação em dispositivos terminais, os reportes não têm como propósito a depuração. Os reportes são feitos através

de impressões no terminal e muitos dos problemas que ocorrem durante a apresentação não são reportados. Além disso, edições na apresentação só são feitas a partir de comandos de edição, que são limitados quando o propósito é a depuração.

2.3 SEPARAÇÃO DE INTERESSES NA IMPLEMENTAÇÃO DE MÁQUINAS DE APRESENTAÇÃO

Com o intuito de mostrar que a sincronização pode ser respeitada mesmo quando parte do sistema multimídia é executado em um dispositivo com recursos limitados, Moreno et al. (2016) propõe uma arquitetura para sistemas multimídia que separa os módulos de uma máquina de apresentação por interesses. Ela está divida em dois módulos principais: *Presentation Manager* e *Multimedia Backend*. Esta representação pode ser vista na Figura 2.2. O *Presentation Manager* tem a responsabilidade de ler a aplicação e criar o plano de apresentação, usando o sub-módulo *Presentation Controller*. Já o *Multimedia Backend* fica responsável por decodificar, processar e renderizar o conteúdo multimídia, usando o sub-módulo chamado *Backend Controller*. Essa separação vai além de apenas modularização. Ambos os módulos estão profundamente separados, por exemplo, pelo isolamento de processos.

Para que ocorra a comunicação, os módulos possuem sub-módulos responsáveis por receberem as mensagens, traduzi-las e repassarem aos sub-módulos que tenham interesse nelas. O sub-módulo *Command Invoker* fica responsável por enviar os comandos ao *Multimedia Backend*, sejam eles a execução de ações ou a obtenção do valor de uma variável. Esses comandos são recebidos pelo *Command Receiver* que valida as requisições e repassa ao *Backend Controller*. Nesse âmbito, o Event Listener previamente registra o interesse em receber notificações de eventos que são gerados pelo *Event Notifier*, repassando ao *Presentation Controller* o evento recebido em uma estrutura de dados esperada.

Para a comunicação entre *Presentation Controller* e *Multimedia Backend*, os autores propõem o uso de mensagens no formato JSON (ECMA International, 2013), pelo fato de serem leves e para evitar problemas de desempenho caso um outro tipo de protocolo fosse utilizado (por exemplo, serviços web SOAP, RPC, RMI, etc.). Como o foco daquele trabalho concentra-se na apresentação do conteúdo, questões de depuração não foram analisadas. Apesar da comunicação entre o *Presentation Manager* e *Multimedia Backend*

poder ser monitorada com intuito de obter os estados dos elementos, as mensagens trafegadas são referentes a uma comunicação restrita, a partir da qual não é possível a obtenção de informações internas de tais componentes, como a criação e estado das instâncias de entidades do modelo hipermídia em uso.

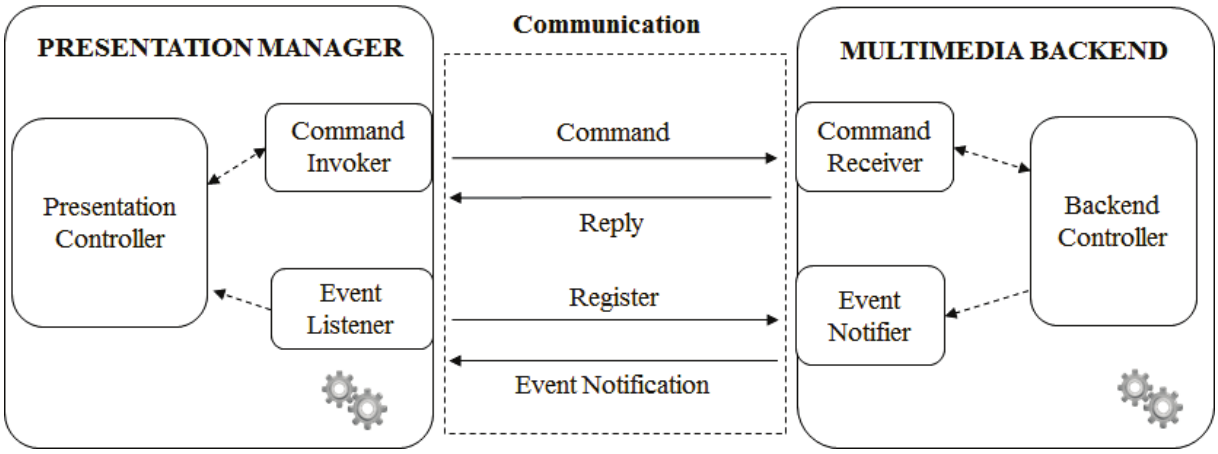

Figura 2.2: Separação na implementação de máquinas de apresentação (MORENO et al., 2016)

2.4 MONITORAMENTO DE APRESENTAÇÕES HIPERMÍDIA

Com o propósito de auxiliar os autores de aplicações e os desenvolvedores de máquinas de apresentação, Santos et al. (2015b) especifica uma arquitetura aberta e genérica. Ela está focada em prover informações como o estado das variáveis, as propriedades dos objetos e os tempos da apresentação da mídia. Seu propósito é tornar possível a detecção de problemas de apresentação causados por erro de programação ou mau funcionamento dos exibidores, como por exemplo, sincronização inválida. Para isto, a arquitetura separa a máquina de apresentação (*Presentation Engine*) e a máquina de monitoramento (*Monitoring Engine*), que se comunicam através de um protocolo específico. Essa separação é mostrada na Figura 2.3.

A *Monitoring Engine* pode enviar mensagens para alterar ou obter o valor do atributo de um objeto, enviar eventos de entrada ou até mesmo comandos arbitrários, como a utilização da API de comandos de edição do Ginga-NCL. Dessa forma, a *Presentation Engine* pode enviar mensagens ao *Monitoring Engine* notificando a respeito das ações internas e também executar comandos específicos do *Monitoring Engine*. Essa comunica-

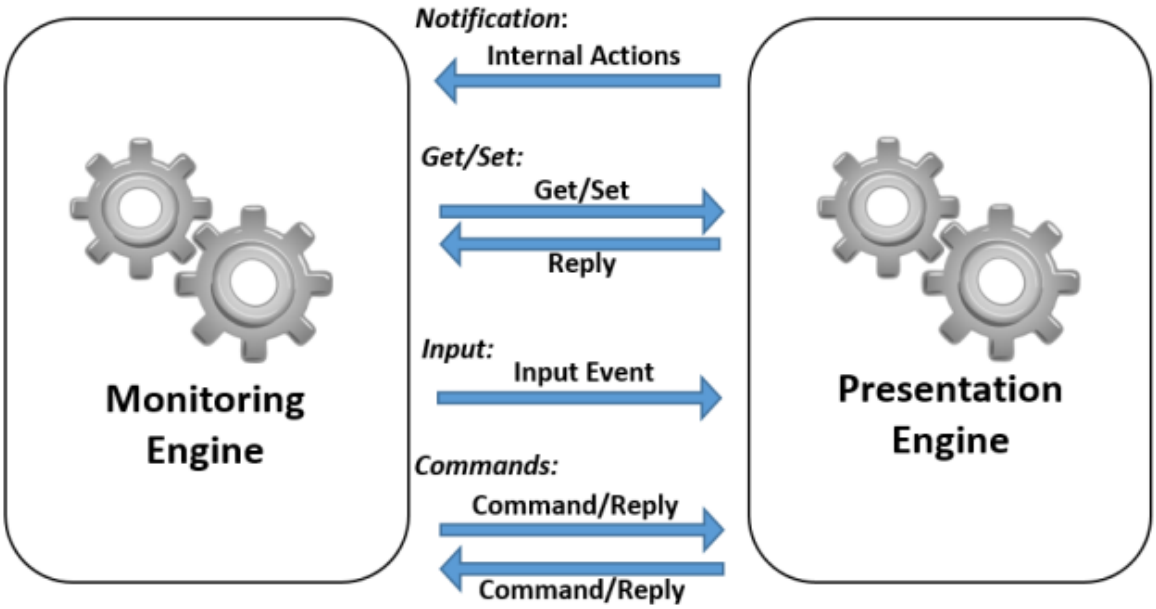

Figura 2.3: Monitoramento de apresentações hipermídia (SANTOS et al., 2015b)

ção entre os módulos é feita através de um protocolo que separa as mensagens em quatro grupos de mensagens. No primeiro estão as mensagens que a *Presentation Engine* envia ao *Monitoring Engine* para notificar sobre ações internas realizadas em objetos de mídia. O segundo grupo estão as mensagens para obter ou alterar o estado de uma variável. No terceiro grupo estão as mensagens enviadas pelo *Monitoring Engine* ao *Presentation Engine* para emular os eventos de entrada. Por fim, o quarto grupo representa as mensagens que contêm comandos atbitrários.

Apesar de possuir uma separação entre essas duas máquinas, elas são especificadas como caixas-pretas, não tendo representado o comportamento de seus sub-módulos. Além disso, as ações que podem ser notificadas são restritas apenas ao início ou finalização de uma aplicação ou o início, pause, retomada, finalização ou seleção de um objeto de mídia. Com isso, não é possível determinar quando um objeto de mídia é criado ou até mesmo a decorrência de eventos que ocasionaram uma determinada ação. Se aplicada a um processo de depuração, a arquitetura inviabiliza saber, por exemplo, qual o elo responsável por uma ação realizada.

Diferentemente daquela proposta, o presente trabalho tem como objetivo definir uma arquitetura para diferentes modelos hipermídia, especificando o comportamento da máquina de apresentação e a comunicação entre os módulos componentes, tornando acessíveis

os eventos que possam ocorrer durante a execução dessa máquina.

2.5 ABORDAGEM SÍNCRONA PARA APRESENTAÇÕES HIPERMÍDIA

Para reduzir a complexidade no mapeamento entre uma estrutura NCL (SOARES; RODRIGUES, 2006) na perspectiva do usuário para as primitivas de máquina, Lima et al. (2017) propõe uma forma de conversão de NCL para Smix, que é uma linguagem síncrona (BENVENISTE; BERRY, 1991) de domínio específico. Smix tem um propósito similar a NCL, porém possui uma semântica mais simples e precisa. Seu comportamento é orientado a eventos. Esta conversão restringe o indeterminismo existente na linguagem como a existência de ações do tipo paralelo em links.

Em NCL, esses links são avaliados de forma arbitrária, ao contrário do formato Smix, que fará com que sejam avaliados sempre na mesma ordem. Isto não traz um impacto considerável na apresentação, pois o resultado acaba sendo imperceptível na maioria dos casos. Apesar de propor uma conversão para um ambiente síncrono, o propósito desta conversão é voltada exclusivamente para a apresentação. Diferentemente daquela proposta, o presente trabalho tem como objetivo a depuração de aplicações hipermídia e não apenas a reprodução.

2.6 MÁQUINAS DE APRESENTAÇÃO COM FACILIDADES VOLTADAS AOS DESENVOLVEDORES

Navegadores web modernos apresentam mecanismos que fornecem ao desenvolvedor acesso aos estados internos (SANTOS et al., 2015b). As recentes versões dos navegadores Web Safari (Apple Inc., 2003), Chrome(Google Inc., 2008) e Firefox(Mozilla Foundation, 2004), tem permitido rastrear layout de página, definir pontos de interrupção em JavaScript (linguagem imperativa), alterar valores de variáveis, monitorar tráfego de rede, obter o aninhamento da aplicação dos estilos definidos em um elemento, dentre outros.

O Safari, por exemplo, apresenta uma *timeline* de renderização de *layout* que permite ao desenvolvedor ver o instante do tempo que o elemento foi renderizado, possibilitando obter-se uma análise da performance na renderização do documento. A Figura 2.4 ilustra essa *timeline* existente no navegador Safari. Apesar de todo esse ferramental disponibili-

zado, não se pode obter uma lógica entre os elementos envolvidos, mas apenas o momento em que certos eventos ocorreram. Quando o modelo hipermídia fica mais complexo, mais informações são necessárias. Sendo assim, toda informação decorrente da instanciação e reprodução do modelo hipermídia se faz necessária, podendo ser filtradas de acordo com o interesse do desenvolvedor, como propõe-se no presente trabalho.

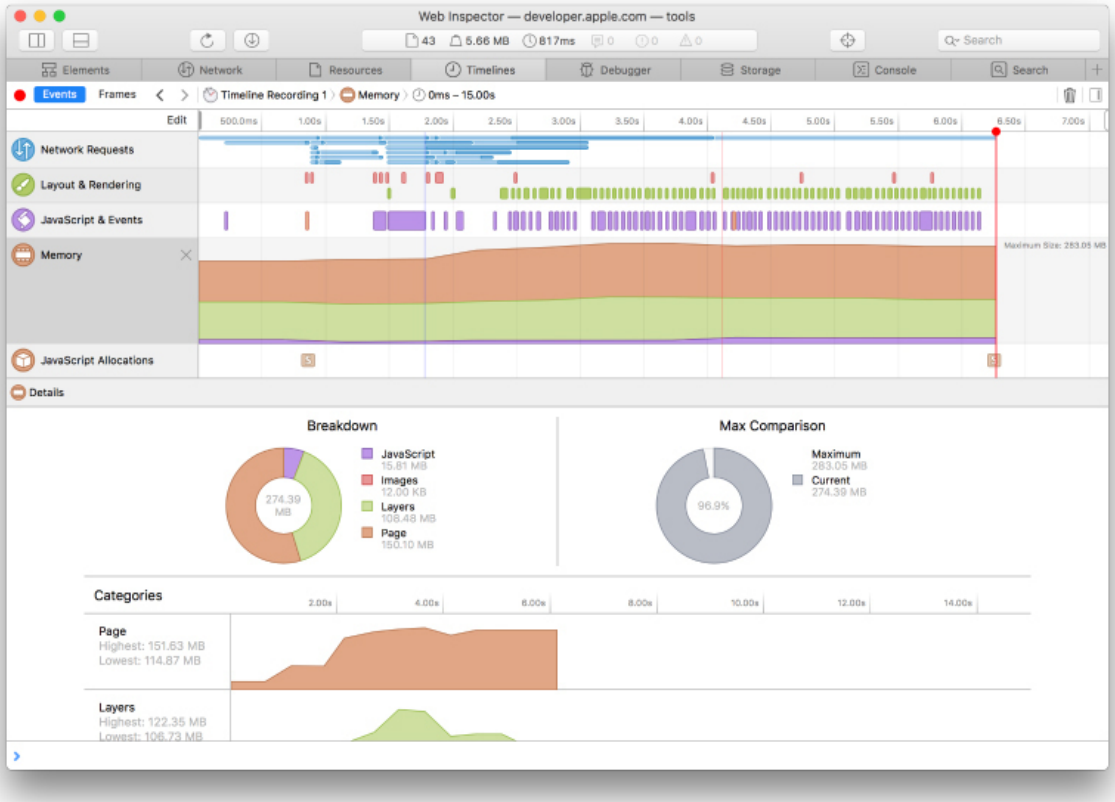

Figura 2.4: *Timeline* de rederização no Web Inspector do Safari[1]

[1]https://developer.apple.com/safari/tools/

3 ARQUITETURA DOHYPE

Em termos gerais, uma máquina de apresentação hipermídia tem a função de carregar um documento e reproduzir o conteúdo conforme codificado. Para isto, tal máquina é normalmente desenvolvida baseada em arquiteturas que visam facilitar a própria implementação da máquina e a partir de *frameworks* de aplicações multimídia.

No entanto, quando o foco é voltado ao autor de aplicações hipermídia, cuja necessidade junto à máquina de apresentação é verificar a corretude do comportamento de seu conteúdo, se faz necessário priorizar o acesso a informações e o controle pleno da apresentação. Isso incorre em uma maior complexidade de implementação da máquina, para que seja capaz de externalizar o maior conjunto possível de informações e manipulações ao seu usuário. Se todo esse aparato estiver sendo monitorado e controlado por uma interface adequada de depuração, diferentes níveis de abstração, visões gráficas e comandos de controle poderão ser oferecidos aos autores.

Este capítulo apresenta uma definição genérica dos componentes arquiteturais, os canais de comunicação e os formatos de mensagens de uma arquitetura para máquinas de apresentação hipermídia voltadas especificamente para o processo de autoria.

A Figura 3.1 ilustra a arquitetura proposta, denominada DoHyPE (***D****ebug-****o****riented* ***H****ypermedia* ***P****resentation* ***E****ngine*). Nela estão representados os componentes e os meios de comunicação entre eles. Ela pode ser divida em cinco partes: o resolvedor de contexto (*Context Resolver*), máquina de apresentação (*Presentation Engine*), os recursos de entrada e saída (*Players*, *Sensors*, *Actuators*, *Remote Controllers*), os componentes de monitoramento e controle (*Monitoring e Controlling*) e os barramentos de comunicação (*Bus*). Todas essas partes serão descritas nas seções a seguir.

3.1 RESOLVEDOR DE CONTEXTO

Para que a comunicação entre os componentes possa existir sem colisões, todos os componentes da arquitetura são identificados de forma única. Sendo assim, a cada criação de qualquer nova instância de um componente, é solicitado ao ***Context Resolver*** um identificador, que será gerado como um código *hash* (ver Seção 3.5.1). O mesmo ***Context Resolver*** pode ser solicitado a mapear um identificador *hash* por ele gerado em dados so-

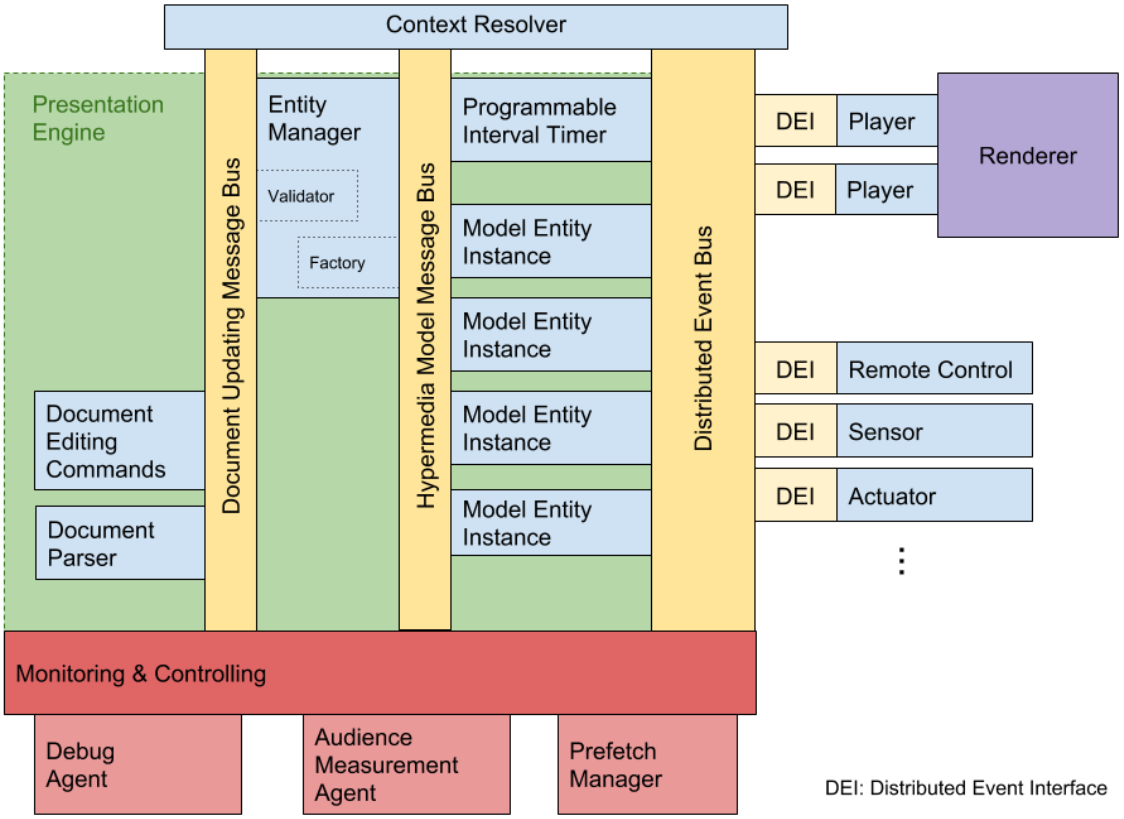

Figura 3.1: Arquitetura DoHyPE

bre o respectivo componente, sempre que necessário. Esses dados podem ser, por exemplo, a hierarquia no qual esse componente está inserido no modelo ou até mesmo informações gerais como linha e coluna do documento hipermídia associados ao componente.

A utilização de *hash* possibilita que os identificadores tenham um comprimento fixo, facilitando a análise das mensagens pelo barramento, além de oferecer resultados que tenham, de preferência, um baixo índice de colisão. Com o intuito de facilitar a comunicação e pelo fato dos recursos de entrada e saída poderem estar distribuídos (ver Seção 3.3), o ***Context Resolver*** guarda também a localização do elemento no momento em que a solicitação de criação de um identificador foi recebida, podendo ela ser local ou remota. Essa abordagem faz com que os componentes da arquitetura se comuniquem através do identificador, abstraindo a localização desses componentes ao remetente.

3.2 MÁQUINA DE APRESENTAÇÃO

A ***Presentation Engine*** é composta por vários outros componentes. Quando uma apresentação é iniciada, o ***Document Parser*** fica responsável por ler um documento e entender a sintaxe da linguagem suportada, emitindo os eventos pertinentes a cada construto reconhecido. O documento dado como entrada pode estar escrito em NCL (SOARES; RODRIGUES, 2005), SMIL (BULTERMAN; RUTLEDGE, 2008), STorM (FREESZ JR. et al., 2017) ou outra linguagem suportada, sendo o ***Document Parser*** responsável por interpretar os elementos da linguagem e comunicar ao ***Entity Manager***, para que este, conhecedor do modelo, realize a criação das instâncias do modelo (***Model Entity Instance***). Essa comunicação permite o uso de diversas linguagens hipermídia em um mesmo documento (documentos híbridos), possibilitando a existência de blocos de códigos de linguagens distintas operando entre si. Para isto, basta que o ***Entity Manager*** esteja preparado para receber tais construtos.

Para manipulações mais sofisticadas como modificação ao vivo ou mesmo a geração dinâmica da aplicação, comandos de edição podem ser usados (SOARES et al., 2009). Nesse âmbito, o componente ***Document Editing Commands*** tem a função de repassar tais ações ao ***Entity Manager***. O ***Entity Manager***, ao receber estas mensagens, resultantes do *parsing* ou comandos de edição, valida se as regras do modelo são respeitadas, como *namespaces*, identificadores, papéis em relacionamentos e afins.

Em DoHyPE, cada instância de uma certa entidade do modelo hipermídia suportado é representada por um componente ***Model Entity Instance*** específico. Ele representa as definições de comportamento de um certo construto. Por exemplo, um ***Model Entity Instance*** pode ser especializado para representar um objeto de mídia, ou um elo que especifica o relacionamento entre objetos de mídia no documento, em conformidade com a semântica definida por um certo modelo hipermídia. Para permitir a implementação do comportamento correto e este poder depender de eventos externos ao construto, componentes ***Model Entity Instance*** podem receber eventos emitidos por outras instâncias e, assim, reagirem mediante as regras do modelo, como eventos de início ou fim natural de um objeto de mídia ou até mesmo eventos de entrada do usuário. Outro exemplo de evento é um intervalo de tempo decorrido, de interesse de qualquer componente, e portanto, programável pelo mesmo. Neste aspecto, o componente ***Programmable Interval Timer*** que é responsável pelo relógio da apresentação, deverá ser previamente notificado

sobre tal interesse e, até que seja notificado do contrário, ele gera o evento endereçado ao(s) componente(s) interessado(s), podendo ser periódico ou não, conforme programado.

3.3 RECURSOS DE ENTRADA E SAÍDA

Para a reprodução de objetos de mídia como vídeo, áudio, imagem, texto ou mesmo outros documentos hipermídia, componentes ***Players*** são instanciados, podendo estar situados remota ou localmente. Esses ***Players*** são os responsáveis por decodificar o conteúdo em si e repassarem ao componente ***Renderer*** o resultado desta decodificação. Em caso de requisitos que demandem a instanciação dinâmica de ***Players***, pode-se, opcionalmente, incluir um componente gerenciador que ficaria responsável por instanciar e informar o identificador de determinado ***Player*** ao componente interessado. Essa decisão centralizaria a instanciação de ***Players***, sendo necessário conhecer, inicialmente, apenas o identificador de tal componente gerenciador.

Por sua vez, o ***Renderer*** fica responsável por fazer a junção dos resultados das diversas instâncias de ***Players*** e exibi-los. A representação adotada não restringe o ***Renderer*** a apresentar o resultado final em meio de apresentação local junto ao ***Player***, podendo, então, ser exibido remotamente. Isto permite que um ***Renderer*** gere fluxos contínuos de áudio e/ou vídeo, com o resultado da junção dos ***Players*** a ele associados e enviar a um dispositivo de interesse específico. Assim como os ***Players***, dispositivos de entrada e saída podem estar situados local ou remotamente, por exemplo, conectados por uma rede de comunicação. No caso dos dispositivos de entrada, estes podem gerar eventos, como pressionamento de teclas, clique em tela ou até mesmo eventos atrelados a sensores, e as instâncias de entidade do modelo podem reagir a esses eventos de entrada, conforme as regras do modelo. Quando são dispositivos de saída, como atuadores e afins, eventos podem ser a eles enviados com intuito de controlarem esses dispositivos.

3.4 COMPONENTES DE MONITORAMENTO E CONTROLE

Durante a apresentação hipermídia, toda comunicação para o intercâmbio de mensagens entre os componentes ocorre por meio de barramentos, sendo eles o ***Document Updating Message Bus***, ***Hypermedia Model Message Bus*** e ***Distributed Event Bus***. Os barramentos podem tanto ser monitorados para análises do que está ocorrendo em tempo

de apresentação, quanto usados para o controle da apresentação, injetando mensagens que alterem estados de componentes da arquitetura. Pelo fato de serem compartilhados, os barramentos permitem que componentes de monitoramento e controle sejam incorporados e passem a obter e/ou gerar mensagens advindas/destinadas de/a qualquer componente. Um componente de monitoramento pode gerar reportes ao usuário sobre o comportamento da apresentação, enquanto componentes de controle podem gerar mensagens com eventos de interesse do mesmo usuário. Um componente de monitoramento relevante e que é a principal motivação da presente proposta é o ***Debug Agent***. Como existem diferentes perfis de usuários, desde os que escrevem código diretamente aos que utilizam ferramentas de geração semi-automática, o ***Debug Agent*** ideal deve ser capaz de filtrar e traduzir as mensagens de acordo com o perfil do autor, adaptando o nível de abstração dos reportes e a quantidade de informação reportada. Além disso, o *Debug Agent* pode também ser um componente de controle, ao permitir ao usuário injetar mensagens que alterem qualquer aspecto dinâmico das instâncias de entidades do modelo, como o estado de eventos de objetos mídia, suas propriedades, entre outros. Com a quantidade de informações que pode ser obtida e a possibilidade de se controlar a apresentação, o *Debug Agent* pode disponibilizar funcionalidades que facilitam a análise da apresentação, incluindo, por exemplo, a execução evento-a-evento (modo *lockstep*).

Um outro exemplo de componente de monitoramento que possui interesse em analisar as mensagens trafegadas pelo barramento é o ***Audience Measurement Agent***. Através destas análises, ele consegue fazer inferências, como a classificação de determinados objetos de mídia pela quantidade de acessos através do conteúdo hipermídia ou até mesmo recomendar conteúdos. Obviamente, tal componente seria adequado para uma instanciação de DoHyPE voltada não exclusivamente à depuração, o que está fora do escopo deste trabalho.

As mensagens capturadas a partir dos barramentos também podem ser usadas para que um componente de gerenciamento de pré-busca (***Prefetch Manager***), construa um plano e, de acordo com este, sinalize via mensagens ações que disparem a recuperação do conteúdo de objetos de mídia previamente às respectivas apresentações. As instâncias de ***Players*** associados a tais objetos de mídia também podem ser criadas antecipadamente como parte das ações do plano de pré-busca.

3.5 COMUNICAÇÃO ENTRE COMPONENTES DOHYPE

Cada comunicação dentro da arquitetura DoHyPE ocorre utilizando o barramento respectivamente mais adequado, conforme a fase em que se encontra a apresentação e/ou os componentes envolvidos. Com intuito de obter maior desempenho, todos os barramentos possuem um comportamento baseado no *design pattern Publish-subscribe* (EUGSTER et al., 2003), no qual os componentes podem se registrar dizendo que tipos de eventos estão interessados em serem notificados. Além disso, os componentes podem notificar aos barramentos quais os identificadores de origem são de interesse, possibilitando os barramentos a aplicação de filtros na propagação de mensagens.

Em um primeiro instante, os componentes interessados em utilizar o barramento de comunicação se conectam a ele, demonstrando o interesse em utilizá-lo. O barramento pode, com isso, saber destinar corretamente as mensagens, controlar a duplicação de identificadores conectados e evitar que componentes se comportem como outros já conectados. Ao concluir as operações, um componente pode encerrar a sua conexão, notificando ao barramento.

Observa-se que, para atingir a generalidade e a abrangência necessárias, DoHyPE especifica componentes em diferentes níveis de abstração. Pode-se perceber que a arquitetura engloba componentes responsáveis pelo processamento e análise do documento codificado, pela instanciação e controle do modelo hipermídia, e pela operação dessa instância com os recursos de entrada e saída pertinentes durante a apresentação (Parsing/Edição - Validação/Instanciação - Apresentação/Manutenção).

Com o intuito de oferecer certo grau de isolamento conforme o tipo de mensagem trafegada e a abstração/interesse dos componentes envolvidos, a arquitetura possui três barramentos de comunicação distintos: ***Document Updating Message Bus***, que é responsável por fazer a ligação entre os componentes que solicitam a criação ou alteração da instância do modelo; ***Hypermedia Model Message Bus***, que tem a função de estabelecer a comunicação do ***Entity Manager*** com os componentes ***Model Entity Instance***; ***Distributed Event Bus***, que tem o papel de interligar as instâncias da entidade do modelo com os recursos de entrada e saída, sejam eles locais ou não. De uma forma geral, os dois primeiros barramentos (***Document Updating Message Bus*** e ***Hypermedia Model Message Bus***) são necessariamente de comunicação local e o barramento ***Distributed Event Bus*** é de comunicação distribuída.

As mensagens não são dependentes de meios de comunicação específicos e, portanto, podem ser intercambiadas através de mecanismos de comunicação entre processos e/ou protocolos de rede diversos.

3.5.1 DESCRIÇÃO DAS MENSAGENS

Para a comunicação entre os componentes, as mensagens devem incluir um conjunto de informações de controle. São elas: *moreFragments*, *messageType*, *bodySize*, *timestamp*, *sequenceNumber*, *source*, *destination*. Tais informações encontram-se descritas a seguir.

Pelo fato do transporte da mensagem poder ser feito por um meio de comunicação que possua um tamanho de mensagem restrito e que não implemente por si a fragmentação necessária, a informação ***moreFragments*** é utilizada para identificar se existem mais fragmentos a serem recebidos, no caso em que ocorre a divisão da mensagem original para a transmissão. Sendo assim, todas as mensagens em sequência, que são fragmentos de uma original, terão ***moreFragments*** sinalizada, até um último fragmento, no qual ***moreFragments*** terá um valor indicando que não há mais fragmentos a serem recebidos. No caso particular em que não há fragmentação da mensagem, basta ***moreFragments*** possuir um valor que não sinaliza a existência de fragmentos.

A informação **messageType** especifica o tipo da mensagem que está sendo trafegada. Através do tipo da mensagem será realizada a filtragem de acordo com os interesses expressados pelos componentes quando instanciados (*subscribe*). Um exemplo é o interesse apenas em mensagens que são do tipo Entrada do Usuário (ver Tabela 3.1). Nesse caso, o componente só irá receber mensagens que possuem o tipo Entrada do Usuário. Os tipos de mensagens definidos para a arquitetura DoHyPE podem ser vistos na Tabela 3.1. Esses tipos não estão limitados apenas aos definidos pela arquitetura, pois a instanciação de DoHyPE permite sua extensão, podendo ser adicionadas novas mensagens que representem eventos ou comportamentos específicos do modelo hipermídia.

A informação **bodySize** especifica o tamanho do *payload* que está sendo transportado. Essa informação permite que se saiba o espaço que o corpo ocupa na mensagem, eliminando, em alguns casos, o uso de delimitadores que indicam o fim da mensagem.

A informação **timestamp** especifica o momento cronológico em que o primeiro byte da mensagem foi gerado, em milissegundos. Apesar de se ter uma limitação temporal de 1 milissegundo de precisão, o propósito deste campo é identificar a ocorrência das

mensagens ao longo de uma linha de tempo, não representando um problema a ocorrência de duas ou mais mensagens com o mesmo instante de tempo. Seu instante inicial é 0 e tem sua origem no início da apresentação. O relógio de referência para a determinação do seu valor está atrelado ao componente *Programmable Interval Timer* e é acessível por todos os componentes, inclusive remotos. Neste caso, a interface ***Distributed Event Interface*** (DEI) disponibiliza um relógio sincronizado, além dos serviços de comunicação apropriados ao *Distributed Event Bus*. Os relógios distribuídos entre as instâncias de DEI são sincronizados por protocolos como o *Precision Time Protocol* (PTP) (IEEE. . . , 2008), que possui acurácia em faixa abaixo de microssegundo.

A informação **sequenceNumber**, por sua vez, especifica um número que identifica a mensagem de forma sequencial, sequência esta que considera o conjunto de mensagens geradas por um componente de origem especificamente. Portanto, a contagem e atribuição de números de sequência fica sob responsabilidade de cada componente individualmente. Cada componente considerará o valor inicial de **sequenceNumber**. É importante ressaltar que no caso de fragmentação, cada fragmento possui seu próprio número de sequência.

As informações **source** e **destination** especificam os identificadores do componente de origem e do componente de destino, respectivamente. Conforme apresentado na seção anterior, esses identificadores são representações em códigos *hash*, gerados pelo *Context Resolver*. Apesar da informação **destination** especificar o componente de destino, em algumas situações o destinatário não é apenas um componente, ou não é necessário conhecer quem tem interesse na mensagem. Para esses casos, a informação **destination** pode receber um endereço que não representa um componente em específico, como por exemplo, um endereço com valor 00000000000000000000000000000000. O algoritmo de *hash* adotado é o *Message-Digest Algorithm* (MD5) (RIVEST, 1992), cuja probabilidade de colisão pode ser considerada aceitável no âmbito da arquitetura DoHyPE, em que o objetivo é apenas a geração de identificadores. Os identificadores não são usados como chaves para proteção da informação em si.

Além das informações de controle até aqui descritas, a mensagem carrega os dados (informação **data**) a serem enviados. Esses dados tem semântica específica para cada tipo de mensagem. A interpretação desses dados fica sob responsabilidade do componente destinatário. Como um exemplo, um componente *Model Entity Instance* pode controlar um componente *Player*, enviando mensagens para iniciar, pausar e parar a

reprodução do conteúdo. Para isso, o componente *Model Entity Instance* inicialmente notifica o *Player* qual é o conteúdo a ser reproduzido. Um exemplo dessa representação encontra-se ilustrado na Tabela 3.2. Como a quantidade de dados a serem enviados é pequena (possui 23 caracteres), não haverá fragmentação. Sendo assim, **moreFragments** não irá sinalizar fragmentação. Considerando que o evento de preparação do conteúdo tenha o código 307, o componente *Model Entity Instance* que possui identificador 3a9b0e692230a41bc35198566506b841 envia essa mensagem ao *player* em execução com identificador 239f86c6cae648133a4beeb6f97f9299, notificando-o para fazer o carregamento do conteúdo.

3.6 SUPORTE A DEPURAÇÃO

No processo de autoria com linguagens imperativas, o uso de ferramentas de depuração é uma prática comum, como o GDB (STALLMAN et al., 2002), a interface gráfica de usuário para depuração DDD (ZELLER; LUTKEHAUS, 1996), dentre outras ferramentas. Essas ferramentas fornecem algumas facilidades ao desenvolvedor para analisar a reprodução da aplicação, como o controle do fluxo de execução (passo-a-passo e *breakpoint*), monitoramento de pilhas de chamadas de funções, valores de variáveis, resolvedor de símbolos e interpretador de expressões.

Quando a autoria é realizada com linguagem declarativa, certas facilidades de depuração disponíveis em linguagens imperativas podem não ser aplicáveis ou restritas a certos construtos da linguagem. É o caso de *breakpoints* marcados em linhas de código. Em aplicações hipermídia baseadas no paradigma declarativo, o fluxo de execução é definido pela semântica dos construtos da linguagem, cuja ordem de apresentação/ocorrência não é refletida pela sequência definida por suas linhas de código. Apesar disso, certas facilidades podem ser disponibilizadas ao autor pelo componente ***Debug Agent***.

Informações de apresentação da aplicação, como valores de variáveis e estados de objetos, podem ser fornecidas ao autor. Essas informações podem ser obtidas tanto por meio de solicitação explícita aos respectivos componentes, quanto através do constante monitoramento dos barramentos da arquitetura. Sendo assim, o componente ***Debug Agent*** tem a capacidade de fornecer ao autor a obtenção e modificação desses valores. Outra facilidade que pode ser disponibilizada é a possibilidade de pausar a apresentação caso um evento em específico ocorra ou uma expressão se torne válida. Para isto, um

evento específico da arquitetura pode ser gerado, notificando a todos os componentes que devem entrar em um estado de pausa de execução. Dessa forma, cada componente fica responsável por implementar como irá reagir. Por exemplo, os ***Players*** podem realizar a pausa da reprodução do conteúdo. Os componentes que reagem a eventos de interação do usuário passam a ignorar esses eventos de interação durante esse estado. O componente responsável pelo relógio da reprodução (***Programmable Interval Timer***) suspende o incremento temporal.

Da mesma forma, a retomada da apresentação pode ser realizada através de um evento específico da arquitetura. Quando essa retomada for solicitada, os componentes voltam aos seus estados normais de reprodução. Esses eventos de apresentação são específicos da arquitetura de forma a não gerar colisões com eventos similares e com a lógica de negócio próprios do modelo suportado em uma instanciação de DoHyPE. Fazendo uso desses eventos de pausa e retomada da apresentação de DoHyPE, o componente ***Debug Agent*** pode fornecer ao autor a capacidade de pausar e reproduzir a aplicação a cada evento gerado, ou periodicamente, ou de outras formas úteis de controlar a apresentação para depuração.

Tipo	Descrição
Geral	
ERROR	Representa que houve erro na comunicação.
OK	Representa que houve sucesso na comunicação.
RESPONSE_SUCESS	Representa que a mensagem é de resposta e com sucesso.
RESPONSE_ERROR	Representa que a mensagem é de resposta e com erro.
Barramento (Bus)	
CONNECT	Requisição de conexão ao barramento.
DISCONNECT	Requisição de desconexão do barramento.
INTERESTED	Registro de interessado em um evento.
INTERESTED_UNSET	Retira um interessado em um evento.
INTERESTED_FILTER_ADD	Registro de interessado em um evento de uma origem específica.
INTERESTED_FILTER_DELETE	Remove o interessado em um evento de uma origem específica.
Programmable Interval Timer	
SINGLE_SHOT_REGISTER	Solicitar ao timer um evento único.
SINGLE_SHOT_UNREGISTER	Remover uma solicitação ao timer de evento único
PERIODIC_REGISTER	Solicitar ao timer um evento periódico.
PERIODIC_UNREGISTER	Remover uma solicitação ao timer de evento periódico.
Entrada do Usuário	
KEY_PRESSED	Uma tecla foi pressionada.
KEY_RELEASED	Uma tecla, anteriormente pressionada, foi liberada.
TOUCH_START	Um display teve toque iniciado.
TOUCH_END	Um toque, anteriormente iniciado, foi finalizado.
Media Players	
PREPARE	Executa a ação Prepare em um MediaPlayer.
PLAY	Executa a ação Play em um MediaPlayer.
STOP	Executa a ação Stop em um MediaPlayer.
PAUSE	Executa a ação Pause em um MediaPlayer.
SET_PROPERTY	Altera uma propriedade de um MediaPlayer.
ADD_ANCHOR	Adiciona uma âncora temporal em um MediaPlayer.
Atualização de Documentos	
END	Sinaliza que o Parser chegou ao fim do documento.

Tabela 3.1: Tipos de mensagens da arquitetura DoHyPE

Informação	**Valor**
moreFragments	false
messageType	307
bodySize	23
timestamp	1535385822284
sequenceNumber	28
source	3a9b0e692230a41bc35198566506b841
destination	239f86c6cae648133a4beeb6f97f9299
data	{“src”: “conteudo.mp4”}

Tabela 3.2: Exemplo de mensagem de notificação a um *player* para carregamento do conteúdo

4 PLAYER DE PROVA DE CONCEITO - DOHYPE(QT) FOR NCL

Como prova de conceito, este trabalho propõe uma instanciação da arquitetura DoHyPE sobre o *Framework* Qt (PROJECT, 2018) e com suporte a apresentação e depuração de documentos hipermídia NCL. A instanciação tira proveito do *Framework* Qt para a implementação dos barramentos de comunicação entre os componentes e para o suporte a diversos dos recursos de entrada e saída, conforme descrito na Seção 4.2. Para o suporte a documentos NCL, a instanciação segue uma abordagem que restringe o conjunto de entidades representadas, sem perda relevante de expressão, conforme descrito na Seção 5.2. A implementação foi feita utilizando a linguagem C++, explorando o paradigma Orientado a Objetos. Além disso, foi definido um formato para as mensagens que visa otimizar a entrega e a interpretação. Dessa forma, a mensagem foi divida em duas partes: o cabeçalho e o corpo. O cabeçalho carrega as informações de controle, enquanto o corpo contém os dados da mensagem. Essas partes serão descritas na próxima Seção.

4.1 PROTOCOLO DE COMUNICAÇÃO

Com intuito de reduzir o custo computacional em interpretar o cabeçalho da mensagem, foi adotado um cabeçalho de tamanho fixo. Isso auxilia o barramento realizar a decodificação das mensagens trafegadas e encaminhar ao destinatário mais rapidamente, tendo um gasto computacional menor ao interpretar o cabeçalho. A Figura 4.1 ilustra o formato do cabeçalho adotado. Todos os campos foram definidos baseados na descrição feita na Seção 3.5.1.

O campo **moreFrame** é representado pelo campo **m**. As mensagens em sequência, que são fragmentos de uma original, terão o campo **m** com o valor 1, até um último fragmento, no qual **m** tenha valor 0. No caso particular em que não há fragmentação da mensagem, basta o campo **m** ter valor 0. Os próximos 15 bits são dedicados ao campo **type** (**messageType**), que especifica o tipo da mensagem que está sendo trafegada.

O campo **size** tem um tamanho de 16 bits e representa o campo **bodySize**. Ele especifica o tamanho do *payload* que está sendo transportado, em bytes. Sendo assim, os

dados transportados no corpo da mensagem têm um limite de tamanho de 65535 bytes. Caso ocorra de se ter uma mensagem maior do que este limite, basta dividir o corpo da mensagem em tamanhos de até 65535 e atribuir o campo **m** para 0, quando for o último fragmento da divisão.

O campo **timestamp** especifica o momento cronológico em que o primeiro byte da mensagem foi gerado, em milissegundos. Por ter 48 bits, seu valor máximo pode chegar a aproximadamente $2,81 * 10^{14}$. O campo **seqnum** representa o campo **sequenceNumber** especifica um número que identifica a mensagem de forma sequencial. Cada componente considerará o valor inicial de **seqnum** como 0 e, por ter 48 bits, seu valor máximo pode chegar a aproximadamente $2,81 * 10^{14}$.

Os campos **source** e **destination** especificam os identificadores do componente de origem e do componente de destino, respectivamente. O resultado do algoritmo *hash* sempre terá um tamanho de 128 bits. Dessa forma, os campos **source** e **destination** possuem um tamanho fixo de 128 bits.

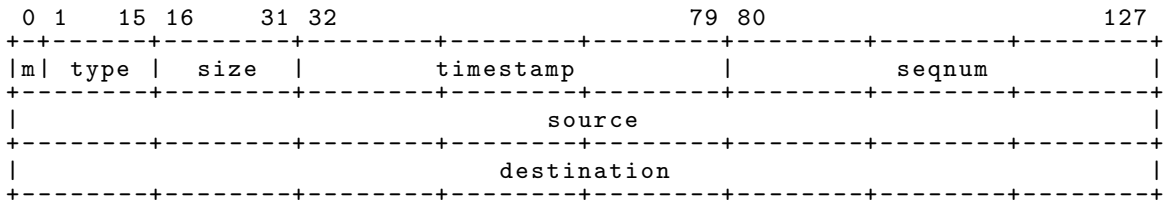

Figura 4.1: Cabeçalho da mensagem

Para permitir uma estruturação dos dados incluídos no corpo da mensagem, de forma menos verbosa e mais leve, esta instanciação adota o formato JSON. Cada informação a ser passada poderá ser representada utilizando o formato chave e valor. Uma estruturação genérica para ilustrar o formato do corpo de uma mensagem pode ser vista na Figura 4.2.

```
{"chave1": "valor1",
 "chave2": "valor2",
 "chave3": {"chave3_1": "valor3_1",
            "chave3_2": "valor3_2",
            "chave3_3": "valor3_3",
            ...}
}
```

Figura 4.2: Estrutura do corpo das mensagens

4.2 DOHYPE(QT)

Qt (PROJECT, 2018) é um *framework* para o desenvolvimento de interfaces gráficas multiplataforma em linguagem C++. Ele possui um conjunto de ferramentas que facilitam desde o desenvolvimento de propósito geral até mesmo o desenvolvimento específico multimídia. Além disso, possui uma documentação sólida e seu código é mantido pelo Qt Project.

Por ser multiplataforma, uma das características do Qt é a utilização de um conjunto de classes e tipos, abstraindo ao desenvolvedor como são implementadas na plataforma. Para facilitar a implementação de interfaces, Qt possui o módulo *QGraphicView*, que é um conjunto de bibliotecas padronizadas que permitem a manipulação de variados itens gráficos em 2D e a renderização desses itens. Essa manipulação contempla a criação desses elementos e a notificação de eventos, como seleção, clique e outros. Qt possui uma característica central que é o mecanismo de comunicação entre objetos: *Signals & Slots*. Esse mecanismo se baseia no conceito de *callback*, no qual um trecho de código é executado quando um evento ocorre. Para isto, o Qt permite que um sinal (ocorrência de um evento) seja emitido e repassado aos objetos que tenham interesse nesse sinal. O interesse é notificado através de um mecanismo de conexão, onde um objeto conecta um sinal a um *slot* (método ou função). O Qt permite que o repasse seja feito tanto na *thread* do objeto que deseja receber o sinal (*Qt::QueuedConnection*) quanto na *thread* que foi realizada a sinalização (*Qt::DirectConnection*). Na linguagem C++, esses *slots* são métodos definidos em uma classe. Em especial, na versão C++11, *slots* podem ser funções anônimas, conhecidas como expressões *lambda*. Esses recursos fornecidos tanto pelo *framework* Qt quanto pela linguagem C++ auxiliaram na implementação.

A Figura 4.3 mostra, de forma resumida, o diagrama de classes da DoHyPE(QT). Nela está descrita a representação dos componentes da arquitetura DoHyPE. A implementação da comunicação da DoHyPE(Qt) visa abstrair ao desenvolvedor o protocolo de comunicação utilizado. Para isso, a comunicação é divida em duas partes: as classes que recebem e as que enviam mensagens. As classes que recebem mensagens, são especializações da classe *DoHyPEReceiverPeer*, que é uma interface que garante a existência do método *onReceiveMessage* em suas heranças. Sendo assim, essas classes que são especializações de *DoHyPEReceiverPeer*, implementam o tratamento do conteúdo recebido, que são: o cabeçalho (instância da classe *DoHyPEMessageHeader*), o corpo da mensagem (conteúdo

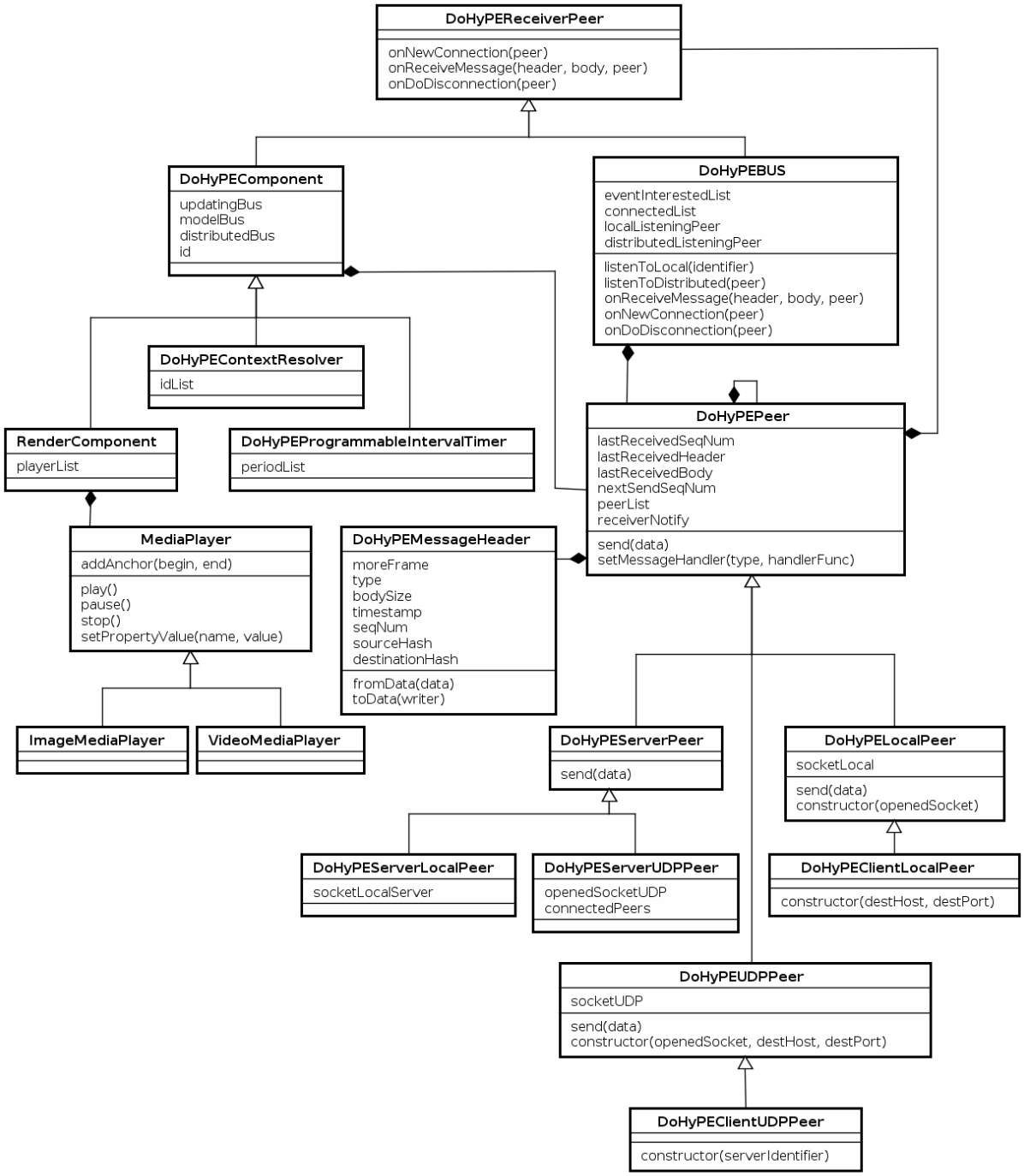

Figura 4.3: Diagrama de classe da biblioteca DoHyPE(Qt)

textual, do tipo *String*) e uma instância da classe *DoHyPEPeer*, que representa o emissor da mensagem. Além disso, essas classes que são filhas de *DoHyPEReceiverPeer*, são notificadas quando uma nova conexão é realizada (*onNewConnection*) ou quando é encerrada (*onDoDisconnection*). Apesar de existir protocolos de comunicações que não são orientados à conexão, como o protocolo UDP, essa abstração é necessária para a criação

de objetos (instâncias) que representam o emissor. Com o intuito de agilizar a comunicação, a implementação dos barramentos faz uso dessa abstração de conexão, permitindo gerenciar o recebimento e o repasse das mensagens mais rapidamente. Essa abstração encontra-se discutida mais a frente.

A classe *DoHyPEPeer* representa um *endpoint* da comunicação, estabelecendo uma interface que permite o envio de mensagens ao destinatário através do método *send*. As classes que herdam de *DoHyPEPeer* ficam responsáveis por destinar corretamente o conteúdo ao protocolo de comunicação. Como ela abstrai o protocolo de comunicação utilizado, a classe *DoHyPEPeer* tem a função de receber o conteúdo do protocolo de comunicação (abstrações como *sockets* e afins), criar a instância que representa o cabeçalho da mensagem (*DoHyPEMessageHeader*) e repassar a uma instância de *DoHyPEReceiverPeer*, que é previamente notificada como sendo a receptora de mensagens (*receiverNotify*). Além disso, a classe *DoHyPEPeer* tem o papel de fazer a ordenação das mensagens recebidas (utiliza o campo *seqnum* do cabeçalho como referência), gerenciar o número de sequência das mensagens que são enviadas e atribuir o valor do *timestamp* no cabeçalho da mensagem a ser enviada. Para facilitar a implementação de classes que herdam de *DoHyPEPeer*, ela permite o registro de tratadores de tipos de mensagens específicos (campo *type* do cabeçalho) através do método *setMessageHandler*. Sendo assim, o tratador (*handlerFunc*) específico de cada tipo é executado quando a mensagem é recebida. Essa classe também permite o registro de um tratador quando não for encontrado tratador registrado para um tipo de mensagem. Os tipos de mensagens podem ser vistos na Tabela 3.1, que agrupa os tipos de mensagens por componentes da arquitetura ou pelo grupo que as representam.

As classes *DoHyPELocalPeer* e *DoHyPEUDPPeer* são implementações da classe *DoHyPEPeer* para comunicações IPC e UDP, respectivamente. *DoHyPELocalPeer* é implementada utilizando o módulo que representa *socket* para conexões locais (*QLocalSocket* do *framework* Qt). Para a instanciação dessa classe, é necessário fornecer qual *socket* será utilizado para o envio e recebimento de mensagens. A classe *DoHypeUDPPeer* é implementada utilizando o módulo que representa *socket* UDP (*QUdpSocket* do *framework* Qt). Como UDP não é orientado à conexão, na instanciação dessa classe é necessário fornecer o *socket* UDP que será usado para fazer o envio dos datagramas, evitando a criação de um novo *socket* UDP para cada envio. Além disso, é preciso informar o endereço de destino

e a porta do *host* que está sendo representado por esta classe. Isso faz com que, quando usado o método *send*, nenhuma informação de destino seja necessária, criando a abstração de conexão.

A classe *DoHyPEServerPeer* é responsável pela criação do meio pelo qual clientes se conectam aos barramentos. Ela é implementada em duas classes: *DoHyPEServerLocalPeer*, que implementa um servidor que espera conexões locais apenas (IPC) e a classe *DoHyPEServerUDPPeer*, que representa uma abstração de servidor UDP. *DoHyPEServerLocalPeer* utiliza o componente o *QLocalServer* do Qt. Esse componente permite a criação de comunicação local IPC através de um identificador textual apenas, abstraindo ao desenvolvedor como ela é feita. Em ambientes Windows, essa conexão é feita utilizando o recurso Pipe Nomeado (*Named Pipe*). Em ambientes Unix, é utilizado Socket de Domínio Local (Domain Socket). Pelo fato de *QLocalServer* ser orientado a conexão, quando uma nova conexão é solicitada, uma instância da classe *DoHyPELocalPeer* é gerada, fornecendo a essa instância o socket *QLocalSocket* que representa o cliente que solicitou a conexão. Já a *DoHyPEServerUDPPeer* possui uma lista de instâncias *DoHyPEUDPPeer*, identificadas pelo o endereço IP e a Porta pelo qual foi feita a requisição de conexão. Quando *DoHyPEServerUDPPeer* recebe um datagrama, é verificado se já existe *DoHyPELocalPeer* associado a IP e Porta de origem. Caso não exista, uma instância de *DoHyPELocalPeer* é gerada, fornecendo o *socket QUdpSocket* do *DoHyPEServerUDPPeer* e o IP e Porta da origem como destino. Dessa forma, é possível ao *DoHyPEPeer* controlar a ordenação e o sequenciamento de pacotes.

A classe *DoHyPEBUS* é a representação de todos os barramentos da arquitetura. Esses barramentos são orientados a conexão, permitindo a identificação dos integrantes do barramento. Sendo assim, antes de ocorrer qualquer comunicação ou solicitação ao barramento, deverá ser realizada a conexão. Quando o componente realiza a conexão com o barramento, o componente poderá enviar mensagens a outros através do identificador de destino. Esse componente poderá solicitar ao barramento que notifique a ocorrência de mensagens de um tipo específico. Como um tipo de mensagem pode ser proveniente de variadas origens, o barramento pode ser notificado de qual identificador de origem se tem interesse, podendo filtrar o envio de mensagens. A implementação da classe *DoHyPEBUS* permite o recebimento de mensagens pelo protocolo de comunicação local (*DoHyPEServerLocalPeer*) e/ou pelo protocolo de comunicação UDP (*DoHyPEServerUDPPeer*).

Os componentes *Player* da arquitetura DoHyPE são instanciados utilizando classes do módulo Qt que facilitam a decodificação, reprodução e exibição de conteúdos multimídia. A classe *MediaPlayer* tem o propósito de implementar as ações e operações comuns aos componentes *Player*, como a alteração de propriedades de exibição (posicionamento, dimensão, transparência, dentre outros), o controle de reprodução (*play*, *pause* e *stop*) e a criação de âncoras temporais. As propriedades de exibição suportadas pelos componentes *Player* DoHyPE(Qt) estão descritas na Tabela 4.1. Para a exibição do conteúdo de mídia discreta (por exemplo, imagem e texto), a classe *QGraphicsPixmapItem* é utilizada. No caso de imagens, a classe *QPixmap* faz a decodificação de vários formatos de imagem por padrão, como PNG (*Portable Network Graphics*), JPG (*Joint Photographic Experts Group*), BMP (*Windows Bitmap*), GIF (*Graphic Interchange Format*), dentre outros formatos. Sendo assim, a classe *ImageMediaPlayer* tem a função de repassar o conteúdo da mídia e controlar a instância *QGraphicsPixmapItem*. Para mídia contínua (por exemplo, vídeo e áudio), a classe *QMediaPlayer* é utilizada e fornece métodos que permitem o controle da reprodução, como *play*, *pause* e *stop*. Nessa implementação, a classe *RenderComponent* tem a função de gerenciar as instâncias de *MediaPlayer* e fazer a junção dos resultados decodificados por essas instâncias, exibindo a composição final.

A classe *DoHyPEProgrammableIntervalTimer* utiliza *QTimer* do Qt para a gerência de intervalos temporais em milissegundo. *QTimer* permite que intervalos de tempos sejam criados, repetitivos ou não, possibilitando a notificação da ocorrência destes intervalos aos interessados. Sendo assim, *DoHyPEProgrammableIntervalTimer* gerencia a notificação de intervalos temporais, periódicos ou não, de interesse previamente informado pelos componentes.

A classe *DoHyPEComponent* representa todos os componentes da arquitetura que tem o papel de cliente na comunicação. Os barramentos agem como servidores na comunicação, diferentemente dos outros componentes, não sendo descendentes dessa classe. *DoHyPEComponent* implementa o estabelecimento de comunicação com os barramentos da arquitetura. Pelo fato de ser descendente da classe *DoHyPEReceiverPeer*, todas as suas instâncias podem ser receptoras de mensagens de instâncias da classe *DoHyPEPeer*. Além disso, *DoHyPEComponent* armazena o identificador que representa o componente. Sendo assim, todos os componentes da arquitetura que estabelecem comunicação como clientes herdam dessa classe. A Tabela 4.2 lista o meapeamento entre classes DoHyPE(Qt) e cada

Propriedade	Descrição
left	Representa a posição esquerda do conteúdo. Pode ser um valor inteiro (pixels) ou um valor em porcentagem entre 0% e 100%.
top	Representa a posição topo do conteúdo. Pode ser um valor inteiro (pixels) ou um valor em porcentagem entre 0% e 100%.
width	Representa a largura do conteúdo. Pode ser um valor inteiro (pixels) ou um valor em porcentagem entre 0% e 100%.
height	Representa a altura do conteúdo. Pode ser um valor inteiro (pixels) ou um valor em porcentagem entre 0% e 100%.
bounds	Permite alterar a posição e o tamanho do conteúdo. Quatro parâmetros são necessários: *left*, *top*, *width* e *height*, nesta ordem. As representações de cada parâmetro estão descritas acima.
location	Permite alterar a posição do conteúdo. Dois parâmetros são necessários: *left* e *top*, nessa ordem. As representações de cada parâmetro estão descritas acima.
size	Permite alterar o tamanho do conteúdo. Dois parâmetros são necessários: *width* e *height*, nessa ordem. As representações de cada parâmetro estão descritas acima.
zIndex	Permite alterar a posição na precedência de exibição (pilha de exibição). Valores mais altos sobrepõe elementos com valores menores.
transparency	Representa o quão transparente o conteúdo deverá ser exibido. Pode ser um valor entre 0 e 1 ou um valor entre 0% e 100%.
visible	Representa se o conteúdo deve ser exibido ou não. Quando *true* exibe o conteúdo. Caso contrário, esconde o conteúdo
fit	Representa como deverá ser feito o redimensionamento do conteúdo. Se o valor meet for informado, a proporção (aspect ratio) é mantida.
explicitDur	Indica a duração explícita (em segundos) do conteúdo. Quando a duração é atingida, a exibição do conteúdo é encerrada.

Tabela 4.1: Propriedades suportadas pelos componentes *Player* DoHyPE(Qt)

componente da arquitetura DoHyPE.

4.3 DOHYPE(QT) FOR NCL

A *Nested Context Language* (NCL) é uma linguagem declarativa para a descrição de apresentações hipermídia. Ela é um padrão para aplicações interativas no Sistema Brasileiro de Televisão Digital (SBTVD) (ABNT, 2008) e também uma Recomendação ITU-T (ITU-T, 2014) para sistemas IPTV. NCL se baseia no modelo conceitual *Nexted Context Model*

Componente da arquitetura	**Implementa**
Resolvedor de Contexto	
Context Resolver	DoHyPEContextResolver (herda de DoHyPEComponent)
Máquina de apresentação	
Document Parser Document Editing Commands Entity Mananager Model Entity Instance	DoHyPEComponent
Programmable Interval Timer	DoHyPEProgrammableIntervalTimer (herda de DoHyPEComponent)
Recursos de entrada e saída	
Player	MediaPlayer
Renderer	RenderComponent (herda de DoHyPEComponent)
Remote Control Sensor Actuator	DoHyPEComponent
Monitoramento e controle	
Debug Agent Audience Measurement Agent Prefetch Manager	DoHyPEComponent
Barramentos de Comunicação	
Document Updating Message Bus Hypermedia Model Message Bus Distributed Event Bus	DoHyPEBUS

Tabela 4.2: Classes que representam os componentes da arquitetura DoHyPE em DoHyPE(Qt)

(NCM), que tem seu foco na representação e tratamentos de documentos hipermídia.

Para a representação do modelo NCM, cada uma de suas entidades relevantes se tornam uma especialização de *Model Entity Instance* da arquitetura. Decisões para reduzir a complexidade da implementação podem ser feitas, a fim de tornar o número de entidades necessárias menor. Dessa forma, certas entidades do modelo poderão ser implementadas atreladas as entidades das quais fazem parte, por exemplo, os Conectores atrelados aos Elos, Descritores atrelados aos Nós, dentre outros. Assim como descrito no modelo NCM,

toda instância de entidade do modelo possuirá um identificador único (ID) e uma coleção de atributos, podendo ser alterados ou consultados mediante requisições. O cuidado que se deve ter é manter uma estrutura no *Entity Manager* para que se consiga, através dela, identificar a reflexão de uma edição, por exemplo, notificar todos que estão associados a uma região a alteração de uma propriedade.

O elemento **<media>** de NCL é uma representação de Nós de Conteúdo e é representado como uma *Model Entity Instance*. Todo o seu comportamento de apresentação, mudanças de estado (start, stop, abort, pause, resume), será notificado no *Hypermedia Model Event Bus*. Um Elo então, definido como elemento **<link>** na linguagem, pode tomar forma de outro *Model Entity Instance*, capturar esta mensagem e agir conforme as informações dos seus conectores, nele embutidos. O interesse na obtenção destas mensagens é feita pelo próprio **<link>** quando ele é criado, recebendo dos barramentos apenas o que lhe possa ser útil. O Nó Switch, definido como elemento **<switch>** na linguagem, também será representado através de um componente *Model Entity Instance*. Assim como o elemento **<link>**, ele poderá notificar ao *Hypermedia Model Event Bus* interesse em determinados eventos que alteram sua escolha. Para manter toda a estrutura de contextos, o elemento **<context>** também será representado como um componente *Model Entity Instance*.

O elemento **<body>** representa o corpo do documento. Este elemento possui comportamento similar ao **<context>**, que é um Nó de Contexto. Sendo assim, associado a este elemento será criado um **<context>**, que irá representá-lo, evitando se ter exceções para a representação do elemento **<body>**. A seguir será descrito como esses elementos são definidos nessa instanciação.

A instanciação da arquitetura para a linguagem NCL foi realizada utilizando DoHyPE(Qt) como base. Além disso, foi utilizado um conjunto restrito de elementos da linguagem. Nessa instanciação, os construtos que foram implementados são: <*context*>, <*media*>, <*property*>, <*link*>, <*bind*>, <*bindParam*>, <*switch*>, <*rule*>, <*bindRule*>, <*port*> e os conectores (<*causalConnector*>), incluindo os construtos <*simpleCondition*>, <*compoundCondition*>, <*simpleAction*> e <*compoundAction*>. Regiões e descritores não foram implementados. Dessa forma, as propriedades da mídia são atribuídas através do construto <*property*>.

A implementação do modelo da linguagem ocorre sobre as classes e funcionalidades

que a DoHyPE(Qt) oferecem. Pelo fato da DoHyPE(Qt) implementar as funcionalidades básicas da arquitetura, por exemplo, a comunicação e os componentes integrantes dessa comunicação, a instanciação é feita baseada na classe *DoHyPEComponent* apenas. Sendo assim, os componentes do modelo da linguagem NCL são implementados como especialização dessa classe. De uma forma geral, a especialização de *Model Entity Instance* é feita através da classe *DoHyPEComponent*. A Figura 4.4 mostra, de forma resumida, o diagrama de classes da implementação de *DoHyPE(Qt) for NCL*.

A classe *NCLComponent* foi projetada com intuito de facilitar o desenvolvimento das classes que representam as entidades do modelo. Essa classe representa o que há de comum entre as entidades do modelo. A linguagem NCL define uma máquina de estado para os eventos que possui 3 estados: *sleeping*, *paused* e *ocurring*. Pelo fato de um evento de apresentação estar associado ao elemento da linguagem, por exemplo, ao elemento *media*, a classe *NCLComponent* guarda o estado atual de apresentação. Para cada alteração nos estados da máquina de estado, uma mensagem é gerada no barramento, sem um destinatário em específico. O barramento analisa se há componente interessado nesse tipo de mensagem, fazendo a filtragem pela origem (conforme discutido na Seção 4.2), e repassa aos interessados, que reagem conforme as regras do modelo da linguagem. A mudança de estado de apresentação é realizada através dos métodos *start()*, *stop()*, *abort()*, *pause()* e *resume()*.

Assim como discutido na Seção 3.5.1, os tipos de mensagens que podem ser utilizadas não se limitam apenas aos tipos definidos pela arquitetura. Sendo assim, nessa instanciação os tipos de mensagens são estendidos para abranger os eventos e ações específicos do modelo no qual a linguagem NCL se baseia. Os tipos de mensagens adicionadas nessa expansão podem ser vistos na Tabela 4.3.

O evento de atribuição, por exemplo, alteração do valor de uma propriedade, é gerado através do método *set()*. A execução do método cria uma transição de estados com uma durabilidade curta, conforme previsto na especificação do modelo da linguagem para a máquina de estado de eventos de atribuição (SOARES et al., 2007). Sendo assim, quando esse método é invocado, há uma transição natural da máquina de estado do evento de atribuição, saindo do estado *sleeping* para *ocurring*, gerando uma mensagem do tipo *ATTRIBUTION_STARTS*. Quando essa atribuição é concluída, normalmente em um curto período de tempo, essa máquina de estado do evento de atribuição sai do estado de

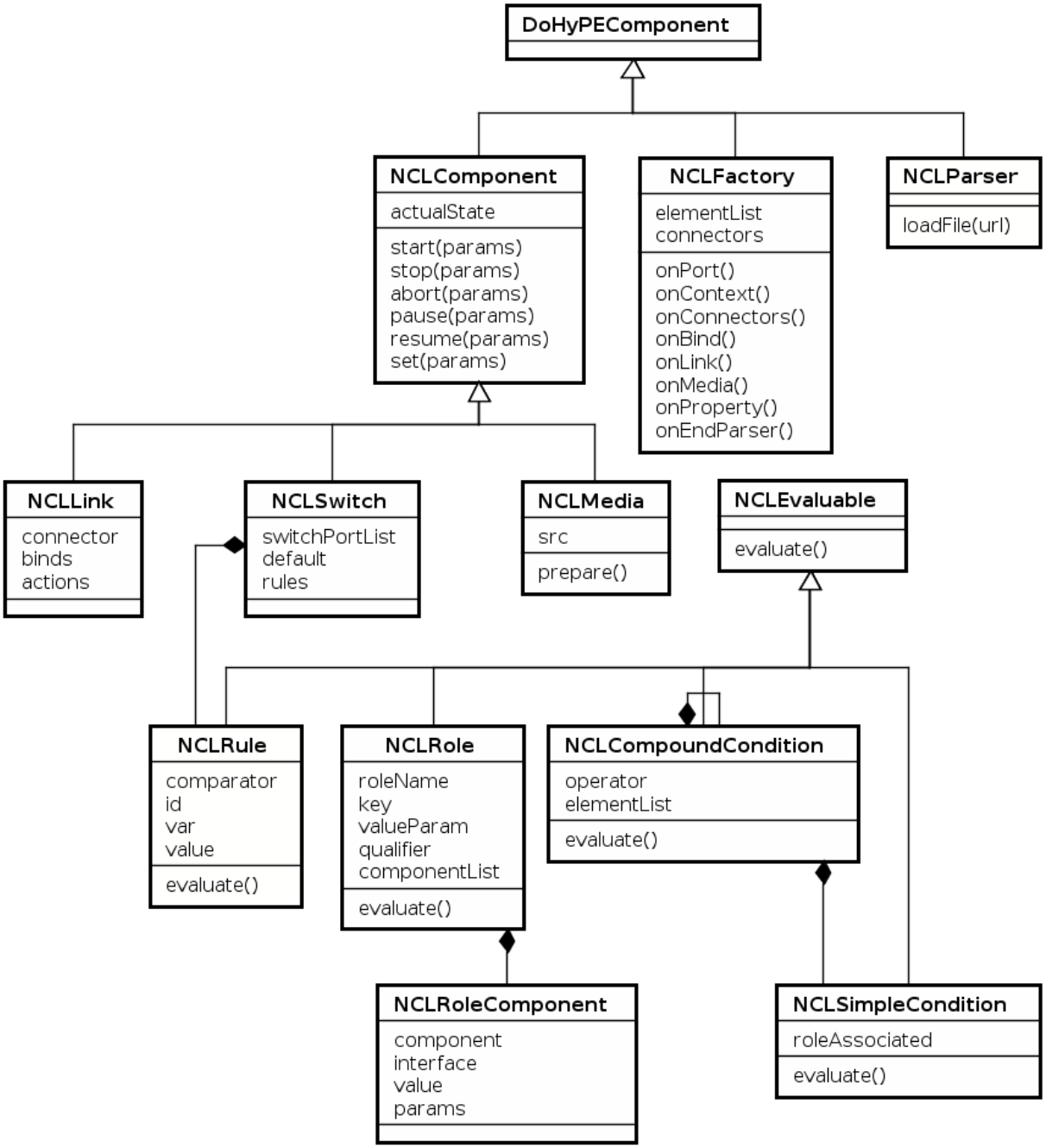

Figura 4.4: Diagrama de classe da *DoHyPE(Qt) for NCL*

ocurring para o estado *sleeping*, gerando uma mensagem do tipo *ATTRIBUTION_STOPS*. Caso haja uma interrupção da atribuição, a máquina de estado passa para *sleeping* através da ação *abort*, gerando uma mensagem do tipo *ATTRIBUTION_ABORTS*.

A máquina de estado referente a eventos de seleção é gerenciada pelo componente da arquitetura responsável pelo evento. Nesse caso, a transição entre os estados gera mensagens de tipos que representam as transições de estados da seleção. Vale ressaltar

Tipo	Descrição
Factory	
MEDIA	Representa um construto *media* encontrado pelo *parser*.
PROPERTY	Representa um construto *property* encontrado pelo *parser*.
CONTEXT	Representa um construto *context* encontrado pelo *parser*.
SWITCH	Representa um construto *switch* encontrado pelo *parser*.
LINK	Representa um construto *link* encontrado pelo *parser*.
BIND	Representa um construto *bind* encontrado pelo *parser*.
RULE	Representa um construto *rule* encontrado pelo *parser*.
BIND_RULE	Representa um construto *bindRule* encontrado pelo *parser*.
PORT	Representa um construto *port* encontrado pelo *parser*.
CONNECTORS	Representa os conectores encontrados pelo *parser*.
Ações	
START	Executa a ação *start* no elemento *NCLComponent*.
STOP	Executa a ação *stop* no elemento *NCLComponent*.
ABORT	Executa a ação *abort* no elemento *NCLComponent*.
PAUSE	Executa a ação *pause* no elemento *NCLComponent*.
RESUME	Executa a ação *resume* no elemento *NCLComponent*.
SET	Executa a ação *set* no elemento *NCLComponent*.
Contexto	
ADD_PORT	Adiciona porta ao contexto de destino.
Transições (notificação de mudança na máquina de estados de eventos)	
PRESENTATION_STARTS	Eventos de apresentação (transição *starts*).
PRESENTATION_STOPS	Eventos de apresentação (transição *stops*).
PRESENTATION_ABORTS	Eventos de apresentação (transição *aborts*).
PRESENTATION_PAUSES	Eventos de apresentação (transição *pauses*).
PRESENTATION_RESUMES	Eventos de apresentação (transição *resumes*).
SELECTION_STARTS	Eventos de seleção (transição *starts*).
SELECTION_STOPS	Eventos de seleção (transição *stops*).
SELECTION_ABORTS	Eventos de seleção (transição *aborts*).
SELECTION_PAUSES	Eventos de seleção (transição *pauses*).
SELECTION_RESUMES	Eventos de seleção (transição *resumes*).
ATTRIBUTION_STARTS	Eventos de atribuição (transição *starts*).
ATTRIBUTION_STOPS	Eventos de atribuição (transição *stops*).
ATTRIBUTION_ABORTS	Eventos de atribuição (transição *aborts*).
ATTRIBUTION_PAUSES	Eventos de atribuição (transição *pauses*).
ATTRIBUTION_RESUMES	Eventos de atribuição (transição *resumes*).

Tabela 4.3: Tipos de mensagens de DoHyPE(Qt) for NCL

que eventos de cliques e pressionamentos de teclas não geram eventos de seleção e são notificados através do barramento *Distributed Event Bus*. Os eventos de seleção são

gerados pelas instâncias da entidade do modelo (*Entity Model Instance*) ou componente *Player*. Ao receberem esses eventos de entrada, esses componentes agem conforme as regras do modelo, gerando no barramento (*Hypermedia Model Message Bus*) mensagens referentes à transição no estado do evento de seleção. Os tipos de mensagens associados às transições podem ser vistos na Tabela 4.4.

Transição (ação)	**Nome da transição**	**Tipo da mensagem**
Eventos de apresentação		
sleeping->ocurring (start)	starts	PRESENTATION_STARTS
ocurring->sleeping (stop)	stops	PRESENTATION_STOPS
ocurring->sleeping (abort)	aborts	PRESENTATION_ABORTS
ocurring->paused (pause)	pauses	PRESENTATION_PAUSES
paused->ocurring (start/resume)	resumes	PRESENTATION_RESUMES
paused->sleeping (stop)	stops	PRESENTATION_STOPS
paused->sleeping (abort)	aborts	PRESENTATION_ABORTS
Eventos de atribuição		
sleeping->ocurring (start)	starts	ATTRIBUTION_STARTS
ocurring->sleeping (stop)	stops	ATTRIBUTION_STOPS
ocurring->sleeping (abort)	aborts	ATTRIBUTION_ABORTS
ocurring->paused (pause)	pauses	ATTRIBUTION_PAUSES
paused->ocurring (start/resume)	resumes	ATTRIBUTION_RESUMES
paused->sleeping (stop)	stops	ATTRIBUTION_STOPS
paused->sleeping (abort)	aborts	ATTRIBUTION_ABORTS
Eventos de seleção		
sleeping->ocurring (start)	starts	SELECTION_STARTS
ocurring->sleeping (stop)	stops	SELECTION_STOPS
ocurring->sleeping (abort)	aborts	SELECTION_ABORTS
ocurring->paused (pause)	pauses	SELECTION_PAUSES
paused->ocurring (start/resume)	resumes	SELECTION_RESUMES
paused->sleeping (stop)	stops	SELECTION_STOPS
paused->sleeping (abort)	aborts	SELECTION_RESUMES

Tabela 4.4: Tipos de mensagens associadas às transições das máquinas de estado de eventos NCL

O *NCLFactory* é a especialização do *Entity Manager*, tendo a função de validar as requisições e criar as instâncias do modelo. A criação é feita quando o *NCLParser* encontra os construtos do documento e notifica ao *NCLFactory*, informando a qual contexto o construto pertence. O *NCLFactory* guarda a lista de conectores que foram definidos no documento e a lista dos elementos que foram criados. No momento da criação, o *NCLFactory* solicita ao *DoHyPEContextResolver* um identificador para a nova instân-

cia, fornecendo ao *DoHyPEContextResolver* o contexto que o elemento faz parte. Dessa forma, tanto a resolução de contexto quanto o identificador podem ser obtidos a qualquer momento junto ao *DoHyPEContextResolver*.

Para cada construto reconhecido pelo *NCLParser*, uma mensagem é enviada ao *NCLFactory* com o tipo da mensagem que representa o construto encontrado. Quando *NCLParser* notifica a descoberta dos elementos <*context*>, <*link*>, <*media*> e <*switch*>, o *NCLFactory* cria as instâncias de *NCLContext*, *NCLLink*, *NCLMedia* e *NCLSwitch*, respectivamente. Quando o construto descoberto é o <*bind*>, o *NCLFactory* faz a atribuição do <*bind*> encontrado ao <*link*> do qual faz parte, através do contexto que é fornecido pelo *NCLParser*. Quando o construto é o <*property*>, o *NCLFactory* envia uma mensagem para alterar o valor da propriedade do componente associado ao contexto informado. Ao encerrar a leitura do documento, o *NCLParser* notifica o *NCLFactory*. O *NCLFactory* então inicia a apresentação, enviando uma mensagem de ação *start* ao *NCLContext* raiz, que é a representação do elemento <*body*> do documento NCL.

O *NCLLink* ao ser criado, recebe o conector associado que permite validar as condições definidas e executar as ações estabelecidas. No papel de condições dos conectores, *NCLCompoundCodition* representa as condições compostas (*compoundCondition*) da linguagem e pode ter um ou mais filhos *NCLSimpleCondition* (condições singulares *simpleCondition*), e/ou outros *NCLCompoundCodition*. *NCLCompoundCodition* e *NCLSimpleCondition* são especializações de *NCLEvaluable*. A classe *NCLEvaluable* representa uma interface que define um método *evaluable* para as classes descendentes. Sendo assim, a classe *NCLSimpleCondition* implementa o método *evaluate()* validando o *NCLRole* associado ao *NCLSimpleCondition*, utilizando o qualificador *qualifier* para operar entre os componentes integrantes desse *NCLRole*. Já *NCLCompoundCodition* implementa o método *evaluate()* validando cada condição filha e fazendo a operação de junção desses resultados com o operador *operator*.

No momento que um *NCLLink* é instanciado, a árvore de condições do conector é criada, permitindo que as associações pelo construto <*bind*> da linguagem sejam feitas. Para cada construto <*bind*> encontrado, o *NCLLink* notifica ao barramento que deseja receber a mensagem que permite validar a condição (*role*) associada ao <*bind*>. A Tabela 4.5 descreve o tipo de mensagem associado a cada *role* da linguagem NCL. Para solicitar ao barramento que notifique a ocorrência de mensagem, o *NCLLink* fornece qual o tipo

de mensagem e o identificador da origem de interesse (atributo *component* do construto <*bind*>). Para cada mensagem recebida pelo *NCLLink*, a condição associada ao <*bind*> realizado, que tem seu valor inicial sendo *false*, é alterada para *true*. Cada vez que essa alteração é realizada, as condições são validadas novamente. Quando há sucesso na validação de todas as condições do conector, as ações estabelecidas são executadas e as condições são alteradas para o valor inicial.

O *NCLSwitch* tem um comportamento diferente do *NCLLink*. Ao invés de aguardar a ocorrência de mensagens de diferentes origens, o *NCLSwitch* solicita os valores das propriedades do componente associado as regras (*Rule*) criadas no documento. Dessa forma, a regra é validada e a decisão de escolha é feita. Essa decisão dura até que a ação de *stop* do *NCLSwitch* seja executada.

Role	**Tipo de mensagem associada**
onBegin	PRESENTATION_STARTS
onEnd	PRESENTATION_STOPS
onPause	PRESENTATION_PAUSES
onAbort	PRESENTATION_ABORTS
onResume	PRESENTATION_RESUMES
onSelection	SELECTION_STARTS
onBeginSelection	SELECTION_STARTS
onEndSelection	SELECTION_STOPS
onAbortSelection	SELECTION_ABORTS
onPauseSelection	SELECTION_PAUSES
onResumeSelection	SELECTION_RESUMES
onBeginAttribution	ATTRIBUTION_STARTS
onEndAttribution	ATTRIBUTION_STOPS
onPauseAttribution	ATTRIBUTION_PAUSES
onResumeAttribution	ATTRIBUTION_RESUMES
onAbortAttribution	ATTRIBUTION_ABORTS

Tabela 4.5: Tipo de mensagem associada a cada *Role* da linguagem NCL

O *NCLMedia* controla o *Player* responsável por reproduzir o conteúdo da mídia. Quando a máquina de estado do *NCLMedia* é alterada, a transição responsável por essa alteração é refletida ao *Player*. Dessa forma, se a instância de *NCLMedia* é pausada (método *pause()* do NCLComponent), essa instância envia mensagem ao *Player* associado para que a reprodução do conteúdo também seja pausada. O mesmo ocorre para as outras ações. Para cada âncora temporal de *NCLMedia* definida no documento, uma âncora é criada junto ao *Player* associado. No momento em que o *Player* é instanciado, uma

âncora temporal é criada automaticamente, representando o conteúdo em sua totalidade temporal. Quando ocorre evento associado a âncora temporal, por exemplo, início ou o fim, o *NCLMedia* é notificado pelo *Player*. Dessa forma, o *NCLMedia* envia mensagem no barramento *Hypermedia Model Message Bus* associada ao evento ocorrido, informando no conteúdo da mensagem a âncora responsável. Assim, os *NCLLink* que têm condições atreladas ao *NCLMedia* reagem conforme estabelecido no documento, resolvendo interface de ligação. No caso particular da âncora ser a que representa o conteúdo total, não é preciso informar a âncora no conteúdo da mensagem.

5 CASOS DE USO

A abordagem arquitetural de DoHyPE permite que a depuração seja realizada com a agregação de diversos ferramentais. Um destes é a visualização temporal do momento em que os eventos ocorrem, como início de uma mídia, o disparo de um elo ou até mesmo a adaptação de conteúdo. Isto fornece ao autor um meio de identificar as razões de uma ação não ocorrer em seu documento hipermídia. Outro ferramental é a alteração de atributos que podem ser injetados durante a apresentação, possibilitando se ver em tempo real o resultado, não sendo necessário reiniciar a apresentação para isso.

5.1 AGENTE DE DEPURAÇÃO COM SUPORTE A SALTO TEMPORAL

A partir dos eventos gerados, pode se criar uma estrutura de dados que represente a sincronização entre os objetos de mídia, permitindo que se tenha saltos temporais na apresentação. Uma destas estruturas é o *Hypermedia Temporal Graph* (HTG) (COSTA et al., 2008), que guarda em si todas as relações de ações existentes, previsíveis ou não. No caso de eventos não-determinísticos, como por exemplo interação do usuário, o autor pode informar qual ação a ser tomada quando este evento ocorrer.

Para que essa estrutura possa ser criada, o *Debug Agent* monitora os barramentos e consegue determinar quais as relações de ações existem. Isso é possível pois todas as etapas da apresentação utilizam barramentos para a comunicação entre os componentes. Além disso, a especificação da arquitetura DoHyPE separa os barramentos por níveis de abstração: Parsing/Edição - Validação/Instanciação - Apresentação/Manutenção. Dessa forma, as relações existentes entre as ações podem ser determinadas. Além disso, pelo fato de ser possível saber em qual estado um componente está, as operações que devem ser realizadas para a execução do salto podem ser determinadas, por exemplo, parar uma mídia em reprodução ou exibir uma mídia que ainda não foi iniciada. Essas operações necessárias para a realização do salto são feitas através da injeção de mensagens pelo *Debug Agent* ao barramento, com destino aos componentes específicos.

5.2 AGENTE DE DEPURAÇÃO COM SUPORTE A ENCADEAMENTO DE EVENTOS

O protocolo de comunicação utilizado no barramento para a comunicação entre os componentes da arquitetura DoHyPE, possui um cabeçalho que fornece o *timestamp* da criação da mensagem e o identificador da origem (remetente). Com o identificador de origem, o *Debug Agent* consegue estabelecer a trilha de eventos, pois consegue determinar os responsáveis por cada evento, em cascata. Dessa forma, é possível oferecer um ferramental ao autor que permita saber a pilha de eventos ocorridos, dado um evento.

Com a informação do *timestamp*, o *Debug Agent* consegue situar um evento no tempo. Assim, além de saber a trilha de eventos, possibilita ao autor saber quando (momento temporal da apresentação) um determinado evento da trilha aconteceu. O *Debug Agent* pode dispor esses eventos numa linha do tempo, permitindo saber quando cada evento da trilha aconteceu de forma visual. Além disso, o *Debug Agent* pode oferecer uma Visão Espacial, permitindo ao autor navegar através dessa linha do tempo e obter uma captura de tela daquele instante selecionado.

A Figura 5.1 ilustra o documento NCL que será utilizado para descrever esse caso de uso. Um exemplo de visualização em tempo real é ilustrado pela Figura 5.2. Quando a apresentação é iniciada, o contexto que representa o <*body*> é também iniciado. Nesse momento, as mídias declaradas como portas do <*body*> são iniciadas, sendo elas video1, video2, video3 e imageBG. Essas ações de início de apresentação realizadas nesses componentes, tanto no contexto quanto nas mídias, são feitas através de mensagens do tipo *PRESENTATION_STARTS*. Decorrido um tempo de exibição do video3 (âncora temporal), as imagens image1 e image2 são iniciadas, sendo então exibidas. Logo após, devido a uma âncora temporal existente em video1, a mídia audio1 é iniciada. Em um espaço curto de tempo, a mídia video1 termina sua exibição, encerrando junto com ela a mídia video2 devido à existência de um elo. No momento corrente da apresentação, que é a parte mais à direita da janela, as mídias audio1, video3, image1, image2 e imageBG estão em apresentação. Outras funcionalidades podem ser disponibilizadas ao autor, como a lista de mídias que serão encerradas ou iniciadas com seu início ou término, bastando clicar na representação da mídia na janela. Esta visualização foi possível ser criada pelo fato dos eventos de início ou fim de reprodução dos objetos de mídia e âncoras terem sidos cap-

turados. É importante mencionar que esta mesma trilha de eventos pode ser construída pelo depurador para outros tipos de eventos, além dos eventos de apresentação. A Figura 5.3 ilustra a apresentação do documento NCL no *player* DoHyPE(Qt) for NCL.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<ncl id="main" xmlns="http://www.ncl.org.br/NCL3.0/EDTVProfile">
<head>
        <connectorBase>
            <causalConnector id="onBegin1StartN">
                <simpleCondition role="onBegin" max="1"/>
                <simpleAction role="start" />
            </causalConnector>
            <causalConnector id="onEnd1StopN">
                <simpleCondition role="onEnd" max="1"/>
                <simpleAction role="stop" />
            </causalConnector>
        </connectorBase>
</head>
<body>
        <port id="video1Port" component="video1"/>
        <port id="video2Port" component="video2"/>
        <port id="video3Port" component="video3"/>
        <port id="imgBGPort" component="imageBG"/>
        <media id="imageBG" src="imageBG.jpg">
                <property name="bounds" value="0,0%,100%,100%"/>
                <property name="transparency" value="80%"/>
        </media>
        <media id="video1" src="video1.mp4">
            <property name="bounds" value="0,54%,50%,50%"/>
            <area id="fsArea" begin="15s"/>
                <property name="fit" value="meet"/>
        </media>
        <media id="video2" src="video2.mp4">
                <property name="bounds" value="50%,54%,50%,50%"/>
                <property name="fit" value="meet"/>
        </media>
        <media id="video3" src="video3.mp4">
            <area id="tsArea" begin="9s"/>
                <property name="bounds" value="22.5%,0%,100%,54%"/>
                <property name="fit" value="meet"/>
        </media>
        <media id="audio1" src="audio1.mp3"/>
        <media id="image1" src="image1.jpg">
                <property name="bounds" value="0,88%,17%,17%"/>
                <property name="transparency" value="50%"/>
                <property name="fit" value="meet"/>
        </media>
        <media id="image2" src="image2.png">
                <property name="bounds" value="80%,93.5%,20%,20%"/>
                <property name="transparency" value="50%"/>
                <property name="fit" value="meet"/>
        </media>
        <link xconnector="onBegin1StartN">
            <bind role="onBegin" component="video3" interface="tsArea"/>
            <bind role="start" component="image1"/>
            <bind role="start" component="image2"/>
        </link>
        <link xconnector="onBegin1StartN">
            <bind role="onBegin" component="video1" interface="fsArea"/>
            <bind role="start" component="audio1"/>
        </link>
        <link xconnector="onEnd1StopN">
            <bind role="onEnd" component="video1"/>
            <bind role="stop" component="video2"/>
        </link>
</body>
</ncl>
```

Figura 5.1: Documento NCL para o caso de uso Trilha de Eventos

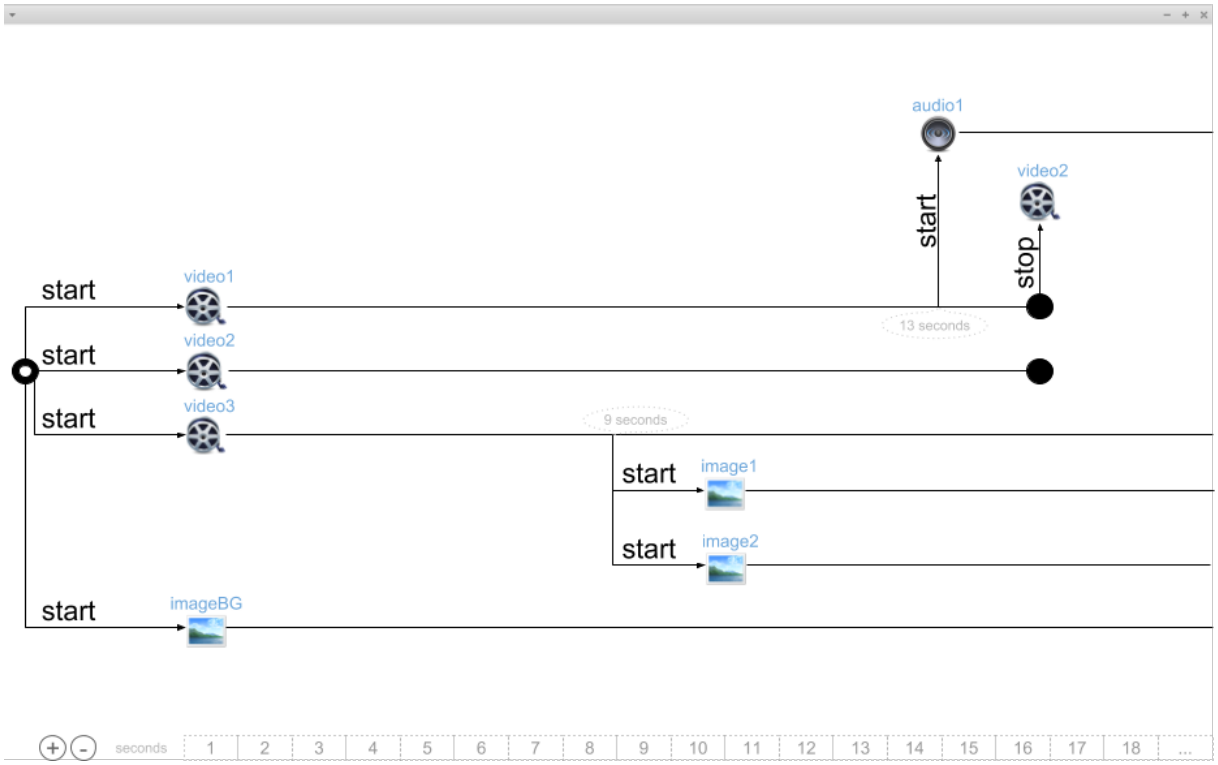

Figura 5.2: Ilustração de visão temporal dos eventos

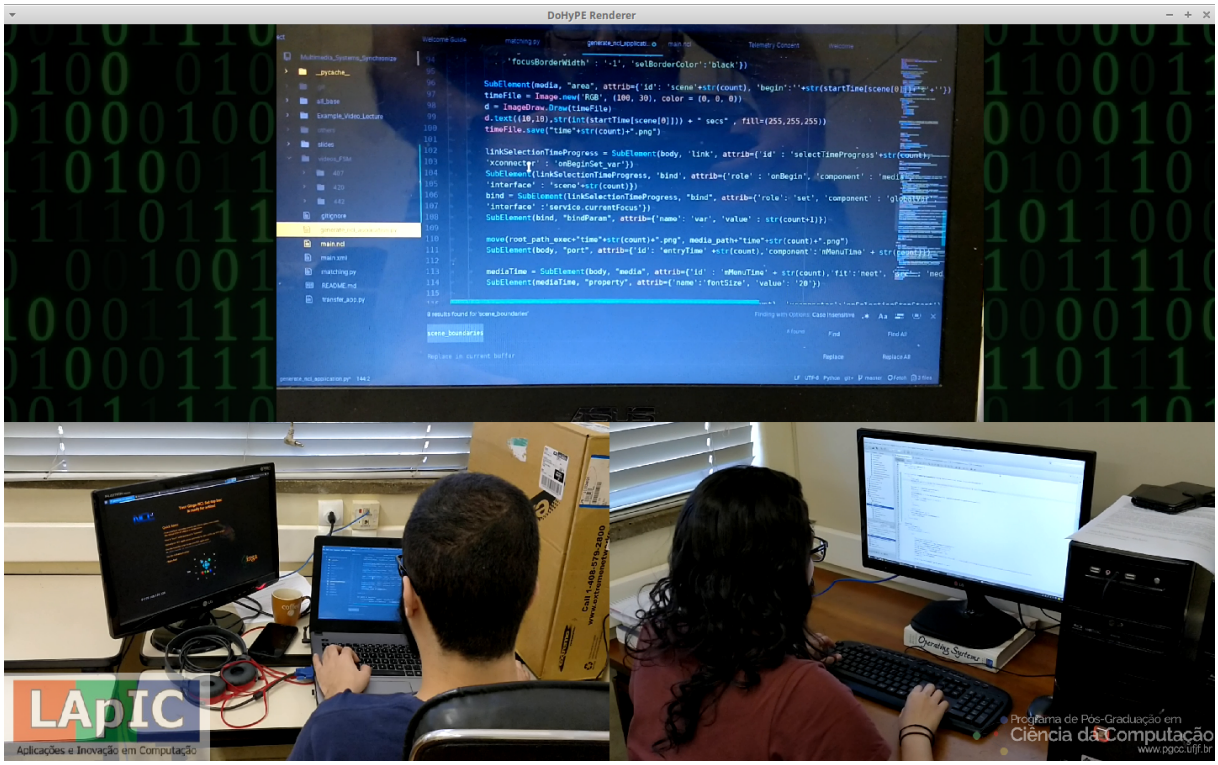

Figura 5.3: Apresentação do documento NCL (Figura 5.1) no *player* DoHyPE(Qt) for NCL

6 RESULTADOS OBTIDOS

Este capítulo apresenta os resultados obtidos nos experimentos realizados, com o intuito de avaliar a comunicação entre os módulos da arquitetura e o impacto que essa comunicação apresenta na reprodução da aplicação hipermídia. Para isso, foi utilizado uma aplicação hipermídia NCL com um conjunto de mídias e apenas um elo. O elo está condicionado ao início da primeira mídia para iniciar a reprodução das outras. Foram mensurados valores como o volume total de mensagens trafegadas pelos barramentos (quantidade e tamanho) e o atraso na reprodução de cada mídia do documento.

Para a realização dos experimentos, foi utilizado um computador com as seguintes configurações: Processador Intel® Core™ 2 Duo E6750, memória de 4GB e Sistema Operacional Xubuntu na versão 18.04.1 LTS. Pelo fato de utilizar mais do que 1024 descritores de arquivos (*socket*), foi necessário realizar uma alteração na configuração do Linux para permitir um limite maior de descritores de arquivos abertos.

Na execução dos experimentos, a mídia utilizada é uma imagem no formato JPEG, com tamanho 128 de largura e 128 de altura. Cada barramento é um processo do Sistema Operacional. Da mesma forma, a Máquina de Apresentação e o *Renderer* são processos distintos. A comunicação entre os módulos da Máquina de Apresentação é baseada em IPC, diferentemente do *Renderer*, que utiliza o protocolo UDP. Além disso, em cada cenário foram realizadas 9 iterações para a determinação da mediana e do desvio padrão. A coleta dos dados referentes ao tráfego da comunicação foi feita em cada barramento da instanciação da arquitetura. A coleta dos dados referentes aos atrasos na reprodução da mídia foi realizada no *Renderer*. Dessa forma, é possível coletar o momento em que uma mídia deveria ser iniciada e o momento em que ela foi realmente iniciada.

Para este experimento, foi utilizado um documento NCL com 10000 elementos <*media*> e apenas um elemento <*link*>. A porta do contexto que representa o elemento <*body*> é a primeira <*media*> definida. Dessa forma, o <*link*> está condicionado ao início da reprodução do elemento <*media*> associado a porta do <*body*> e tem como ação iniciar todos os outros elementos <*media*>. Esse documento está representado na Figura 6.1. Por questões de simplificação e otimização do espaço utilizado, não foram especificados todos os elementos NCL utilizados na Figura 6.1.

6.1 ATRASOS NA REPRODUÇÃO DO CONTEÚDO

A Tabela 6.1 mostra a quantidade de conexões, o total de mensagens recebidas e a quantidade total de dados (*bytes*) trafegados nos barramentos. Como pode ser visto, no barramento *Document Updating Message Bus* há 3 conexões. Em conformidade com a arquitetura, essas 3 conexões são os componentes: *Document Parser*, *Entity Manager* e o *Context Resolver*. O número de conexões do *Hypermedia Model Message Bus* é referente aos 10001 *Model Entity Instance*, sendo 10000 *media* e 1 *link*. Além disso, existem três componentes conectados à ele, que são *Context Resolver*, o *Entity Manager* e o *Programmable Interval Timer*. Por último, o *Distributed Event Bus* possui 10000 conexões referentes ao *Model Entity Instance* (*media*), a conexão do *Context Resolver*, do *Renderer* e do *Programmable Interval Timer*.

A Figura 6.2 ilustra o atraso para iniciar a reprodução do elemento *media* que está no papel de ação do <*link*>. Nela está representado o Desvio Padrão calculado para os experimentos realizados. Como representado no gráfico, os elementos que são iniciados mais posteriormente tem seu atraso um pouco maior. Isso ocorre por causa da notificação elemento a elemento a ser iniciado.

```
<?xml version="1.0" encoding="iso-8859-1"?>
<ncl id="main" xmlns="http://www.ncl.org.br/NCL3.0/EDTVProfile">
<head>
    <connectorBase>
        <causalConnector id="onBegin1StartN">
                <simpleCondition role="onBegin" max="1"/>
                <simpleAction role="start" max="unbounded"/>
            </causalConnector>
    </connectorBase>
</head>
<body>
    <port id="video1Port" component="image0"/>
        <media id="image0" src="img.jpg"/>
        <media id="image1" src="img.jpg"/>
        ...
        <media id="image9999" src="img.jpg"/>
        <link xconnector="onBegin1StartN" id="link1">
                <bind role="onBegin" component="image0"/>
                <bind role="start" component="image1"/>
                ...
                <bind role="start" component="image9999"/>
        </link>
</body>
</ncl>
```

Figura 6.1: Documento NCL utilizado no experimento

Barramento	Conexões	Mensagens	Dados (bytes)
Document Updating Message Bus	3	20007	2818371
Hypermedia Model Message Bus	10004	29966	1698394
Distributed Event Bus	10004	29961	2766406

Tabela 6.1: Tráfego de dados nos barramentos

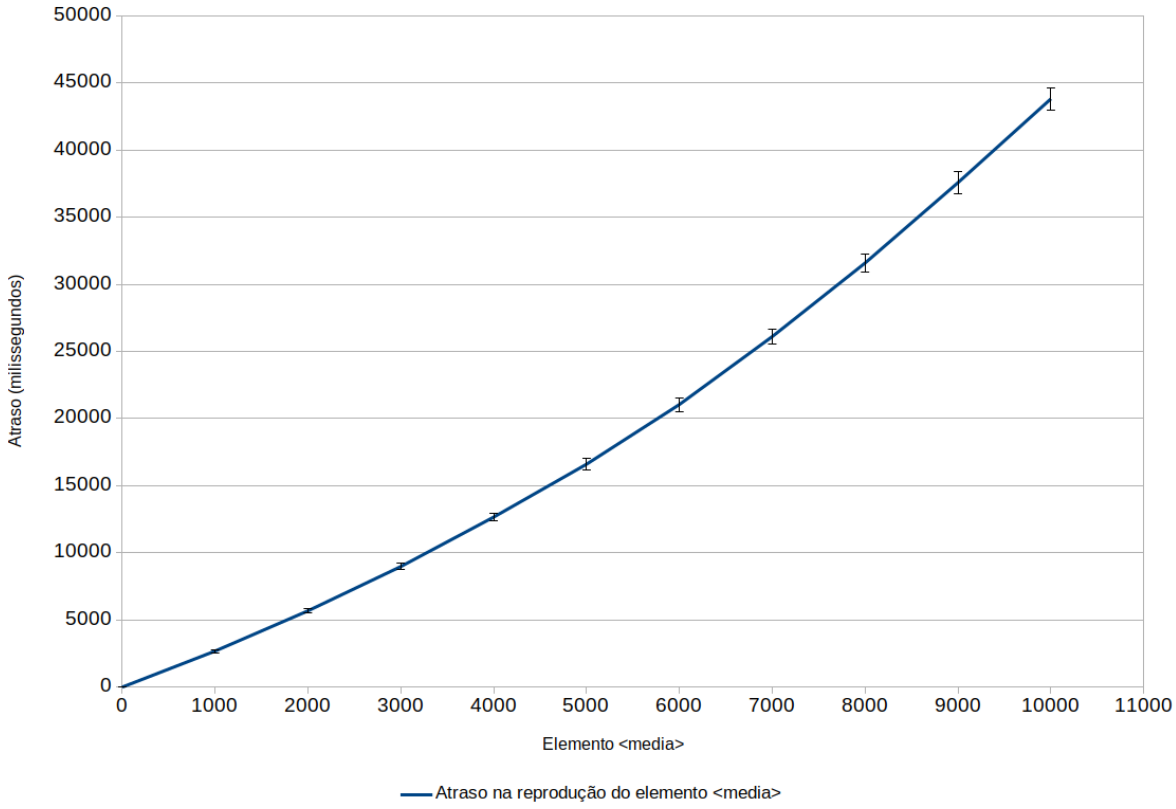

Figura 6.2: Gráfico do atraso na reprodução do elemento *media*

6.2 AUMENTO DOS DADOS TRAFEGADOS POR ELEMENTOS <*MEDIA*>

A Figura 6.3 ilustra o crescimento do número de conexões, mensagens e dados trafegados nos barramentos. O primeiro gráfico representa o crescimento do número de conexões estabelecidas ao barramento de acordo com o aumento da quantidade de elementos <*media*>. Como pode ser visto, o número de conexões realizadas ao barramento *Document Updating Message Bus* se mantém estável. Isso ocorre pelo fato das entidades do modelo não utilizarem esse barramento para comunicação. Diferentemente, os outros barramentos possuem a quantidade de conexões e dados proporcional ao número de elementos <*media*> existentes.

O segundo gráfico da Figura 6.3 representa o número de mensagens de acordo com o crescimento de elementos <*media*>. Observa-se que os barramentos *Hypemedia Model Message Bus* e *Distributed Event Bus* possuem uma quantidade maior de mensagens trafegadas. Além disso, essa quantidade aumenta proporcionalmente a medida que são encontrados elementos <*media*>. Isso ocorre por causa desses barramentos serem utilizados para a comunicação entre as entidades do modelo e para o controle dos componentes *Player* pelas instâncias de entidade do modelo.

O último gráfico da Figura 6.3 representa a quantidade de dados, em bytes, que foram trafegados nos barramentos a medida que cresce o número de elementos <*media*> . Pode-se notar que, apesar do barramento *Hypemedia Model Message Bus* possuir um dos maiores números de conexões, ele é o que teve o menor tráfego de dados. Isso acontece pois as mensagens trafegadas pelo *Hypemedia Model Message Bus* possuem poucos ou nenhum dados no corpo da mensagem. Já o corpo da mensagem que trafega pelo *Distributed Event Bus* possui dados que são necessários aos *Players*, por exemplo, a origem do conteúdo. O mesmo ocorre com o *Document Updating Message Bus*, pelo fato dessas informações repassadas aos *Players* serem identificadas no documento pelo *Document Parser* e repassadas ao *Entity Manager*.

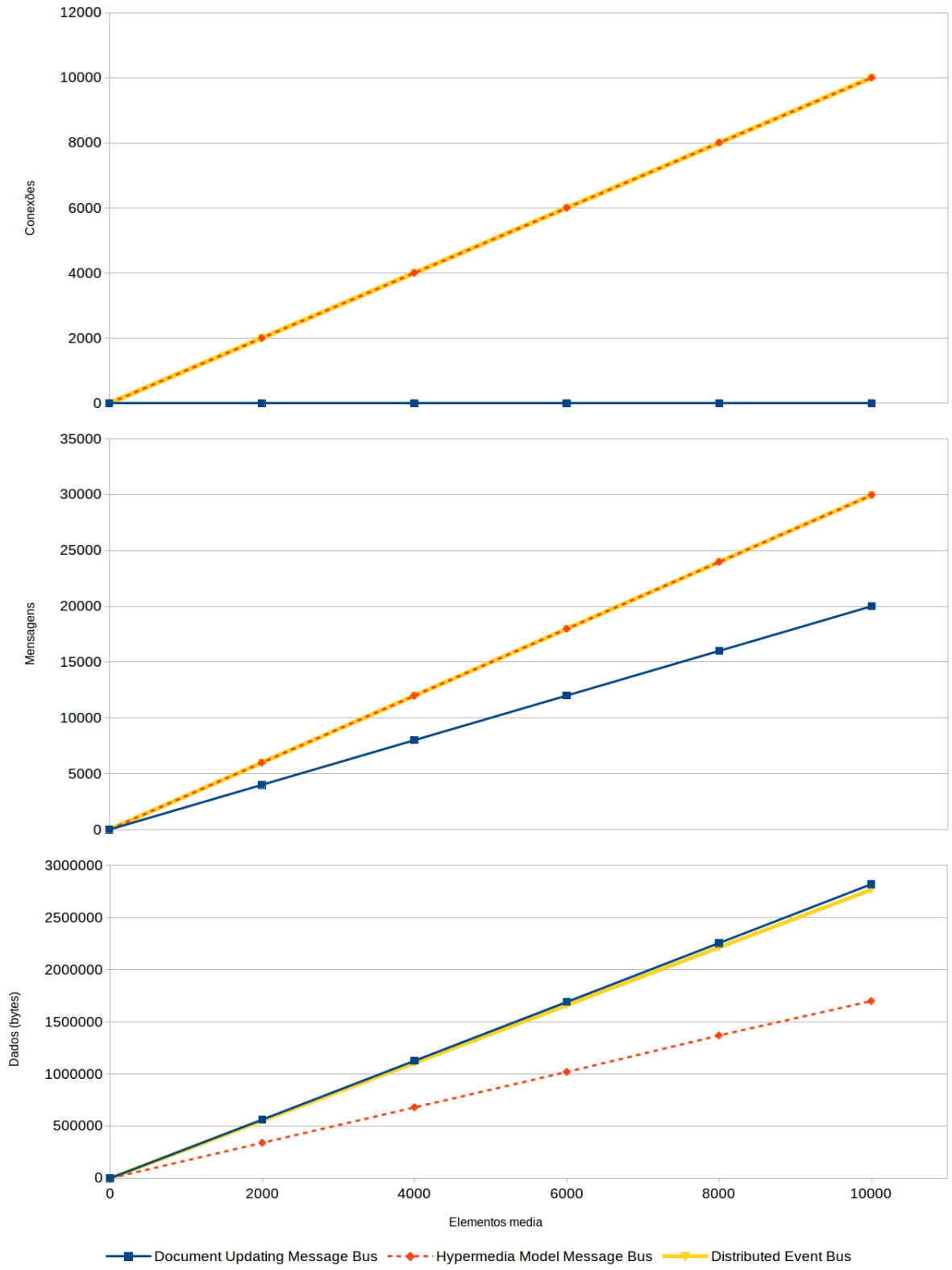

Figura 6.3: Gráficos da quantidade de conexões, mensagens e dados nos barramentos por elementos <*media*>

7 CONCLUSÃO

Nesta dissertação foi descrita uma arquitetura denominada DoHyPE, que visa auxiliar o processo de autoria de aplicações hipermídia. Ela proporciona ao desenvolvedor não apenas a apresentação do documento, mas também mecanismos que permitem uma depuração detalhada, que incluem o acesso a informações do que está acontecendo na apresentação, assim como o controle a partir da injeção de eventos pelo depurador. Para isso, é proposta uma arquitetura na qual a comunicação entre os componentes é compartilhada, permitindo o monitoramento e a inserção de mensagens.

Foi também apresentada uma instanciação da arquitetura para o suporte a documentos hipermídia escritos em Linguagem NCL. Pelo fato do *framework* Qt possuir diversas funcionalidades que facilitam o desenvolvimento multimídia, ele foi utilizado para simplificar este desenvolvimento. Para isso, a instanciação foi realizada em duas etapas. Na primeira etapa, foi feita a instanciação da arquitetura em Qt (*DoHyPE(Qt)*). Na segunda etapa, foi realizada a *DoHyPE(Qt) for NCL*, que é a instanciação da arquitetura para a linguagem NCL usando *DoHyPE(Qt)* como base. Além disso, casos de uso da arquitetura foram apresentados.

Os resultados mostraram que quanto maior o número de instâncias de entidade do modelo, maior o número de conexões existente nos barramentos de comunicação das entidades do modelo e o barramento distribuído. Além disso, apesar do barramento para comunicação das entidades do modelo possuir o maior número de conexões, é o que teve o menor tráfego de dados. Os resultados também mostraram que o barramento utilizado para a edição do documento permanece com a quantidade de conexões constante, mesmo com o aumento no número de instâncias de entidade do modelo.

Espera-se com este trabalho que a fase de autoria de aplicações hipermídias se torne uma tarefa menos árdua, possibilitando a ampliação na gama de aplicações e uso de serviços multimídia, como os serviços de TV Digital, IPTV e IBB. Como trabalho futuro propõe-se analisar o impacto da comunicação entre os componentes da arquitetura proposta em máquinas de apresentação não voltadas a autoria. Outra proposta é a possibilidade de descrições formais de modelos hipermídias que a própria ferramenta de exibição/depuração pode carregar e gerar as entidades do modelo respectivo. Dessa forma,

as entidades do modelo são geradas automaticamente através do documento que as descreve, tornando o exibidor/depurador dinâmico para reproduzir documentos hipermídia de modelos distintos. Isso torna a implementação do exibidor/depurador não dependente de um modelo específico. Outra proposta é realizar um estudo de como a Máquina de Apresentação pode ser usada com a instanciação de mais de um modelo concomitantemente, por exemplo, a utilização de NCL em conjunto com SMIL. Essa instanciação de múltiplos modelos permite a existência de elementos de diferentes linguagens hipermídia interoperando entre si.

REFERÊNCIAS

ANTONACCI, M. J.; MUCHALUAT-SAADE, D.; RODRIGUES, R.; SOARES, L. F. G. Ncl: Uma linguagem declarativa para especificação de documentos hipermídia na web. **VI Simpósio Brasileiro de Sistemas Multimídia e Hipermídia-SBMídia2000, Natal, Rio Grande do Norte**, 2000.

Apple Inc. **Safari**. 2003. `https://developer.apple.com/safari/tools/`.

ASSOCIAÇÃO BRASILEIRA DE NORMAS TÉCNICAS. **NBR 15606-2**, abril 2008. Disponível em: <`http://www.abnt.org.br/imagens/Normalizacao_TV_Digital/ABNTNBR15606-2_2007Vc_2008.pdf`>. Acesso em: Junho, 2013.

AZEVEDO, R. G. A.; ARAÚJO, E. C.; LIMA, B.; SOARES, L. F. G.; MORENO, M. F. Composer: meeting non-functional aspects of hypermedia authoring environment. **Multimedia Tools and Applications**, v. 70, n. 2, p. 1199–1228, May 2014. ISSN 1573-7721. Disponível em: <https://doi.org/10.1007/s11042-012-1216-8>.

AZEVEDO, R. G. A.; NETO, C. d. S. S.; TEIXEIRA, M. M.; SANTOS, R. C. M.; GOMES, T. A. Textual authoring of interactive digital tv applications. In: **Proceedings of the 9th European Conference on Interactive TV and Video**, 2011. (EuroITV '11), p. 235–244. ISBN 978-1-4503-0602-7. Disponível em: <http://doi.acm.org/10.1145/2000119.2000169>.

BENVENISTE, A.; BERRY, G. The synchronous approach to reactive and real-time systems. **Proceedings of the IEEE**, v. 79, n. 9, p. 1270–1282, Sep 1991. ISSN 0018-9219.

BULTERMAN, D. C.; RUTLEDGE, L. W. **SMIL 3.0: Flexible Multimedia for Web, Mobile Devices and Daisy Talking Books**, 2008.

COSTA, R. M. d. R.; MORENO, M. F.; SOARES, L. F. G. Intermedia synchronization management in dtv systems. In: **Proceedings of the Eighth ACM Symposium on Document Engineering**, 2008. (DocEng '08), p. 289–297. ISBN 978-1-60558-081-4. Disponível em: <http://doi.acm.org/10.1145/1410140.1410203>.

DAMASCENO, J.; SANTOS, J. dos; MUCHALUAT-SAADE, D. Editec: Hypermedia composite template graphical editor for interactive tv authoring. In: **Proceedings of the 11th ACM Symposium on Document Engineering**, 2011. (DocEng '11), p. 77–80. ISBN 978-1-4503-0863-2. Disponível em: <http://doi.acm.org/10.1145/2034691.2034708>.

ECMA International. **The JSON Data Interchange Format**, October 2013.

EUGSTER, P. T.; FELBER, P. A.; GUERRAOUI, R.; KERMARREC, A.-M. The many faces of publish/subscribe. **ACM Comput. Surv.**, ACM, New York, NY, USA, v. 35, n. 2, p. 114–131, jun. 2003. ISSN 0360-0300. Disponível em: <http://doi.acm.org/10.1145/857076.857078>.

FREESZ JR., M. A.; YUNG, L.; MORENO, M. Storm: A hypermedia authoring model for interactive digital out-of-home media. In: **Proceedings of the 23rd Brazillian Symposium on Multimedia and the Web**, 2017. (WebMedia '17), p. 41–48. ISBN 978-1-4503-5096-9. Disponível em: <http://doi.acm.org/10.1145/3126858.3126889>.

Google Inc. **Google Chrome**. 2008. `https://www.google.com/chrome/`.

HOSSFELD, T.; SKORIN-KAPOV, L.; HEEGAARD, P. E.; VARELA, M. A new qoe fairness index for qoe management. **Quality and User Experience**, v. 3, n. 1, p. 4, Feb 2018. ISSN 2366-0147. Disponível em: <`"https://doi.org/10.1007/s41233-018-0017-x"`>.

IEEE Standard for a Precision Clock Synchronization Protocol for Networked Measurement and Control Systems. **IEEE Std 1588-2008 (Revision of IEEE Std 1588-2002)**, p. 1–300, July 2008.

ITU-R. **Integrated broadcast broadband system**, sep 2017. Disponível em: <https://www.itu.int/rec/R-REC-BT.2075-1-201701-I/en>.

ITU-T. **IPTV functional architecture**, sep 2008. Disponível em: <`https://www.itu.int/rec/dologin_pub.asp?lang=e&id=T-REC-Y.1910-200809-I!!PDF-E&type=items`>.

ITU-T. **Nested context language (NCL) and Ginga-NCL**, nov 2014. Disponível em: <https://www.itu.int/rec/T-REC-H.761-201411-I/en>.

LIMA, G. F.; AZEVEDO, R. G. d. A.; COLCHER, S.; HAEUSLER, E. H. Converting ncl documents to smix and fixing their semantics and interpretation in the process. In: **Proceedings of the 23rd Brazillian Symposium on Multimedia and the Web**, 2017. (WebMedia '17), p. 109–116. ISBN 978-1-4503-5096-9. Disponível em: <http://doi.acm.org/10.1145/3126858.3126876>.

MATTOS, D. Paulo de; SILVA, J. Varanda da; MUCHALUAT-SAADE, D. C. Next: Graphical editor for authoring ncl documents supporting composite templates. In: **Proceedings of the 11th European Conference on Interactive TV and Video**, 2013. (EuroITV '13), p. 89–98. ISBN 978-1-4503-1951-5. Disponível em: <http://doi.acm.org/10.1145/2465958.2465964>.

MEKAHLIA, F. Z.; GHOMARI, A.; YAZID, S.; DJENOURI, D. Temporal and spatial coherence verification in smil documents with hoare logic and disjunctive constraints: a hybrid formal method. **Journal of Integrated Design and Process Science**, IOS Press, v. 20, n. 3, p. 39–70, 2016.

MORENO, M. F.; SANTOS, R.; LIMA, G.; MORENO, M.; SOARES, L. F. Deepening the separation of concerns in the implementation of multimedia systems. In: **Proceedings of the 31st Annual ACM Symposium on Applied Computing**, 2016. (SAC '16), p. 1337–1343. ISBN 978-1-4503-3739-7. Disponível em: <http://doi.acm.org/10.1145/2851613.2851769>.

Mozilla Foundation. **Mozilla Firefox**. 2004. `https://developer.mozilla.org/en-US/docs/Mozilla/Firefox`.

PARNIN, C.; ORSO, A. Are automated debugging techniques actually helping programmers? In: ACM. **Proceedings of the 2011 international symposium on software testing and analysis**, 2011. p. 199–209.

PROJECT, Q. **Qt Documentation**. 2018. `http://doc.qt.io/qt-5/reference-overview.html`.

PUC-RIO. **Ginga-NCL Virtual Set-top Box**. 2013. `http://www.softwarepublico.gov.br/dotlrn/clubs/ginga/gingancl/`. Accesso em 27 de Junho de 2013.

RIVEST, R. L. **The MD5 Message-Digest Algorithm**, April 1992. `http://www.rfc-editor.org/rfc/rfc1321.txt`. Disponível em: <http://www.rfc-editor.org/rfc/rfc1321.txt>.

SANTOS, J. A. F. dos. Multimedia and hypermedia document validation and verification using a model-driven approach. **Master's thesis, UFF**, 2012.

SANTOS, J. dos; BRAGA, C.; MUCHALUAT-SAADE, D. C. A rewriting logic semantics for ncl. **Science of Computer Programming**, Elsevier, v. 107, p. 64–92, 2015.

SANTOS, R. C. M.; MORENO, M. F.; SOARES, L. F. G. An architecture to assist multimedia application authors and presentation engine developers. In: **2015 IEEE International Conference on Multimedia and Expo (ICME)**, 2015. p. 1–6. ISSN 1945-7871.

SILVA, J. Debugging techniques for declarative languages: Profiling, program slicing and algorithmic debugging. **Ai Communications**, IOS Press, v. 21, n. 1, p. 91–92, 2008.

SOARES, L. F. G.; MORENO, M. F.; MORENO, M. F. Transmissão de aplicações e comandos de edição ao vivo em sistemas de tv digital. **PUC-Rio. Rio de Janeiro**, 2009.

SOARES, L. F. G.; RODRIGUES, R. F. **Nested Context Model 3.0. Part 1 - NCM Core**, May 2005.

SOARES, L. F. G.; RODRIGUES, R. F. **Nested Context Model 3.0: Part 8 - NCL (Nested Context Language) NCL Digital TV Profiles**, 10 2006.

SOARES, L. F. G.; RODRIGUES, R. F.; MORENO, M. F. Ginga-ncl: the declarative environment of the brazilian digital tv system. **Journal of the Brazilian Computer Society**, scielo, v. 12, p. 37 – 46, 03 2007. ISSN 0104-6500. Disponível em: <`http://www.scielo.br/scielo.php?script=sci_arttext&pid=S0104-65002007000100005&nrm=iso`>.

SOMMERVILLE, I. **Engenharia de Software**, 2011.

STALLMAN, R. M.; PESCH, R.; SHEBS, S. et al. **Debugging with GDB: The GNU Source-Level Debugger**, 2002. viii + 344 p. ISBN 1-882114-88-4. Disponível em: <http://www.gnupress.org/book7.html>.

WATT, D. A.; FINDLAY, W.; HUGHES, J. **Programming language concepts and paradigms**, 1990.

ZELLER, A.; LUTKEHAUS, D. Ddd - a free graphical front-end for unix debuggers. **SIGPLAN Notices**, v. 31, n. 1, p. 22–27, 1996.